PBeFS — Parallel Best-First Search

A manual of the algorithms, their proofs and their code

Donato Meoli

Contents

PBeFS — Parallel Best-First Search **7**
Abstract . . . 7
License . . . 7
Conventions . . . 7
Table of contents (high-level) . . . 8

1. Introduction and notation **9**
1.1 The state-space problem . . . 9
1.2 What makes a problem hard . . . 11
1.3 The measures of a search . . . 12
1.4 The conventions of the pseudocode . . . 14
1.5 The concepts of the library . . . 15
1.6 Workers, executors and options of the parallel searches . . . 22
1.7 The plan of the book . . . 23
1.8 Notes and references . . . 24
1.9 Exercises . . . 25

2. Best-first search and A* **26**
2.1 Searching a tree and searching a graph . . . 26
2.2 The priority . . . 28
2.3 A* in the library . . . 29
2.4 Admissible and consistent heuristics . . . 31
2.5 The correctness of A* . . . 34
2.6 Consistent heuristics . . . 36
2.7 Reopening with an inconsistent heuristic . . . 37
2.8 The ties within the last layer . . . 37
2.9 Pruning the move back to the parent . . . 38
2.10 The cost of a search and its counters . . . 39
2.11 The goal test at generation . . . 41
2.12 Greedy best-first search . . . 42
2.13 Enforced hill-climbing . . . 49
2.14 Iterated searches . . . 50
2.15 Width-based search . . . 52
2.16 Benchmark . . . 56
2.17 Notes and references . . . 57
2.18 Exercises . . . 58

3. The data structures of search **60**
3.1 The operations of a search . . . 60
3.2 Heaps . . . 61
3.3 The two-level bucket queue . . . 62
3.4 Packed keys . . . 65
3.5 Breaking the ties . . . 66
3.6 CLOSED: a hash table with open addressing . . . 67
3.7 The arena of the nodes . . . 69
3.8 Prefetching the lookups . . . 70
3.9 Perfect hashing . . . 71

3.10 Transposition tables . . . 72
3.11 Immediate and delayed duplicate detection . . . 72
3.12 The sets of data structures . . . 72
3.13 Benchmark . . . 74
3.14 Notes and references . . . 74
3.15 Exercises . . . 75

4. Memory-bounded and external search 76
4.1 What limits A* . . . 76
4.2 Beam search . . . 77
4.3 Depth-first branch-and-bound . . . 79
4.4 Iterative deepening . . . 79
4.5 Recursive best-first search . . . 84
4.6 External search with delayed duplicate detection . . . 86
4.7 Symbolic search . . . 91
4.8 Novelty and width-based search . . . 92
4.9 Partial expansion . . . 93
4.10 Fringe search . . . 95
4.11 The bounds of iterative deepening . . . 97
4.12 Real-time search . . . 99
4.13 Hierarchical path-finding . . . 101
4.14 Baselines: breadth-first and cost-bounded depth-first search . . . 102
4.15 Benchmark . . . 103
4.16 An example . . . 104
4.17 Notes and references . . . 104
4.18 Exercises . . . 105

5. Heuristics and pattern databases 107
5.1 Relaxations and abstractions . . . 107
5.2 The heuristics of the domains of the library . . . 108
5.3 The abstraction of a pattern . . . 109
5.4 Ranking the placements . . . 110
5.5 Building a table . . . 111
5.6 Additive pattern databases . . . 113
5.7 Consistency . . . 116
5.8 The reflection about the main diagonal . . . 117
5.9 The encodings of the entries . . . 117
5.10 Files and the cache of the tables . . . 120
5.11 Sizes . . . 121
5.12 When the heuristic does not help . . . 121
5.13 The pattern databases of any abstraction . . . 122
5.14 Ranking the k-permutations . . . 125
5.15 The pattern databases of the permutation puzzles . . . 127
5.16 The heuristics of the permutation puzzles . . . 130
5.17 Benchmark . . . 132
5.18 Notes and references . . . 133
5.19 Exercises . . . 134

6. Bounded-suboptimal, anytime and incremental search 136
6.1 The problem and the interface . . . 136
6.2 The weight and the rules of a weighted search . . . 138
6.3 Weighted A* and weighted IDA* . . . 142
6.4 The weighted and anytime parallel searches . . . 143
6.5 Anytime search . . . 144
6.6 Searches with a second order . . . 148
6.7 Batch weighted A* . . . 153
6.8 Incremental search . . . 154
6.9 Hash-distributed incremental search . . . 158
6.10 Benchmark . . . 163

6.11 Notes and references . . . 164
6.12 Exercises . . . 164

7. Bidirectional and external-memory search 167
7.1 The problem and the interface . . . 167
7.2 The two fronts . . . 169
7.3 The lower bounds . . . 172
7.4 One node at a time: MM, fMM, BAE* and the bidirectional Dijkstra's algorithm . . . 173
7.5 Expanding pairs: NBS and the must-expand graph . . . 175
7.6 Expanding clusters: DVCBS . . . 176
7.7 GBFHS and the bidirectional breadth-first search . . . 177
7.8 The two directions in parallel . . . 178
7.9 Parallel external-memory bidirectional search . . . 180
7.10 A* with immediate duplicate detection on the disk . . . 183
7.11 Benchmark . . . 185
7.12 Notes and references . . . 186
7.13 Exercises . . . 186

8. Parallel A* in shared memory 188
8.1 The interface of the parallel searches . . . 188
8.2 The overheads of a parallel search . . . 190
8.3 Correctness and termination of a parallel search . . . 191
8.4 Simple Parallel A* . . . 193
8.5 The shared closed list . . . 197
8.6 MultiQueue A* . . . 198
8.7 The master-worker farm . . . 202
8.8 Parallel A* for slow expansions . . . 205
8.9 Parallel A* for slow evaluations of the edges . . . 210
8.10 The searches compared . . . 215
8.11 Benchmarks . . . 216
8.12 Notes and references . . . 218
8.13 Exercises . . . 218

9. Hash-distributed search 220
9.1 The search and its interface . . . 220
9.2 Hash Distributed A* . . . 222
9.3 The messages and the end of the search . . . 228
9.4 The work distributions . . . 230
9.5 Incremental Zobrist hashing . . . 236
9.6 HDA* over MPI . . . 237
9.7 The engineering of the implementation . . . 239
9.8 Benchmark . . . 240
9.9 Notes and references . . . 241
9.10 Exercises . . . 241

10. Automatic work distribution 243
10.1 The problem and the interface . . . 243
10.2 The sample and the transition structure . . . 247
10.3 The estimates . . . 249
10.4 The key as a linear map . . . 251
10.5 Cutting the transition graphs . . . 253
10.6 State abstractions: AHDA* and DAHDA* . . . 256
10.7 Operator-based Zobrist hashing: OZHDA* . . . 257
10.8 Kernel selection: KZHDA* . . . 258
10.9 The automatic choice . . . 260
10.10 The engineering of the implementation . . . 262
10.11 Benchmark . . . 262
10.12 Notes and references . . . 263
10.13 Exercises . . . 263

11. Parallel best-nblock-first search 265
11.1 The problem and the interface . . . 265
11.2 Structured duplicate detection . . . 268
11.3 Parallel best-nblock-first search . . . 270
11.4 Livelock and the hot nblocks . . . 271
11.5 The pseudocode . . . 272
11.6 Analysis . . . 277
11.7 The implementation . . . 282
11.8 The benchmark . . . 285
11.9 Notes and references . . . 286
11.10 Exercises . . . 286

12. Further parallel schemes 288
12.1 The searches and their interface . . . 288
12.2 Parallel structured duplicate detection . . . 290
12.3 K-best-first search . . . 294
12.4 A portfolio of A* searches . . . 296
12.5 Synchronous messages and the lending of nodes . . . 297
12.6 Transposition-table driven scheduling . . . 298
12.7 Parallel iterative deepening below a frontier . . . 301
12.8 Parallel greedy best-first search . . . 305
12.9 Two baselines . . . 311
12.10 The searches compared . . . 312
12.11 Benchmark . . . 313
12.12 Notes and references . . . 313
12.13 Exercises . . . 314

13. Branch-and-bound and its parallel schemes 316
13.1 The problem and its interface . . . 317
13.2 Nodes, bounds and the incumbent . . . 319
13.3 The sequential search . . . 321
13.4 Correctness and the reported bound . . . 324
13.5 The selection rules . . . 326
13.6 A* as a special case . . . 330
13.7 What a parallel search has to guarantee . . . 330
13.8 One pool of open nodes . . . 332
13.9 One pool per worker . . . 338
13.10 The schemes compared and chosen by name . . . 346
13.11 Reoptimization . . . 347
13.12 The interface for a branch-and-bound solver . . . 348
13.13 Lazy children, indexed children and the modes . . . 349
13.14 The search skeletons . . . 351
13.15 The racing ramp-up . . . 355
13.16 Checkpoints . . . 357
13.17 Master, hubs and workers over MPI . . . 359
13.18 Intervals of leaves . . . 360
13.19 The example problems . . . 364
13.20 Benchmark . . . 367
13.21 Notes and references . . . 369
13.22 Exercises . . . 370

14. The example domains 373
14.1 The domains and their concepts . . . 373
14.2 The sliding-tile puzzles . . . 375
14.3 The sequence score of Nilsson . . . 379
14.4 Pathfinding on grids . . . 380
14.5 Explicit graphs and the map of Romania . . . 383
14.6 Four problems of the textbook . . . 383
14.7 The traveling salesperson problem . . . 388

14.8 Slow expansions . . . 390
14.9 The pancake problems . . . 391
14.10 TopSpin . . . 392
14.11 Rubik's cube . . . 393
14.12 The Towers of Hanoi with four pegs . . . 394
14.13 The road graphs of DIMACS . . . 394
14.14 The navigation of a square robot . . . 395
14.15 The boards of Burns et al. and the refinements of the grids . . . 396
14.16 Problems not in the library . . . 399
14.17 Running the domains . . . 399
14.18 Notes and references . . . 400
14.19 Exercises . . . 401

15. Multiple sequence alignment and the instances 403
15.1 The problem and the interface . . . 403
15.2 The costs . . . 405
15.3 The pairwise heuristic . . . 406
15.4 The features, the abstractions and the nblocks . . . 407
15.5 Levelled graphs and the hyperplane work distribution . . . 408
15.6 The instance sets . . . 410
15.7 The readers of the instance files . . . 412
15.8 Running the instances . . . 414
15.9 The benchmark . . . 414
15.10 Notes and references . . . 415
15.11 Exercises . . . 415

16. Classical planning 416
16.1 SAS+ tasks and their STRIPS view . . . 416
16.2 The translator from PDDL . . . 418
16.3 The domain Planning . . . 419
16.4 The delete relaxation: $h^{\max}$, h^{add} and h^{FF} . . . 423
16.5 The landmark-cut heuristic . . . 425
16.6 Abstraction heuristics . . . 426
16.7 Features and abstractions for the parallel searches . . . 431
16.8 The goal test at the generation . . . 432
16.9 The searches on planning tasks . . . 433
16.10 The tasks of the data . . . 433
16.11 The benchmark . . . 434
16.12 Notes and references . . . 435
16.13 Exercises . . . 435

17. Multi-objective search 436
17.1 The problem and its interface . . . 436
17.2 The framework of the sequential searches . . . 438
17.3 The frontiers and the queue of OPEN . . . 440
17.4 NAMOA* with dimensionality reduction . . . 441
17.5 Searches on several orders of the objectives . . . 441
17.6 Hash-distributed multi-objective search . . . 443
17.7 The domains . . . 445
17.8 Benchmark . . . 446
17.9 Notes and references . . . 446
17.10 Exercises . . . 446

18. Symbolic search 448
18.1 The diagrams and their runtime . . . 448
18.2 The problem and its interface . . . 449
18.3 The encoding and the relations . . . 450
18.4 The heuristic as a partition of the states . . . 451
18.5 Pattern databases, perimeters and abstractions . . . 451
18.6 BDDA* . . . 453

18.7 The bidirectional search 453
18.8 The k cheapest plans 454
18.9 The parallel images 455
18.10 The domains 455
18.11 The library of BDDs and the limits 456
18.12 Notes and references 456
18.13 Exercises 456

Glossary **457**

Bibliography **458**

PBeFS — Parallel Best-First Search

Donato Meoli *Dipartimento di Informatica, Università di Pisa* `donato.meoli@outlook.com`

A manual of the algorithms of PBeFS, their proofs and their code

---

Abstract

PBeFS is a C++-20 library for the best-first search of an implicit state-space graph, i.e., A* and its variants, and of a tree with an incumbent, i.e., branch-and-bound, sequentially and in parallel on C++ threads, OpenMP, FastFlow and MPI. A problem is given to the library as a `Domain` (the initial state, the successors of a state with the costs of the edges, the goal test and the heuristic) or as a branch-and-bound problem (the root, the branching, the bound and the value of a solution), and every search of the library runs on every problem that models the concepts it needs.

This manual describes each search of the library in the same way: the idea, the invariants it keeps, figures of the state of the search before and after a step, the pseudocode (the same as in the Doxygen comments of the headers), the analysis (optimality, bounded suboptimality, termination of the parallel searches) as lemmas and theorems with their proofs, the excerpt of the code that implements it, taken from the headers at build time, and the engineering that makes the implementation fast. Each chapter ends with the benchmark of its searches and with exercises, each of which is checked by a test of the library. The manual is a companion to the API reference and to the README of the repository, and it is available both as a web site and as a PDF.

License

PBeFS is distributed under the GNU Lesser General Public License v3.0 (LGPL-3); this manual is distributed under the same terms.

Conventions

- **Class and function names** are typeset in `monospace`, e.g., `AStar`, `HDAStar::solve()`, and so are the files of the repository, e.g., `include/PBeFS/astar/AStar.h`; a precise pointer into the source has the form `file:line`.
- **Each chapter** starts with a line `[Source: ...]` that lists the files it describes.
- **Mathematical notation** is fixed in Chapter 1; in particular, s_0 is the initial state, $c(u, v)$ the cost of the edge (u, v), g, h and $f = g + h$ the usual functions of A*, z the incumbent and p the number of workers.
- **Pseudocode** has numbered lines and the conventions of the Doxygen comments of the headers, which are listed in §1.4.
- **Lemmas, theorems and exercises** are numbered within their chapter; the end of a proof is marked by □. Each exercise names the test (`ctest -R PBeFS.<case>`) that checks its solution.
- **Benchmark tables** are written by `manual/build/tables.py` from the results of the benchmark tools of the repository; a table whose results have not been measured yet on an idle machine is an empty table marked as a placeholder.
- **Status admonitions** flag the maturity of a feature explicitly. Four forms appear in the text:
  - **Status — mature.** The feature is stable, tested and in tagged releases. Implicit unless stated.

  - **Status — beta.** The feature is in `main` but its interface or semantics may change in incompatible ways.
  - **Status — under development.** The feature is partly present or is being written on a branch; the manual describes what exists today and the direction of the work.
  - **Status — planned.** The feature is documented as intended but is not yet written. The manual indicates explicitly that it cannot be used today.

When in doubt about a claim made in this manual, the authoritative sources are, in order of decreasing specificity: the headers of the repository at https://github.com/dmeoli/PBeFS, and the API reference.

Table of contents (high-level)

- **Part I — Sequential search.** Chapter 1 introduces the problem, the notation and the concepts of the library; Chapter 2 treats best-first search and A*; Chapter 3 the data structures of a search, i.e., the open and closed lists, the bucket and radix queues and the hash tables; Chapter 4 the searches whose memory is bounded or on disk, among them iterative deepening and the delayed duplicate detection; Chapter 5 the heuristics and the pattern databases; Chapter 6 the bounded-suboptimal, anytime and incremental searches; Chapter 7 the bidirectional and external-memory searches.
- **Part II — Parallel search.** Chapter 8 treats the parallel searches on a shared memory; Chapter 9 the hash-distributed search and its distributions; Chapter 10 the automatic choice of the distribution; Chapter 11 the parallel best-nblock-first search; Chapter 12 further parallel schemes.
- **Part III — Branch-and-bound.** Chapter 13 treats branch-and-bound and its parallel schemes.
- **Part IV — Problems.** Chapter 14 describes the example domains of the library; Chapter 15 the multiple sequence alignment and the sets of instances; Chapter 16 classical planning.
- **Part V — Multi-objective search.** Chapter 17 treats the searches whose edges cost a vector and whose answer is the Pareto front, sequential and parallel.
- **Part VI — Symbolic search.** Chapter 18 treats the searches on sets of states represented by binary decision diagrams, with their heuristics, abstractions and parallel images.
- **Back matter.** The Glossary and the Bibliography.

1. Introduction and notation

[Source: include/PBeFS/astar/Domain.h (the concepts of a problem), include/PBeFS/astar/Solution.h (Solution, SearchStats, SearchOptions), include/PBeFS/astar/Node.h (cost_slack, infinite_cost), include/PBeFS/core/SearchUtils.h (ParallelParams, run_workers, pin_worker), include/PBeFS/astar/HDAStar.h (ParallelOptions)]

We are concerned with finding a path of least cost from an initial state to a goal state of a graph that is far too large to be written down, but whose states and edges can be generated on demand from a few rules (the moves of a puzzle, the roads of a map, the actions of a plan). This chapter fixes the problem and the notation shared by the whole book (§1.1), discusses what makes an instance of the problem hard (§1.2) and by which measures a search is judged (§1.3), and lists the conventions of the pseudocode (§1.4). It then describes how a problem is handed to the library, i.e., the concept `Domain` and its refinements, and what a search returns (§1.5), how the parallel searches are run on threads, OpenMP, FastFlow or MPI (§1.6), and the plan of the book (§1.7). Thus, the searches themselves start in Chapter 2, while the problems used as examples throughout are described in Chapter 14.

1.1 The state-space problem

The model of this book is the *deterministic* one with *perfect information*: the agent knows the state of the world, and an action applied to a state leads to one state only, known in advance. Stochastic outcomes (Markov decision processes), partial observations, adversaries (two-player games) and the interleaving of planning with acting (real-time search) are not treated, and for them we refer to the textbook of Russell and Norvig [13]. Our definitions follow Edelkamp and Schrödl [10] (chapter 1) and Russell and Norvig [13] (chapter 3).

Definition 1.1 (State-space problem). A *state-space problem* is a tuple $P = (S, A, s_0, T, w)$, where S is a set of *states*, $s_0 \in S$ the *initial state*, $T \subseteq S$ the set of *goal states*, and A a set of *actions*. Each action $a \in A$ is a partial map $a : S \to S$ (defined on the states where a is applicable), and w assigns to the application of a to s a cost $w(s, a) \geq 0$. A *solution* is a sequence of actions $\pi = (a_1, \ldots, a_k)$ such that the states $u_0 = s_0$ and $u_i = a_i(u_{i-1})$ are defined and $u_k \in T$; its *cost* is $\sum_{i=1}^{k} w(u_{i-1}, a_i)$. One asks for a solution of least cost. When each action costs 1 the problem is said to have *unit costs*, and the cost of a solution is then its length.

In fact, the problem is a shortest-path problem: the *state-space graph* of P is the directed graph $G = (V, E)$ with $V = S$ and an edge $(u, v) \in E$ whenever $v = a(u)$ for some action a applicable to u, with the cost $c(u, v)$ of the cheapest such action. A solution is then a path from s_0 to a node of T, and its cost is the sum of the costs of its edges. Since only the cheapest action between two states matters, we state the problem on the graph from now on, and we use *state* and *node* interchangeably whenever no confusion arises (the distinction is made in Chapter 2, where a node is the record that a search keeps for a state). Several actions, or several paths, usually lead to the same state; hence, the graph is not a tree in general, and a search that does not recognize the states it has already met (a *duplicate*) may do the same work many (possibly exponentially many) times.

The graph of Figure 1.1 is small enough to be stored as a list of edges, or as an adjacency matrix of $|V|^2$ entries. However, the graphs of interest are not: the 15-puzzle has $16!/2 \approx 1.05 \cdot 10^{13}$ states reachable from any of its states (a rather "small" puzzle, see §14.2), whose adjacency matrix would have more than 10^{26} entries, and a map of a video game, a routing problem or a planning task are of the same kind. Such a graph is given *implicitly*.

Definition 1.2 (Implicit graph). An *implicit state-space graph* is given by the initial state s_0, a predicate $\text{goal} : V \to \{\text{false}, \text{true}\}$ that recognizes the goals, and an *expansion* procedure that, given a state u, enumerates the successors v of u, i.e., the states with $(u, v) \in E$, each with the cost $c(u, v)$ of its edge. A state is *generated* when it is first produced by an expansion (or when it is s_0), and *expanded* when its own successors are enumerated.

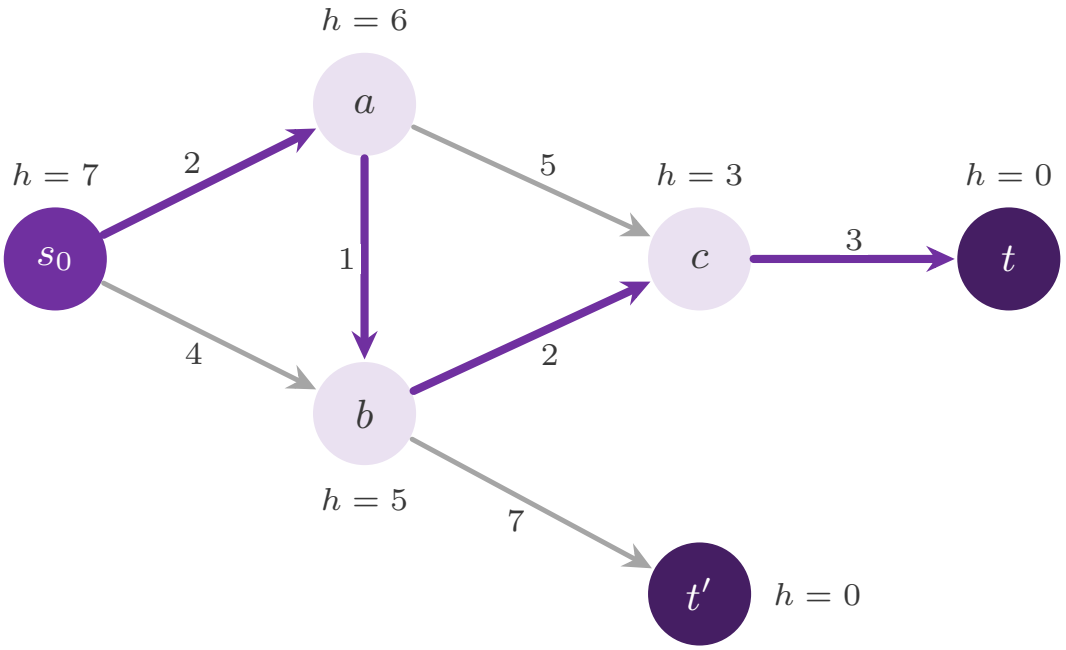


Figure 1.1. A small state-space graph: the initial state s_0, two goals t and t' (dark violet), the cost of each edge on it and the estimate h of the cost to the nearest goal above or below each state. The path of least cost, s_0, a, b, c, t (violet), costs $2 + 1 + 2 + 3 = 8$, while the goal t' is reached at best at cost $3 + 7 = 10$.

Hence, the searches of this book only see the graph through these three objects, and the part of the graph that they hold in memory is the part they have generated; the rest of V is never built (Figure 1.2). We assume throughout that the graph is *locally finite*, i.e., that each state has finitely many successors (while V may be infinite), and that the costs are nonnegative, $c(u, v) \geq 0$. Edges of cost 0 are allowed (e.g., the moves of the player of Sokoban that push no box). Negative costs are not, since with them the optimal cost may not exist and the searches that expand the states by a nondecreasing priority are not correct.

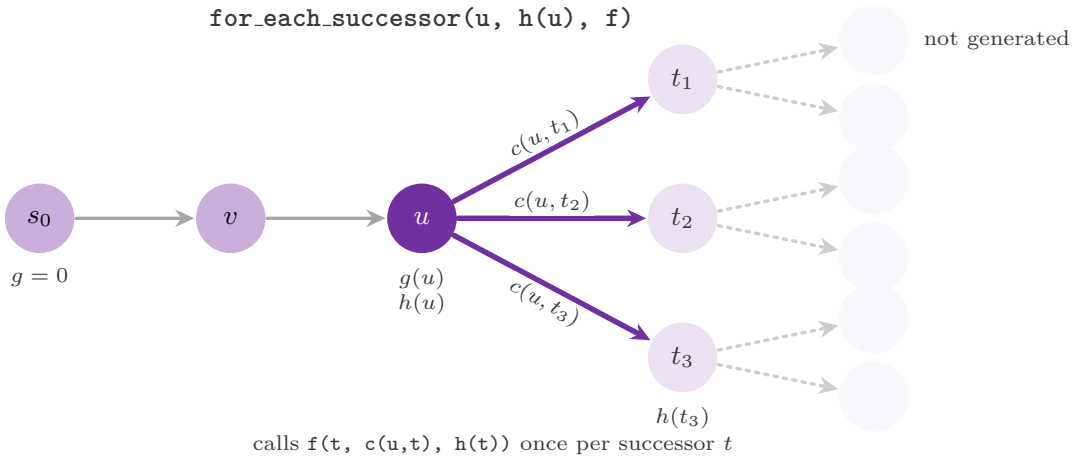


Figure 1.2. An implicit graph. The search has generated s_0, v and u; the expansion of u, i.e., the call `for_each_successor( u , h( u ) , f )` of the `Domain`, generates t_1, t_2 and t_3, each passed to `f` with the cost of its edge and its heuristic value. The successors of t_1, t_2 and t_3 (dotted) do not exist for the search until these states are expanded in turn.

The cost functions

We denote by $g^*(u)$ the least cost of a path from s_0 to u, by $h^*(u)$ the least cost of a path from u to a goal ($+\infty$ if no goal is reachable from u), and by $C^* = h^*(s_0)$ the optimal cost, which is the value sought (in Figure 1.1, $C^* = 8$). A search keeps, for each state u it has generated, the cost $g(u)$ of the best path from s_0 to u found so far; this is the cost of an actual path, and hence $g(u) \geq g^*(u)$ at any time. It also keeps the estimate $h(u)$ of

$h^*(u)$ given by the problem (the *heuristic*); their sum

$$f(u) = g(u) + h(u)$$

estimates the cost of a solution through u, and it is the priority of A* [9]. Two properties of h decide what the searches can guarantee, and we define them here, since the whole book uses them.

Definition 1.3 (Admissible and consistent heuristics). A heuristic $h : V \to [0, +\infty]$ is *admissible* if $h(u) \le h^*(u)$ for each state u, i.e., if it never overestimates the cost of reaching a goal. It is *consistent* if $h(t) = 0$ for each goal t and

$$h(u) \le c(u, v) + h(v) \qquad \text{for each edge } (u, v) \in E,$$

i.e., if h satisfies the triangle inequality along each edge.

Consistency is stated edge by edge, and the property that $f = g + h$ does not decrease along a path (sometimes called *monotonicity*) is a consequence of it, not its definition; a consistent heuristic is admissible (Lemma 2.4), while the converse does not hold. These facts, and what each property guarantees to A*, are proved in Chapter 2. In Figure 1.3, the graph of the test `reopen` of the library shows the gap between the two properties. Its h is admissible, being at most h^* on each state, but it drops by 3 along an edge of cost 1, and A* run on it closes v_2 through the expensive edge before finding the cheap path through v_1 (the run is traced in Chapter 2).

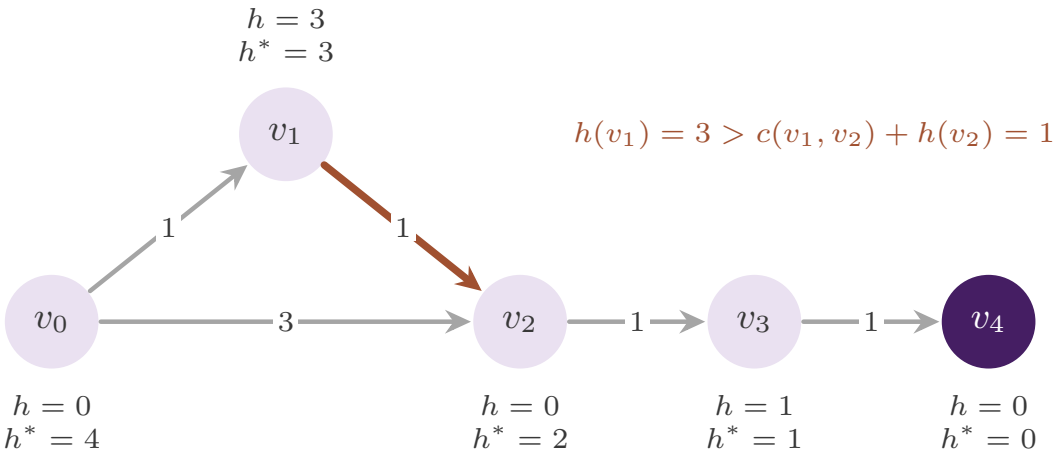


Figure 1.3. An admissible heuristic that is not consistent (the graph of the test `reopen`, whose edges are arcs): $h \le h^*$ on every state, but on the edge (v_1, v_2) the value of h falls from 3 to 0 at the cost of 1, which violates the inequality of Definition 1.3.

1.2 What makes a problem hard

The model of §1.1 is somewhat simple, but it contains problems that are NP-hard (e.g., the traveling salesperson problem of §14.7) and even PSPACE-complete ones (e.g., Sokoban and, in general, classical planning); hence, the form of a problem is (arguably) no indication of its difficulty. We now list the features of an instance that typically decide how hard it is for the searches of this book, in the order of section 2.3 of the textbook of Jinnai on heuristic search [164].

The size of the state space. The number of states reachable from s_0 bounds the work of a search that detects duplicates. It is $9!/2 = 181440$ for the 8-puzzle (which the test `astar` checks by a breadth-first search of the whole space), $16!/2 \approx 1.05 \cdot 10^{13}$ for the 15-puzzle and $25!/2 \approx 7.8 \cdot 10^{24}$ for the 24-puzzle (these counts are proved in §14.2). An infinite state space (e.g., one with a real-valued variable) is allowed by Definition 1.2, but then a search may not stop when no goal is reachable.

The branching factor. The branching factor of a state is the number of its successors. That of a problem is either their maximum $b_{\max}$, which bounds the growth of a search, or their mean over the states, which is (often) what is measured and what is estimated in the literature; the two coincide on a problem whose states all have

the same number of successors. A search that expands all the states up to depth d meets at most $b_{\max}^d$ paths at depth d (Lemma 1.4 and Figure 1.4). A "large" branching factor is not in general a source of difficulty (at least, not of the kind that the parallel searches suffer from), since a dense graph also has short paths, but it multiplies the cost of each level of the search; the exact mean branching factors of the puzzles and of the grids of the library are derived in Chapter 14.

Lemma 1.4 (Paths of a given length). If each state has at most $b_{\max}$ successors, there are at most $b_{\max}^d$ paths of d edges that start at s_0, and at most $\sum_{k=0}^{d} b_{\max}^k$ states at distance at most d edges from s_0.

Proof. By induction on d: there is one path of 0 edges, and each path of $d+1$ edges is a path of d edges followed by one of the at most $b_{\max}$ edges that leave its last state. Each state at distance $k \le d$ is the last state of at least one path of k edges, which gives the second bound. □

The bound is attained by a tree, and on a graph with duplicates the number of distinct states is usually much smaller than the number of paths; this is why the searches of this book detect duplicates.

Dead ends and irreversible actions. A problem is *reversible* if each action has an inverse of the same cost, as in the puzzles and the grids of unit costs, and from any state reached from s_0 one can then go back to s_0. In an irreversible problem (e.g., Sokoban, where a box pushed into a corner cannot be pulled back) the search may enter a region with no goal, a *dead end*; the searches that keep all the generated states (all those of this book but the depth-first ones) are favored there over local searches, which may spend their whole budget in such a region.

The existence of a solution. When no goal is reachable, a search that detects duplicates proves it on a finite graph by exhausting the states reachable from s_0 (the status `unsolvable` of a `Solution`, see §1.5). This may cost as much as the hardest solvable instance of the same size (or more, since no goal stops the search early), while on an infinite graph it does not stop at all. Note that the test `limits` checks the first behavior on a grid split by a wall. The n-queens problem with $n = 2$ or $n = 3$, or the missionaries and cannibals with four of each and a boat of two seats, are (rather) small unsolvable instances (§14.6).

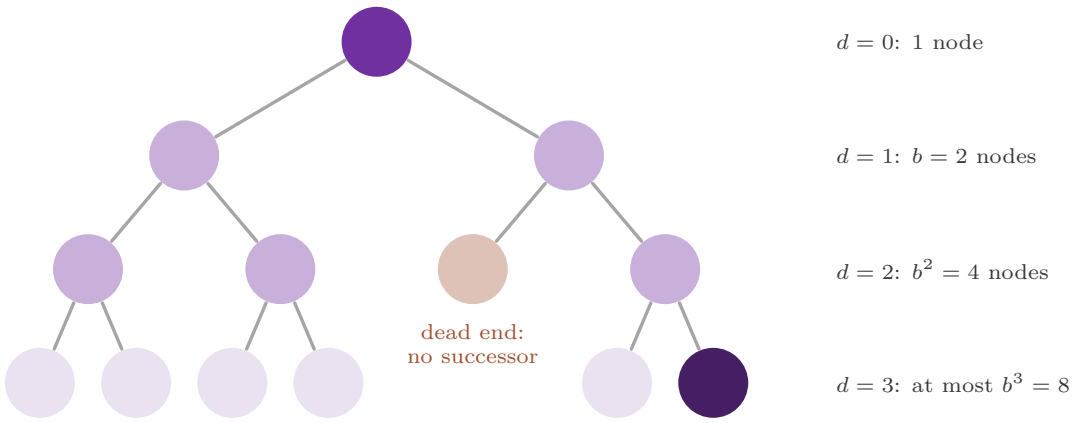


Figure 1.4. The growth of a search tree with branching factor $b = 2$: up to b^d paths at depth d. A dead end (rust) is a state with no successor or, more generally, one from which no goal is reachable.

1.3 The measures of a search

A search is judged by four properties, as in section 3.3 of [13]: it is *complete* if it returns a solution whenever one exists (and reports that none exists otherwise, when the graph is finite), *optimal* (or *admissible*) if the solution it returns has cost C^*, and its *time* and *space* are the number of states it expands and the number of states it stores, as functions of the instance. For the uninformed searches these are usually given in terms of the branching factor b (the maximum $b_{\max}$ of §1.2), the depth d of the shallowest goal, the maximum depth m of the state space and, with costs, C^* and the least edge cost ε. Table 1.1 transcribes figure 3.21 of [13], which assumes a tree; the searches themselves (breadth-first, the algorithm of Dijkstra as uniform-cost search, depth-first, iterative deepening) are studied, with the proofs of these properties, in Chapter 2 and Chapter 4, and the bidirectional search in Chapter 7.

Table 1.1. Completeness, optimality, time and space of the uninformed searches on a tree (from figure 3.21 of [13]), with $k = 1 + \lfloor C^*/\varepsilon \rfloor$: (a) complete if b is finite, (b) complete if every edge costs at least $\varepsilon > 0$, (c) optimal if the costs are all equal, (d) with breadth-first search in both directions.

search	complete	optimal	time	space
breadth-first	yes (a)	yes (c)	$O(b^d)$	$O(b^d)$
uniform-cost	yes (a, b)	yes	$O(b^k)$	$O(b^k)$
depth-first	no	no	$O(b^m)$	$O(bm)$
iterative deepening	yes (a)	yes (c)	$O(b^d)$	$O(bd)$
bidirectional	yes (a, d)	yes (c, d)	$O(b^{d/2})$	$O(b^{d/2})$

We remark that the condition (b) is the right one on an infinite graph, and that it is not enough that the graph has no cycle of nonpositive cost. In fact, let s_0 have the successors v_1, at cost $1/2$, and a goal t, at cost 2, and let each v_i have the single successor v_{i+1}, at cost $2^{-(i+1)}$: the graph has no cycle at all, but $g^*(v_i) = 1 - 2^{-i} < 1 < 2$ for every i, and a uniform-cost search, which always expands a state of least g, expands $v_1, v_2, \ldots$ for ever without reaching t. The next lemma is what (b) gives instead.

Lemma 1.5 (Finitely many cheap paths). On a locally finite graph whose edges all cost at least $\varepsilon > 0$, for every $C \geq 0$ only finitely many paths that start at s_0 cost at most C.

Proof. A path of k edges costs at least $k\varepsilon$, and hence a path of cost at most C has at most $\lfloor C/\varepsilon \rfloor$ edges. The paths of at most $\lfloor C/\varepsilon \rfloor$ edges that start at s_0 are finitely many, by the induction of Lemma 1.4 with the finite number of successors of each of the finitely many states met at each step in place of $b_{\max}$. □

Hence, a search that expands the states by nondecreasing g (or by nondecreasing f with an admissible h) expands finitely many states before it reaches a cost larger than C^*, which is the argument of the termination theorems of Chapter 2.

The counters of a run

In the library, a run is measured by the counters of `SearchStats` (`Solution.h:73`), which all the searches fill: the expansions (`expanded`, including the `reexpanded` states, i.e., the expansions of a state already expanded with a larger g), the generated successors (`generated`), those dropped as duplicates with a g not smaller than the known one (`duplicates`), the distinct states stored at the end (`stored`, which Chapter 4 redefines for IDA*, the longest path held, and for the search on the disk, the states written to its closed files) and the wall-clock time of the search (`seconds`). Thus, the ratio `generated / expanded` is the mean branching factor met by the search, usually smaller than that of the problem when the search skips the move back to the parent (Lemma 1.8). Furthermore, a parallel search with p workers (the threads or processes that run it, see §1.6) records the expansions E_i of each worker i (`expanded_by`) and its stored states (`stored_by`), and counts the states sent to another worker (`sent`) and the messages that carry them (`messages`). From these, the methods `load_balance()` and `communication_overhead()` of `SearchStats` (`src/Solution.cpp:36` and `:51`) compute the load balance as Kishimoto, Fukunaga and Botea [2] define it (on the expansions of each worker) and the communication overhead as Jinnai and Fukunaga [5] define it,

$$\mathrm{LB} = \frac{\max_i E_i}{\frac{1}{p}\sum_{i=0}^{p-1} E_i}, \qquad \mathrm{CO} = \frac{\text{sent}}{\text{generated}},$$

with the conventions that $\mathrm{LB} = 1$ when no worker has expanded anything or `expanded_by` is empty (a sequential search fills it with one entry, which also gives $\mathrm{LB} = 1$) and $\mathrm{CO} = 0$ when nothing has been generated.

Proposition 1.6 (Range of the load balance). For $p \geq 1$ workers, $1 \leq \mathrm{LB} \leq p$; $\mathrm{LB} = 1$ if and only if all the workers expand the same number of states, and $\mathrm{LB} = p$ if and only if a single worker expands all of them.

Proof. With $\Sigma = \sum_i E_i > 0$ and $M = \max_i E_i$, one has $\Sigma/p \leq M$, since the mean of nonnegative numbers is at most their maximum, with equality if and only if they are all equal; and $M \leq \Sigma$, with equality if and only if all the other E_i are 0. Dividing by Σ/p gives $1 \leq \mathrm{LB} = pM/\Sigma \leq p$, and the conventions above give $\mathrm{LB} = 1$ when $\Sigma = 0$. □

We remark that no such bound holds in general for CO: it is at most 1 when each generated state is sent at most once, which is the case of the distributions of HDA*, but the definition does not require it. Also, the third measure of [5], the search overhead, compares the expansions of the parallel search with those of the sequential one and is computed from two runs; these measures and their use are the subject of Chapter 8 and Chapter 9.

1.4 The conventions of the pseudocode

The pseudocode of the book is that of the Doxygen comments of the headers, and it follows the code of the library line by line, with the following conventions (the same for the graph searches and for branch-and-bound).

- The lines are numbered, a block is given by its indentation, and the keywords are **procedure**, **while** ... **do**, **for each** ... **do**, **if** ... **then**, **else**, **loop**, **return**, **continue** and **break**; ▷ starts a comment.
- `x` ← `e` assigns, ∞ is the infinite cost and $\emptyset$ the empty set, and the relations are `=`, $\neq$, `<`, $\leq$, $\geq$, $\in$ and $\notin$.
- OPEN is the set of the generated states not yet expanded, from which a search takes the next state to expand, and CLOSED the set of the expanded states (in the library, the table that holds the node of each generated state, see Chapter 2); s_0 is the initial state, and `g(u)`, `h(u)`, `f(u) = g(u) + h(u)` and `parent(u)` are the fields of the node of the state `u`, while `c(u, v)` is the cost of the edge (u, v). Also, `z` is the *incumbent*, i.e., the cost of the best goal found so far (∞ if none), and `p` is the number of workers, indexed by `i` from 0 to $p - 1$.
- `lock L` and `unlock L` take and release the lock `L`, and `send m to j` puts the message `m` in the outbox of the worker `j`, which may deliver it later, in a batch.
- The node limits, the counters of the statistics and the prefetches are left out, since they change neither the outcome nor the order of the search.

Table 1.2 collects the notation of the book (cf. Table 1.1 of [164]), and each symbol has one meaning throughout, but for the letter z, which a few chapters also use for a key of Zobrist hashing (with a subscript or an argument, $z(s)$).

Table 1.2. The notation of the book.

symbol	meaning	in the library
$G = (V, E)$	the state-space graph	a `Domain`
s_0	the initial state	`initial_state()`
T	the goal states	`is_goal( s )`
$c(u, v)$	the cost of the edge (u, v)	`c` in `f( t , c , h_t )`
u, v, t	states (nodes)	`state_type`
$g(u)$	cost of the best path found to u	the field `g`
$g^*(u)$	least cost of a path to u	
$h(u)$	heuristic estimate	`heuristic( s )`
$h^*(u)$	least cost from u to a goal	
$f(u)$	$g(u) + h(u)$, the priority of A*	the key of OPEN
C^*	the optimal cost, $h^*(s_0)$	`Solution::cost`
z	the incumbent	the parallel searches
b, d	branching factor, depth of a goal	
p, i	number of workers, a worker	`threads`
OPEN	the generated, unexpanded states	`OpenList.h`
CLOSED	the expanded states	`NodeTable.h`
π	a solution, as a path	`Solution::path`

As an example, Algorithm 1.1 is the procedure by which all the searches of the library enumerate the successors of a state, i.e., the function template `for_each_child` (`Domain.h:705`). Its argument `o` is the *operator* by which `u` was generated (a small integer that names the move, e.g., the cell left by the blank of a puzzle), and `no_op` (the value 255, `Domain.h:340`) is the operator of s_0, which no move generated; when the problem knows its operators (the refinement `OperatorDomain` of §1.5), the procedure passes to each successor its own operator and skips the move back to the parent.

Algorithm 1.1 (The successors of a state).

```
1  procedure CHILDREN(u, h(u), o, f)   ▷ o: the operator that generated u
2     if the Domain is an OperatorDomain then
3        for each successor v of u not reached by the inverse of o do
4           f(v, c(u, v), h(v), the operator of v)
5     else
6        for each successor v of u do
7           f(v, c(u, v), h(v), no_op)
```

The test of line 2 is resolved at compile time (`if constexpr`), and hence it costs nothing at run time; the inverse of `no_op` leads nowhere, so that at s_0 line 3 enumerates all the successors. The skip of line 3 is justified by Lemma 1.8 below.

1.5 The concepts of the library

The headers are in three layers, each with a header that includes it whole: `core.h` (the helpers and the parameters of the parallel searches, in `include/PBeFS/core`), `include/PBeFS/astar.h` (the graph searches and their domains, in `include/PBeFS/astar`) and `bnb.h` (the tree searches, in `include/PBeFS/bnb`), while `PBeFS.h` includes the three of them, save the searches over MPI.

All the searches of the library are class or function templates over the problem, which is a type satisfying the C++20 concept `Domain` of `Domain.h:169` or one of its refinements; since all the calls are resolved at compile time, the inner loop of a search pays no virtual call, which is one of the choices of a fast implementation recommended by Burns et al. [4]. A refinement adds what a particular search, or a particular work distribution, needs; hence, a problem implements only the methods required by the searches it is meant for, and a search that needs more than the problem gives is rejected at compile time. Both the concepts and the searches that use them are drawn in Figure 1.5.

The concept Domain

A type `D` is a `Domain` (`Domain.h:169`) if it has

- a regular `state_type`, i.e., one that is copyable and compared with `==`, and an arithmetic `cost_type`, the type of g, h and the edge costs (the aliases `state_t< D >` and `cost_t< D >` name them);
- the method `initial_state()`, which gives s_0, the goal test `is_goal( s )`, and `heuristic( s )`, the value $h(s)$;
- the method `hash( s )`, a hash of 64 bits of the state;
- the expansion `for_each_successor( s , h_s , f )` of Definition 1.2, which calls `f( t , c , h_t )` for each successor t of s, with $c = c(s,t)$ and $h_t = h(t)$.

The value $h_s = h(s)$ is passed in so that one can compute $h(t)$ incrementally from it when the heuristic allows it (e.g., the Manhattan distance of the sliding-tile puzzle, which changes by 1 at each move, see Proposition 14.5). The successors are given to a callable instead of being returned in a container, and therefore no allocation is paid per expansion.

The hash is used by CLOSED and, mixed again with a seed, by the work distribution of HDA* that hashes the whole state (Chapter 9); with the default data structures, CLOSED takes the slot of a state from the high bits of the product of its hash by $\lfloor 2^{64}/\phi \rfloor$, $\phi = (1+\sqrt{5})/2$ being the golden ratio, i.e., by Fibonacci hashing (Chapter 3). The hash must therefore be "well spread", i.e., close to uniform in all its bits on the states met by a search, and no worse than that in its high bits. All the domains of the library obtain it by passing an integer image of the state to the mixing function of SplitMix64 (the addition of the constant $\lfloor 2^{64}/\phi \rfloor$ followed by the finalizer, two rounds of an exclusive or with a right shift and a multiplication by an odd constant, and a last exclusive or with a shift). That this function spreads the bits well is an empirical property, measured by its authors and not provable in this sense; what can be proved is that it loses nothing.

Lemma 1.7 (The mixer is a bijection). The mixing function of SplitMix64 is a bijection of the words of 64 bits. Hence, on a domain whose hash is the mixer applied to the state itself, a word of 64 bits (all the domains of Chapter 14 but `NQueens` and the puzzles of more than 16 cells), two distinct states have distinct hashes.

Proof. The function is a composition of maps of the words of 64 bits into themselves, each of which has an inverse: the addition of a constant modulo 2^{64} (inverted by its subtraction), the multiplication by an odd constant modulo 2^{64} (inverted by the multiplication by its inverse modulo 2^{64}, which exists since the constant is coprime with 2), and $x \mapsto x \oplus (x \gg k)$ with $1 \le k < 64$. For the latter, the k highest bits of $y = x \oplus (x \gg k)$ are those of x, and each lower bit $x_j = y_j \oplus x_{j+k}$ follows from a higher bit of x already known, from the top down. □

Of course, distinct hashes may still fall in the same slot of CLOSED, whose size is much smaller than 2^{64}; the lemma only states that the state never loses information on its way to the hash. Nothing else is required; in particular, the concept does not require the heuristic to be admissible or consistent, which no compiler can check: the searches state in their documentation which property they need for which guarantee, and the domains of the library state which property their heuristic has (Chapter 14 proves each one).

Declaring a domain and running a search

The example below is compiled and run by the tester of the library (in the case `manual`, `ctest -R PBeFS.manual`). It declares a domain whose states are the integers from 0 to n, with the moves $s \to s+1$ at cost 1 and $s \to s+3$ at cost 2 and the goal n, and it checks by a `static_assert` that the type satisfies the concept. Then it calls the function `astar( domain , open , options )` of `AStar.h`, i.e., the sequential A* of Chapter 2, with the default OPEN list (the bucket queue for integer costs and the heap otherwise, Chapter 3) and the default options, and reads the `Solution` it returns, whose fields are described below.

```
struct Line {

 using state_type = int;
 using cost_type = int;

 int n;                                          // the goal

 int initial_state( void ) const { return( 0 ); }
 bool is_goal( int s ) const { return( s == n ); }
 int heuristic( int s ) const { return( ( n - s ) / 3 * 2 + ( n - s ) % 3 ); }
 std::uint64_t hash( int s ) const {
  return( std::uint64_t( s ) * 0x9E3779B97F4A7C15ULL );
  }

 template< class F >
 void for_each_successor( int s , int , F && f ) const {
  if( s + 1 <= n )
   f( s + 1 , 1 , heuristic( s + 1 ) );
  if( s + 3 <= n )
   f( s + 3 , 2 , heuristic( s + 3 ) );
  }

 };

static_assert( Domain< Line > );
```

```
const auto sol = astar( Line{ 10 } );    // the bucket queue: int costs
// sol.status, sol.cost, sol.path from 0 to 10, and the counters sol.stats
```

The heuristic of the example is the least cost of covering the distance $n - s$ with steps of 3 at cost 2 and of 1 at cost 1, i.e., $2\lfloor (n-s)/3 \rfloor + (n-s) \bmod 3$. This is h^* itself, since a step of 3 is cheaper per unit and no move may pass the goal, and it is therefore consistent (h^* satisfies the inequality of Definition 1.3 by the definition of a least cost). The tester checks that the cost is 7 (three steps of 3 and one of 1), that the path goes from 0 to 10, and that A* with $h = 0$ gives the same cost.

The refinements

Each refinement of `Domain` adds a few methods, which serve the parallel searches of the later chapters; we name each search here with one clause, and the chapter cited gives it in full. Hash Distributed A* (HDA*, Chapter 9) gives each state an *owner*, a worker chosen by a function of the state (its *work distribution*), and sends each generated state to its owner; the distributions are written HDA*[Z] for Zobrist hashing, HDA*[AZ] for Abstract Zobrist hashing, HDA*[P] for the perfect hash modulo p, and AHDA* for the abstraction of the state. The Safe Parallel Best-NBlock-First search (Chapter 11) partitions the states into *nblocks*, and PA*SE (Chapter 8) expands in parallel the states that a pairwise heuristic proves independent. In the order of Figure 1.5, the refinements are the following.

- `FeatureDomain` (`Domain.h:193`) describes a state as a set of propositions taken from a universe of `num_features()` features, and `for_each_feature( s , f )` calls `f( i )` for each feature i that holds in s; the Zobrist hashing of HDA*[Z] needs no more than this [2].
- `IncrementalFeatureDomain` (`Domain.h:209`) also gives, by `for_each_feature_change( s , t , f )`, the features that hold in exactly one of s and its successor t; the Zobrist key of t is then computed from that of s in as many lookups as the features that the move changes (Proposition 9.15).

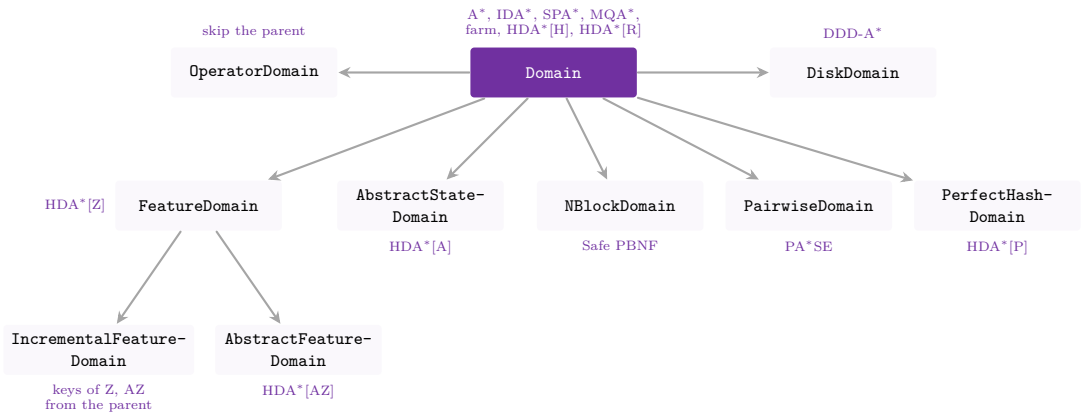


Figure 1.5. The concept `Domain`, its refinements (an arrow goes from a concept to one that requires it) and, in violet, the searches or the work distributions of HDA* that need each one. `DiskDomain` (in `DDDAStar.h`) is a `Domain` with integer costs and trivially copyable states. The refinements read by the sequential searches are listed in Table 1.3.

- `AbstractFeatureDomain` (`Domain.h:221`) adds a projection `abstract_feature( i )` of the features onto `num_abstract_features()` abstract ones, the projection of the Abstract Zobrist hashing of HDA*[AZ] [5].
- `AbstractStateDomain` (`Domain.h:267`) maps a state to an abstract state, coded as an integer by `abstract_state( s )`, which AHDA* uses to keep the successors of a state on the same worker [3].
- `NBlockDomain` (`Domain.h:281`) partitions the state space by an abstraction into `num_nblocks()` nblocks, `nblock( s )` being the nblock of s, and enumerates by `for_each_nblock_successor( b , f )` the abstract successors of the nblock b, i.e., its duplicate detection scope in the Parallel Best-NBlock-First search [3]; the two conditions that the abstraction must satisfy are those of Definition 11.1.
- `PairwiseDomain` (`Domain.h:295`) gives a heuristic `pairwise_heuristic( s , t )` between any two states, which must be forward-backward consistent, as PA*SE [6] requires.
- `OperatorDomain` (`Domain.h:356`) knows the operator by which a state was generated: its second `for_each_successor( s , h_s , op , f )` skips the successor reached by the inverse of `op`, and passes to `f` the operator of each successor (Lemma 1.8).
- `PerfectHashDomain` (`Domain.h:408`) gives an injective map `perfect_hash( s )` of the states into the integers, which the work distribution HDA*[P] of Burns et al. [3] reduces modulo p (see Chapter 3 for perfect hashing and Chapter 9 for its defects as a distribution).

In the searches, the operator is stored in each node, and in each message of HDA*, through the type `OperatorOf< D >` (`Domain.h:682`), which is an empty type when `D` is not an `OperatorDomain`; with `[[no_unique_address]]` on it, a node or a message of a problem without operators does not grow by a single byte.

The incremental Zobrist key of `IncrementalFeatureDomain` rests on an identity of the exclusive or, which Proposition 9.15 states and proves. A Zobrist key assigns to each feature i a random word R_i of 64 bits, and to a state s the exclusive or $z(s) = \bigoplus_{i \in F(s)} R_i$ of the words of the set $F(s)$ of its features [2]; then $z(t) = z(s) \oplus \bigoplus_{i \in \Delta} R_i$, where Δ is the set of the features that hold in exactly one of s and t, since the words of the features common to both appear twice and cancel. On the sliding-tile puzzle a move changes two features (a tile leaves a cell and enters another, Figure 1.6), and therefore the key of a successor costs two lookups instead of one per tile; the test `hda-thread` checks, on random walks of the three puzzles, that the keys and the owners computed from the parent coincide with those computed from scratch.

The second refinement with a statement on which the searches rely is `OperatorDomain`, whose skip of the parent is one of the optimizations of Burns et al. [4]. We prove it once for all the searches of the library, sequential and parallel; Proposition 2.16 is its instance for A*.

Lemma 1.8 (Skipping the parent). Let an operator generate u from v, and let its inverse lead from u back to v at the same cost, $c(u, v) = c(v, u)$. In a search where the cost g of a stored state never increases, and where v stays stored once it has been expanded, assume that the current $g(u)$, and the operator stored with it, were set when u was generated from v, i.e., that $g(u) = g(v) + c(v, u)$ for the value $g(v)$ of that moment. Then the generation of v from u would give v a cost not smaller than its current one, i.e., it would be a duplicate that the search drops.

Proof. At the expansion of u, the current cost $g'(v)$ of v is not larger than the value $g(v)$ it had when u was generated, since g never increases; hence, $g(u) + c(u, v) = g(v) + 2\,c(v, u) \geq g(v) \geq g'(v)$, the costs being

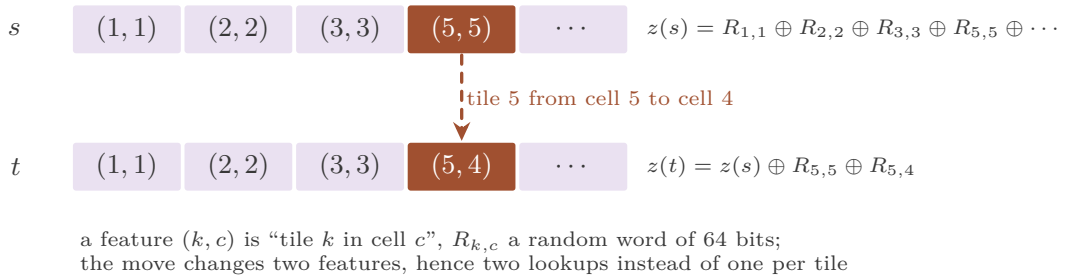


Figure 1.6. The incremental Zobrist key on the 8-puzzle. A feature (k, c) is "tile k is in cell c"; the move of tile 5 from cell 5 to cell 4 changes the two features in rust, and the key of t follows from that of s by two exclusive ors (Proposition 9.15).

nonnegative. A generated state whose cost is not smaller than the known one is a duplicate: the search only counts it, and changes neither OPEN nor CLOSED. Hence, not generating it changes neither the outcome nor the order of the expansions. □

The searches of the library that keep CLOSED never increase a g (a node is reopened only with a smaller one) and never drop an expanded state, and therefore the lemma applies to all of them, including the parallel ones, where the cost of v known at its owner is at most the cost with which v generated u. The hypothesis on the cost of the inverse is not checked by the concept, and Exercise 1.3 shows what is lost without it. Among the domains of the library, the sliding-tile puzzles have operators, whose operator is the cell left by the blank (Figure 1.7), and so do the grids, the pancakes, Rubik's cube, Top Spin, the Towers of Hanoi, the permutation puzzles and the planning tasks (Chapter 14); IDA*, which has no CLOSED, prunes the parent of the other domains by comparing a successor with the state two places back on its path (Chapter 4).

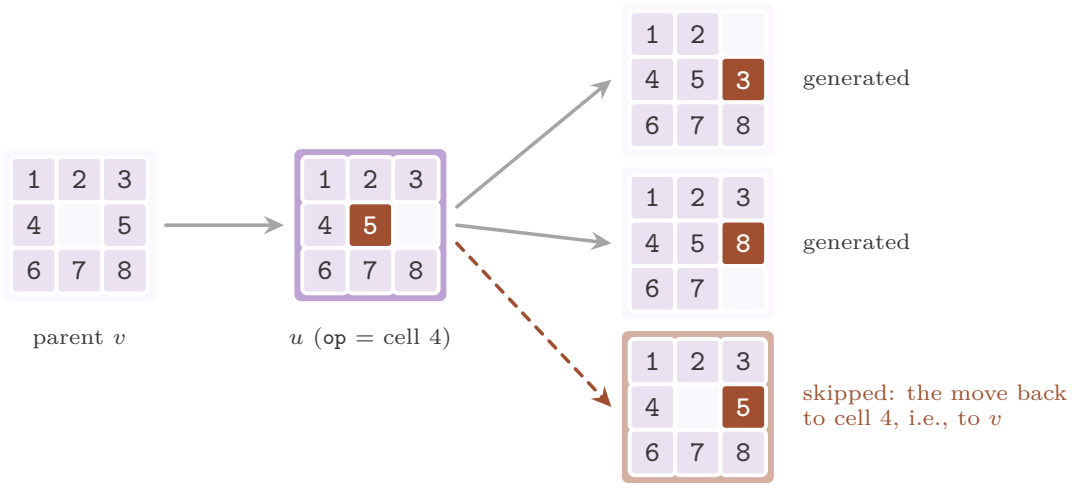


Figure 1.7. Skipping the parent on the 8-puzzle. The state u was generated from v by moving the blank from cell 4 to cell 5, and its operator is therefore 4; of the three moves of the blank of u, the one back to cell 4 would regenerate v at a larger cost, and `for_each_successor( u , h , 4 , f )` does not generate it (Lemma 1.8).

The refinements of the sequential searches

Figure 1.5 draws the refinements by which the parallel searches distribute their work. The sequential searches read other refinements, which Table 1.3 lists with the searches that read them. As above, a refinement only adds methods to `Domain` (or to `FeatureDomain` or `OperatorDomain`, which it then requires); a search that does not read it compiles and runs unchanged on a problem that offers it.

Table 1.3. The refinements of `Domain` read by the sequential searches (`Domain.h`), the methods they add, and the searches that read them, with the section or the chapter that describes each one.

concept	adds	read by
`TreeOperatorDomain`	`for_each_tree_successor( s , h , op , f )`	IDA* (§4.4)
`PartialExpansionDomain`	`for_each_successor_delta( s , h , op , δ , f )`	EPEA* (§4.9)
`MultiHeuristicDomain`	`num_heuristics()`, `heuristic( s , i )`	the greedy searches (§2.12)
`PreferredDomain`	`preferences( s )`	the greedy searches, EHC (§2.12, §2.13)
`DeferredDomain`	`for_each_successor_state( s , f )`	lazy GBFS (§2.12)
`GoalFeatureDomain`	`for_each_goal_feature( f )`	SIW, BFWS (§2.15)
`RelevantFeatureDomain`	`for_each_relevant_feature( s , f )`	BFWS (§2.15)
`DistanceDomain`	`distance( s )`	the focal searches (Chapter 6)

We describe them in the order of the table.

- `TreeOperatorDomain` is an `OperatorDomain` whose `for_each_tree_successor( s , h , op , f )` gives the successors of `for_each_successor( s , h , op , f )` but some more. It skips those of the pairs of moves (`op`, then the move to the successor) that the domain declares redundant, because another pair that it keeps reaches the same state at a cost not larger; two turns of the same face of Rubik's cube, and two moves of the same disk of the Towers of Hanoi, are the examples of Korf [59]. The function template `for_each_tree_child` calls it, and falls back on `for_each_child` (Algorithm 1.1) for the other domains. The contract of the concept, which the compiler cannot check, is that every path is matched by one with the same ends, a cost not larger, and only moves that are kept. On the domains of the library a pruned pair is of one of two kinds. Either its two moves compose into one move or none (two turns of a face, two moves of a disk), and replacing them shortens the path; or they commute and come in the order that the domain does not keep (two turns of opposite faces of the cube, two commuting reversals of Top Spin), and swapping them keeps the length and lowers the number of pairs out of order. Induction on the length, and then on that number, proves the contract for them. Hence a search of the tree of the paths, which detects no duplicates, stays complete and optimal, and IDA* reads the concept (§4.4). The searches with CLOSED do not, since the kept pair may go through a state closed with another operator, whose successors would then be pruned for good. Rubik's cube, the Towers of Hanoi, Top Spin and the permutation puzzles of the pattern databases implement it (Chapter 14).
- `PartialExpansionDomain` is an `OperatorDomain` whose `for_each_successor_delta( s , h_s , op , δ , f )` calls `f( t , c , h_t , op_t )` only for the successors t of s, but the parent, whose $\Delta f = c + h_t - h_s$ equals δ. It returns the least Δf larger than δ among the others, or `infinite_cost()` if there is none. This is the operator selection function of Enhanced Partial Expansion A*, which never generates a successor that it does not put in OPEN (§4.9); with real costs, Δf equals δ up to the slack of `cost_slack` (see below). The sliding-tile puzzles and the grids implement it.
- `MultiHeuristicDomain` gives $m = $ `num_heuristics()` ≥ 1 heuristics, `heuristic( s , i )` being $h_i(s)$ and h_0 the `heuristic( s )` of `Domain`. For any domain, `heuristics_of( d )` and `heuristic_of( d , s , i )` give m and $h_i(s)$, with $m = 1$ and $h_0 = h$ on a domain that is not a `MultiHeuristicDomain`, so that the searches need not tell the two cases apart. The greedy searches alternate their queues over the heuristics, break the ties of one by another, or keep the Pareto front of their vectors [98] (§2.12). On the sliding-tile puzzles h_0 is the Manhattan distance and h_1 the number of misplaced tiles.
- `PreferredDomain` gives, by `preferences( s )`, a predicate p computed once per expansion of s, where $p(t)$ is true if the successor t of s is *preferred*: e.g., the helpful actions of the planner FF [101], or the preferred operators of a heuristic [97]. The greedy searches put the preferred successors in queues of their own, while enforced hill-climbing may expand them alone or first (§2.13). On the sliding-tile puzzles a move is preferred if the tile that it moves gets closer to its goal cell, i.e., if it lowers the Manhattan distance, and on the grids if it lowers the distance to the goal.
- `DeferredDomain` gives, by `for_each_successor_state( s , f )`, the successors t of s with the costs of their edges, as `f( t , c )`, without computing $h(t)$. GBFS with deferred (lazy) evaluation then computes h only for the states that it selects [97], which pays when the heuristic costs far more than a move, as in planning. No domain of the library implements it, their heuristics costing no more than a move, and the test `gbfs` wraps the 15-puzzle in one to count the evaluations.

- `GoalFeatureDomain` is a `FeatureDomain` whose goal is a conjunction of features, enumerated by `for_each_goal_feature( f )`: a state is a goal if and only if it has all of them. The serialized width search achieves them one at a time, and the best-first width search orders the states by the number $\#g$ of those not yet achieved [104], [105] (§2.15). On the sliding-tile puzzles the goal features are the tiles in their goal cells, and on the grids the column and the row of the goal.
- `RelevantFeatureDomain` is a `FeatureDomain` whose `for_each_relevant_feature( s , f )` enumerates the features of a relaxed plan from s. In the best-first width search, $\#r$ counts the features of this set that are reached on the path to a state [105]. No domain of the library implements it, and the search then takes $\#r = 0$.
- `DistanceDomain` gives, by `distance( s )`, an estimate $d(s)$ of the number of edges of a cheapest path from s to a goal, which need not be admissible; it is h when every edge costs 1, as on the sliding-tile puzzles. By it, the focal searches of Chapter 6 prefer the nodes closer to a goal in edges rather than in cost [52]. On the grids, whose moves cost differently, d is the number of moves to the goal on a grid without obstacles, i.e., the Manhattan distance with 4 moves and the larger of the two differences of the coordinates with 8.

Each of the other refinements of `Domain.h` belongs to searches with a chapter of their own, which describes it. `BidirectionalDomain`, `BackwardOperatorDomain` and `MinCostDomain` (whose `min_edge_cost()` is the ε of the bounds of the bidirectional searches) are in Chapter 7. `PredecessorDomain` and `EventDomain`, with the events `EdgeCostChanged` and `GoalChanged` of the incremental searches, are in Chapter 6, and so is `BatchHeuristicDomain`. `EdgeDomain`, `LazyEdgeDomain` and `HeterogeneousEdgeDomain`, whose `expensive_edge( s , a )` lets GePA*SE evaluate the cheap edges of a state together and each expensive one apart [122], are in Chapter 8. Last, `FeatureGroupDomain` and `FeatureTransitionDomain` are in Chapter 10, and `LevelDomain` in Chapter 15.

Real costs

A `cost_type` may be a floating-point type, as on the grids whose diagonal moves cost $\sqrt{2}$ times the cost of a cell. Two sums of the same edge costs taken in another order may then differ in their last bits, and a search that compared for equality the f of two nodes reached along such paths would take them for different values. Hence the searches that compare costs for equality, or test an inequality that a rounding could reverse, allow the slack `cost_slack( x , τ )` of `Node.h`. It is 0 for an integer `cost_type` and $\tau \max\{1, |x|\}$ for a real one, τ being `real_cost_tolerance` $= 10^{-9}$ by default, and two values x and y are taken as equal when $|x - y| \leq$ `cost_slack( x )`. The slack is relative, since the rounding error of a sum grows with its value, and it is at least τ, since a cost near 0 has no relative error to speak of. PEA* and EPEA* use it, through the option `tolerance` of their `PartialOptions`, and so do the selection function of the grids (§4.9) and the test of independence of PA*SE (Chapter 8). Also, the largest value of a `cost_type`, `infinite_cost()` (the infinity of a real type, the largest integer otherwise), stands for "no path" and, as a heuristic value, for a dead end.

We now show that the tolerance covers the paths of any practical length for the type `double`, whose unit roundoff is $u = 2^{-53} \approx 1.1 \cdot 10^{-16}$. Let $x_1, \ldots, x_n \geq 0$ be the costs of the edges of a path, and $S = \sum_i x_i$. Adding them from left to right gives $\hat{S} = \sum_i x_i \prod_j (1 + \delta_j)$, with $|\delta_j| \leq u$, where the product is over the at most $n - 1$ additions that follow the term x_i. Each product lies in $[1 - \gamma_{n-1}, 1 + \gamma_{n-1}]$, with $\gamma_k = ku/(1 - ku)$ for $ku < 1$ (Higham [170], chapters 3 and 4), and the terms are nonnegative; hence $|\hat{S} - S| \leq \gamma_{n-1} S$. Two orders of the same terms thus give sums that differ by at most $2\gamma_{n-1}S$, which is less than $\tau \max\{1, S\}$ as long as $2\gamma_{n-1} < \tau$, i.e., for paths of fewer than $4 \cdot 10^6$ edges with $\tau = 10^{-9}$. (The code compares with the computed sum rather than with S, which changes the bound by a factor $1 + \gamma_{n-1}$ only.) Conversely, two costs that differ by less than the slack are taken as equal. We do not claim that this is harmless; a domain whose distinct path costs may be that close should use integer costs, or a smaller tolerance where the search takes one.

What a search returns

Every graph search returns a `Solution< State , Cost >` (`Solution.h:153`), whose fields are those of Table 1.4; the method `solved()` is true if and only if the status is `solved`. The values of the enumeration `Status` (`Solution.h:46`) are `solved` (a path was found, optimal under the conditions on h stated by the search), `unsolvable` (OPEN emptied without reaching a goal), `node_limit` (the limit of `SearchOptions` was reached first, see below) and `disk_limit` (the limit on the bytes written to the disk was reached first, which only the external searches of Chapter 4 have). The field `bound` is a lower bound on C^* that the search has proven, i.e., the cost itself when the path is optimal and less than it for the searches that only bound the suboptimality of their path, and `improvements` lists the paths that an anytime search found on its way, with their costs, their bounds, the expansions and the time at which each was found; both are described with those searches in Chapter 6.

Table 1.4. The fields of `Solution` (`Solution.h:153`) and their defaults; `path.front()` is s_0 and `path.back()` the goal, and the path is empty unless the status is `solved`.

field	type	meaning	default
`status`	`Status`	how the search ended	`unsolvable`
`cost`	`Cost`	the cost of the path	`Cost{}` (0)
`bound`	`Cost`	a proven lower bound on C^*	`Cost{}` (0)
`path`	`vector< State >`	s_0 to the goal, if solved	empty
`improvements`	`vector< Improvement >`	the paths of an anytime search	empty
`stats`	`SearchStats`	the counters of §1.3	all 0

Next, the options shared by all the searches are those of `SearchOptions` (`Solution.h:188`), in Table 1.5. The first is a limit on the stored states (per worker, in a parallel search), after which the search stops with the status `node_limit`; IDA*, which stores only a path, applies it to the expansions instead, and the search on the disk to the states written to its closed files (Chapter 4). The second is the number of states the search is expected to store, by which a sequential search sizes CLOSED at the start (otherwise CLOSED is doubled, and rehashed, as it fills); `AStar` reads it, and so do most of the other sequential searches (the greedy and width-based searches of Chapter 2, and those of Chapter 4 that keep a table of the states), while the parallel searches size their tables otherwise. The next four options of the table are read by the searches that they concern. The weight w of the heuristic orders OPEN by $g + wh$, in the searches of Chapter 6 and in the greedy searches of §2.12, and the dropping of the duplicates and the anytime search apply with a weight above 1 (Chapter 6). The goal test at the generation of a state is made by A* with the guarantees of §2.11, and by the greedy searches as in §2.12.

A search returns, as `cost`, the cost of the path that it found, which is the g of the goal when the parents of the nodes of the path have not changed since the goal was generated; when they have (a node of the path reached later by a cheaper path, with its parent changed), the path that the parents give is cheaper than that g. The function template `path_cost( d , path )` of `Domain.h` gives the cost of a path, each step taken by the cheapest edge between its two states (0 for an empty path), and `AStar` returns the least of the two values; the function `check_path` of `test/test.h` verifies, for each search that the tester runs, that the path goes from s_0 to a goal along edges of the domain and that its cost, so computed, is the one returned.

Table 1.5. The fields of `SearchOptions` (`Solution.h:188`) and of `ParallelParams` (`SearchUtils.h:98`), with their defaults; 0 means "no limit" or "unknown" in the first two; the node limit holds per worker in a parallel search, and `expected_nodes` is read by the sequential searches that keep a table of the states.

field	type	meaning	default
`node_limit`	`size_t`	stop after this many states	0
`expected_nodes`	`size_t`	the states CLOSED is sized for	0
`weight`	`double`	the weight w of h in $g + wh$	1
`drop_duplicates`	`bool`	reopen only beyond what w allows	`false`
`anytime`	`bool`	go on after the first goal	`false`
`goal_at_generation`	`bool`	test the goal when generated	`false`
`threads`	`unsigned`	the number p of workers	1
`batch`	`size_t`	the states in a message	64
`flush_period`	`unsigned`	steps between two flushes	64
`seed`	`uint64_t`	the seed of the random draws	1

1.6 Workers, executors and options of the parallel searches

A parallel search of the library runs p *workers*, indexed by $i = 0, \ldots, p-1$; in shared memory each worker is a thread, and with MPI each one is a process. In shared memory, the searches are templates over the *executor* that runs their workers, i.e., a type that offers `run( p , body )`, which calls `body( i )` for $i = 0, \ldots, p-1$ at once, each call on a separate thread, and returns when all of them have returned. Three executors are provided: a pool of C++ threads (`ThreadPool`, the default), an executor on OpenMP (`OpenMPExecutor`) and one on FastFlow [12] (`FastFlowExecutor`); OpenMP and FastFlow are optional dependencies, and a program can use an executor only if the library was built with its runtime. Since the executor is a template parameter, the same search can be compared across runtimes, and the same runtime across searches, which the benchmarks of Chapter 8 and Chapter 9 do.

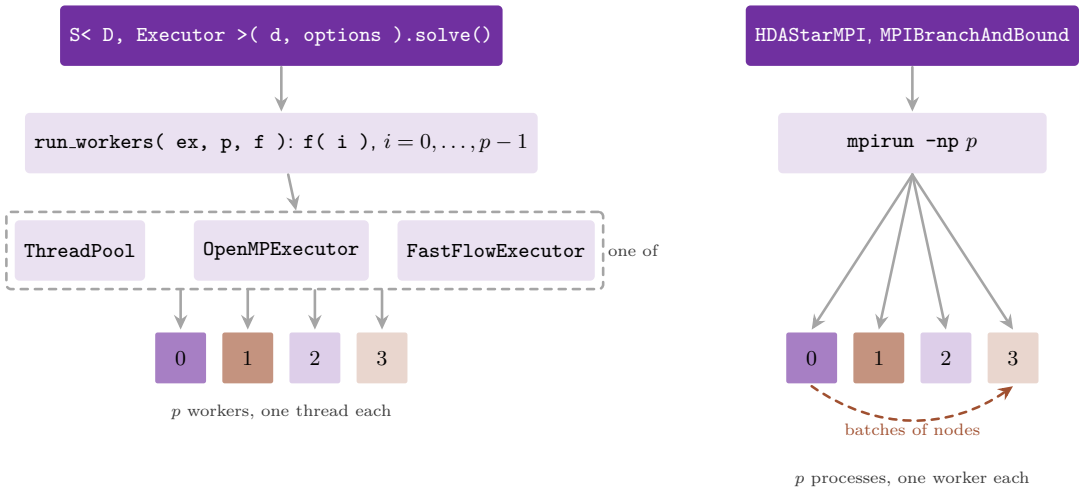


Figure 1.8. How the workers are run. A shared-memory search calls `run_workers`, which runs the function of a worker for $i = 0, \ldots, p-1$ on the chosen executor, one thread per worker; a search over MPI is started by `mpirun` as p processes, one worker each, which exchange the states in batches of messages.

Every search starts its workers through the function `run_workers( ex , p , f )` (`SearchUtils.h:125`), which is Algorithm 1.2 (Figure 1.8).

Algorithm 1.2 (Running the workers).

```
1  procedure RUNWORKERS(ex, p, f)
2     body ← the procedure that, given i,
3        pins its thread to the i-th CPU, unless it runs on the caller
4        calls f(i)
5     if ex ≠ ∅ then
6        ex.run(p, body)          ▷ the executor given by the program
7     else
8        E ← a new executor of p threads
9        E.run(p, body)           ▷ E is destroyed on return
```

Line 3 is the function `pin_worker( i )` (`SearchUtils.h:79`): if the environment variable `PBEFS_PIN` is set to a nonempty value other than `0`, it binds the thread to the i-th CPU (modulo their number) of the affinity mask of the process, where the system allows it, and otherwise it does nothing (on a machine with several sockets, the binding is usually needed for the timings to be repeatable). One of the calls may run on the thread of the caller, which is left where it is. Since `run( p , body )` returns when all the calls have returned, RUNWORKERS returns if and only if each $f(i)$ does, and the termination of a parallel search is that of its workers, proved search by search in the later chapters. A program that runs many searches can give each of them a common executor by `set_executor( ex )`, which saves the start of the threads at each search (the benchmarks of the library do not use it, and their times include that start).

The parameters of the parallel searches are those of `ParallelParams` (`SearchUtils.h:98`, Table 1.5). Besides the number of workers, the searches that send their states to an owner read the number of states packed in a message and the number of steps (expansions or evaluations) between two flushes of an outbox that is not full. The seed is that of the random draws of the searches that make some (the queues of the MultiQueue A* and of

several branch-and-bound schemes), while a work distribution of HDA* takes its own seed in its constructor. In turn, the options of the parallel graph searches are `ParallelOptions` (`HDAStar.h:125`), which derives from both `SearchOptions` and `ParallelParams` and adds nothing, so that a node limit is per worker (the expected number of states is not used by the parallel searches); the searches with further parameters derive their options from it (`PBNFOptions` in `SafePBNF.h`, `PASEOptions` in `PASE.h`). The defaults of `ParallelParams`, 64 states per message flushed every 64 steps, are those of HDA*, and each search sets different ones where needed. Also, its method `normalize()`, which each search calls at its construction, turns `threads`, `batch` or `flush_period` left at 0 into 1. Finally, the searches over MPI (`HDAStarMPI` and the branch-and-bound of `MPIBranchAndBound.h`) take the number of workers from the number of processes, one worker per process; their termination detection, which must count the messages in flight, is described in Chapter 9.

Two more helpers of `SearchUtils.h` serve the reproducibility of the experiments. First, `random_unit` (`SearchUtils.h:157`) and `random_below` (`SearchUtils.h:166`) draw a real number in $[0, 1)$ and an integer in $[0, n)$ directly from the words of `std::mt19937_64`, whose sequence the C++ standard fixes, rather than through the distributions of `<random>`, which differ between standard libraries; hence, a random grid, a set of random cities or a random knapsack generated from a seed is the same on all platforms. If x is the word drawn, the first returns $\lfloor x/2^{11} \rfloor \cdot 2^{-53}$ and the second $x \bmod n$, and the next lemma states what they give when the words are uniform (an assumption on the generator, which is only empirical).

Lemma 1.9 (The two draws). Let x be uniform on $\{0, \dots, 2^{64} - 1\}$. Then $\lfloor x/2^{11} \rfloor \cdot 2^{-53}$ is uniform on the 2^{53} multiples of 2^{-53} in $[0, 1)$, all exactly representable in double precision; and, for $1 \le n \le 2^{64}$, the probability that $x \bmod n = k$ differs from $1/n$ by less than 2^{-64} for each $k \in \{0, \dots, n-1\}$.

Proof. The map $x \mapsto \lfloor x/2^{11} \rfloor$ takes exactly 2^{11} words to each of the integers $0, \dots, 2^{53} - 1$, and an integer below 2^{53} times a power of 2 is a double, which gives the first claim. For the second, write $2^{64} = qn + r$ with $0 \le r < n$: the residue k has $q + 1$ preimages if $k < r$ and q otherwise, and hence a probability $(q + 1)/2^{64}$ or $q/2^{64}$; both differ from $1/n = (q + r/n)/2^{64}$ by at most $\max(r/n, 1 - r/n)/2^{64} < 2^{-64}$. □

Hence, the relative bias of the second draw is less than $n \cdot 2^{-64}$ (Lemire [151] removes it at the price of a rejection). This is negligible for the values of n with which the library draws, i.e., the costs of the cells of a grid (at most 255), the weights and the profits of a random knapsack, and the worker to which a node is sent, unless the range of a knapsack is chosen close to 2^{64}. Finally, `Stopwatch` (`SearchUtils.h:176`) is the wall clock (`std::chrono::steady_clock`) by which the searches measure their `seconds`.

1.7 The plan of the book

The chapters of the book and their dependencies are drawn in Figure 1.9.

- Chapter 2 gives the best-first search, its instances (breadth-first, depth-first, Dijkstra, A*), the reopening of the closed nodes, and the properties of A* with their proofs, i.e., completeness, optimality with an admissible heuristic, no reopening with a consistent one, and optimal efficiency, together with the goal test at the generation of a state; then the searches that give up the optimality for speed, i.e., greedy best-first search with the OPEN lists of the satisficing planners, enforced hill-climbing, the iterated searches of LAMA and the width-based searches.
- Chapter 3 treats the data structures of the searches: OPEN as a heap or as buckets, the breaking of ties, CLOSED as a hash table, perfect hashing, the arena of the nodes, and immediate against delayed duplicate detection.
- Chapter 4 treats the searches with bounded memory: beam search, depth-first branch-and-bound, iterative deepening, IDA* and its controlled reexpansion, transposition tables, recursive best-first search, external search with delayed duplicate detection on the disk, the symbolic and novelty-based searches, partial expansion (PEA* and EPEA*), fringe search, the real-time searches and the hierarchical path-finding on grids.
- Chapter 5 treats the heuristics, from the relaxations to the pattern databases with their compressions, and what to do when a heuristic does not work.
- Chapter 6 gives the bounded-suboptimal searches (weighted A* in any order of expansion, the duplicate dropping, the optimistic, focal and potential searches, the batch weighted A*), the anytime searches (AWA*, ARA* and the anytime parallel searches), and the incremental searches LPA*, D* Lite and their hash-distributed versions.
- Chapter 7 gives the front-to-end bidirectional searches (the bidirectional Dijkstra, MM, fMM, BAE*, NBS, DVCBS, GBFHS), the two directions on two workers, and the parallel searches on external memory.
- Chapter 8 gives the parallel searches on a shared OPEN (SPA*, the MultiQueue A*, the farm and PA*SE).

- Chapter 9 gives HDA*, its work distributions and the measures of §1.3, on threads and over MPI.
- Chapter 10 gives the distributions of HDA* built from a sample of the instance, their estimates and the automatic choice among them.
- Chapter 11 gives the Safe Parallel Best-NBlock-First search.
- Chapter 12 gives the further parallel schemes: PSDD and its variants, KBFS, the portfolio of A*, PRA*, TDS, the parallel IDA* and the parallel greedy searches.
- Chapter 13 gives branch-and-bound as a best-first search with an incumbent, and its parallel schemes.
- Chapter 14 describes the example domains of the library, with the proofs of the properties of their heuristics.
- Chapter 15 gives the multiple sequence alignment, its hyperplane work distribution, and the sets of instances of the repository.
- Chapter 16 gives classical planning as a domain of the library: the SAS$^+$ tasks, their heuristics and the searches on them.
- Chapter 17 gives the searches whose edges cost a vector, which return the Pareto front of the paths, and their parallel versions.

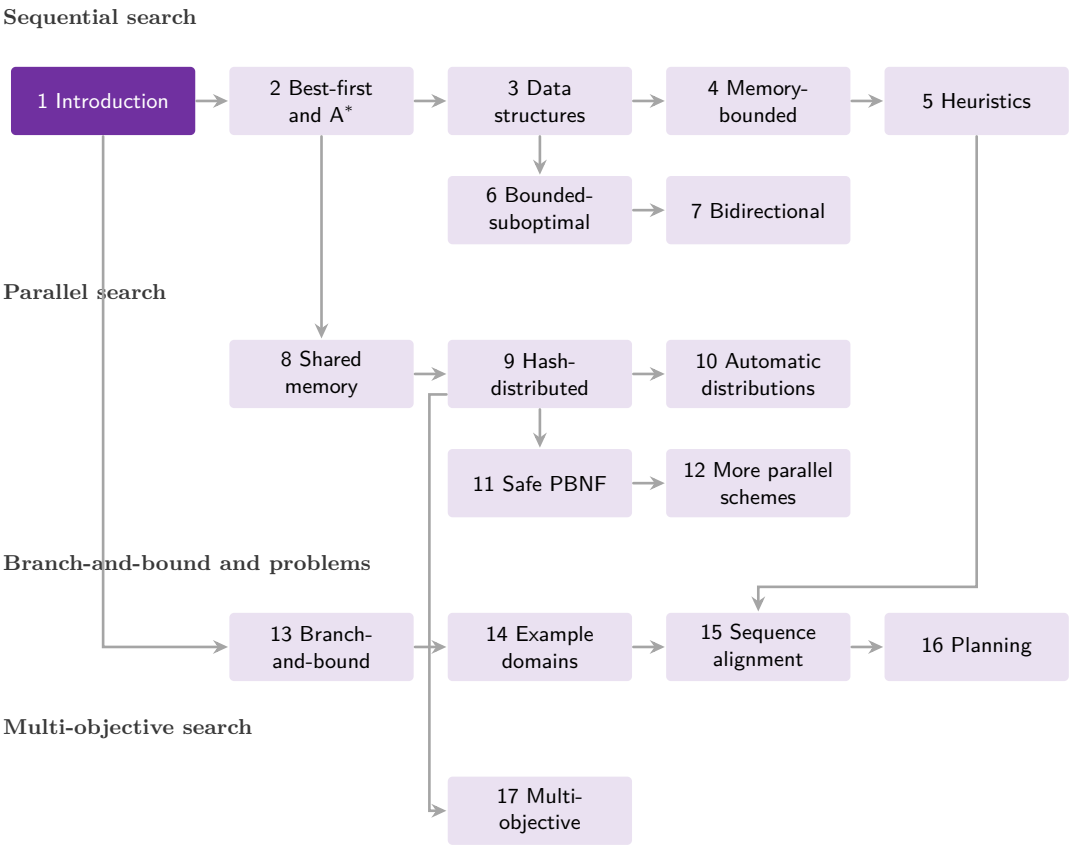


Figure 1.9. The chapters of the book; an arrow goes from a chapter to one that builds upon it. Every chapter uses the notation of this one and the domains of Chapter 14.

1.8 Notes and references

The formulation of §1.1 is that of Edelkamp and Schrödl [10] (chapter 1), whose book is the reference on heuristic search, and of Russell and Norvig [13] (chapter 3), from which the measures of §1.3 and Table 1.1 are taken; the classical treatment of A* is that of Nilsson [8], after the paper of Hart, Nilsson and Raphael [9] that introduced it, and the textbook of Jinnai [164] covers the same ground (its chapters 1 to 3) with the notation that Table 1.2 adapts. Our parallel searches follow the survey of Fukunaga et al. [1], and the measures LB and CO are those of Kishimoto, Fukunaga and Botea [2] and of Jinnai and Fukunaga [5]. Burns et al. [4] list the choices that make a search code fast (templates instead of virtual calls, a packed state, an incremental heuristic, the pruning of the parent, and so on), and the concepts of §1.5 are the way in which the library makes them available to any problem, instead of a single puzzle.

We remark that the interface of the library is narrower than that of a general planner: a `Domain` exposes

successors, not actions with preconditions and effects, and therefore the domain-independent heuristics of classical planning cannot be computed from it (this is the subject of Chapter 16). Also, a concept checks the signatures of the methods, not their semantics: an inadmissible heuristic, a hash with poorly spread bits or an "inverse" operator that costs less than the operator all compile. They then void the guarantees of the searches; the tests of the library check the domains that it ships (Chapter 14).

1.9 Exercises

Exercise 1.1 (`ctest -R PBeFS.reopen`). On the graph of Figure 1.3, compute h^* and verify that h is admissible; find every edge on which the inequality of consistency fails. Then run A* on it by hand, with the ties broken in favor of the larger g, and show that v_2 is expanded twice, the second time with $g(v_2) = 2$. The test checks that the cost found is 4 and that `reexpanded` is positive.

Exercise 1.2 (`ctest -R PBeFS.hda-thread`). Prove the analogue of the identity of Proposition 9.15 for the Abstract Zobrist key, in which the words are indexed by the abstract features, $z_A(s) = \bigoplus_{i \in F(s)} R_{A(i)}$: show that the key of a successor follows from the abstract features of the features of Δ, and that two features of Δ with the same abstract feature cancel. The test checks both keys against those computed from scratch on random walks of the 8-, 15- and 24-puzzle.

Exercise 1.3 (`ctest -R PBeFS.idastar`). Lemma 1.8 needs the inverse operator to cost as much as the operator. Give a graph with an operator whose inverse is cheaper, on which skipping the parent loses the optimal path; then explain why comparing a successor with the state two places back on the path (as IDA* does on the domains without operators) is safe on any graph with nonnegative costs. The test runs IDA* on the grids and on the map of Romania, which has no operators, and checks the optimal costs.

Exercise 1.4 (`ctest -R PBeFS.limits`). Describe the `Solution` returned by a search stopped by a node limit (its status, its cost and its path), and the one returned on an instance with no reachable goal. Then show that A* on a grid of 50×50 cells whose column 25 is blocked expands the 1250 cells on the side of the start before it reports `unsolvable`, as the test checks.

Exercise 1.5 (`ctest -R PBeFS.grid`). Write a wrapper `Blind< D >` that satisfies `Domain` whenever `D` does and has $h = 0$, which makes A* on it the algorithm of Dijkstra (see the class `Blind` of `test/test.cpp`); explain why it does not satisfy `OperatorDomain` even when `D` does, and what is lost by that. The test compares A* with and without the heuristic on random grids.

Exercise 1.6 (`ctest -R PBeFS.astar`). Using the counters of §1.3, estimate the mean branching factor of the 8-puzzle met by A* on the instance of figure 3.28 of [13], and explain why `generated / expanded` is smaller than the mean number of moves of a state (which is derived in Proposition 14.4). Finally, show that $9!/2$ states are reachable from the goal. The test checks this count by a breadth-first search of the whole space, and A* against the breadth-first distances of 200 of these states; the estimate of the branching factor is left to the reader, who can read the counters of the runs.

Exercise 1.7 (`ctest -R PBeFS.rubik`). The `TreeOperatorDomain` of Rubik's cube prunes, after a turn of a face, the turns of the same face and, when that face is the second of its pair of opposite faces, the turns of the first one. Show that a turn has 15 successors in the tree after a turn of the first face of a pair and 12 after one of the second, and derive the numbers 18, 243 and 3240 of the nodes at depths 1, 2 and 3 of the tree; then prove that the pruning keeps, for every path, a path with the same ends and at most the same number of turns. The test counts the tree to depth 5 against table 1 of Korf [59].

Exercise 1.8 (`ctest -R PBeFS.gbfs`). The test wraps the 15-puzzle in a `DeferredDomain` that counts the calls of its heuristic. Explain why the greedy search with lazy evaluation on it evaluates exactly the states that it selects and finds new, so that `evaluated` equals the calls counted, and why this number is smaller than `generated`. What would the same search count on the puzzle itself, which is not a `DeferredDomain`?

Exercise 1.9 (`ctest -R PBeFS.classic`). On a grid with 8 moves and cells of real costs, give two paths from s_0 to the same cell made of the same moves in another order, and show that their costs computed in `double` may differ; bound the difference as in the argument on the real costs of §1.5, and check that `cost_slack` covers it for any path of the grids of the test. The test checks that EPEA*, whose selection function compares the values of Δf up to the slack, returns the cost of A* on such grids.

2. Best-first search and A*

[Source: include/PBeFS/astar/AStar.h (AStar, astar), include/PBeFS/astar/Node.h (Node, trace_path), include/PBeFS/astar/Solution.h (Solution, SearchStats, SearchOptions, Status), include/PBeFS/a star/Domain.h (OperatorDomain, for_each_child), include/PBeFS/astar/ExplicitGraph.h; include/PB eFS/astar/GreedySearch.h (GreedySearch, GreedyOptions, GreedyNode, GreedyOpen, HeuristicCache, Qu eueSpec, gbfs), include/PBeFS/astar/EnforcedHillClimbing.h (EnforcedHillClimbing, EHCOptions, eh c), include/PBeFS/astar/IteratedSearch.h (IteratedSearch, IteratedOptions, lama_phases), include /PBeFS/astar/WidthSearch.h (NoveltyTable, IteratedWidth, SerializedWidth, BestFirstWidth, WidthO ptions, iw, siw, bfws, features_of, added_features); the tests astar, grid, graph, limits and reopen of test/test.cpp, and gbfs, gbfs-queues, ehc, iterated, width and planning]

We are concerned in this chapter with the sequential best-first search of the graph of a Domain (cf. Chapter 1, whose notation we use throughout), and in particular with A* [9], on which the other searches of the library are built. We first describe the generic best-first search, the difference between a search of the tree of the paths and a search of the graph, and the priorities that give breadth-first search, depth-first search, the algorithm of Dijkstra and A*. Then we state A* as the library implements it, and we prove its correctness under an admissible heuristic and its stronger properties under a consistent one. The last sections discuss the reopening of the closed nodes, the breaking of the ties, the pruning of the move back to the parent and the counters by which a search is measured. The data structures of OPEN and CLOSED, with the cost of their operations, are the subject of Chapter 3, and we only point to them here. Also, the searches that hold less than CLOSED in memory (IDA* and the search with delayed duplicate detection on the disk) are those of Chapter 4, and the heuristics themselves are those of Chapter 5.

The second part of the chapter, from §2.11 to §2.15, first gives the goal test at the generation of a state, by which A* saves the expansions of its last layer while keeping its guarantees (§2.11). It then turns to the searches that give up the optimality of the path for speed, as the satisficing planners do: greedy best-first search with the OPEN lists of the planners (§2.12), enforced hill-climbing (§2.13), the iterated searches of LAMA (§2.14) and the width-based searches (§2.15); the parallel greedy searches are those of Chapter 12. The benchmark, the notes and the exercises of both parts close the chapter (§2.16 to §2.18).

Throughout the chapter we assume that the costs of the edges are nonnegative, $c(u, v) \geq 0$, and that the heuristic is nonnegative, $h(u) \geq 0$, as it is in all the domains of the library. The second assumption is used only to have $h(t) \leq h^*(t) = 0$ imply $h(t) = 0$ at a goal t (a heuristic that is positive at a goal may make A* return a path that is not optimal, cf. Exercise 2.6).

2.1 Searching a tree and searching a graph

A best-first search keeps the nodes it knows in two sets: OPEN, the *frontier*, i.e., the nodes generated but still unexpanded, and CLOSED, the nodes already expanded. At each iteration it removes from OPEN a node of least priority $f(u)$, tests whether it is a goal and, if it is not, *expands* it, i.e., generates its successors by the domain (which are then *generated*) and puts them in OPEN. A state is thus, at any time, in one of three conditions: not generated yet, generated and waiting in OPEN, or expanded and in CLOSED (the faint, light and darker states of the figures of this chapter).

In its simplest form, a search of this kind does not look at what it has already seen: each successor is put in OPEN as a new node, even if its state has been generated before. Such a search explores the *tree of the paths* from s_0, in which a state typically appears many times, once for each path that reaches it, and it is called a *tree search*. Figure 2.1 shows a graph in which two paths reach c, and the tree that a tree search explores on it: c and t appear twice, and so does everything below them. On a graph with many short cycles, e.g., a grid, the paths of

length d are exponentially many in d, while the states within distance d are only polynomially many, and a tree search may even fail to terminate on a finite graph, since a cycle gives paths of any length (cf. section 3.1 of [10]).

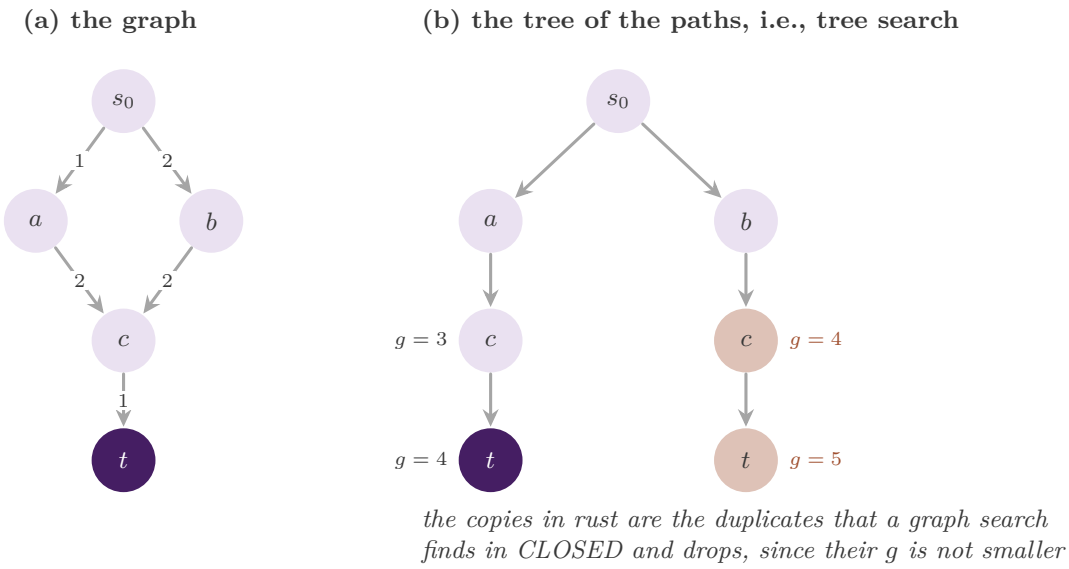


Figure 2.1. A graph in which two paths reach c (left), and the tree of its paths from s_0, which a tree search explores (right): the second copies of c and t, in rust, are the *duplicates* that a graph search detects in CLOSED and drops, since their g is not smaller than that of the first copies.

A *graph search* keeps, instead, one node per state: it looks each successor up among the states already generated (the *duplicate detection*), and it drops the successor if its state is known with a g that is not larger. This needs CLOSED to be a dictionary of the generated states, and the library keeps one (the `NodeTable` of Chapter 3), but it bounds the nodes by the states. One case remains: a successor v whose state is known, but which has now been reached through a cheaper path. If v is still in OPEN, its g is lowered and its priority with it; if v has already been expanded, it is *reopened*, i.e., moved from CLOSED back to OPEN with the new g, so that the cheaper path is propagated to its successors when v is expanded again. Whether a search must reopen depends on the priority and, for A*, on the heuristic (see §2.6 and §2.7). Dropping a duplicate without comparing the values of g is safe only when the first copy of a state is known to be reached through a path of least cost, e.g., in a breadth-first search with unit costs, or when an optimal path is not required.

Algorithm 2.1 (best-first graph search with a priority f).

```
 1  procedure BEST-FIRST(s₀, f)
 2     CLOSED ← {s₀}, with g(s₀) ← 0 and parent(s₀) ← nil
 3     OPEN ← {s₀}
 4     while OPEN ≠ ∅ do
 5        u ← a node of OPEN of least f(u); remove it from OPEN
 6        if u is a goal then return the path to u, of cost g(u)
 7        mark u as expanded
 8        for each successor v of u do
 9           g' ← g(u) + c(u, v)
10           if v ∉ CLOSED then          ▷ a new state
11              add v to CLOSED, with g(v) ← g' and parent(v) ← u
12              put v in OPEN
13           else if g' < g(v) then      ▷ a cheaper path to v
14              g(v) ← g';  parent(v) ← u
15              put v in OPEN, whether it was expanded or not
16     return unsolvable
```

As in the library, CLOSED holds all the generated states, each with a flag that records whether it has been expanded (line 7); the closed list of the textbooks is the subset of CLOSED with the flag set, and OPEN is a queue of the others. At line 6 the path is rebuilt by following the pointers `parent` from the goal back to s_0 (the function `trace_path` of `Node.h:168`). Note that the goal test is done when a node is *selected* at line 5, rather than when it is generated at line 10. In fact, a goal generated through a path of cost g is not necessarily reached

optimally, since a node of OPEN of smaller priority may still lead to a cheaper one; only the test at the selection gives the guarantee of Theorem 2.9.

2.2 The priority

Algorithm 2.1 is a family of searches, one for each priority f; the classical ones are the following, with $d(u)$ the number of edges of the path found to u (cf. chapter 3 of [13] and chapter 2 of [10]).

- *Breadth-first search* (BrFS), $f(u) = d(u)$, with the ties broken first in, first out: the nodes are expanded by layers of depth, and the first goal selected is one with the fewest edges, which is optimal when all the costs are equal.
- *Depth-first search* (DFS), $f(u) = -d(u)$, with the ties broken last in, first out: the search follows one path as deep as it goes and backtracks only at a dead end; OPEN then behaves as a stack, and the recursive formulation does without it altogether, keeping only the current path (which is the basis of the memory-bounded searches of Chapter 4).
- *The algorithm of Dijkstra*, also called uniform-cost search, $f(u) = g(u)$: the nodes are expanded by nondecreasing cost from s_0, and the first goal selected is optimal for any nonnegative costs.
- A^*, $f(u) = g(u) + h(u)$, with h the heuristic of the domain: $f(u)$ estimates the cost of a solution through u, and with an admissible h the first goal selected is optimal (§2.5). With $h = 0$, A* is the algorithm of Dijkstra.
- *Greedy best-first search*, $f(u) = h(u)$, which goes towards the goal that looks closest and is not optimal in general, and *weighted* A^*, $f(u) = g(u) + w\,h(u)$ with $w \geq 1$, which returns a path of cost at most $w\,C^*$ when h is admissible.

For a graph with branching factor b (the largest number of successors of a state), an optimal solution of d edges and cost C^* (the optimal cost of Table 1.2 of §1.4), costs of at least $\varepsilon > 0$, and largest depth m of a state, the first four have the following properties, the time and the space being measured in nodes generated.

search	complete	optimal	time and space
breadth-first	finite b	equal costs	$O(b^{d+1})$
depth-first, graph	finite graphs	no	$O(b^m)$
Dijkstra	finite b, $c \geq \varepsilon$	yes	$O(b^{1+D})$
A*	as Dijkstra	h admissible	see the text

Here $D = \lfloor C^*/\varepsilon \rfloor$, and the cost of A* is Proposition 2.19. The rows of breadth-first search and of the algorithm of Dijkstra are Proposition 2.20, which derives them from the theorems on A* with $h = 0$; the bound of breadth-first search has the exponent $d + 1$ because the goal is tested when it is selected (with the test at the generation, as in section 3.4 of [13], it would be d). The row of the depth-first search of a graph, which the library does not implement, is taken from section 3.4 of [13]: it terminates on a finite graph since it expands each state at most once, and it is not optimal since it returns the first goal that it reaches, whatever the cost of its path.

With a consistent heuristic, the nodes expanded by A* are a subset of those expanded by the algorithm of Dijkstra, but for the ties of the last layer: by Theorem 2.14, A* expands only nodes with $g^*(u) + h(u) \leq C^*$, hence with $g^*(u) \leq C^*$, while the algorithm of Dijkstra expands all those with $g^*(u) < C^*$ (Theorem 2.15 with $h_1 = 0$ gives the same conclusion when h is positive on the states that are not goals). Note that the depth-first search of a tree needs only $O(b\,m)$ memory, the current path and the siblings of its nodes, which is the reason for its use in Chapter 4; the depth-first search of a graph keeps CLOSED and needs as much memory as the states it visits. Consider the graph of five states s_0, a, b, c, t drawn in Figure 2.2, whose seven edges (s_0, a), (s_0, b), (a, b), (a, c), (b, c), (b, t) and (c, t) carry their costs and whose optimal path s_0, a, b, c, t costs 7 (the figure is read in §2.3). Breadth-first search expands s_0, a, b, c and t and returns the path s_0, b, t of two edges and cost 10; depth-first search (with the successors pushed in the order a, b) expands s_0, b, t and returns the same path; the algorithm of Dijkstra and A* both expand s_0, a, b, c, t and return the optimal path. That A* expands no fewer nodes than Dijkstra there is due to the size of the example. On the grid of Figure 2.6, a map of 9×7 cells with 4 moves of unit cost and a wall between s_0 and t (the figure is read in §2.6), a run of Algorithm 2.2 with $h = 0$ and the ties broken first in, first out expands 56 cells, while A* with the Manhattan distance and the ties broken in favor of the larger g, and last in, first out among equal f and g (the rule of the bucket queue of the library), expands 26 of them. These counts come from a separate simulation of Algorithm 2.2 with these rules of the ties, not from a run of the library, whose heap breaks the ties among equal f and g in no fixed order.

Of these searches, the library implements A* (`AStar.h`), and gets the algorithm of Dijkstra from it with a domain whose heuristic is zero. A wrapper of a `Domain` that returns 0 from `heuristic` and passes 0 as the heuristic of each successor is all that one needs, and the tests use one (`Blind`, `test/test.h:346`) to check A* on the grids against the costs of Dijkstra (test `grid`). On a domain with unit costs the same wrapper gives a breadth-first search, up to the order of the ties within a layer. The breadth-first search is also a search of its own, `BreadthFirst`, with the baselines of §4.14; greedy best-first search is `GreedySearch` (§2.12), and weighted A* is `AStar` with the weight of `SearchOptions`, the subject of Chapter 6 (a wrapper of the domain whose heuristic is $w\,h$ gives it too, cf. Exercise 2.7).

> **Status — planned.** The library has no depth-first search of a graph as a search of its own. The depth-first searches of a tree are those of Chapter 4.

2.3 A* in the library

The interface

A search of the library takes a type `D` that models the concept `Domain` of `Domain.h:169`. Such a type has the type `state_type` (a copyable value compared with `==`), the arithmetic type `cost_type` of g, h and the costs, and the members `initial_state()`, `is_goal( s )`, `heuristic( s )`, `hash( s )` (a well spread 64-bit hash, used by CLOSED) and `for_each_successor( s , h_s , f )`. The last one calls `f( t , c , h_t )` for each successor t of s, with $c = c(s,t)$ and $h_t = h(t)$; the value $h(s)$ is passed in, and hence the domain can compute $h(t)$ incrementally, as the sliding-tile puzzles do with the Manhattan distance (cf. Chapter 1 and Chapter 14).

The class template `AStar< D , Open >` (`AStar.h:153`) is constructed from the domain, which must outlive it, and from the `SearchOptions` of `Solution.h:188`. These are `node_limit`, the number of stored nodes at which the search gives up and returns `Status::node_limit` (0, the default, for no limit; Proposition 2.17 states when the test is made), and `expected_nodes`, the number of nodes that CLOSED is sized for at the start, instead of growing by doublings as it fills (0, the default, for a small initial table). Then `solve()` runs the search and returns a `Solution` (`Solution.h:153`), i.e., the `Status` (`solved`, `unsolvable` or `node_limit`), the `cost` of the path, the `path` itself as the sequence of its states from s_0 to the goal (empty unless the status is `solved`), and the counters `stats` of §2.10; `solved()` tells whether the status is `solved`. The OPEN list `Open` defaults to the bucket queue (`BucketOpenList`) for integer costs and to the heap (`HeapOpenList`) otherwise. The function `astar( domain , open , options )` (`AStar.h:326`) chooses it at run time by an `OpenKind`, i.e., `automatic` (the default of the class), `heap` or `buckets`; for a domain whose costs are not integers it runs the heap whatever the `OpenKind`, since the bucket queue needs integer keys. Both lists are described in Chapter 3. Their declarations are the following (quoted by hand, since in the header they come with their bodies).

```
template< Domain D , class Open = /* buckets if cost_t< D > is integral,
                                     else the heap */ >
class AStar {
 public:
 explicit AStar( const D & domain , const SearchOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
astar( const D & domain , OpenKind open = OpenKind::automatic ,
       const SearchOptions & options = {} );
```

The following example, from the test case `manual` of the library (`ctest -R PBeFS.manual`), solves an instance of the 8-puzzle with the Manhattan distance by both lists, and then with a node limit.

```
const EightPuzzle d( EightPuzzle::parse( "7 2 4 5 0 6 8 3 1" ) );
SearchOptions o;
o.expected_nodes = 1 << 12;                // CLOSED sized at the start
const auto s = astar( d , OpenKind::automatic , o );   // the buckets
const auto t = astar( d , OpenKind::heap , o );        // the heap
// the path goes from the initial state to a goal, one move per cost unit
const bool path = s.solved() && ( s.status == Status::solved ) &&
                  ( s.path.front() == d.initial_state() ) &&
```

```
                d.is_goal( s.path.back() ) &&
                ( s.path.size() == std::size_t( s.cost ) + 1 );
const auto & st = s.stats;      // expanded, reexpanded, generated,
                                // duplicates and stored
o.node_limit = 5;               // stop after storing 5 nodes
const auto l = astar( d , OpenKind::automatic , o );
// l.status == Status::node_limit, and l.path is empty
```

The examples of the repository (`examples/`) do the same on all the domains of the library.

The algorithm

In the library, the search is Algorithm 1 of the survey of Fukunaga et al. [1], i.e., Algorithm 2.1 with $f(u) = g(u) + h(u)$. Two choices on the data structures change its form, while its outcome stays the same. First, OPEN holds *entries* (f, g, u) rather than nodes, ordered by increasing f and, for equal f, by decreasing g (§2.8). Second, neither OPEN list of the library can lower the key of an entry: a node reached through a cheaper path is pushed again with its new g, and the old entry, which records the g of its push, becomes *stale*, i.e., its node is closed or has a different g. A stale entry is dropped when it reaches the top of OPEN. We give the pseudocode of the Doxygen comment of `AStar.h`.

Algorithm 2.2 (A*).

```
 1  procedure A*(s₀)
 2     CLOSED ← {s₀}, with g(s₀) ← 0 and parent(s₀) ← nil
 3     OPEN ← {(f(s₀), g(s₀), s₀)}
 4     while OPEN ≠ ∅ do
 5        (f, g, u) ← the first entry of OPEN; remove it from OPEN
 6        if u is closed or g ≠ g(u) then continue      ▷ a stale entry
 7        close u
 8        if u is a goal then return the path to u, of cost g(u)
 9        for each successor v of u do
10           g' ← g(u) + c(u, v)
11           if v ∉ CLOSED then
12              add v to CLOSED, with g(v) ← g' and parent(v) ← u
13              push (g' + h(v), g', v) in OPEN
14           else if g' < g(v) then           ▷ a cheaper path: reopen v
15              g(v) ← g';  parent(v) ← u;  v is not closed
16              push (g' + h(v), g', v) in OPEN
17     return unsolvable
```

We now comment on the lines that are not obvious. Line 6 drops the stale entries. An entry whose node is closed was pushed before the node was expanded with a smaller (or equal) g, and an entry whose g differs from that of the node was overtaken by a cheaper path, whose entry is in OPEN with a smaller f. In fact, the flag *closed* of line 7 records that the node has been expanded *with its current* g; line 15 clears it when a cheaper path is found, whether the node was waiting in OPEN or already expanded, which is the reopening of §2.7. Also, the node keeps a second flag, *expanded*, which is never cleared and by which the expansions of a node already expanded are counted (`Node.h:82`). With a consistent heuristic the branch at lines 14 to 16 is taken only for nodes still in OPEN (Theorem 2.13).

Algorithm 2.2 leaves out, as all the pseudocode of the book (§1.4), the counters, the node limit and the prefetches. Since the claims of §2.10 are about the counters, we also give the loop of `AStar::solve()` (`AStar.h:180`) with all of them. Its expansion is split in two passes: the first asks the domain for all the successors of u, computes the hash of each one and *prefetches*, i.e., asks the processor to start loading into its cache, the slot of CLOSED where the lookup of that successor will start; the second does the lookups, whose cache misses thus overlap instead of being paid one after the other (§3.8).

Algorithm 2.3 (A* as `AStar::solve()` runs it, with a node limit L, $L = 0$ meaning none).

```
1  procedure SOLVE(s₀, L)
2     CLOSED ← {s₀}, with g(s₀) ← 0, h(s₀) and parent(s₀) ← nil
3     OPEN ← {(h(s₀), 0, s₀)}
4     while OPEN ≠ ∅ do
```

```
 5       (f, g, u) ← the first entry of OPEN; remove it from OPEN
 6       if u is closed or g ≠ g(u) then continue      ▷ a stale entry
 7       close u;  expanded ← expanded + 1
 8       if u was expanded before then reexpanded ← reexpanded + 1
 9       mark u as expanded
10       if u is a goal then return (solved, g(u), the path to u)
11       K ← ∅                                  ▷ pass 1: the successors
12       for each (v, c(u, v), h(v), o) of CHILDREN(u, h(u), op(u)) do
13          compute the hash of v and prefetch its slot of CLOSED
14          append (v, c(u, v), h(v), o) to K
15       generated ← generated + |K|
16       for each (v, c, h(v), o) of K, in order do  ▷ pass 2: lookups
17          g' ← g(u) + c
18          if v ∉ CLOSED then
19             add v to CLOSED, with g(v) ← g', parent(v) ← u, op(v) ← o
20             push (g' + h(v), g', v) in OPEN
21          else if g' < g(v) then
22             g(v) ← g';  parent(v) ← u;  op(v) ← o;  v is not closed
23             push (g' + h(v), g', v) in OPEN
24          else duplicates ← duplicates + 1
25       if L > 0 and |CLOSED| ≥ L then return (node_limit)
26    return (unsolvable)
```

Here CHILDREN is Algorithm 1.1 of §1.4, which passes the operator o of each successor and skips the move back to the parent in a domain that knows its operators (§2.9); `op(u)` is the operator stored in the node, `no_op` for s_0. Line 19 is the lookup and the insertion of CLOSED (`find_or_insert` of `NodeTable`, which builds the node only when the state is new), and `stored` is |CLOSED| when the search ends, since CLOSED keeps one node per state. Also, line 7 sets the flag *closed* and line 9 the flag *expanded* of `Node.h:81` and `Node.h:82`; the path of line 10 is rebuilt by `trace_path` (`Node.h:168`).

Proposition 2.17 proves that Algorithm 2.3 does what Algorithm 2.2 does, and what the node limit changes.

A trace

Figure 2.2 and Figure 2.3 follow Algorithm 2.2 on a graph of five states with a consistent heuristic (the values g/h are written next to each state, in rust where they change, and the entries of OPEN are listed in the order of the queue, the new ones in rust and the stale ones struck out). In step (3) the expansion of a finds a path to b of cost 3, cheaper than the edge of cost 4 from s_0: b is still in OPEN, and the entry $(8, 4, b)$ becomes stale. In step (4) the goal t is generated with $g = 9$, but it is not tested, and in step (5) the expansion of c lowers its g to 7; in step (6) t is selected and the path s_0, a, b, c, t of cost 7 is returned, while three stale entries are still in OPEN and are never popped.

2.4 Admissible and consistent heuristics

Much of the behavior of A* depends on two properties of h, which we now define (the heuristics of the library and how they are obtained are discussed in Chapter 5). Let $h^*(u)$ be the least cost of a path from u to a goal ($+\infty$ if there is none), and $k(u, v)$ the least cost of a path from u to v ($+\infty$ if there is none).

Definition 2.1 (Admissible heuristic). A heuristic h is *admissible* if $h(u) \le h^*(u)$ for each state u.

Definition 2.2 (Consistent heuristic). A heuristic h is *consistent* if $h(t) = 0$ for each goal t and $h(u) \le c(u, v) + h(v)$ for each edge (u, v).

The inequality of Definition 2.2 is the triangle inequality of h along the edges (Figure 2.4): the estimate from u is never larger than the cost of an edge plus the estimate from its head. Some textbooks define consistency over all the pairs of states, with $k(u, v)$ in place of $c(u, v)$, and call the edgewise property *monotonicity*; the two are equivalent, as the next lemma shows, and the property of f along a path that is sometimes also called monotonicity is a consequence of either (Lemma 2.12).

Lemma 2.3 (Edgewise and pairwise consistency). A heuristic satisfies $h(u) \le c(u, v) + h(v)$ for each edge (u, v) if and only if it satisfies $h(u) \le k(u, v) + h(v)$ for each pair of states u, v.

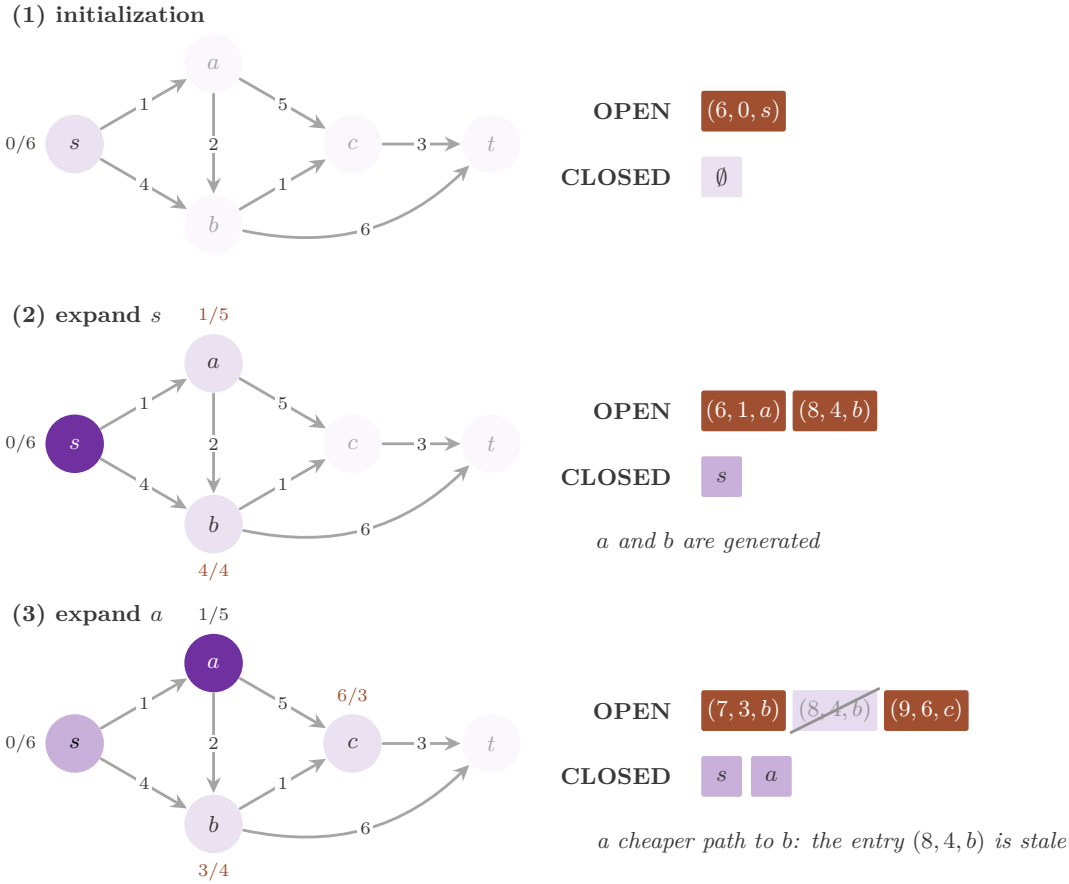


Figure 2.2. Algorithm 2.2 on a graph of five states, steps (1) to (3), the initial state s_0 being labelled s: the node being expanded is violet, the closed ones darker, those not generated yet faint; g/h is written next to each state, in rust where it changes. The entries (f, g, u) of OPEN are in the order of the queue, the new ones in rust, the stale ones struck out.

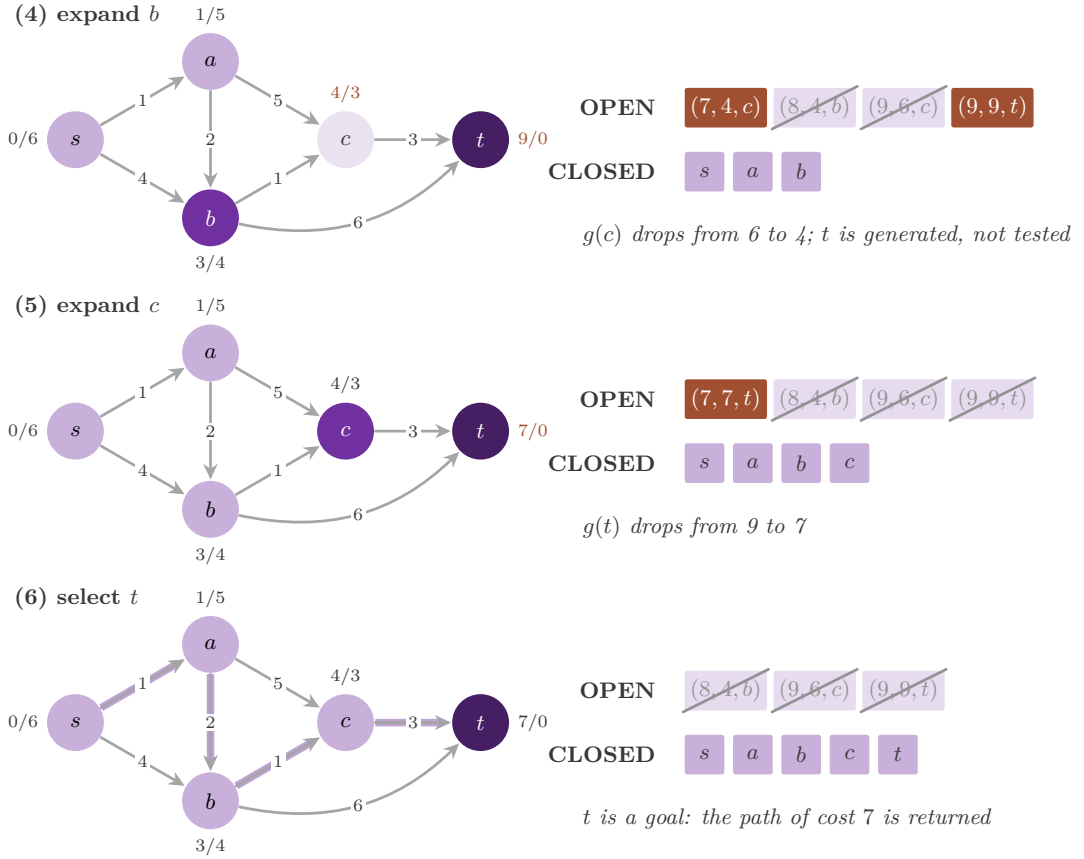


Figure 2.3. Algorithm 2.2 on the graph of Figure 2.2, steps (4) to (6): the goal is generated in step (4) with $g = 9$, improved in step (5) to $g = 7$, and returned when it is selected in step (6); the optimal path is drawn in violet.

Proof. Since $k(u,v) \le c(u,v)$ for each edge, the pairwise inequality implies the edgewise one. Conversely, if v is not reachable from u, then $k(u,v) = +\infty$ and there is nothing to prove; otherwise, let $u = w_0, w_1, \ldots, w_q = v$ be a path of least cost. Summing the edgewise inequalities $h(w_{i-1}) - h(w_i) \le c(w_{i-1}, w_i)$ for $i = 1, \ldots, q$, the left-hand side telescopes to $h(u) - h(v)$ and the right-hand side is $k(u,v)$. □

Lemma 2.4 (Consistency implies admissibility). A consistent heuristic is admissible.

Proof. If no goal is reachable from u, then $h^*(u) = +\infty$. Otherwise, let t be a goal with $k(u,t) = h^*(u)$; by Lemma 2.3, $h(u) \le k(u,t) + h(t) = h^*(u)$, since $h(t) = 0$. □

Of course, the converse does not hold. On the graph of Figure 2.7, whose states $v_0, \ldots, v_4$ are joined by the arcs (v_0, v_1), (v_1, v_2), (v_2, v_3) and (v_3, v_4) of cost 1 and by the arc (v_0, v_2) of cost 3, each directed from the state of the smaller index to that of the larger one (the figure draws them as lines), with $s_0 = v_0$ and the goal v_4 (the figure is read in §2.7), the heuristic $h = (0, 3, 0, 1, 0)$ is admissible, since $h^* = (4, 3, 2, 1, 0)$, but $h(v_1) = 3 > c(v_1, v_2) + h(v_2) = 1$. The sequence score of Nilsson (`SequenceScore.h`, used by the example `8Puzzle`) is a heuristic of the sliding-tile puzzles that is not even admissible: A* then usually expands fewer nodes, but it may return a path that is not optimal (`SequenceScore.h:50`).

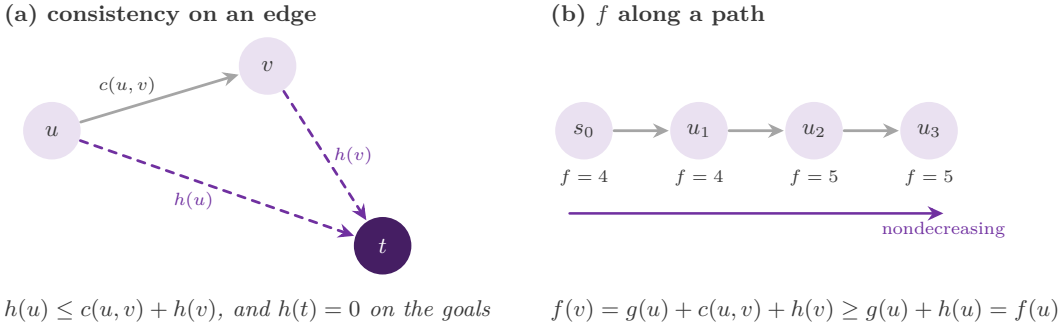


Figure 2.4. Consistency (left): the estimate $h(u)$ is at most the cost of the edge (u,v) plus the estimate $h(v)$, i.e., h satisfies the triangle inequality along the edges. Its consequence (right): $f = g + h$ does not decrease along a path, which is Lemma 2.12.

Both properties are preserved by the maximum of heuristics, which is used by the pattern databases of Chapter 5 (the maximum with the reflection of a table) and in general to combine several heuristics of a domain.

Lemma 2.5 (The maximum of heuristics). Let $h_1, \ldots, h_m$ be heuristics and $h(u) = \max_i h_i(u)$. If all the h_i are admissible, so is h; if all are consistent, so is h.

Proof. The first claim is immediate, since $h_i(u) \le h^*(u)$ for each i. For the second, $h(t) = \max_i h_i(t) = 0$ at a goal; for an edge (u,v), let i be an index with $h(u) = h_i(u)$: then $h(u) = h_i(u) \le c(u,v) + h_i(v) \le c(u,v) + h(v)$. □

We say that a heuristic h_2 is *more informed* than h_1 if $h_2(u) > h_1(u)$ for each state u that is not a goal; Theorem 2.15 below shows that, between two consistent heuristics, the more informed one never makes A* expand a node that the other does not expand.

2.5 The correctness of A*

We now prove that A* with an admissible heuristic returns an optimal path, and that it terminates. Our argument rests on one invariant on OPEN (cf. Lemma 1 of [9] and section 2.1 of [1]), which we state for any state w (a goal being a special case), since §2.6 needs it in that form. Let $g^*(w)$ be the least cost of a path from s_0 to w, and recall that $C^* = h^*(s_0)$.

Lemma 2.6 (Values of g). At any time of Algorithm 2.2, $g(u)$ is the cost of a path from s_0 to u, and hence $g(u) \ge g^*(u)$, for each node u in CLOSED.

Proof. By induction on the assignments to g: $g(s_0) = 0$ is the cost of the empty path, and lines 12 and 15 set $g(v) = g(u) + c(u,v)$, the cost of the path to u of value $g(u)$ followed by the edge (u,v). □

Lemma 2.7 (The invariant of OPEN). Let w be a state reachable from s_0 that, at the start of an iteration of the loop at line 4, is not closed with $g(w) = g^*(w)$. Then each path of least cost P from s_0 to w has a node u that is not closed, has $g(u) = g^*(u)$ and has in OPEN the entry $(g^*(u) + h(u), g^*(u), u)$, which is not stale.

Proof. Let u be the first node of P that is not closed with its optimal g; it exists, since w is such a node. If $u = s_0$, then s_0 has not been expanded yet, and it is in OPEN with $g = 0$ by line 3. Otherwise the predecessor p of u on P is closed with $g(p) = g^*(p)$; since the prefix of P to u is a path of least cost, $g^*(u) = g^*(p) + c(p, u)$. When p was last expanded, with $g(p) = g^*(p)$, line 10 computed $g' = g^*(u)$ for u. Then either u was new, or it had $g(u) > g'$, or it already had $g(u) \leq g' = g^*(u)$. In the first two cases lines 12 to 16 set $g(u) = g^*(u)$ and pushed the entry $(g^*(u) + h(u), g^*(u), u)$, while in the third case $g(u) = g^*(u)$ by Lemma 2.6 and such an entry had been pushed when $g(u)$ took that value. Since $g(u)$ cannot fall below $g^*(u)$, it has not changed since then, and by the choice of u the node has not been closed with it; hence the entry is still in OPEN, and it is valid. □

In other words (Figure 2.5), the expanded prefix of an optimal path is followed by one node of OPEN that carries its optimal g. Its expansion generates the next node of the path with its optimal g, which takes its place. Note that the proof needs the reopening of lines 14 to 16: the next node might have been closed before with a larger g, and without the reopening it would never be put back in OPEN.

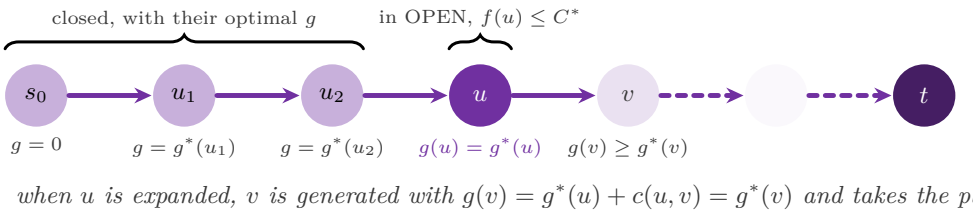


Figure 2.5. The invariant of OPEN (Lemma 2.7): on an optimal path from s_0 to a goal, the nodes before u are closed with their optimal g, and u is in OPEN with $g(u) = g^*(u)$; if h is admissible, $f(u) \leq C^*$.

Lemma 2.8 (Every selected node has $f \leq C^*$). If h is admissible and a goal is reachable from s_0, each entry (f, g, u) that is selected at line 5 and is not stale, before a goal is returned, has $f \leq C^*$, and therefore $g^*(u) + h(u) \leq C^*$.

Proof. Let t^* be a goal with $g^*(t^*) = C^*$. Before the search returns, t^* is not closed with $g(t^*) = C^*$ (closing it would end the search at line 8), and by Lemma 2.7 an optimal path to t^* has a node u' with an entry of key $f' = g^*(u') + h(u') \leq g^*(u') + h^*(u') = C^*$ in OPEN, the inequality being the admissibility of h and the equality the optimality of the path through u'. The entry selected at line 5 has the least key in OPEN, and hence $f \leq f' \leq C^*$; finally, $g^*(u) + h(u) \leq g(u) + h(u) = f$ by Lemma 2.6, since a selected entry that is not stale has $g = g(u)$. □

Theorem 2.9 (Admissibility of A*). If h is admissible and A* returns a path, the path is optimal, i.e., its cost is C^*.

Proof. Let the search return at line 8 with the goal t selected from a non-stale entry $(f, g(t), t)$. Since $h(t) \leq h^*(t) = 0$ and $h \geq 0$, $h(t) = 0$ and $f = g(t)$; by Lemma 2.8, $g(t) \leq C^*$, and by Lemma 2.6, $g(t)$ is the cost of a path from s_0 to t, and therefore $g(t) \geq C^*$. □

It remains to prove that A* terminates. On a finite graph this holds for any heuristic, admissible or not; on an infinite graph it needs a positive lower bound on the costs, and the absence of cycles of nonpositive cost is not enough.

Lemma 2.10 (The parent pointers form a tree). At any time of Algorithm 2.2, the pointers `parent` of the nodes of CLOSED form a tree rooted at s_0, and $g(\text{parent}(v)) + c(\text{parent}(v), v) \leq g(v)$ for each node $v \neq s_0$.

Proof. The inequality holds when `parent(v)` is set, with equality, and is preserved afterwards, since g never increases and a later change of `parent(v)` sets it again. Suppose that the pointers form a cycle $v_1, \ldots, v_q$, with v_{i-1} the parent of v_i (indices modulo q), and let v_j be the node of the cycle whose pointer was set last, at line 15, to $u = v_{j-1}$. Just before that assignment the other pointers of the cycle were already in place, and the inequality along the path $v_j, v_{j+1}, \ldots, v_{j-1} = u$ gives $g(u) \geq g(v_j)$, the costs being nonnegative; the new value $g(u) + c(u, v_j)$ is then not smaller than the old $g(v_j)$, while line 14 requires it to be strictly smaller. Hence there is no cycle, and since each node but s_0 has a parent, the pointers form a tree rooted at s_0. □

Theorem 2.11 (Termination of A*). Algorithm 2.2 terminates (i) on any finite graph, and (ii) on any graph with finite branching whose costs are at least some $\varepsilon > 0$, provided that h is admissible and that a goal is reachable from s_0.

Proof. Each iteration either drops a stale entry or closes a node, and the entries are pushed only by lines 3, 13 and 16, once per value taken by g of a node. Hence, it suffices to show that the pairs $(v, g(v))$ taken by the nodes

are finitely many. (i) By Lemma 2.6, each value of $g(v)$ is the cost of a path from s_0 to v, and the successive values of $g(v)$ are strictly decreasing. A path decomposes into a simple path and a multiset of simple cycles, whose costs are nonnegative. Since the simple paths and the simple cycles of a finite graph are finitely many, there is an $\eta > 0$ not larger than the cost of each simple cycle of positive cost, and a path whose cost is at most the first value of $g(v)$ contains at most that value over η cycles of positive cost. Thus the costs of such paths are finitely many, and so are the values of $g(v)$. (ii) By Lemma 2.8, each node u expanded before the end has $g(u) \le f \le C^*$; by Lemma 2.10, $g(u)$ is at least the cost of the path of the tree from s_0 to u, which therefore has at most C^*/ε edges. Under a finite branching, the paths of at most C^*/ε edges are finitely many, and so are the nodes that may be expanded and their successors. The values of g are then finitely many by (i), applied to the finite subgraph that these nodes span. □

We remark that the condition of (ii) cannot be weakened to the absence of cycles of nonpositive cost. On an infinite chain $s_0, w_1, w_2, \ldots$ whose i-th edge costs 2^{-i}, with a goal reached from s_0 by one edge of cost 1 and $h = 0$, each node of the chain has $g < 1$ and is selected before the goal, and the search never ends. On a finite graph without a reachable goal, A* ends with OPEN empty and returns `Status::unsolvable`, which the test `limits` checks on a grid split by a wall; on an infinite graph it may end only by the node limit.

2.6 Consistent heuristics

With a consistent heuristic A* has stronger properties, which we now prove. The values of f do not decrease along a path, and the nodes are selected by nondecreasing f; each node is closed with its optimal g and never reopened; and the set of the expanded nodes is determined by C^*, up to the ties on the last value of f.

Lemma 2.12 (Monotonicity of f). If h is consistent and v is a successor of u with $g(v) = g(u) + c(u, v)$, then $f(v) \ge f(u)$.

Proof. $f(v) = g(u) + c(u, v) + h(v) \ge g(u) + h(u) = f(u)$, by Definition 2.2. □

Theorem 2.13 (No reopening). If h is consistent, each node is closed by Algorithm 2.2 with $g(u) = g^*(u)$; hence no closed node is ever reopened, each state is expanded at most once, and the keys of the selected non-stale entries are nondecreasing.

Proof. Suppose that a node u is selected from a non-stale entry with $g(u) > g^*(u)$. By Lemma 2.7 applied to $w = u$, an optimal path to u has a node w' with an entry of key $g^*(w') + h(w')$ in OPEN, and $w' \ne u$, since $g(u) \ne g^*(u)$. Now, by Lemma 2.3, $h(w') \le k(w', u) + h(u)$, and $g^*(w') + k(w', u) = g^*(u)$ since w' is on an optimal path to u; hence $g^*(w') + h(w') \le g^*(u) + h(u) < g(u) + h(u)$. Thus the entry of w' has a smaller key than that of u, and u would not have been selected. Each node is therefore closed with its optimal g, which no path can improve, and line 14 never holds for a closed node; a state is then expanded only when its node is closed, i.e., once. Finally, when u is expanded with key $f(u)$, all the keys in OPEN are at least $f(u)$, and by Lemma 2.12 so are the keys pushed by the expansion; hence the next selected key is not smaller. □

Thus, the keys of the selected entries grow by layers of equal f, and C^* is the last layer. Equivalently, A* is the algorithm of Dijkstra on the reduced costs $c'(u, v) = c(u, v) - h(u) + h(v)$, which consistency makes nonnegative. Indeed, along any path $s_0 = w_0, \ldots, w_q = u$ the reduced costs telescope to $\sum_i c(w_{i-1}, w_i) + h(u) - h(s_0)$: the reduced cost of a path is its cost plus $h(u) - h(s_0)$, which depends only on its ends, the least reduced cost from s_0 to u is $g^*(u) + h(u) - h(s_0)$, and the key $g(u) + h(u)$ of A* is the key of the algorithm of Dijkstra on c' plus the constant $h(s_0)$, which orders the entries in the same way. The next two results characterize the nodes that A* expands (Figure 2.6); the first part holds for any admissible heuristic and is Lemma 2.8 again.

Theorem 2.14 (Necessary and sufficient conditions of expansion). Let h be admissible and a goal be reachable. (i) Every node u expanded by Algorithm 2.2 before the goal it returns has $g^*(u) + h(u) \le C^*$. (ii) If h is consistent, each node u with $g^*(u) + h(u) < C^*$ is expanded before the search returns.

Proof. (i) is Lemma 2.8. (ii) Suppose that the search returns, selecting a goal t with key C^* (by Theorem 2.9), while u has not been expanded. By Lemma 2.7, an optimal path to u then has a node w with an entry of key $g^*(w) + h(w)$ in OPEN. As in the proof of Theorem 2.13, this key is at most $g^*(u) + h(u) < C^*$, and t, of key C^*, would not have been selected before it. □

A* with a consistent heuristic thus expands all the nodes with $f^*(u) = g^*(u) + h(u) < C^*$, none with $f^*(u) > C^*$, and some of those with $f^*(u) = C^*$, which ones depending on the order of the ties (§2.8). The comparison of two heuristics follows.

Theorem 2.15 (A more informed heuristic expands no more nodes). Let h_1 and h_2 be consistent, and h_2 more

informed than h_1. Every node expanded by A* with h_2, but the goal it returns, is expanded by A* with h_1.

Proof. Let u be a node that is not a goal and is expanded with h_2. By Theorem 2.14 (i), $g^*(u) + h_2(u) \leq C^*$, and since $h_1(u) < h_2(u)$, $g^*(u) + h_1(u) < C^*$; by Theorem 2.14 (ii), u is expanded with h_1. □

If h_2 only dominates h_1, i.e., $h_2(u) \geq h_1(u)$, the same argument covers the nodes with $g^*(u) + h_2(u) < C^*$, and the nodes of the last layer may differ by the ties (cf. section 2.1 of [1]). Theorem 2 of [9] also shows that A* is optimal, in the number of expanded nodes, among the admissible searches that use the same information; the precise sense of that claim, which depends on the ties, is discussed in chapter 2 of [10].

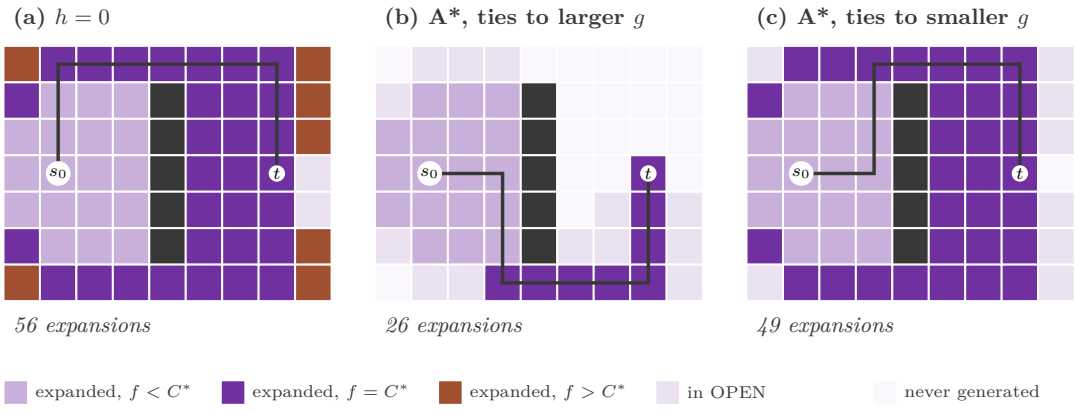


Figure 2.6. The cells expanded on a grid with 4 moves of unit cost, from s_0 to t around a wall, by (a) the algorithm of Dijkstra (A* with $h = 0$, the ties broken first in, first out), and by A* with the Manhattan distance, breaking the ties of f (b) in favor of the larger g and (c) in favor of the smaller one. The colours classify the expanded cells by $f^* = g^* + h$, with h the Manhattan distance ($C^* = 12$): both runs of A* expand the 18 cells with $f^* < C^*$ and none with $f^* > C^*$, but (b) expands 8 of the 31 cells with $f^* = C^*$, and (c) all of them; the path returned is drawn as a dark line.

2.7 Reopening with an inconsistent heuristic

When h is admissible but not consistent, Theorem 2.13 fails: a node may be closed with a g larger than g^*, and it is then reopened when a cheaper path to it is found, as line 14 of Algorithm 2.2 allows. Figure 2.7 shows the graph of the test `reopen` (`test/files.cpp:623`), in which $h(v_1) = 3$ makes v_1 look expensive. After v_0, A* expands v_2 (key 3) with $g(v_2) = 3$, through the edge of cost 3, and then v_1 (key 4), which finds the path of cost 2 to v_2 and reopens it. Then v_2 is expanded a second time with key 2, smaller than the key 4 of the previous expansion (the keys of the selected entries are thus not monotone), and the search returns the path v_0, v_1, v_2, v_3, v_4 of cost 4, which is optimal, as Theorem 2.9 guarantees. The test checks the cost and that the counter `reexpanded` is positive.

Reopening keeps A* optimal, but it may cost a great deal: with an inconsistent heuristic the number of expansions may grow much faster than the number of states; a family of graphs on which it grows exponentially in the number of states is discussed in chapter 2 of [10], and we do not reproduce it here. The analysis of Felner et al. [31] discusses when inconsistent heuristics are nevertheless useful. In the library a domain does not declare whether its heuristic is consistent, and `AStar` always reopens. The counter `reexpanded` of the `SearchStats` reports how many expansions were of nodes already expanded; it is zero when the heuristic is consistent (by Theorem 2.13), and it is the first quantity to look at when a search is slower than expected (cf. Chapter 5). The pattern databases with the blank minimized, for instance, are admissible but not consistent, and A* reopens nodes on them. Note that a search that cannot reopen, such as the search with delayed duplicate detection of Chapter 4, refuses a successor whose f is smaller than that of its parent.

2.8 The ties within the last layer

Theorem 2.14 leaves free the order of the nodes with the same f. This order matters on the domains with few distinct values of f, where the last layer $f = C^*$ may hold a large part of the nodes generated. Both OPEN lists of the library break the ties in favor of the larger g, i.e., of the node with the smaller h, which is closer to a goal if the heuristic is informative. That on the domains with unit costs the search then usually reaches a goal after expanding a small part of the last layer is an empirical observation, made in the experiments of Burns et al. [4],

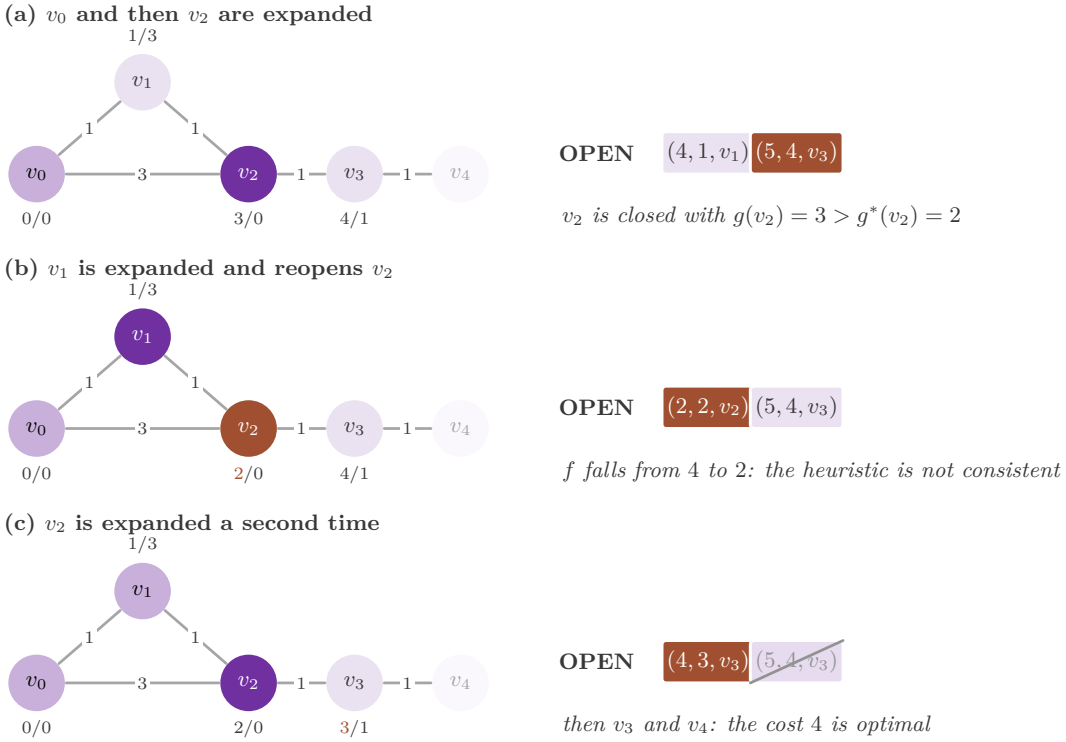


Figure 2.7. A* with the admissible but inconsistent heuristic $h = (0, 3, 0, 1, 0)$ on the graph of the test `reopen`, whose arcs go from left to right. (a) v_2 is closed with $g = 3$; (b) the expansion of v_1 finds $g(v_2) = 2$ and reopens v_2 (rust); (c) v_2 is expanded again and lowers $g(v_3)$, making the old entry of v_3 stale.

and not a theorem: an adversarial order of the successors can make either rule expand the whole layer. On the grid of Figure 2.6, the simulation of §2.2 expands 26 cells with the ties in favor of the larger g, against 49 with the ties in favor of the smaller one; the difference is entirely in the layer $f^* = C^*$ (8 of its 31 cells against all of them), as Theorem 2.14 requires. Among the entries with the same f and g, the bucket queue takes the last one pushed, which tends to follow one path down the layer, while the heap takes any. Also, the library can be built with one of three *sets of data structures*, i.e., three implementations of OPEN, CLOSED and the arena of the nodes chosen at compile time (`Structures.h`, §3.12); they break these last ties differently, and hence they may expand different nodes of the last layer, while by Theorem 2.14 they expand the same nodes below it when h is consistent.

2.9 Pruning the move back to the parent

On a domain whose moves are reversible at the same cost, such as the sliding-tile puzzles and the grids with symmetric costs, each expansion generates the parent of the node it expands. Such a successor is a duplicate that CLOSED would drop, since it is reached with a g larger than that of the parent; but detecting it costs a hash and a lookup, i.e., one of the about b lookups of each expansion. A domain that models the concept `OperatorDomain` (`Domain.h:356`) avoids it: its `for_each_successor( s , h_s , op , f )` receives the operator `op` that generated s (`no_op` at the root), skips the successor that the inverse of `op` leads to, and passes to `f` the operator of each successor it generates, which the node stores in its field `op` (Figure 2.8, left). The function `for_each_child` (`Domain.h:705`) calls this form when the domain has it and the plain one otherwise, and all the searches of the library go through it.

Proposition 2.16 (Parent pruning changes no outcome). If the inverse of each operator has the same cost, skipping the successor that the inverse of the operator of u leads to changes neither the path returned by Algorithm 2.2 nor the order of its expansions; it changes only the counters `generated` and `duplicates` of §2.10, each by the number of the successors skipped.

Proof. Let $p = \text{parent}(u)$ be the state skipped, so that $g(u) \geq g(p) + c(p, u)$ by Lemma 2.10. The skipped successor would have $g' = g(u) + c(u, p) \geq g(p) + 2\,c(p, u) \geq g(p)$, and p is in CLOSED; hence line 14 would not hold, and the successor would only be counted as a duplicate, without any push. □

A search that keeps no CLOSED, such as IDA*, cannot detect the duplicate at all, and without the pruning it would walk back and forth along each edge; on a domain without operators it compares each successor with the state two places back on its path (Figure 2.8, right, and Chapter 4). The pruning by operator and its use in the fast implementations of A* are discussed by Burns et al. [4].

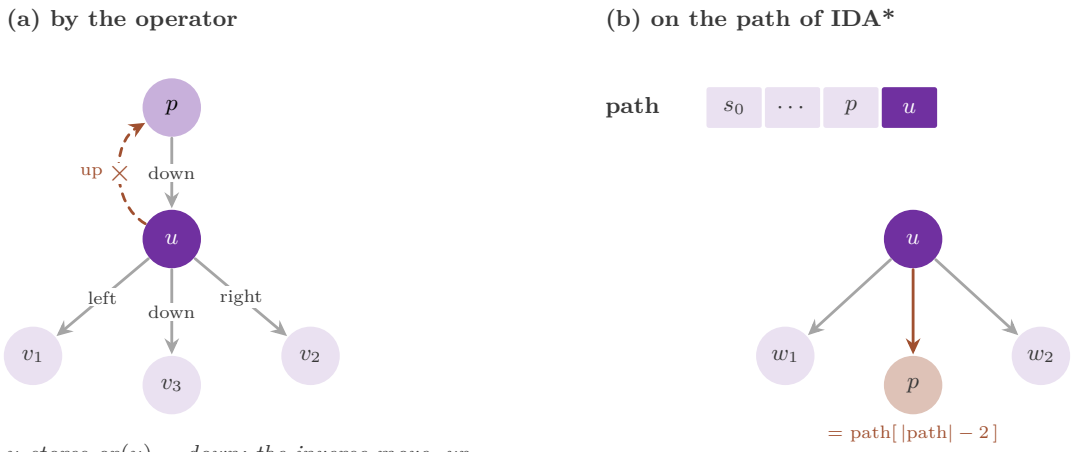


Figure 2.8. Parent pruning. (a) In an `OperatorDomain`, u records the operator that generated it (here the blank moved down), and its expansion does not generate the inverse move, so that p is never looked up. (b) Without operators, a search that keeps only the current path skips the successor equal to the state before u on the path.

2.10 The cost of a search and its counters

The `SearchStats` of a `Solution` (`Solution.h:73`) report what a search did. For `AStar` the counters are the following, written by the lines of Algorithm 2.3 given in parentheses.

- `expanded`, the expansions, the reexpansions included (line 7; an iteration that drops a stale entry is not an expansion);
- `reexpanded`, the expansions of nodes already expanded before (line 8, §2.7);
- `generated`, the successors generated, after the parent pruning (line 15);
- `duplicates`, the successors whose state was already in CLOSED with a g not larger, and which were dropped (line 24);
- `stored`, the distinct nodes stored at the end, i.e., the states generated and the root, which measure the memory;
- `seconds`, the wall-clock time of `solve()`, from its first line to its return (§2.16).

The other fields of `SearchStats` are for the parallel searches: `sent` and `messages` count the nodes and the batches of nodes sent to another worker, and `expanded_by` and `stored_by` hold the expansions and the nodes of each worker, from which `load_balance()` and `communication_overhead()` compute the load balance and the communication overhead (Chapter 8 and Chapter 9). For `AStar`, `sent` and `messages` are 0, and the two vectors have the single entries `expanded` and `stored`. We first prove that Algorithm 2.3, from which these counters are read, is Algorithm 2.2, and then the relations between them.

Proposition 2.17 (Algorithm 2.3 is Algorithm 2.2). Let both algorithms run with the same OPEN list and the domain enumerate the successors of each state in the same order. (i) With $L = 0$, Algorithm 2.3 pops the same entries, closes the same nodes in the same order and returns the same status, cost and path as Algorithm 2.2. (ii) With $L > 0$, it does the same until the first expansion after which $|\text{CLOSED}| \geq L$, and then returns `node_limit`; at that moment $L \leq |\text{CLOSED}| \leq \max(L-1, 1) + b$, with b the largest number of successors of a state, and a goal that is selected is always returned, since the test of line 10 precedes that of line 25.

Proof. (i) The counters and the hashes are written but never read by the search, and a prefetch changes the content of the cache, not that of CLOSED or of OPEN. Pass 1 calls only the domain, which reads neither list; hence K is the sequence of the successors that Algorithm 2.2 visits at its line 9, in the same order. During pass 2, $g(u)$ does not change: an improvement of u through one of its own successors would need $g(u) + c(u, u) < g(u)$,

i.e., a negative cost. Thus each iteration of lines 16 to 24 does, on the same CLOSED, what lines 10 to 16 of Algorithm 2.2 do for the same successor (a state that occurs twice in K is found at its second occurrence where the first left it, in both algorithms). Lines 19 and 22 also store the operator, which only CHILDREN reads, and it reads it to skip a successor that Algorithm 2.2 would treat as a duplicate (Proposition 2.16). By induction on the iterations, the two algorithms go through the same states of CLOSED and OPEN. (ii) The test of line 25 is made only after an expansion, and before it the two algorithms agree by (i). When the search stops, either the last expansion was the first one, made with $|\text{CLOSED}| = 1$, or the test failed after the expansion before it, with $|\text{CLOSED}| \leq L - 1$, the iterations that drop a stale entry adding no node; and an expansion adds at most b nodes. □

Note that a search that stops at the limit may have found no goal only because it stopped, and also that a search whose OPEN would have become empty at the same iteration returns `node_limit` rather than `unsolvable`.

Proposition 2.18 (The counters of A*). Let I be the number of the improvements of g at line 22 of Algorithm 2.3. When `solve()` returns, (i) `generated` = (`stored` -1) $+I+$ `duplicates`; (ii) the entries pushed in OPEN are `stored` $+I$, at most I of them are dropped as stale, and the iterations of the loop at line 4 are at most `stored` $+I$; (iii) `expanded` $-$ `reexpanded` is the number of the distinct states expanded, which is at most `stored`; (iv) if h is consistent, `reexpanded` $= 0$ and the I improvements are all of nodes in OPEN.

Proof. (i) Each element of K goes through exactly one of the branches of lines 18, 21 and 24; the first adds a node to CLOSED, which starts with s_0 and never loses a node, the second is an improvement and the third a duplicate. (ii) The entries are pushed by lines 3, 20 and 23, i.e., $1 + (\texttt{stored} - 1) + I$ times. Let v be a node improved I_v times, so that it has $1 + I_v$ entries, pushed with strictly decreasing values of g (line 21); call *latest* the entry pushed last at a given time, the only one whose g equals $g(v)$. The flag *closed* of v is set only by the pop of its latest entry (line 7, after the test of line 6), and it is cleared by each later push (line 22); hence the latest entry is never popped while v is closed, and its pop is not stale. A stale pop of v is thus the pop of an entry that is no longer the latest, and there are at most I_v of these, since the last entry pushed is always the latest. Summing over the nodes, at most I entries are stale; each iteration pops one entry, and an entry is popped once. (iii) Line 7 counts each expansion and line 8 those of the nodes whose flag *expanded*, never cleared, was already set; their difference is the number of the nodes on which the flag is set, i.e., of the states expanded at least once, which are in CLOSED. (iv) By Theorem 2.13, no closed node is reopened, i.e., line 21 holds only for nodes that are not closed, and each state is expanded at most once. □

For instance, a search whose initial state is a goal returns at line 10 of its first iteration, with `expanded` $= 1$, `generated` $= 0$ and `stored` $= 1$, a path of one state and the cost 0 (the edge cases checked by `check_astar_edges`, `test/test.h:474`). From the counters one can derive the *effective branching factor* `generated / expanded`, the fraction of the successors that are duplicates and the rate of expansions per second; the number I of the improvements is not counted on its own, but it follows from (i).

The time of a best-first search is thus the number of its expansions times the time of an expansion, which is made of (i) the extraction of an entry from OPEN, (ii) the generation of the successors by the domain, (iii) the evaluation of their heuristic, (iv) the duplicate detection in CLOSED and (v) the insertions in CLOSED and OPEN. We state the cost of `AStar` with the costs of the data structures of Chapter 3 as parameters.

Proposition 2.19 (The cost of A*). Let X, Γ and N be the values of `expanded`, `generated` and `stored` at the end of a search, I the improvements, b the largest number of successors of a state and t_D a bound on the time that the domain takes to generate one successor, with its heuristic and its hash. Then Algorithm 2.3 takes time

$$O\big(X + \Gamma\, t_D\big) + \Gamma\, t_{\text{look}} + (N + I)\,(t_{\text{push}} + t_{\text{pop}}) + O(N),$$

with t_{look}, t_{push} and t_{pop} the times of a lookup or insertion of CLOSED and of a push and a pop of OPEN, and the last term the rebuilding of the path; it holds N nodes, at most $N + I$ entries of OPEN, the slots of CLOSED and b successors at a time. In particular: (i) with the heap of $d = 4$ children (Proposition 3.3) the term of OPEN is $O((N + I)\log(N + I))$; (ii) with the bucket queue, integer costs and a consistent h (Proposition 3.5, under its hypotheses) it is $O(N + I + f_{\max})$, with $f_{\max}$ the largest key pushed; (iii) with uniform hashing and the load factor of CLOSED at most 7/10 (Proposition 3.11) the expected value of t_{look} is $O(1)$, amortized over the doublings; (iv) if h is consistent and a goal is reachable, $I \leq \Gamma$, X is at most the number of the states u with $g^*(u) + h(u) \leq C^*$, and $N \leq 1 + \Gamma \leq 1 + b\,X$.

Proof. Each iteration of the loop pops one entry, and those that are not stale run lines 7 to 25 once. Lines 11 to 15 call the domain once per successor, i.e., Γ times overall, besides an $O(1)$ work per expansion; lines 16 to 24 make one lookup per successor and one push per new node or improvement; the pops are at most the pushes, which are $N + I$ by Proposition 2.18 (ii); and the path of line 10 has at most N states, since the pointers `pa`

`rent` form a tree on CLOSED (Lemma 2.10). The memory follows in the same way, the entries in OPEN at any time being at most those pushed. Then (i), (ii) and (iii) are the propositions quoted. For (iv), $I \le \Gamma$ by Proposition 2.18 (i); by Theorem 2.13 each state is expanded at most once, and by Theorem 2.14 (i) only if $g^*(u) + h(u) \le C^*$; finally, $N - 1 \le \Gamma \le b\,X$ by Proposition 2.18 (i). □

On the sliding-tile puzzles and on the grids, t_D is a few instructions (the Manhattan distance of a successor is that of its parent plus or minus one), and the time goes mostly to the data structures, i.e., to t_{look}, t_{push} and t_{pop}; with a costly heuristic, such as a pattern database whose tables do not fit in the cache, t_D may dominate instead (Chapter 5). Also, A* stores every state that it generates, i.e., $N \ge X$; whether a search runs out of memory before it runs out of time depends on the domain and on the machine, and it is not a property that the bounds above decide.

The uninformed searches of §2.2 are A* with $h = 0$, and their properties follow from the results of this chapter.

Proposition 2.20 (The uninformed searches). Let $h = 0$, so that Algorithm 2.2 is the algorithm of Dijkstra, and the costs be nonnegative. (i) If it returns a path, the path is optimal; it terminates on a finite graph, and on a graph of finite branching with costs at least $\varepsilon > 0$ and a reachable goal. (ii) In the latter case it expands each state at most once, only states with $g^*(u) \le C^*$, and at most $\sum_{i=0}^{D} b^i$ of them, $D = \lfloor C^*/\varepsilon \rfloor$; it thus generates and stores $O(b^{1+D})$ nodes. (iii) With unit costs, the key $g(u)$ of a node selected from a non-stale entry is the least number of edges of a path from s_0 to u, the search is a breadth-first search up to the order of the ties among the nodes of equal depth, and it generates and stores $O(b^{d+1})$ nodes, with $d = C^*$ the number of edges of an optimal path.

Proof. The heuristic $h = 0$ is consistent (Definition 2.2: $0 \le c(u, v) + 0$, and 0 at the goals), hence admissible by Lemma 2.4. (i) is Theorem 2.9 and Theorem 2.11. (ii) By Theorem 2.13 each state is expanded at most once, with $g(u) = g^*(u)$, and by Theorem 2.14 (i) only if $g^*(u) \le C^*$. By Lemma 2.10 the path of the tree of the pointers `parent` from s_0 to u costs at most $g(u) \le C^*$, and since each of its edges costs at least ε it has at most D edges. A node has at most b children in that tree, so that the nodes at depth i of the tree are at most b^i; the expanded ones are thus at most $\sum_{i \le D} b^i$, each generates at most b successors, and the stored nodes are at most one more than the successors generated (Proposition 2.18 (i)). (iii) With unit costs, $g^*(u)$ is the least number of edges of a path to u, which is the key of u when it is selected (by Theorem 2.13), and the keys selected are nondecreasing: the nodes are expanded by layers of depth, which is breadth-first search up to the order within a layer. The bound is (ii) with $\varepsilon = 1$ and $C^* = d$. □

2.11 The goal test at generation

Algorithm 2.2 tests the goal when a node is selected, which makes the first goal selected an optimal one; a goal generated earlier, at the expansion of its parent, waits in OPEN until no node of smaller f is left. The option `goal_at_generation` of `SearchOptions` (off by default) lets `AStar` keep the best goal generated so far and stop as soon as OPEN can no longer improve it, which saves the expansions of the nodes of $f = C^*$ that come before a goal in the order of the ties, and all of the last layer of f when the parent of an optimal goal has a smaller f (e.g., with the blind heuristic and positive costs, the case of the planning tasks of Chapter 16). Thus the search keeps the *incumbent* $\hat{u}$, the goal of least g among those that it has generated with $h = 0$, and precedes line 5 of Algorithm 2.2 with the line 4' of the Doxygen comment of `AStar.h`.

Algorithm 2.4 (The line 4' of A* with the goal test at generation; the stale entries at the top of OPEN are dropped before the test).

```
4' if û exists and no entry of OPEN has f < g(û) then
      return the path to û, of cost g(û)
```

We update the incumbent after each successor v that lines 11 to 16 put in OPEN, i.e., that is new or reached through a cheaper path: if $h(v) = 0$, v is a goal and $\hat{u}$ does not exist or has $g(\hat{u}) > g(v)$, then $\hat{u} \leftarrow v$, and the counter `goals_generated` of `SearchStats` counts these updates. Since a goal has $h = 0$ under an admissible heuristic, each goal generated is a candidate; with an inadmissible heuristic a goal of positive h is never the incumbent, and it is still found by line 8. This test is made only on a successor with $h = 0$, so that a domain whose heuristic is informative pays a single comparison per successor, and the goal test of line 8 is kept: a goal selected there also ends the search. Also, the incumbent is a node of CLOSED, and a later improvement of its g (the node reached by a cheaper path, lines 14 to 16) lowers $g(\hat{u})$ with it. With a weight w of the heuristic (Chapter 6), the keys of OPEN are $f' = g + wh$, and line 4' compares the least key with $g(\hat{u})$ on the scale of the

keys (the method `live` of `Weight`).

Theorem 2.21 (The goal test at generation keeps the guarantees). Let h be admissible and a goal be reachable. Then Algorithm 2.2 with line 4' returns a path of cost C^*; with the priority $f' = g + wh$, $w \geq 1$, it returns a path of cost at most wC^*.

Proof. If the search returns at line 8, the claim is Theorem 2.9 (and, with the weight, the bound of Exercise 2.7), since line 4' only adds a way of returning, and it changes neither OPEN nor CLOSED otherwise: the stale entries that it drops are those that line 6 would drop. Let the search return at line 4' with the incumbent $\hat{u}$. Since $g(\hat{u})$ is the cost of a path from s_0 to the goal $\hat{u}$ (Lemma 2.6), $g(\hat{u}) \geq C^*$, and it suffices to prove $g(\hat{u}) \leq C^*$. Let t^* be a goal with $g^*(t^*) = C^*$. At the start of the iteration t^* is not closed with $g(t^*) = C^*$, since closing it would have ended the search at line 8; by Lemma 2.7, an optimal path to t^* has a node u' whose entry $(g^*(u') + h(u'), g^*(u'), u')$ is in OPEN and is not stale. Its key is at most $g^*(u') + h^*(u') = C^*$, by admissibility, and line 4' returns only if no live entry has a key smaller than $g(\hat{u})$; hence $g(\hat{u}) \leq g^*(u') + h(u') \leq C^*$. With the priority f', the key of the entry of u' is $g^*(u') + wh(u') \leq w(g^*(u') + h^*(u')) = wC^*$, and the same argument gives $g(\hat{u}) \leq wC^*$. □

The proof needs the reopening of lines 14 to 16 through Lemma 2.7, as that of Theorem 2.9 does, and it does not need a consistent heuristic. Next, we state what the test saves, and that it never costs an expansion.

Proposition 2.22 (The expansions with the goal test at generation). Let both searches run on the same domain with the same OPEN list. (i) The sequence of the nodes expanded by Algorithm 2.2 with line 4' is a prefix of that of Algorithm 2.2 without it. (ii) If h is consistent, s_0 is not a goal, and some optimal path from s_0 to a goal t has $g^*(p) + h(p) < C^*$ at the predecessor p of t, the search with line 4' expands exactly the nodes u with $g^*(u) + h(u) < C^*$, each once.

Proof. (i) Line 4' reads OPEN and drops only stale entries, which the search without it drops at line 6 without expanding anything; the incumbent is written but not read by lines 5 to 16. Hence the two searches pop the same live entries in the same order until the first one returns, and the one with line 4' returns either at the same time (line 8) or earlier (line 4'). (ii) By Theorem 2.13, the keys of the selected entries are nondecreasing and each node is closed with its optimal g, and by Theorem 2.14 (ii), each node of $g^* + h < C^*$ is expanded; hence all of them are expanded before any node of key at least C^*, p among them, with $g(p) = g^*(p)$. The expansion of p generates t with $g = g^*(p) + c(p, t) = C^*$, and $h(t) = 0$; thus, when the first entry of key at least C^* reaches the top of OPEN, the incumbent exists and has $g(\hat{u}) = C^*$ (it cannot be smaller, by Theorem 2.21), and line 4' returns. Each node expanded before has key, i.e., $g^* + h$, smaller than C^*, and is expanded once by Theorem 2.13. □

Without line 4', A* also expands the nodes of $f = C^*$ that precede the first goal in the order of the ties, i.e., between none and all of them (§2.8); with the blind heuristic and unit costs these are the states at distance C^* from s_0, the last and usually the largest layer of the breadth-first search. We remark that the hypothesis on p of (ii) fails when the optimal goals are reached only from nodes of $f = C^*$, e.g., with the perfect heuristic $h = h^*$, where every node of an optimal path has $f = C^*$; then line 4' saves only the ties that come after the incumbent. On the planning tasks of the library, with the blind heuristic, the test `planning` checks that the search with the test at generation expands exactly the states closer to s_0 than the optimal cost, that it counts at least one goal in `goals_generated`, and that it finds the cost of the search without it, which counts none and expands no fewer states. Of course, the greedy searches of §2.12 read the same option, with another meaning: with eager evaluation they return a goal as soon as it is generated, and the bound of Theorem 2.24 is then lost.

2.12 Greedy best-first search

The searches of the rest of the chapter give up the optimality of the path for speed, as the planners do when any plan will serve (the *satisficing* planners, cf. Chapter 16). Greedy best-first search (GBFS) is the best-first search of Algorithm 2.1 with the priority h: it expands the node that the heuristic places closest to a goal, whatever its g. It goes back to the Graph Traverser of Doran and Michie [96], and it is the limit $w \to \infty$ of the order by $g + wh$ of weighted A* [46], which the same class runs with a finite weight. In `GreedySearch.h`, the class template `GreedySearch` implements GBFS with the OPEN lists of the satisficing planners of the last two decades as options that compose, i.e., the deferred evaluation of the heuristic, several queues alternated over several heuristics, the queues of the preferred successors and their boosting, the type-based exploration, the epsilon-greedy selection, the Pareto queues and the breaking of the ties.

The interface

The kinds of evaluation, of breaking of the ties and of queue, and the description of a queue, are the following declarations of `GreedySearch.h`.

```
enum class GreedyEvaluation {
 eager ,     ///< when it is generated, the key being its own h
 lazy        ///< when it is selected, the key being the h of its parent
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// how the ties of the key of a queue are broken

enum class TieBreak {
 fifo ,      ///< the first pushed
 lifo ,      ///< the last pushed
 low_g ,     ///< the smaller g, then the first pushed
 high_g ,    ///< the larger g, then the first pushed
 high_g_lifo ,  ///< the larger g, then the last pushed
 random      ///< at random
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// the kind of a queue of OPEN

enum class QueueKind {
 best ,      ///< the least key, the ties broken by the next heuristic of
             ///< the queue and then by the TieBreak
 type ,      ///< a type at random, then a node of that type at random
 epsilon ,   ///< a node at random with probability epsilon, else the best
 pareto      ///< a vector at random among those not dominated
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// a queue of OPEN: its kind, the heuristics it looks at, and whether it
/// holds only the preferred successors
/** A queue of kind best orders by the first heuristic in heuristics, the
 * ties broken by the second, if any; one of kind type takes the type of a
 * node from the first heuristic (and from g, if
 * GreedyOptions::type_with_g); one of kind epsilon takes the best node by
 * the first heuristic; one of kind pareto looks at the vector of all of
 * them. */

struct QueueSpec {

 QueueKind kind = QueueKind::best;           ///< the kind of the queue
 std::vector< std::size_t > heuristics = { 0 };  ///< the heuristics [see
                                             ///< MultiHeuristicDomain]
 bool preferred_only = false;                ///< only the preferred
                                             ///< successors [see
                                             ///< PreferredDomain]

 };  // end( struct( QueueSpec ) )
```

A queue of kind `best` orders by the first heuristic of its list `heuristics`, the ties broken by the second, if there is one, and then by the `TieBreak`. A queue of kind `type` takes its types from the first heuristic (and from g), one of kind `epsilon` takes the best node by the first heuristic or a random one, and one of kind `pareto` looks at the vector of all its heuristics. With `preferred_only`, a queue receives only the preferred successors of a `PreferredDomain` (§1.5). The heuristics are indices in $[0, m)$, where m is `heuristics_of( d )`, i.e., the number of heuristics of a `MultiHeuristicDomain`, and 1 for any other domain. The options are

```
struct GreedyOptions : public SearchOptions {
 GreedyOptions( void );          // weight = +infinity, i.e., GBFS
 GreedyEvaluation evaluation = GreedyEvaluation::eager;
```

```
 std::vector< QueueSpec > queues = { QueueSpec{} };
 TieBreak tie = TieBreak::fifo;
 int boost = 0;
 double epsilon = 0.2;
 bool type_with_g = true;
 bool reopen = false;
 bool improve_open = true;
 double bound = std::numeric_limits< double >::infinity();
 bool prune_by_f = false;
 bool preferred_first = false;
 bool randomize_successors = false;
 std::uint64_t seed = 1;
 };
```

i.e., those of `SearchOptions` and the following ones, with their defaults in parentheses: the evaluation (eager), the list of the queues (one queue of kind `best` on h_0), the breaking of the ties (FIFO), the boost of the queues of the preferred successors (0, none), the probability ϵ of the queues of kind `epsilon` (0.2), and whether a type includes g (`type_with_g`, true). Then come whether a closed node reached through a cheaper path is reopened (`reopen`, false) and whether a node in OPEN so reached takes the cheaper path (`improve_open`, true); with `improve_open` false a node keeps the first path found, as in the parallel greedy searches of Chapter 12. Last, `bound` prunes the successors with $g \geq$ `bound` (or $g + h \geq$ `bound` with `prune_by_f`), and the iterated searches of §2.14 set it; `preferred_first` puts the preferred successors in OPEN before the others, `randomize_successors` puts the successors in OPEN in a random order, and `seed` seeds the random choices. The constructor of `GreedyOptions` sets the `weight` of `SearchOptions` to $+\infty$, which is GBFS. A finite weight $w \geq 0$ makes the key of a node $g + wh$, and a negative one is refused. Of the other options of `SearchOptions`, `goal_at_generation` makes the search with eager evaluation return a goal as soon as it is generated, while `node_limit` and `expected_nodes` are those of `AStar`. The options are quoted by hand, since their constructor has a body, and so are the search and its function, which in the header come with their bodies.

```
template< Domain D >
class GreedySearch {
 public:
 explicit GreedySearch( const D & domain , const GreedyOptions & options = {} );
 void set_cache( HeuristicCache< D > * cache ) noexcept;
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
gbfs( const D & domain , const GreedyOptions & options = {} );
```

The constructor throws `std::invalid_argument` if the options name no queue, a queue with no heuristic, a heuristic out of $[0, m)$, or a negative weight. By `set_cache` the search is given a `HeuristicCache` (described below), which must outlive it, and `gbfs( d , o )` is `GreedySearch< D >( d , o ).solve()`. A node is a `GreedyNode`, i.e., the `Node` of Chapter 3 with the index of the node in place of the key of the owner. The first heuristic is the field `h`, and the index finds the others in an array of the search. OPEN is a `GreedyOpen< P >`, whose interface is

```
template< class P >
class GreedyOpen {
 public:
 GreedyOpen( const GreedyOptions & o , std::size_t m );
 template< class Cost >
 void push( const P & p , const Cost * hv , Cost g , bool preferred );
 bool pop( P & p );
 void boost( void );
 [[nodiscard]] bool empty( void ) const;
 [[nodiscard]] std::size_t size( void ) const;
 };
```

where `push` inserts the payload `p` into every queue that takes it. The payload is a node of the eager search or the

index of a record of the lazy one, and it comes with the m values `hv` of the heuristics that make its keys, its g and its preference. Then `pop` removes the first entry of the nonempty queue of least priority (and returns false if all are empty), `boost` lowers the priorities of the queues of the preferred successors, and `size` counts the entries of all the queues, the stale ones included. The keys are kept as `double`, which is exact for the integers below 2^{53}.

The algorithm

We give the pseudocode of the Doxygen comment of `GreedySearch.h`, with $Q_1, \ldots, Q_q$ the queues of OPEN, each with a priority π_j (initially 0), $h_0, \ldots, h_{m-1}$ the heuristics, $h = h_0$, and $best_i$ the least value of h_i seen so far.

Algorithm 2.5 (GBFS with eager evaluation, and its lazy variant).

```
 1  procedure GBFS(s₀)
 2     CLOSED ← {s₀}, with g(s₀) ← 0, parent(s₀) ← nil, and the values
         h_i(s₀) of the heuristics;  best_i ← h_i(s₀) for each i
 3     push s₀ in each Q_j that is not for the preferred successors only
 4     loop
 5        j ← a nonempty queue of least π_j;  π_j ← π_j + 1
 6        if there is none then return unsolvable
 7        u ← the first node of Q_j, removed
 8        if u is closed or its g changed since it was pushed then
             continue
 9        close u;  if u is a goal then return the path to u
10        P ← the preferred successors of u
11        for each successor v of u do
12           g' ← g(u) + c(u, v);  if g' ≥ bound then continue
13           if v ∉ CLOSED then
14              add v to CLOSED, with g(v) ← g', parent(v) ← u and the
                  values h_i(v);  if h(v) = ∞ then continue  ▷ dead end
15              if h_i(v) < best_i for some i then
16                 best_i ← h_i(v);  π_j ← π_j − boost for each queue of
                     preferred successors
17              push v in each Q_j, those of the preferred successors
                  only if v ∈ P
18           else if g' < g(v) and (v is not closed or reopening) then
19              g(v) ← g';  parent(v) ← u;  v is not closed;  push v
                  again as at line 17
20     ▷ with lazy evaluation, the entries are the pairs (u, v) pushed
         with the keys of u, and v is looked up, evaluated and expanded
         when its entry is selected, if it is not in CLOSED
```

Lines 5 and 6 are the method `pop` of `GreedyOpen`, which in fact first looks for a nonempty queue and then raises its priority. Line 8 drops the stale entries, as line 6 of Algorithm 2.2 does; an entry may lie in several queues, and the copies that surface after its node has been expanded are stale. In a queue of kind `best` or `epsilon`, the key of a node is h with the infinite weight and $g + wh$ with a finite one. "Reopening" at line 18 is the option `reopen` for a closed v, and `improve_open` for a v in OPEN. Three details of the code complete the pseudocode: with `prune_by_f`, line 12 tests $g' + h(v) \geq$ `bound` instead; a dead end, i.e., a state with $h(v) = $ `infinite_cost()`, is closed at once and never expanded, nor improved at line 18; and line 15 does not boost when $best_i$ was still ∞, so that the evaluation of s_0 boosts nothing. With `goal_at_generation` and eager evaluation, the goal test moves from line 9 to the generation of a new state, after line 14 (and to s_0, before line 3).

With *lazy* (deferred) evaluation [97], the expansion of u does not evaluate its successors. For each successor v that the bound does not prune by g, it pushes a *record* (u, v, g'), whose keys are the values of the heuristics of u, its parent. When a record is selected, v is looked up. A new v is added to CLOSED, evaluated (the pruning by $g + h$ is made here) and expanded at once, its goal test included; a known v is expanded again only if $g' < g(v)$ and `reopen` holds, and it is otherwise a duplicate. Thus each node of CLOSED has been expanded with its current g, unless it is a dead end or a state pruned by $g + h$. The heuristic is computed once per new state selected instead of once per state generated, and the price is a key that lags by one move, together with the records of the states already expanded, which are dropped only when they surface. On a `DeferredDomain` the successors come from `for_each_successor_state`, without their h. On any other domain they come from `for_each_successor`, and the record keeps the h that it passes: the domain pays the evaluation anyway, and the search only avoids that of the heuristics beyond the first. The progress of line 15, and the boost, are tested when

a state is expanded.

The queues of OPEN

We alternate the queues as Röger and Helmert [98] do: `pop` takes the nonempty queue of least priority, the first one among the ties, and raises its priority by 1, so that each queue counts the entries taken from it. Hence, while every queue is nonempty and there is no boost, the queues are taken in turn. A queue that stays empty for a while keeps its low priority, and when it receives entries again it is taken many times in a row, until its count catches up. One queue may order by a heuristic and break its ties by another (the second index of its list), which is the lexicographic combination of two heuristics, while two queues on two heuristics alternate them.

The queues with `preferred_only` hold only the successors that the `PreferredDomain` marks as preferred, by the predicate `preferences( u )` computed once per expansion of u; their entries are therefore also in the other queues. When some heuristic h_i reaches a value smaller than all those seen so far (line 15), `boost` lowers the priorities of these queues by `boost`, and they are then taken about `boost` times in a row. This is the boosting of the preferred operators of Richter and Helmert [97], and `boost` $= 1000$ is the value of LAMA [102]. The root is pushed only in the queues that are not of the preferred successors (line 3). Hence an eager search whose queues are all of the preferred successors expands nothing, while a lazy one, which expands s_0 before it pushes anything, follows the preferred successors only.

A queue of kind `type` implements the type-based exploration of Xie et al. [99]. The *type* of a node is the pair (h, g) of its first heuristic and its g, or h alone without `type_with_g`, and the queue keeps a bucket for each type; `pop` draws a nonempty bucket uniformly at random, and then a node of it uniformly at random. We hash a type to 64 bits by the mixer of SplitMix64 (§1.5) applied to the bits of h and of g. Two types whose hashes collide therefore share a bucket, which changes the probabilities of the draw and nothing else. Such a queue is meant to be alternated with a queue of kind `best`, as in the test `gbfs-queues`, and its types are made of the values of the heuristic and of g, not of the key $g + wh$.

A queue of kind `epsilon` is the epsilon-greedy selection of Valenzano et al. [100]. With probability ϵ, `pop` takes an entry uniformly at random among those of the queue, and otherwise the entry of least key, as a queue of kind `best` does; with $\epsilon = 0$ it is a queue of kind `best`, and with $\epsilon = 1$ a random walk on OPEN. A queue of kind `pareto` is the Pareto queue of Röger and Helmert [98], which groups its entries by the vector $(h_{i_1}, \ldots, h_{i_r})$ of the values of its heuristics. Its `pop` draws uniformly at random a vector that no other vector of the queue *dominates* (i.e., none is at most as large in each component and smaller in one), and then takes the first entry of that vector (the last with the tie `lifo`). We compute the set of the undominated vectors again, by comparing each pair of vectors, whenever a vector appears in the queue or leaves it. This costs $O(V^2 r)$ for V distinct vectors, which is acceptable when the heuristics take few values, as in planning, and costly otherwise.

Last, the `TieBreak` applies to the ties of the keys in the queues of kind `best` and `epsilon`. With `fifo` and `lifo` the order of insertion decides, with `low_g` and `high_g` the value of g and then the order, with `high_g_lifo` the larger g and then the last pushed, and with `random` a random word of 64 bits drawn at the insertion. A queue of kind `best` is an implicit d-ary heap of the default data structures (Chapter 3), which compares its entries by the key, the second heuristic, g (with `low_g` or `high_g`) and the order. All the random choices are drawn from one generator seeded by `seed`, and a search is therefore reproducible, as the test `gbfs-queues` checks for each kind.

The cache of the heuristic values

A search that restarts, as the iterated searches of §2.14 do, evaluates again many of the states of the searches before it. The class template `HeuristicCache< D >` keeps the m values of the heuristics of each state that a search evaluated, in a table of the states of the kind of CLOSED. A search that is given the cache by `set_cache` thus evaluates a state once over all the searches that share it, which is what makes the restarts of Richter, Thayer and Ruml cheap [103]. Its interface is

```
template< Domain D >
class HeuristicCache {
 public:
 explicit HeuristicCache( const D & domain );
 cost_t< D > * find_or_insert( const state_t< D > & s , std::uint64_t hash ,
                               bool & fresh );
 [[nodiscard]] std::uint64_t hits( void ) const noexcept;
 [[nodiscard]] std::uint64_t misses( void ) const noexcept;
 };
```

where `find_or_insert` returns the slot of the m values of s and tells by `fresh` whether s was not there, in which case the search evaluates s and fills the slot; `hits` and `misses` count the lookups of the two kinds. The counter `evaluated` of a greedy search counts the states whose heuristics it computed, those found in the cache excluded. Without a `DeferredDomain`, the lazy search counts only the states that it selects, although the domain computes h for all the successors that it generates.

Analysis

We now prove what the code guarantees. The searches are complete on a finite graph under three conditions on the options and one on the heuristic, whatever the queues, the ties and the weight, and the weighted search keeps the bound of weighted A* with eager evaluation and a weaker one with lazy evaluation. We assume, as in the rest of the chapter, nonnegative costs and a nonnegative heuristic.

Proposition 2.23 (Termination and completeness of `GreedySearch`). On a finite graph, (i) `solve()` terminates, with either evaluation and any options. (ii) If at least one queue is not of the preferred successors only, the bound is $+\infty$, there is no node limit, and $h_0(u) = \infty$ only for states u from which no goal is reachable, then the search returns a path whenever a goal is reachable from s_0.

Proof. (i) With eager evaluation, an entry is pushed when a node is created, at most once per state, or when its g decreases (line 19); the successive values of the g of a node are costs of distinct paths, of which only finitely many are smaller than the first value, as in the proof of Theorem 2.11 (i). Hence the entries pushed, each in at most q queues, are finitely many, each iteration removes one, and the loop ends at the latest when the queues are empty. With lazy evaluation, the records are pushed by the expansions, at most b each, and a state is expanded when it enters CLOSED or when its g decreases, i.e., finitely many times for the same reason. (ii) We first show that each node of CLOSED that is not a dead end is expanded before the queues become empty. With eager evaluation, consider the entry of a node v pushed last, whose g is the current $g(v)$; it lies in a queue that is not of the preferred successors only, since all the entries enter those queues, and it is removed from it before the queues are empty. At its removal either v is not closed, and it is expanded (line 9), or it is closed, and it was then expanded with its current g, since only the expansion closes a node. With lazy evaluation each node of CLOSED but the dead ends was expanded when it entered CLOSED. Now let $s_0 = u_0, u_1, \ldots, u_k$ be a path to a goal, and suppose that the search ends with the queues empty. No u_j is a dead end, by the hypothesis on h_0, and none is pruned. The root is expanded; if u_j is expanded, its successor u_{j+1} enters CLOSED (eager) or gets a record that is selected before the queues are empty and that puts it in CLOSED (lazy), and it is therefore expanded. Hence u_k is expanded, and the search returns at its goal test (or earlier, at another goal, or at the generation of a goal with `goal_at_generation`), a contradiction. □

The conditions of (ii) are necessary: a search with the queues of the preferred successors only drops the other successors for good, a finite bound prunes the paths that reach it, a node limit stops the search, and a heuristic that declares a dead end where a goal is reachable removes that state. On an infinite graph GBFS need not terminate even when a goal is reachable, since it may follow forever a path along which h does not increase (an exercise), which A* avoids by the growth of g (Theorem 2.11 (ii)).

Theorem 2.24 (The bound of the weighted search). Let h_0 be admissible, a goal be reachable, and the search run with eager evaluation, a finite weight $w \geq 0$, the single queue of kind `best` on h_0 of the default options (with any `TieBreak`), `reopen` and `improve_open`, no `goal_at_generation`, and the bound $+\infty$. Then the path returned costs at most $\max\{1, w\}\, C^*$; in particular it is optimal for $w \leq 1$.

Proof. With these options Algorithm 2.5 is Algorithm 2.2 with the priority $g + wh_0$, the ties broken by the `TieBreak` instead of the larger g, and the dead ends never pushed: line 18 reopens a node, closed or not, whenever its g decreases, as lines 14 to 16 of Algorithm 2.2 do. A dead end has $h_0 = \infty$, which by admissibility means that no goal is reachable from it, and it lies on no path to a goal. Hence the proof of Lemma 2.7 applies unchanged, the order of the ties playing no part in it: at the start of each iteration before the return, an optimal path to a goal has a node u' with a live entry of $g = g^*(u')$, whose key is $g^*(u') + wh_0(u')$. If $w \geq 1$, this is at most $w(g^*(u') + h^*(u')) = wC^*$; if $w \leq 1$, it is at most $g^*(u') + h^*(u') = C^*$. The goal t returned at line 9 was selected from a live entry of least key, whose key is $g(t) + wh_0(t) = g(t)$, since $0 \leq h_0(t) \leq h^*(t) = 0$; hence $g(t) \leq \max\{1, w\}\, C^*$, and the cost of the path that the parents give is at most $g(t)$, by the inequality of Lemma 2.10. □

The bound is that of Pohl [46]. The test `gbfs` checks, on 200 states of the 8-puzzle, that $w = 1$ returns the optimal cost and $w = 2$ a cost at most $2C^*$, and, on instances of the 15-puzzle of Korf, the bound with $w = 3$.

Without `improve_open` a node of OPEN keeps its first path and the bound is lost, and with `goal_at_generation` the first goal generated is returned, whatever its g. With lazy evaluation the key of a record is the g of the successor with the h of its parent, and the bound holds only up to the cost of one edge.

Proposition 2.25 (The bound of the lazy search). Under the hypotheses of Theorem 2.24 but with lazy evaluation, the path returned costs at most $\max\{1, w\}\, C^* + \bar{c}$, where $\bar{c}$ is the largest cost of an edge of an optimal path. With $w = 1$ the search may return a path that is not optimal.

Proof. Let $P = (s_0 = u_0, \dots, u_k)$ be an optimal path. Before the search returns, let $j+1$ be the least index such that u_{j+1} has not been expanded with $g = g^*(u_{j+1})$; it exists, since the goal u_k expanded would have ended the search, and $j + 1 \geq 1$, since the root is expanded first with $g = 0$. When u_j was expanded with $g^*(u_j)$, it pushed the record $R = (u_j, u_{j+1}, g^*(u_{j+1}))$, with the key $g^*(u_{j+1}) + wh_0(u_j)$. If R had been selected, u_{j+1} would have been expanded with $g^*(u_{j+1})$: as a new state, since it is not a dead end and the bound prunes nothing; as a known state whose g is larger, by `reopen`; and a known state whose g is not larger has $g = g^*(u_{j+1})$ and was expanded with it, since a node of CLOSED takes its g only when it is expanded. Hence R is still in the queue, and the record of the goal t, selected with the least key, has the key $g(t) + wh_0(p) \geq g(t)$, p its parent, at most that of R, which is at most $g^*(u_j) + c(u_j, u_{j+1}) + wh^*(u_j)$ by admissibility. For $w \geq 1$ this is $wC^* + c(u_j, u_{j+1}) - (w-1)g^*(u_j) \leq wC^* + \bar{c}$, since $g^*(u_j) + h^*(u_j) = C^*$; for $w < 1$ it is at most $C^* + c(u_j, u_{j+1})$. For the second claim, let s_0 have the successors a, at cost 1, and the goal t, at cost 10, and let a reach t at cost 5, with $h(s_0) = 0$, $h(a) = 5$ and $h(t) = 0$, which is admissible (and consistent), so that $C^* = 6$. With $w = 1$, the expansion of s_0 pushes (s_0, a) with the key 1 and (s_0, t) with the key 10; the expansion of a pushes (a, t) with the key $6 + 5 = 11$, and the record (s_0, t), of key 10, is selected first and returns the path of cost 10, within the bound $6 + 5 = 11$. □

GBFS itself bounds nothing, but which states it may expand is characterized by the analysis of Heusner, Keller and Helmert [112], which the parallel greedy searches of Chapter 12 build upon. The *high-water mark* of a state s is

$$\mathrm{hwm}(s) = \min_{p} \max_{s' \in p} h(s'),$$

the minimum being over the paths p from s to a goal ($+\infty$ if there is none), i.e., the largest h that a search must accept on its way from s to a goal; a state s is a *progress state* if $\mathrm{hwm}(s) > \min_t \mathrm{hwm}(t)$ over its successors t (and so is a goal). The *bench* of a progress state s has the level $\ell = \min_t \mathrm{hwm}(t)$, the inner states reachable from s through states that are not progress states and have $h \leq \ell$, and the exits, the progress states of $h = \ell$ among the successors of s and of its inner states. The *bench transition system* (BTS) is the set of the benches reached from that of s_0 through their exits that are not goals, and Theorem 1 of [112] states that a state is expanded by GBFS with some breaking of the ties if and only if it lies in a bench of the BTS; we do not reproduce its proof. The two facts below are the part of this analysis that the library uses and that we prove.

Lemma 2.26 (The recurrence of the high-water mark). For a goal s, $\mathrm{hwm}(s) = h(s)$; for any other state, $\mathrm{hwm}(s) = \max\{h(s), \min_t \mathrm{hwm}(t)\}$, the minimum being over the successors t of s. Hence, on a finite graph, the high-water marks are the values of a shortest-path problem in which a path costs the largest h along it, solved by the algorithm of Dijkstra run backwards from the goals with max in place of $+$.

Proof. A path from a goal s may stop at s, and its value is then $h(s)$, while any other path from s contains s and has a value of at least $h(s)$. A path from a state s that is not a goal goes through some successor t, and its value is the larger of $h(s)$ and of the value of its suffix from t, whose least value is $\mathrm{hwm}(t)$. The operation max is monotone and $\max\{a, x\} \geq x$, which are the two properties of $+$ with nonnegative costs on which the correctness of the algorithm of Dijkstra rests: a state is settled at its final value when it is the least among those not yet settled, since any later candidate is at least as large. □

Proposition 2.27 (GBFS stays below the high-water mark). Let the search be GBFS, i.e., run with the infinite weight, eager evaluation and the single queue of kind `best` on $h_0 = h$, with any `TieBreak`, any `reopen` and `improve_open`, the bound $+\infty$, and $h = \infty$ only on dead ends. If a goal is reachable, each state expanded before the search returns has $h \leq \mathrm{hwm}(s_0)$.

Proof. Let $P = (s_0 = u_0, \dots, u_k)$ be a path to a goal with $\max_j h(u_j) = \mathrm{hwm}(s_0)$. Before the search returns, the goal u_k has not been expanded; let u_j be the first state of P never expanded. If $j = 0$, the entry of s_0 is in the queue, since its g stays 0. Otherwise u_{j-1} was expanded and generated u_j, which entered CLOSED and, not being a dead end, the queue; the entry of u_j pushed last is not stale, since u_j has never been closed, and it has not been removed, since its removal would have expanded u_j. Thus the queue holds a live entry of key $h(u_j) \leq \mathrm{hwm}(s_0)$, and the live entry selected, whose key is the h of its node, has a key not larger. □

The test `pgbfs-rounds` computes the high-water marks of random graphs by the algorithm of Lemma 2.26,

builds the BTS from them, and checks on 60 random graphs that GBFS with the FIFO and the LIFO breaking of the ties (run as the parallel greedy search with one worker, which the same test shows to be GBFS) expands only states of the BTS, before using the BTS on the parallel searches.

2.13 Enforced hill-climbing

Enforced hill-climbing (EHC) is the local search of the planner FF of Hoffmann and Nebel [101]. From the current state s, a breadth-first search looks for a state of heuristic value smaller than $h(s)$, which becomes the current state, and the path to it is appended to the plan. The search ends at a goal, and it fails if a breadth-first search exhausts the states reachable from s without finding a better one. Its options, in `EnforcedHillClimbing.h`, are

```
enum class PreferredUsage {
 ignore ,     ///< as the others
 prune ,      ///< only the preferred successors are generated
 first        ///< the preferred successors are visited first
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// the options of EnforcedHillClimbing

struct EHCOptions : public SearchOptions {

 /// the use of the preferred successors [see PreferredDomain]
 PreferredUsage preferred = PreferredUsage::ignore;

 /// if the search fails, a GreedySearch from the initial state with the
 /// options greedy, as FF does
 bool fallback = false;

 /// the options of the search of the fallback
 GreedyOptions greedy;

 };  // end( struct( EHCOptions ) )
```

i.e., the use of the preferred successors of a `PreferredDomain` (the helpful actions of FF), and a fallback. The preferred successors are treated as the others with `ignore`, generated alone with `prune`, and visited before the others with `first`. If the search fails and `fallback` holds, a `GreedySearch` from s_0 runs with the options `greedy`, whose node limit and expected nodes are replaced by those of the `EHCOptions`; FF does the same when its search with the helpful actions fails. The search and its function are declared as

```
template< Domain D >
class EnforcedHillClimbing {
 public:
 explicit EnforcedHillClimbing( const D & domain ,
                                const EHCOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
ehc( const D & domain , const EHCOptions & options = {} );
```

and the pseudocode of the Doxygen comment of the header is the following.

Algorithm 2.6 (Enforced hill-climbing).

```
1  procedure EHC(s0)
2     s ← s0;  plan ← ⟨s0⟩
3     while s is not a goal do
4        Q ← ⟨s⟩;  visited ← {s}
```

```
 5        loop
 6           if Q = ∅ then return failure      ▷ or a complete search
 7           u ← the first state of Q, removed
 8           for each successor v ∉ visited of u (the preferred ones
               only, or first, if so asked) do
 9              visited ← visited ∪ {v};  parent(v) ← u
10              if v is a goal or h(v) < h(s) then
11                 append the path from s to v to plan;  s ← v
12                 continue at line 3
13              append v to Q
14     return plan
```

Each iteration of the loop at line 3 is a *phase*. We keep the values of h over the whole search, in a table of the states, so that a state is evaluated once even if several phases meet it; a phase marks the states that it visits by its number, instead of clearing a set. A successor of $h = $ `infinite_cost()` is a dead end, marked as visited but not appended to Q. In the `Solution`, the path is the plan, i.e., the concatenation of the paths of the phases, and its cost is the sum of their costs, each of which is the g of the better state in the breadth-first search of its phase. The counters are summed over the phases (and the fallback, if it runs). Thus `evaluated` counts each state once and `expanded` a state once per phase that expands it, while the node limit bounds the states evaluated, i.e., those of the table; a search stopped by it returns `node_limit` and runs no fallback.

Proposition 2.28 (Termination and completeness of EHC). On a finite graph, (i) `solve()` terminates, after at most as many phases as the distinct values of h on the states reachable from s_0. (ii) If `preferred` is `ignore`, a goal is reachable from each state reachable from s_0, and h is finite on these states, it returns a plan. (iii) Otherwise it may fail without the fallback; with the fallback, and without a node limit, it is as complete as `GreedySearch` with the options `greedy` (Proposition 2.23). The plan is a path from s_0 to a goal, which may visit a state more than once, and its cost is not bounded in terms of C^*.

Proof. (i) A phase is a breadth-first search in which each state is appended to Q at most once, and it ends on a finite graph; a phase that does not fail moves s to a goal, which ends the search, or to a state of strictly smaller h, and the values of h of the successive current states are therefore distinct. (ii) Let s be the current state of a phase. A goal t is reachable from s by a path $s = v_0, \dots, v_r = t$, $r \geq 1$; without the pruning of the preferred successors and of the dead ends, which do not exist here, the breadth-first search from s visits all the states reachable from s unless it stops before, and in particular it visits t, at which line 10 stops it. Hence no phase fails, and the search ends at a goal by (i). (iii) With `prune`, a goal may be reachable only through a successor that is not preferred, and a phase fails if all the states that it reaches from s through preferred successors have $h \geq h(s)$. The plan is a path since each phase appends a path of the breadth-first tree from s; a state of a phase, of $h \geq h(s)$, may be visited again by a later phase, whose current state has a smaller h. Finally, a phase stops at the first better state in the order of the breadth-first search, i.e., by the number of edges, whatever the cost of the path to it and of the rest of the plan. □

The test `ehc` checks, with each use of the preferred successors and the fallback, that the search solves 200 states of the 8-puzzle and two instances of Korf, that it crosses a plateau of $h = 5$ on a graph of four states, and that it fails on a graph where the only successor of s_0 has a larger h and leads to no goal.

2.14 Iterated searches

A satisficing planner with time to spare does not stop at its first plan. LAMA [102] and the other planners of its kind run a sequence of greedy and weighted searches, each bounded by the cost of the best plan that those before it found, so that each new plan is cheaper. Restarting Weighted A* of Richter, Thayer and Ruml [103] restarts in the same way with smaller and smaller weights; its authors report that the restarts, which forget OPEN and CLOSED but keep the values of the heuristic, often find the better plans sooner than a search that keeps them. In `IteratedSearch.h`, the class template `IteratedSearch` runs such a sequence of `GreedySearch`es, its *phases*, with the options

```
struct IteratedOptions {

 /// the phases, in order; the node limit of each is its own
 std::vector< GreedyOptions > phases;
```

```
/// each phase is bounded by the cost of the incumbent
bool pass_bound = true;

/// the bound prunes by \f$ g + h \f$ (for an admissible h) instead of g
bool prune_by_f = false;

/// the last phase is repeated as long as it finds a cheaper path
bool repeat_last = false;

/// a phase that finds no path does not end the search, once a path is
/// known (before, the search always goes on)
bool continue_on_fail = false;

/// the phases share the values of the heuristics [see HeuristicCache]
bool share_cache = true;

/// the most phases run, the repetitions included (0: no limit)
std::size_t max_phases = 0;

};  // end( struct( IteratedOptions ) )
```

i.e., the `GreedyOptions` of each phase, in order, with the node limit of each phase holding for that phase only. With `pass_bound`, each phase is bounded by the cost c of the incumbent through its `bound`, which prunes the successors with $g \geq c$ or, with `prune_by_f`, with $g + h \geq c$ (for an admissible h). With `repeat_last`, the last phase is repeated as long as it finds a cheaper path. Once a path is known, a phase that finds none ends the search, unless `continue_on_fail` holds; before, the search always goes on. With `share_cache` (the default), the phases share a `HeuristicCache`, and a state is evaluated once over the whole sequence. Last, `max_phases` bounds the phases run, the repetitions included (0 for no limit). The class is

```
template< Domain D >
class IteratedSearch {
 public:
 struct Phase {
  cost_t< D > cost;
  SearchStats stats;
  };
 IteratedSearch( const D & domain , const IteratedOptions & options );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void );
 [[nodiscard]] const std::vector< Phase > & phases( void ) const noexcept;
 };

[[nodiscard]] IteratedOptions lama_phases( std::size_t m , bool preferred );
```

whose constructor throws `std::invalid_argument` if there is no phase. Then `solve()` returns the best path with the counters summed over the phases, `evaluated` being the misses of the shared cache when the phases share one; its status is `node_limit` if no path was found and some phase hit its limit. For each phase run by the last `solve()`, `phases()` gives the cost (`infinite_cost()` if it found no path) and the counters. The pseudocode of the header is the following.

Algorithm 2.7 (The iterated searches).

```
1  procedure ITERATED(phases)
2     c ← ∞;  best ← none
3     for each phase P (the last one again while it improves, if so
        asked) do
4        π ← the path found by P with the bound c, if any
5        if π exists then c ← cost(π);  best ← π
6        else if a path is known and failing ends the search then stop
7     return best
```

The function `lama_phases( m , preferred )` returns the phases of LAMA [102] on a domain with m heuristics. They are a lazy GBFS and then lazy searches ordered by $g + wh$ with $w = 5, 3, 2, 1$, each reopening the closed

nodes, with `repeat_last` and `continue_on_fail`. Each phase alternates a queue of kind `best` on each heuristic and, if `preferred`, a queue of the preferred successors on each heuristic, boosted by 1000, the preferred successors entering OPEN first.

Theorem 2.29 (The iterated searches). (i) With `pass_bound`, the costs of the paths found by the successive phases decrease strictly. (ii) Let the graph be finite, $h_0 = \infty$ only on the states from which no goal is reachable, and h_0 admissible if `prune_by_f`; let the last phase reopen the closed nodes (with eager evaluation, also `improve_open`), have a queue that is not of the preferred successors only, and hit no node limit; and let `pass_bound`, `repeat_last` and `continue_on_fail` hold and `max_phases` be 0. If some phase finds a path, the path returned is optimal.

Proof. (i) A phase run with the bound c prunes the successors with $g \geq c$ (with `prune_by_f`, $g + h \geq c$, which prunes more), and a goal that it returns was generated, or pushed in a record, with $g < c$; hence its cost, the g of the goal, is smaller than c. (ii) We first show that a phase as the last one, run with a bound c on a graph that has a path $P = (s_0 = u_0, \ldots, u_k)$ of cost smaller than c to a goal, returns a path. Let $g_P(u_j)$ be the cost of the prefix of P to u_j, and let $u_0, \ldots, u_k$ be neither dead ends, by the hypothesis on h_0, nor pruned: $g_P(u_j) \leq \text{cost}(P) < c$, and, with `prune_by_f`, $g_P(u_j) + h_0(u_j) \leq g_P(u_j) + h^*(u_j) \leq \text{cost}(P) < c$ by admissibility; the same holds for any $g \leq g_P(u_j)$. We prove by induction on j that, unless the phase returns before, u_j is expanded with some $g(u_j) \leq g_P(u_j)$. The root is expanded with $g = 0$. When u_j is expanded with $g(u_j) \leq g_P(u_j)$, its successor u_{j+1} is generated with $g' \leq g_P(u_{j+1})$, which is not pruned; then either u_{j+1} is new, or it is known with a larger g and the reopening (and `improve_open`) gives it g', or it is known with $g(u_{j+1}) \leq g'$. In each case, by the argument of the proof of Proposition 2.23 (ii), u_{j+1} is expanded with its current g, which is at most $g_P(u_{j+1})$, before the queues become empty. Hence u_k is expanded, and the phase returns a path, of cost smaller than c by (i), at the latest then. Now, with `continue_on_fail` no phase that fails ends the search, so that all the phases are run, and with `repeat_last` and no limit on the phases the search ends only when the last phase, run with the cost c of the incumbent as its bound, finds no cheaper path; by (i) this happens after finitely many repetitions, the costs of the paths below the first c being finitely many on a finite graph. By the claim above, no path of cost smaller than c exists, i.e., $c \leq C^*$; and $c \geq C^*$, since c is the g of a goal, i.e., the cost of a path. Hence $c = C^*$, and the path returned, whose cost is at most c, is optimal. □

The phases of `lama_phases` satisfy the hypotheses of (ii) on the options: the last phase is lazy, of weight 1, reopens the closed nodes and has a queue of kind `best` on each heuristic, and the phases are repeated while they improve. Thus LAMA, as the library runs it, ends with an optimal path on a finite graph when it is given the time, which is the anytime behavior that Richter and Westphal describe [102]; the test `iterated` checks it on 20 states of the 8-puzzle, together with the strict decrease of (i) phase by phase. It also runs three phases on an instance of Korf with and without the shared cache, and checks that the cache changes no path and lowers `evaluated`.

We remark that the cost of a phase is the g of its goal. It may exceed the cost of the path that the parents give, when a node of that path was reopened after the goal was generated. The bound passed to the next phase is then larger than necessary, and that phase may return a path that is not cheaper than the one before. The conclusion of (ii) does not change, since its proof uses only that c is at least the cost of a path.

2.15 Width-based search

The width-based searches of Lipovetzky and Geffner [104], [105] need no heuristic. They prune, or postpone, the states that make nothing new true in terms of the features of a `FeatureDomain` (§1.5). Let $F(s)$ be the set of the features of a state s, and call a *tuple* a set of features. The *novelty* $w(s)$ of a state s, generated after the states $s_1, \ldots, s_n$, is the size of the smallest tuple $T \subseteq F(s)$ such that $T \not\subseteq F(s_i)$ for each i. In words, T is true in s and in none of the states generated before it; $w(s) = +\infty$ if there is no such tuple, i.e., if $F(s) \subseteq F(s_i)$ for some i. Of course, the search IW(i) is stated in §4.8 in terms of the atoms of a factored state, with a bound on its size. Here we describe the classes of `WidthSearch.h`, whose header gives no pseudocode, and we prove what they guarantee.

The novelty table

The class `NoveltyTable` keeps the tuples of at most k features seen so far, out of n features, and gives $\min\{w(s), k+1\}$:

```
class NoveltyTable {
```

```
public:
NoveltyTable( std::size_t n , unsigned k );
[[nodiscard]] unsigned width( void ) const noexcept;
unsigned record( const std::vector< std::size_t > & F ,
                 const std::vector< std::size_t > & N );
void clear( void );
};
```

In the table, the tuples of one feature are the bits of a vector of n bits, and those of two the bits of a triangle of $n(n-1)/2$ bits. Those of $m > 2$ features are the codes $\sum_j i_j n^j$ of their sorted features, in a hash set for each m, which requires $n^k < 2^{64}$; the constructor throws `std::invalid_argument` for $k = 0$ or $n^k \geq 2^{64}$. The method `record( F , N )` takes the sorted features $F = F(s)$ of a state and a sorted subset $N \subseteq F$. It records the tuples of at most k features of F that contain at least one feature of N, and returns the size of the smallest of them that was not recorded before, or $k+1$ if none was; `clear` forgets all the tuples. Two functions prepare the arguments. `features_of( d , s , F )` writes the sorted features of s into F, and `added_features( d , s , t , Fs , F , N )` writes into N the features of the successor t (sorted in F) that its parent s (with the features F_s) does not have. On an `IncrementalFeatureDomain` the latter takes them from the changes of the move, without scanning F_s, and on any other `FeatureDomain` it takes the difference $F \setminus F_s$.

Lemma 2.30 (The incremental novelty is exact). Let the table hold all the tuples of at most k features of the states $s_1, \ldots, s_n$, and nothing else, and let s be a state, t a successor of s, and s one of the s_i. Then `record( F (t) , N )` with $N \supseteq F(t) \setminus F(s)$ returns $\min\{w(t), k+1\}$, where the novelty is taken with respect to $s_1, \ldots, s_n$, and afterwards the table holds all the tuples of at most k features of $s_1, \ldots, s_n, t$.

Proof. A tuple $T \subseteq F(t)$ that contains no feature of N is contained in $F(t) \setminus N \subseteq F(s)$, and it is therefore in the table already. Those that `record` visits are exactly the others: the single features of N, the pairs of a feature of N with another feature of F, and the tuples of $m \geq 3$ features of F with at least one in N. Hence, after the call, all the tuples of t of at most k features are in the table, and a tuple of t is new, i.e., contained in no $F(s_i)$, if and only if `record` inserts it; the size of the smallest such tuple is $w(t)$ when $w(t) \leq k$, and `record` returns $k+1$ otherwise. □

With $N = F(t)$ the lemma needs no parent, and the searches below call `record( F , F )` on the root. Their invariant is that each state that they keep has all its tuples in the table; a state pruned by its novelty has all its tuples of at most k features in the table already, and recording it changes nothing. In the test `width`, `record( F , F )` is checked against a brute-force enumeration of all the tuples, for $k = 1, 2, 3$, on 300 random sets of 12 features.

Iterated width

`IteratedWidth< D >` is IW(k) on a `FeatureDomain` D: the breadth-first search that prunes each generated state of novelty larger than k, i.e., the IW(i) of §4.8 with three choices of the code. First, the goal test is done when a state is generated (and at s_0); second, a state already generated is a duplicate, which is dropped without being evaluated again; third, the novelty of a successor is computed from the features that the move adds, by Lemma 2.30. The options of the width-based searches are

```
struct WidthOptions : public SearchOptions {

 /// the width k of IW, the largest width of SIW, and the width of the
 /// novelty of BFWS
 unsigned width = 2;

 /// BFWS: the states of novelty larger than width are pruned (k-BFWS)
 bool prune = false;

 /// BFWS: orders by the heuristic instead of the goal features to achieve,
 /// and computes the novelty for each value of it
 bool use_heuristic = false;

 /// BFWS: counts the features of the relaxed plans reached [see
 /// RelevantFeatureDomain]
 bool relevant = true;
```

```
};  // end( struct( WidthOptions ) )
```

i.e., those of `SearchOptions` and the width k (2 by default), which is the k of IW, the largest width of SIW and the width of the novelty of BFWS, with three options of BFWS that we describe below. We quote by hand the classes and their functions.

```
template< FeatureDomain D >
class IteratedWidth {
 public:
 explicit IteratedWidth( const D & domain , const WidthOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void );
 template< class Target >
 [[nodiscard]] Solution< state_t< D > , cost_t< D > >
 search( const state_t< D > & s0 , unsigned k , Target && target );
 [[nodiscard]] std::uint64_t pruned( void ) const noexcept;
 };

template< GoalFeatureDomain D >
class SerializedWidth {
 public:
 explicit SerializedWidth( const D & domain , const WidthOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void );
 [[nodiscard]] std::size_t calls( void ) const noexcept;
 [[nodiscard]] unsigned max_width( void ) const noexcept;
 };

template< FeatureDomain D >
class BestFirstWidth {
 public:
 explicit BestFirstWidth( const D & domain , const WidthOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< FeatureDomain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
iw( const D & domain , const WidthOptions & options = {} );
// siw( domain , options ) on a GoalFeatureDomain, bfws( domain , options )
// on a FeatureDomain, with the same signature
```

`IteratedWidth::search( s0 , k , target )` runs IW(k) from the state `s0` to the first state t, of sorted features F, with `target( t , F )`, and `solve()` is `search` from s_0 to a goal with the width of the options; `pruned()` tells how many states the last search pruned by their novelty. The `Solution` has the status `unsolvable` when the search ends without reaching a target, whether or not it pruned a state, and `node_limit` when the stored states reach the limit. In the counters, `evaluated` counts the novelty tests, one per state that is not a duplicate, the root included.

The guarantee of IW(k) rests on the width of a problem, which we define on the paths of the graph, as Lipovetzky and Geffner define it on the plans of a planning problem [104]. A tuple T is *true* in s if $T \subseteq F(s)$; a path *for* T is a path from s_0 to a state in which T is true, and it is *optimal for* T if it has the least number of edges among them.

Definition 2.31 (Width). The width of a problem is the least k for which there is a sequence of tuples $T_0, T_1, \ldots, T_m$ of at most k features each such that (i) T_0 is true in s_0; (ii) every path optimal for T_i can be extended by one edge into a path optimal for T_{i+1}, for each $i < m$; and (iii) every path optimal for T_m is a path of least number of edges to a goal.

Theorem 2.32 (IW(k) finds an optimal path on a problem of width at most k). If the width of a problem is at most k and there is no node limit, IW(k) returns a path to a goal with the least number of edges; with unit costs, an optimal one.

Proof. Let $T_0, \ldots, T_m$ be the sequence of Definition 2.31 and d_i the number of edges of a path optimal for T_i, so that $d_{i+1} = d_i + 1$ by (ii). The search generates the states by nondecreasing depth, since the queue is FIFO and

a successor is one edge deeper than its parent, and the path that the parents give to a kept state has as many edges as its depth. We claim that, for each i, unless the search returns before, it keeps a state of depth d_i in which T_i is true. For $i = 0$, the root is kept, at depth $0 = d_0$. Let s be a kept state of depth d_i with T_i true; its path is optimal for T_i, and by (ii) some successor s' of s has T_{i+1} true, and its path is optimal for T_{i+1} (the successor is not the parent of s, which has depth $d_i - 1 < d_{i+1}$ and therefore T_{i+1} false, and hence the skip of the parent does not remove it). The expansion of s generates s', unless the search returns before. Consider the first state s'' generated, in the order of the search, in which T_{i+1} is true; it has depth at most d_{i+1}, since s' is generated (as a new state or as a duplicate of one generated before), and at least d_{i+1}, since its path has T_{i+1} true at its end. The tuple T_{i+1} is true in none of the states generated before s'', so that $w(s'') \leq |T_{i+1}| \leq k$, and s'' is kept, by Lemma 2.30 and the invariant of the table. Hence the claim. For $i = m$, the kept state of depth d_m with T_m true has a path optimal for T_m, which by (iii) leads to a goal with the least number of edges; the goal test at its generation returns it, unless the search has returned before at another goal, generated before it and thus not deeper. □

Our proof follows that of Theorem 4 of [104], with the goal test at the generation and the duplicates of the code; we remark that it needs the novelty to be taken with respect to all the states generated before, the pruned ones included, which the table gives, since a pruned state adds no tuple. The size of IW(k) is that of Proposition 4.19, which we restate in terms of features.

Proposition 2.33 (The size of IW(k)). With n features, IW(k) keeps at most $1 + \sum_{j=1}^{k} \binom{n}{j}$ states, and expands at most as many; for a fixed k this is $O(n^k)$.

Proof. Each state kept after the root has novelty at most k, and its recording adds at least one tuple of at most k features to the table, which never loses one; the tuples of j features are $\binom{n}{j}$. An expanded state is a kept one. □

On a grid whose features are the column and the row of a cell, each cell is a distinct tuple of two features. Hence IW(2) prunes no cell, and it is a breadth-first search with duplicate detection, which is optimal with unit costs; the test `width` checks it against A* on three random grids of 50×50 cells. The same test checks that IW(1), which keeps only the cells of a new row or of a new column, either fails or finds a path not cheaper.

Serialized iterated width

`SerializedWidth< D >` (SIW) runs on a `GoalFeatureDomain`, whose goal is the conjunction of the goal features G (§1.5), and achieves them one at a time [104]. From the current state s, with $A = F(s) \cap G$ the goal features that it has, it runs IW(1), IW(2), …, IW(k) from s, k being the `width` of the options, until one reaches a target, i.e., a goal or a state that has all the features of A and at least one more goal feature; the target becomes the current state, and the path to it is appended to the plan. SIW fails when IW(k) reaches no target, or when an IW(j) that pruned no state reaches none (then no target is reachable from s, and the larger widths are not tried). The counters are summed over the calls of IW (`stored` is the largest of their values), the path is the concatenation of their paths and the cost the sum of their costs, and `calls()` and `max_width()` give the calls of IW and the largest width used by the last search.

Proposition 2.34 (SIW). On a finite graph and without a node limit, SIW terminates after at most $|G|$ calls of `search` that succeed, and it returns a path from s_0 to a goal or fails; it may fail although a goal is reachable.

Proof. Each successful call moves s to a goal, which ends the search, or to a state with all the goal features of s and one more; the number of goal features of the current state thus increases strictly, and it is at most $|G|$, since a state with all of them is a goal. Each call is a breadth-first search on a finite graph, and at most k calls are made from the same current state. Each call appends a path from the current state, and the plan is therefore a path. SIW fails, for instance, when every path from s that keeps the features of A leads to no state with more goal features, while a goal is reachable only by undoing a feature of A, as on the sliding-tile puzzles, where a tile in its goal cell must often be moved away. □

In the test `width`, SIW runs with $k = 2$ on the random grids and with $k = 8$ on an instance of the 8-puzzle, and checks the paths.

Best-first width search

`BestFirstWidth< D >` (BFWS) is the best-first search of Lipovetzky and Geffner [105], in the variant BFWS(f_5) of that paper, which uses the novelty to order rather than to prune. It orders OPEN by the triple $(w, \#g, g)$, the ties broken by the order of insertion, where $\#g$ is the number of goal features that a state does not have. The novelty w is computed in a separate `NoveltyTable` of width k for each value of the pair $(\#g, \#r)$, where $\#r$ counts the features of a relaxed plan reached on the path to the state. Precisely, if the domain is a `Relevant`

`FeatureDomain` and the option `relevant` holds (the default), the relevant features of a node are those of `for_each_relevant_feature` at the nearest ancestor where $\#g$ decreased (or at the root), computed when that ancestor is expanded, and $\#r$ counts those added on the path from it; otherwise $\#r = 0$. With `use_heuristic`, or on a domain that is not a `GoalFeatureDomain`, the heuristic h replaces $\#g$ both in the order and in the partition of the novelty, and a successor of \$h = \$ `infinite_cost()` is a dead end. With `prune`, the states of novelty larger than k are pruned (k-BFWS).

Each successor generated is evaluated, and its novelty is recorded in the table of its pair: from the features that the move adds when the pair is that of its parent, and from all its features otherwise, so that the invariant of Lemma 2.30 holds in each table. Then the successor is looked up. A state already generated is a duplicate, unless its g decreases and it is not closed, in which case it is pushed again with the smaller of its two novelties. The closed nodes are never reopened, and the goal test is done at the expansion.

Proposition 2.35 (BFWS). On a finite graph, without `prune` and without a node limit, BFWS returns a path whenever a goal is reachable through states that are not dead ends; with `prune` it may fail.

Proof. Without `prune` each successor that is not a dead end is kept: a new state is pushed, and a known one is either pushed again with a smaller g or dropped, having been pushed when it was new. Each entry is removed before OPEN is empty, and the entry pushed last of a node is stale only if the node is closed, i.e., expanded; hence each node of CLOSED is expanded before OPEN is empty, and the induction along a path to a goal of the proof of Proposition 2.23 (ii) applies. Finally, the entries are finitely many, since a node is pushed again only when its g decreases. With `prune`, a state on every path to a goal may have novelty larger than k, which removes it. □

The test `width` runs BFWS on $\#g$ and on h on the random grids and on two instances of the 15-puzzle of Korf, and k-BFWS with $k = 2$ on the same instances, checking that it either fails or returns a path not cheaper than the optimal one. No domain of the library is a `RelevantFeatureDomain`, and therefore $\#r = 0$ in all the runs of the tests; the computation of the relevant features is not exercised by the tester.

2.16 Benchmark

Our benchmark of this chapter is the sequential A* on the 100 instances of the 15-puzzle of Korf [7], with the Manhattan distance and the goal with the blank in the first cell, run by `tools/AStarBench` from `tools/bench/campaign.sh` (its table A). Each instance is solved in a process of its own, pinned to one core and to the memory of its node, first with the bucket queue, then with the heap, and then with the bucket queue again; the second run of the bucket queue is an A/A comparison, which measures the noise of the machine. When the reference solver of Burns et al. [4] is available to the campaign, it is run on the same instances and its row is added to the table. Costs are integers and the search is exact: each instance is solved to its optimal cost, which is known from [7]. The machine and the compiler are those stated in the note of the table. The clock is the `Stopwatch` started at the first line of the private method `run` that `AStar::solve()` calls (`AStar.h:194`) and read before it returns (`AStar.h:298`). Hence it does not include the start of the process, the reading of the instance and the construction of the domain; nor does it include the release of CLOSED and of the arena of the nodes, which happens when the local objects of `solve()` are destroyed, after the clock has been read, while it includes their allocation.

Table 2.1. Sequential A* on the 100 instances of Korf: for each search (the OPEN list, and the reference solver when present), the instances solved, the seconds summed over the instances (each instance the median of its repetitions), the nodes expanded summed over the instances, and the millions of expansions per second, i.e., the ratio of the last two.

search	solved	seconds	expanded	M expansions/s

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The satisficing searches

The satisficing searches are compared on the 100 instances of the 15-puzzle of Korf [7], with the Manhattan distance as h_0, the number of misplaced tiles as h_1, and the moves that lower the Manhattan distance as the

preferred successors, run by `tools/AStarBench` with `--algo gbfs` (with the evaluations `--eval eager` and `--eval lazy` and the queues of `--queues`), `ehc`, `lama`, `iw`, `siw` and `bfws`; these runs are not yet a table of `tools/bench/campaign.sh`. Each instance is solved in a process of its own, pinned to one core, as in §2.16, and the clock is the `Stopwatch` of `solve()`, which does not include the construction of the domain. Since the searches are not exact, the table reports the costs of the paths with the work, and the comparison is between costs and expansions, not only between times; the iterated searches are run to the end of their phases, i.e., until they prove their last path optimal (Theorem 2.29), and their time is therefore not that of their first path.

Table 2.2. The satisficing searches on the 100 instances of Korf: for each search and its options, the instances solved, the costs of the paths summed over the solved instances, their ratio to the sum of the optimal costs, the expansions and the evaluations of the heuristic summed over the instances, and the seconds summed over the instances (each the median of its repetitions).

search	solved	cost	cost / optimal	expanded	evaluated	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

2.17 Notes and references

A* and the proof of its admissibility are due to Hart, Nilsson and Raphael [9], who also stated the consistency assumption and, under it, the optimality of A* in the number of expansions (their Theorem 2). The form of the algorithm with the reopening of the closed nodes, which the library follows, is Algorithm 1 of the survey of Fukunaga et al. [1]. The textbook of Edelkamp and Schrödl [10] treats in its chapters 2 and 3 the general best-first search, the reopening and the worst cases of inconsistent heuristics, and chapter 3 of Russell and Norvig [13] compares the uninformed searches. The textbook of Nilsson [8] treats the graph search and the heuristic estimates in the same framework. The analysis of inconsistent heuristics by Felner et al. [31] shows that they can be useful in practice, mostly in depth-first searches with the bidirectional pathmax, which the library does not implement. The engineering of a fast A* (the bucket queue, the ties broken in favor of the larger g, the parent pruning by operator and the states stored in place) is studied by Burns et al. [4], whose implementation is the reference of the benchmark. The library makes the same choices, adds the prefetch of the lookups of CLOSED and the huge pages (Chapter 3), and keeps one node per state in an arena for the whole search. The parallel searches of Chapter 8 and Chapter 9 keep the structure of Algorithm 2.2 in each worker.

The satisficing searches

Greedy best-first search goes back to the Graph Traverser of Doran and Michie [96], and weighted A*, with its bound wC^*, to Pohl [46]. The OPEN lists of `GreedySearch` are those of the planners: the deferred evaluation and the preferred operators with their boosting are due to Richter and Helmert [97], who compare them on the planning benchmarks; the alternation of queues over several heuristics, the lexicographic combination and the Pareto queues to Röger and Helmert [98], who find the alternation the best of the combinations they compare; the type-based exploration to Xie et al. [99]; and the epsilon-greedy selection to Valenzano et al. [100], who compare the knowledge-free exploration with the enhancements that use knowledge of the problem. Enforced hill-climbing and the helpful actions are those of the planner FF of Hoffmann and Nebel [101], the phases of `lama_phases` those of LAMA as Richter and Westphal describe them [102], and the restarts with the values of the heuristic kept are those of Restarting Weighted A* of Richter, Thayer and Ruml [103]. The width, IW(k) and SIW are due to Lipovetzky and Geffner [104], and BFWS to the same authors [105]; the high-water marks, the benches and the characterization of the states that GBFS may expand to Heusner, Keller and Helmert [112].

As to where the library stands, the following facts are stated from the code. The searches run on any `Domain`, the planning tasks of Chapter 16 among them, and the refinements of §1.5 supply what the planners take from the structure of a task (several heuristics, the preferred successors, the deferred evaluation, the goal and relevant features). The planners of the papers use the heuristics of the delete relaxation and of the landmarks, which on a `Domain` are available only through the planning domain; the bound of Proposition 2.25 on the lazy weighted search, which the papers do not state, and the optimality of the iterated searches of Theorem 2.29 are proved here for the code. BFWS implements the variant f_5 of [105] with the relevant features of a `RelevantFeatureDomain`, which no domain of the library provides, and without the other variants of the paper; the parallel greedy searches, with the theory of the BTS that they need, are those of Chapter 12.

2.18 Exercises

Exercise 2.1 (`ctest -R PBeFS.astar`). Run Algorithm 2.2 by hand on the graph of Figure 2.2 with the ties broken in favor of the smaller g, and then in an arbitrary order; show that the cost returned is the same, as Theorem 2.9 requires. The test checks that both OPEN lists, which break the ties differently, return the optimal cost on 200 states of the 8-puzzle and on instances of the 15-puzzle.

Exercise 2.2 (`ctest -R PBeFS.reopen`). On the graph of Figure 2.7, list the entries of OPEN after each iteration of Algorithm 2.2 and count the expansions and the reexpansions. Which inequality of Definition 2.2 does $h = (0, 3, 0, 1, 0)$ violate? Find the largest consistent heuristic that is not larger than h^* on that graph, and show that with it no node is reopened.

Exercise 2.3 (`ctest -R PBeFS.grid`). Show that A* on a domain whose heuristic is zero is the algorithm of Dijkstra, and that the wrapper `Blind` of `test/test.h:346` gives it from any domain. On a grid with 4 moves of unit cost, compare the cells expanded with $h = 0$ and with the Manhattan distance (cf. Figure 2.6), and explain by Theorem 2.15 why the second never expands a cell that the first does not, but for the ties of the last layer.

Exercise 2.4 (`ctest -R PBeFS.graph`). On the map of Romania of Russell and Norvig [13], with the straight-line distance to Bucharest as the heuristic, run A* by hand from Arad, and check that the path returned passes through Sibiu and costs 418. Prove that the straight-line distance is consistent if the length of each road is at least the distance between its ends.

Exercise 2.5 (`ctest -R PBeFS.limits`). Show that a tree search (Algorithm 2.1 without the lookups of line 10) may not terminate on a finite graph with a reachable goal if some costs are zero, and that the graph search of Algorithm 2.2 terminates on any finite graph (Theorem 2.11). The test checks that A* proves a grid split by a wall unsolvable, and that it stops at the node limit.

Exercise 2.6 (`ctest -R PBeFS.shift`). Let h be consistent and $\delta > 0$. (i) Is $h + \delta$ admissible? Does it satisfy the inequality of Definition 2.2 on the edges? (ii) Show that A* with $h' = h + \delta$ on all the states, goals included, selects the same entries in the same order as with h, and returns an optimal path, although h' is not admissible. (iii) Show by an example that if h is shifted on some states only, the path returned may not be optimal. The test checks (ii) on 200 states of the 8-puzzle, comparing the costs and the expansions with those of the Manhattan distance.

Exercise 2.7 (`ctest -R PBeFS.weighted`). Let h be admissible and $w \geq 1$. Prove that A* with the priority $g + w\,h$ returns a path of cost at most $w\,C^*$, following the proof of Lemma 2.8 with the node u' of the invariant. The test runs `AStar` on a wrapper of the 8-puzzle whose heuristic is twice the Manhattan distance, and checks the bound on 200 states.

Exercise 2.8 (`ctest -R PBeFS.astar`). Give a graph and an admissible but inconsistent heuristic on which $I > 0$ and `reexpanded` > 0, and compute the four counters of Proposition 2.18 by hand; check (i) of that proposition on them. Then show that a node limit $L = 5$ on a start at least six moves from the goal of the 8-puzzle makes the search return `node_limit` with between 5 and $\max(4, 1) + 4 = 8$ stored nodes (Proposition 2.17). The test checks the edge cases of a start that is a goal, a start one move away and the node limit of 5 (`check_astar_edges`, `test/test.h:474`).

Exercise 2.9 (`ctest -R PBeFS.reopen`). Prove that the pointers `parent` of A* never form a cycle even when nodes are reopened (Lemma 2.10), and that the path returned on the graph of Figure 2.7 is the path of the tree from v_0 to v_4. What would break in the proof if an edge had a negative cost?

The satisficing searches

Exercise 2.10 (`ctest -R PBeFS.planning`). Prove (ii) of Proposition 2.22 for the blind heuristic and positive costs directly, i.e., without Theorem 2.14, and show that the number of the expansions is then the number of the states at a distance smaller than C^* from s_0. Give a graph on which the search without the test at generation expands every state at distance C^* before the goal, and one on which it expands none of them. The test checks the count of the expansions on the planning tasks of the library.

Exercise 2.11 (`ctest -R PBeFS.gbfs`). Run Algorithm 2.5 with lazy evaluation and $w = 1$ by hand on the graph of the proof of Proposition 2.25, and then with eager evaluation, and compare the paths returned. Then build, for any $w \geq 1$ and $\bar{c} > 0$, a graph on which the lazy search returns a path of cost larger than wC^*. The test checks the bound of Theorem 2.24 with eager evaluation only, as the comment of the test says.

Exercise 2.12 (`ctest -R PBeFS.gbfs`). Give an infinite graph with a reachable goal on which GBFS does not terminate, and show that a finite weight w and positive costs bounded below make the search terminate. Then show that the dead ends of the test, the successors of s_0 but one, are never expanded, and explain why a dead end must not be declared on a state from which a goal is reachable (Proposition 2.23).

Exercise 2.13 (`ctest -R PBeFS.gbfs-queues`). With two queues of kind `best` on h_0, one of them of the preferred successors only, and `boost` = 1000, describe the sequence of the queues selected after a progress, while both queues are nonempty, and after the queue of the preferred successors becomes empty for a while; show that a queue selected less often than the others for a while is then selected as many times in a row as it fell behind. The test runs all the kinds of queue of this section, with both evaluations, and checks that the same seed gives the same search.

Exercise 2.14 (`ctest -R PBeFS.gbfs-queues`). Show that a queue of kind `epsilon` with $\epsilon = 0$ selects the entries in the order of a queue of kind `best`, so that GBFS with it expands the same states, as the test checks on an instance of Korf. Compute the probability that a queue of kind `type` selects a given node when OPEN holds T types, the type of the node holding N nodes, and compare it with that of a queue of kind `epsilon` with $\epsilon = 1$.

Exercise 2.15 (`ctest -R PBeFS.ehc`). On the graph of the test with a plateau of $h = 5$, follow Algorithm 2.6 and give the plan; then give a graph without dead ends on which EHC with `prune` fails, and one on which the plan found by EHC costs more than any fixed multiple of C^* (Proposition 2.28).

Exercise 2.16 (`ctest -R PBeFS.iterated`). Show that the hypotheses of (ii) of Theorem 2.29 hold for the phases of `lama_phases`, and that they fail when `max_phases` is 3, as in the second part of the test. Then explain why the shared `HeuristicCache` lowers `evaluated` without changing the paths, and bound `evaluated` by the number of the distinct states that the phases evaluate.

Exercise 2.17 (`ctest -R PBeFS.width`). Show that on the grids of the test, whose features are the column and the row of a cell, every problem has width at most 2 by Definition 2.31, with T_i the cell at depth i of a shortest path, so that IW(2) is optimal by Theorem 2.32; then show that IW(1) keeps at most $W + H - 1$ cells on a grid of $W \times H$ cells, and give a grid on which it fails. The test checks `record` against a brute-force enumeration, IW(2) against A*, and IW(1), SIW and BFWS on the same grids.

Exercise 2.18 (`ctest -R PBeFS.width`). Explain why SIW fails on many instances of the sliding-tile puzzles with a small width (Proposition 2.34), and why the test runs it on the 8-puzzle with $k = 8$. Then show that k-BFWS with $k = 2$ may return a path more expensive than BFWS without pruning, and that both may return a path that is not optimal.

3. The data structures of search

[Source: include/PBeFS/astar/OpenList.h (OpenEntry, OpenKind, HeapOpenList, BucketOpenList), include/PBeFS/astar/NodeTable.h (NodeTable), include/PBeFS/astar/Node.h (Node, NodePool, packed_fg, infinite_cost, trace_path), include/PBeFS/astar/Structures.h, include/PBeFS/astar/structures/ (the sets std, boost and the default one), include/PBeFS/astar/AStar.h (AStar, astar); the tests open, astar and ddd of test/test.cpp; tools/bench/structures.sh]

We are concerned in this chapter with the data structures that a best-first search runs on: the OPEN list, i.e., a priority queue of the entries (f, g, u) of §2.3, the CLOSED list, i.e., a dictionary from the generated states to their nodes, and the memory of the nodes themselves. On the domains whose expansion takes a few instructions, such as the sliding-tile puzzles and the grids, these structures take most of the time of a search (§2.10), and the choices made here are those of all the searches of the library, the parallel ones included. We first list the operations that a search performs on the three structures, and we prove that the lazy deletion, which replaces the decrease of a key, is exact. Then we describe the heaps, the two-level bucket queue and the packed keys, each with its invariant and its cost, and we prove what the breaking of the ties changes. In the second half of the chapter we treat CLOSED (a hash table with open addressing), the arena of the nodes, the prefetch of the lookups, the perfect hashes of the domains, and the two variants of duplicate detection that the memory-bounded searches of Chapter 4 use. Finally, we describe the three sets of data structures among which a build of the library chooses, and how a user selects the OPEN list of a search. The pseudocode follows the conventions of §1.4; in it, n is the number of the entries of a structure and b the branching factor of the domain.

3.1 The operations of a search

An iteration of A* (Algorithm 2.2 of §2.3) performs the following operations on its data structures: (i) one extraction of the first entry of OPEN, (ii) for each of the b successors of the node, one evaluation of the hash of its state and one lookup in CLOSED, which inserts the state if it is new, and (iii) for each successor that is new or reached through a cheaper path, one push in OPEN. In the library, the three structures have the following interfaces, which the three sets of §3.12 implement in the same way (a *node* is the record of a state described in §3.7, and the *hash* of a state is the 64-bit value returned by the member `hash` of the `Domain`, cf. Chapter 1).

- *OPEN* (`OpenList.h`): `push( node )` pushes the node with its current f and g, `push( node , g )` pushes it with the given g (and $f = g + h$) without reading `node->g`, `top()` is the entry with the least f and, among those, the largest g, `top_f()` is its f, `pop()` removes and returns it, and `empty()` and `size()` count the entries, the stale ones included. An entry (`OpenEntry`) is the pair of the pointer to the node and of the value of g at the push.
- *CLOSED* (`NodeTable.h`): `find_or_insert( state , hash , make )` looks the state up and, if it is absent, inserts the node returned by `make()`, returning the node of the state and whether it has just been inserted; `find( state , hash )` returns the node or `nullptr`; `prefetch( hash )` brings into the cache the first place probed for the hash; `slots_for( n )` is the number of places that holds n nodes without growing; `size()` and `bytes()` measure the table.
- *The arena of the nodes* (`NodePool`, `Node.h`): `allocate()` returns a new node, whose address never changes until the end of the search; the nodes are never freed one by one.

The hash is computed once, by the search, and passed to each operation of CLOSED; hence neither the lookup, nor the growth of the table, nor the prefetch evaluates it again.

Lazy deletion

Neither OPEN list of the library can lower the key of an entry. A node reached through a cheaper path is pushed again with its new g, and the old entry, which records the g of its push, becomes *stale*: an entry (u, g) is stale when the node u is closed or when $g \neq g(u)$. The search drops a stale entry when it reaches the top (line 6 of Algorithm 2.2), which is the following loop.

Algorithm 3.1 (The selection with lazy deletion).

```
1  function SELECT(OPEN)
2     while OPEN ≠ ∅ do
3        (u, g) ← pop(OPEN)
4        if u is closed or g ≠ g(u) then continue      ▷ a stale entry
5        return u
6     return nil
```

This is the *lazy deletion* of the textbooks. It makes the entries plain values, without the handles by which a heap would find an entry to move it, and which each swap of the heap would have to update. We first show that it selects the right nodes, and then we bound its price, i.e., the number of the entries.

Lemma 3.1 (Lazy deletion is exact). At the beginning of each iteration of Algorithm 2.2, every node of CLOSED that is not closed has exactly one entry in OPEN that is not stale, namely $(u, g(u))$, and a closed node has none. Hence Algorithm 3.1 returns the node that a queue with the decrease of the keys would return, under the same rule for the ties.

Proof. The value $g(u)$ changes only at line 12 (when u is created) and at line 15 (when it decreases), and each of the two lines is followed by a push of $(u, g(u))$ with the new value; also, the flag *closed* is cleared only at line 15. Thus, after the last change of $g(u)$ the entry pushed with it is in OPEN, and it is not stale until u is closed, which happens only when that very entry is selected (line 7). Every other entry of u carries a value of g that u had before, and since $g(u)$ only decreases these values differ from the current one; hence they are stale. The entries that are not stale are thus in a one-to-one correspondence with the open nodes, each carrying its current key $(g(u) + h(u), g(u))$, which is the content of a queue with the decrease of the keys. □

Lemma 3.2 (The entries of OPEN). The entries pushed by A* are $1 + N + I$, with N the nodes stored besides the root and I the improvements of g at line 14 of Algorithm 2.2; for any heuristic they are at most the successors generated plus one, and with a consistent one the improved nodes are all still in OPEN.

Proof. Line 3 pushes the root, line 13 pushes each new node once, and line 16 pushes once per improvement. Each successor generated is either new, or an improvement, or a duplicate, and hence $N + I$ is at most the number of the successors generated. With a consistent heuristic, moreover, no closed node is improved (Theorem 2.13), and the improvements are those of nodes still in OPEN. □

By Lemma 3.1, the entries that become stale are exactly the I entries overtaken by an improvement, and each of them costs at most one extraction, without an expansion (none if it is still in OPEN when the search ends). On the sliding-tile puzzles with the Manhattan distance and on the grids with unit costs, a node of OPEN is rather rarely improved, since its first path is usually a shortest one, and the stale entries are then typically a small fraction of the pushes (an observation on these domains, not a property that we can prove in general).

3.2 Heaps

A *binary heap* keeps n entries in an array, the entry in the place i being not worse than its children in the places $2i + 1$ and $2i + 2$; the first entry is in the place 0. A *d-ary heap* has d children per node, in the places $di + 1, \ldots, di + d$, and the parent of the place $i > 0$ is the place $\lfloor (i - 1)/d \rfloor$ (cf. Johnson [158]). Here "better" is a strict weak order on the keys of the entries, i.e., a strict order whose incomparability is transitive (as the order of A* below, in which the entries with the same f and g are incomparable), and "not worse" its negation, which is then transitive too. A push puts the new entry in the place n and moves it up, past its parent as long as it is better (Figure 3.1). A pop takes the entry of the place 0, moves the last entry there and moves it down, past the best of its children as long as that child is better (Figure 3.2). The default heap moves a hole rather than swapping, i.e., it shifts each parent (or child) into the hole and writes the moving entry once, at its final place.

Algorithm 3.2 (Push and pop in a d-ary heap $A[0..n-1]$).

```
 1  procedure PUSH(A, x)
 2     i ← n;  n ← n + 1
 3     while i > 0 and x is better than A[(i − 1) div d] do
 4        A[i] ← A[(i − 1) div d];  i ← (i − 1) div d
 5     A[i] ← x

 1  function POP(A)                                    ▷ n ≥ 1
 2     r ← A[0];  x ← A[n − 1];  n ← n − 1;  i ← 0
 3     if n = 0 then return r
 4     loop
 5        if d i + 1 ≥ n then break                ▷ i is a leaf
 6        c ← the best of A[d i + 1 .. min(d i + d, n) − 1]
 7        if A[c] is not better than x then break
 8        A[i] ← A[c];  i ← c
 9     A[i] ← x
10     return r
```

Proposition 3.3 (Correctness and cost of a d-ary heap). Algorithm 3.2 keeps the *heap order*, i.e., no entry is better than its parent; hence $A[0]$ is an entry of which no other is better. Let $k = \lceil \log_d((d-1)n+1) \rceil - 1$ be the depth of the place $n-1$, with n the entries after a push or before a pop. A push makes at most k comparisons and a pop at most dk, i.e., $O(\log_d n)$ and $O(d \log_d n)$.

Proof. Order. In PUSH, let the hole be at i; the invariant of the loop is that the array, with the hole filled by any entry not worse than the entries of its subtree and not better than its parent, is in heap order. It holds at the start, since the hole is a new leaf, and line 4 keeps it: the parent moved into the hole is not worse than the entries below it (it was their ancestor) and x is better than it. When the loop stops, x is not better than the parent of the hole, and it is better than the entries moved below it, which were already not worse than the rest of the subtree. In POP the argument is symmetric: line 8 moves into the hole the best child, which is not worse than its siblings and than its own subtree, and line 9 writes x where no child is better than it. Since the order is transitive along the paths to the root, $A[0]$ is an entry of which no other is better. Cost. The levels $0, \ldots, \ell$ of a d-ary heap hold $(d^{\ell+1}-1)/(d-1)$ places, and hence the place $n-1$ has the depth k of the statement. PUSH makes one comparison per level that the entry climbs, at most k; POP makes, per level that the hole descends, at most $d-1$ comparisons to find the best child and one with x, i.e., at most dk. □

In the default set, `HeapOpenList` is a d-ary heap with $d = 4$. Its items are the triples (f, g, u), i.e., they carry f and g next to the pointer to the node, and hence a comparison never follows the pointer. The order is that of A*: the item (f_1, g_1, u_1) is better than (f_2, g_2, u_2) if $f_1 < f_2$, or $f_1 = f_2$ and $g_1 > g_2$; two items with the same f and g are not better than each other, and they leave the heap in an order that the operations do not fix. With $d = 4$ the heap has half the levels of a binary heap, and by Proposition 3.3 a push makes half the comparisons, while a pop makes 4 comparisons per level against 2, on half the levels, i.e., about as many in all. Furthermore, with integer costs an item takes 16 bytes. The four children of a place i are thus the 64 consecutive bytes from the offset $16(4i+1) = 64i+16$, and in an array aligned to a line of 64 bytes (as the arrays on huge pages are, cf. §3.7) they straddle two lines of cache (an array shifted by three places would align them on one), and the descent of a pop reads at most two lines per level, on half the levels of a binary heap. Also, the array of a large heap is allocated on huge pages, and the walk from the root to a leaf, whose places are far apart, then does not miss the translation lookaside buffer at each level. In the set `std` the heap is the binary heap of `std::priority_queue`, and that of the set `boost` is the 4-ary `boost::heap::d_ary_heap`. All three use the same order, and they may differ only in the order of the items with the same f and g.

3.3 The two-level bucket queue

With integer costs and a bounded range of f, a priority queue can do without comparisons: the queue of Dial [65] keeps an array of buckets indexed by the key, and it finds the least key by a scan that moves forward. The order of A* has two keys, the least f and then the largest g. Accordingly, `BucketOpenList`, the default OPEN for integer costs, nests two levels of buckets, as the nested bucket queue of Burns et al. [4] (which they distinguish from the 2-level bucket queue of [10], meant to shrink the array when the values of f are many): an array of *layers* indexed by f, the layer f holding the entries with that f in a set of LIFO *lists*, one per value of g (Figure 3.3). Two bounds make the scans short: *minF*, below which no layer has entries, and, in each layer L, *topG*(L), above which no list of the dense array of the layer (see below) has entries.

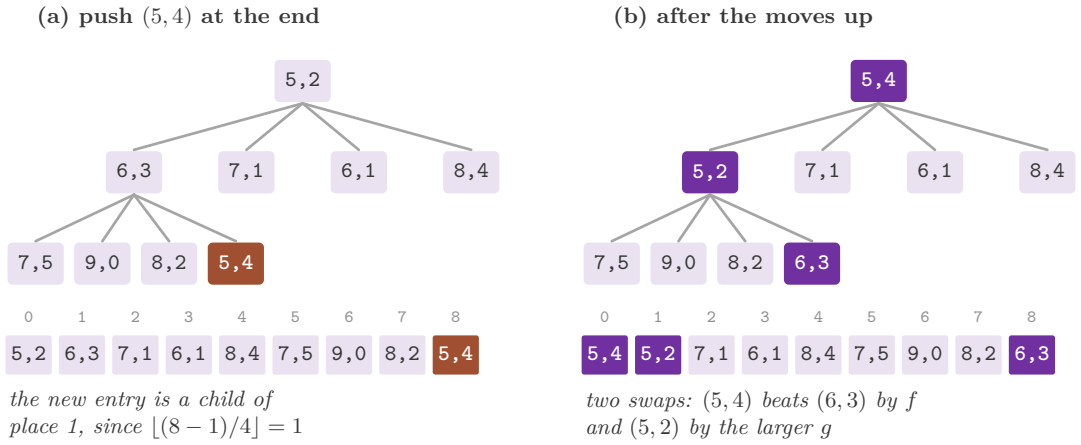


Figure 3.1. A push in a 4-ary heap of entries (f, g), as a tree and as an array (the children of the place i are the places $4i + 1$ to $4i + 4$). (a) The new entry $(5, 4)$ goes to the place 8, a child of the place 1. (b) It moves up twice, past $(6, 3)$, which has a larger f, and past $(5, 2)$, which has the same f and a smaller g.

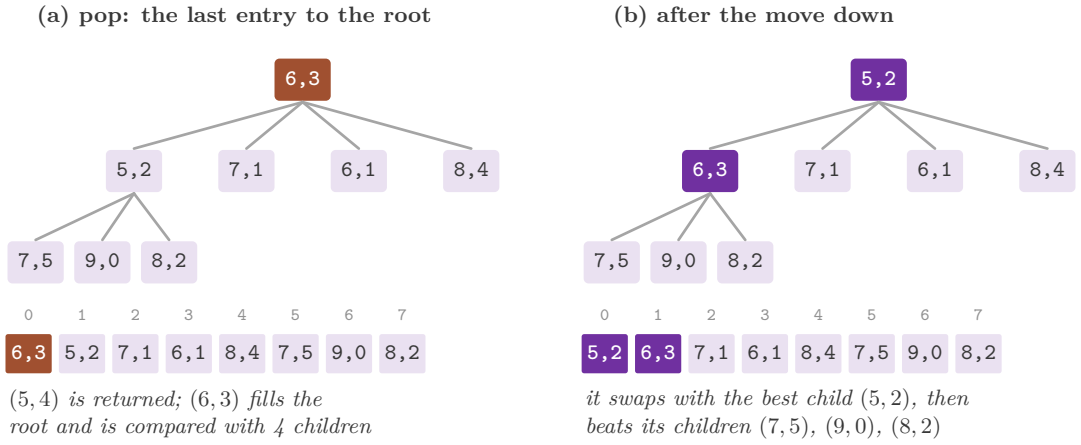


Figure 3.2. A pop from the heap of Figure 3.1. (a) The first entry $(5, 4)$ is returned and the last one, $(6, 3)$, fills the root. (b) It moves down past the best of the four children of the root, $(5, 2)$, and it stops at the place 1, being better than its children; the places read are those of two lines of cache.

The lists of a layer need care. An array of heads indexed by g costs $\Theta(G)$ to rebuild, with G the largest g of the layer, and on a grid g grows to thousands while a layer holds few entries. The default queue therefore keeps in a *dense array* the heads of the lists of $g < D = 256$ (the first $S = 4$ of them inside the layer itself), and those of $g \geq D$ as a sorted vector of at most $R = 64$ *runs* $(g, head)$, the last of which is the top of the layer; a layer that has runs is *sparse*. A push of some $g \geq D$ into a layer that already has R runs turns it *dense*, i.e., it moves the heads of its runs into the dense array, which then covers all its values of g, and the layer stays dense until it empties, when it frees the part of the array of $g \geq S$ and resets *topG* to 0. The items lie in one vector of cells, each with the index of the next cell of its list, and the cells freed by a pop form a free list that a push takes from first; hence a push or a pop calls no allocator once the vector has reached the largest size of the queue.

Algorithm 3.3 (The two-level bucket queue; L is the layer of f, v its runs, $head_L[g]$ its dense array).

```
 1  procedure PUSH(x, f, g)
 2     if g ≥ D and L is not dense then
 3        if |v| = R then MAKE-DENSE(L)                ▷ L is now dense
 4        else insert x at the head of the run of g in v
 5                    ▷ a new last run if g > the last g of v, else the
 6                    ▷ run of g found, or inserted, by binary search
 7     if g < D or L is dense then
 8        insert x at the head of the list head_L[g]
 9        if L had no entry or g > topG(L) then topG(L) ← g
10     count(L) ← count(L) + 1
11     if the queue was empty or f < minF then minF ← f

 1  function FRONT()                                   ▷ queue not empty
 2     while count(the layer of minF) = 0 do minF ← minF + 1
 3     L ← the layer of minF
 4     if v = ∅ then
 5        while head_L[topG(L)] = nil do topG(L) ← topG(L) − 1
 6        return the head of head_L[topG(L)], with (minF, topG(L))
 7     return the head of the last run of v, with (minF, its g)

 1  function POP()
 2     (x, f, g) ← FRONT();  unlink x from its list; its cell to the free list
 3     if x came from the last run of v and that run is now empty then
 4        drop the run
 5     count(L) ← count(L) − 1
 6     if count(L) = 0 then free head_L[S ..];  topG(L) ← 0
 7     return (x, f, g)

 1  procedure MAKE-DENSE(L)
 2     G ← the last g of v;  extend head_L to the places 0 .. G
 3     for each run (g', h') of v do head_L[g'] ← h'
 4     v ← ∅;  topG(L) ← G
```

In the following we write $f_{\max}$ for the largest f pushed and G_L for the largest g pushed into the layer L.

Lemma 3.4 (The invariants of the bucket queue). After each operation of Algorithm 3.3: (i) no layer below *minF* has an entry; (ii) the runs of a layer are sorted by increasing g, all of $g \geq D$ and with entries, there are at most R of them, and a dense layer has none; (iii) in a layer that is not dense every entry of $g \geq D$ is in a run and every other one in the dense array, and in a dense layer all are in the dense array; (iv) no list of the dense array above $topG(L)$ has entries. Hence FRONT returns an entry of the least f, of the largest g among those, and the last one pushed among those with the same f and g.

Proof. (i) A push lowers *minF* to its f if f is smaller (or sets it if the queue was empty), and FRONT moves it forward only past empty layers. (ii) A run is appended only with a g larger than the last one, or inserted at its sorted position; a run is dropped when its list empties; MAKE-DENSE empties v, and it is called when $|v| = R$, before a further run would be created. (iii) This follows from the tests of lines 2 and 7 of PUSH and from MAKE-DENSE, which moves all the runs. (iv) Line 9 of PUSH raises *topG* to every g pushed into the dense array (and sets it to g when the layer was empty), MAKE-DENSE sets it to the largest g of the layer, line 5 of FRONT lowers it only past empty lists, and line 6 of POP resets it only when the layer is empty. For the last claim, by (i) the layer of *minF* after line 2 of FRONT is the least nonempty one; if it has runs, by (ii) and (iii) the last run holds the largest g of the layer, since all the entries of the dense array have $g < D$; otherwise,

by (iv), line 5 stops at the largest nonempty list. The lists are LIFO, since an entry is inserted at the head and taken from it. □

We now give the cost. The two scans (lines 2 and 5 of FRONT) are paid by the moves of the bounds in the opposite direction, which is the argument of the following proposition.

Proposition 3.5 (The cost of the bucket queue). (i) Over any sequence of operations, the steps of the scan of line 2 of FRONT are at most $f_{\max}+1$ plus the sum of the decreases of *minF* at line 11 of PUSH, and the steps of the scan of line 5 in a layer are at most the sum of the raises of *topG* in it (line 9 of PUSH and line 4 of MAKE-DENSE). (ii) Besides the scans, a push costs $O(1)$ amortized time if it goes to the dense array or to the end of the runs, and $O(\log R + R)$ if it inserts a run in the middle; a MAKE-DENSE costs $O(G_L)$; a pop costs $O(1)$ besides FRONT, plus $O(G_L)$ for the release of a dense array at line 6. (iii) Hence, if the least f never decreases between two pops (as in A* with a consistent heuristic, by Theorem 2.13), all the pops of a search cross at most $f_{\max}+1$ layers; and if all the pushes into a layer that is never dense precede its first pop, its scan of line 5 makes at most $D-1$ steps until it empties.

Proof. (i) *minF* never exceeds $f_{\max}$ while the queue holds an entry, increases only by one at line 2, and otherwise changes only at line 11; hence its increases are at most $f_{\max}$ plus its decreases (a push into the empty queue may also raise it, which costs no step). The same argument holds for $topG(L)$, which decreases only at line 5 and starts at the g of the first push at each filling of the layer. (ii) The dense array grows by doubling, which costs $O(1)$ amortized per element, and a cell comes from the free list or from the end of a vector that doubles; the binary search over at most R runs costs $O(\log R)$ and the insertion shifts at most R runs; MAKE-DENSE extends the array to G_L+1 places, and the release frees as many. (iii) Under the first hypothesis no push lowers *minF* after the first pop. Under the second, the raises of *topG* in a dense array of $g<D$ total at most $D-1$, since *topG* only increases until the first pop. □

Thus, in A* with a consistent heuristic the scan of the layers costs $O(f_{\max})$ over the whole search. Within a layer, the pushes and the pops interleave, since the expansion of an entry of the least layer pushes its children of equal f into the same layer. Each such push raises *topG* by at most $c(u,v)$ above the g of the entry just popped, at which FRONT had left *topG*, and therefore the scan of line 5 costs at most $D-1$ steps for the pushes made before the first pop of the layer plus $c(u,v)$ per child pushed into it afterwards. Without the first hypothesis, as in a worker of HDA*, whose least layer empties and fills again each time the nodes sent by the other workers arrive, the bound *minF* moves back, and by (i) the scan may cross a layer many times. Also, a sequence that alternates a push of a large $g<D$ with its pop, in a layer that keeps an entry of $g=0$, costs a scan of $\Theta(D)$ per pop. The runs avoid this for the values $g\geq D$: their top is the last run, found without a scan, and a dense array over them is built only after R distinct values $g\geq D$ have been pushed into the layer, i.e., its $O(G_L)$ is paid by at least R pushes, whereas a dense array alone would cost $\Theta(G_L)$ at each filling, even of one entry. The memory is that of the $f_{\max}+1$ layers, of the cells (as many as the largest size of the queue), of the dense arrays (at most D heads per layer that is not dense, and G_L+1 per dense layer) and of at most R runs per sparse layer. It grows with the range of the keys rather than with the entries, and the queue is therefore used only for integer costs. We remark that the cells are indexed by 32-bit integers, and the queue therefore holds fewer than $2^{32}-1$ entries at a time. The sets `std` and `boost`, which have no bucket queue, use the heap for all the cost types.

The test `open` checks both OPEN lists against an ordered multiset of the pairs $(f,-g)$ on random sequences of 20000 pushes and pops in which the least f moves back below the f of the last pop, as in a worker of HDA*, with g up to 200 and up to 2000, i.e., on both sides of D.

> **Status — planned.** The *radix heap* of Ahuja et al. [160], which needs only that no key pushed be smaller than the last key popped (the case of A* with a consistent heuristic) and costs $O(\log C)$ amortized time per extraction for keys in a range of C, is not an OPEN list of the library; the bucket queue covers the integer costs of its domains, and the heap the others.

3.4 Packed keys

The parallel searches whose OPEN is shared or relaxed (MultiQueue A* and PA*SE of Chapter 8) compare the keys of the entries as single machine words, and a comparison, or an atomic read of a key, is then one instruction. For integer costs, `packed_fg` (`Node.h:114-120`) puts $f=g+h$ in the high half of a word of 64 bits and the complement of g in the low one.

Algorithm 3.4 (The packed key of (g,h)).

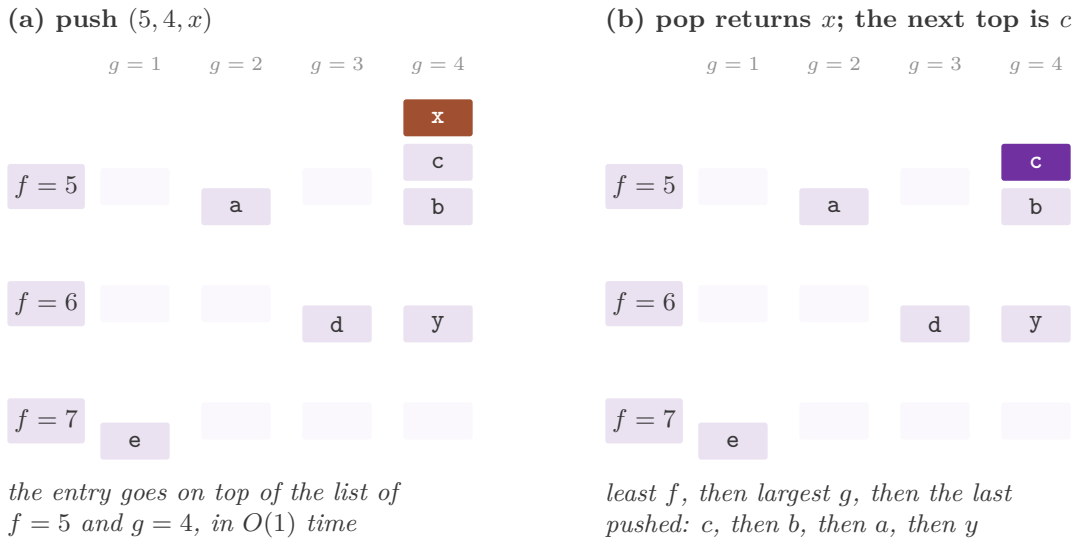


Figure 3.3. The two-level bucket queue: a layer per value of f, and in each layer a LIFO list per value of g (the light places are empty lists). (a) The push of $(5, 4, x)$ puts x on top of the list of $f = 5$ and $g = 4$. (b) The pop returns the entry of the least f, then of the largest g, then the last pushed, i.e., x; the next pops return c, b, a and then y, the top of the layer 6.

```
1  function PACKED(g, h)                    ▷ 0 ≤ g and 0 ≤ g + h < 2^32
2      return (g + h) · 2^32 + (2^32 − 1 − g)
```

The second term is the complement of g in 32 bits, which the code computes as the bitwise negation of g masked to its low 32 bits.

Lemma 3.6 (The order of the packed keys). Let $0 \le g_i$, $0 \le h_i$ and $g_i + h_i < 2^{32}$ for $i = 1, 2$, and let k_i be the packed key of (g_i, h_i). Then $k_1 < k_2$ if and only if $f_1 < f_2$, or $f_1 = f_2$ and $g_1 > g_2$.

Proof. The key is $k_i = 2^{32} f_i + (2^{32} - 1 - g_i)$, and since $0 \le g_i \le f_i < 2^{32}$ (here $h_i \ge 0$ is used) the low half lies in $[0, 2^{32})$. Hence the comparison of the keys is that of the high halves, and for equal high halves that of the low ones, i.e., the comparison of f and then the reverse comparison of g. □

The sum $g + h$ is computed in the cost type before it is widened, and with a signed 32-bit cost the range is therefore $g + h < 2^{31}$, the largest value of the type. Figure 3.4 shows two keys of the same f: the one of the larger g is the smaller word, and comes first.

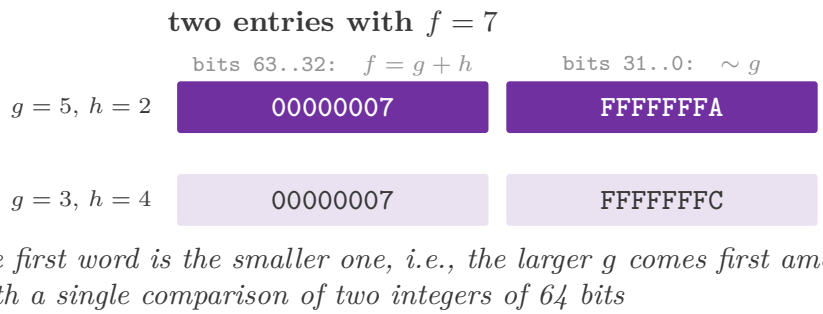


Figure 3.4. The packed keys of two entries with $f = 7$, in hexadecimal: the high half holds f, the low half the complement of g, so that the entry with $g = 5$ has the smaller word and comes before the one with $g = 3$, as in the order of A*.

3.5 Breaking the ties

Both OPEN lists break the ties of f in favor of the larger g, i.e., of the smaller h, since $g = f - h$ when f is fixed; ordering by the least h among the least f, as some textbooks put it, is the same rule. We now make precise

what the choice changes, with a consistent heuristic and the notation of §2.6.

Proposition 3.7 (The ties change only the last layer). With a consistent heuristic and any rule for the ties, A* expands the same nodes with $g^*(u) + h(u) < C^*$ and none with $g^*(u) + h(u) > C^*$; the rules differ only in the nodes of the last layer, $g^*(u) + h(u) = C^*$, that they expand before a goal.

Proof. The claim is Theorem 2.14, whose proof does not depend on the order of the entries with equal keys. □

Proposition 3.8 (The ties with a perfect heuristic). Let the costs be at least some $\varepsilon > 0$ and $h = h^*$, and let a goal be reachable. With the ties of f broken in favor of the larger g, A* expands the nodes of one optimal path from s_0 to a goal, and no other node.

Proof. Since $h = h^*$, a node v has $g^*(v)+h(v) = C^*$ if and only if it lies on an optimal path, and $g^*(v)+h(v) > C^*$ otherwise; the heuristic is consistent ($h^*(u) \le c(u, v) + h^*(v)$ on every edge), and by Theorem 2.14 no node of the second kind is expanded. Let $x_0 = s_0, x_1, \dots$ be the nodes selected from entries that are not stale, in their order. We show by induction on j that, as long as x_j is not a goal, (a) x_{j+1} is a successor of x_j on an optimal path, with $g(x_{j+1}) = g(x_j) + c(x_j, x_{j+1}) > g(x_j)$, and (b) when x_j is selected, every entry of key C^* in OPEN has $g \le g(x_j)$; (b) holds for x_0, the only entry. Let (b) hold for x_j, not a goal. By Theorem 2.13, $g(x_j) = g^*(x_j)$, and since $h(x_j) = h^*(x_j)$ is finite, x_j has a successor y on an optimal path, with $g^*(y) = g(x_j) + c(x_j, y) > g(x_j)$. The node y is not closed: the nodes closed so far are $x_0, \dots, x_j$, whose values of g are at most $g(x_j)$ by (a), while a closed node has $g = g^*$ (Theorem 2.13). Nor did y have, before the expansion, the entry $(C^*, g^*(y), y)$, whose g exceeds $g(x_j)$, against (b). Hence the expansion of x_j sets $g(y) = g^*(y)$ and pushes that entry (line 13 or 16 of Algorithm 2.2). After the expansion, every entry has a key at least C^*; the entries of key C^* are those that were there, with $g \le g(x_j)$, and those just pushed, which have $g = g(x_j) + c > g(x_j)$ and belong to successors on optimal paths, and none of the latter is stale, since a value g^* cannot be improved. The next entry selected has the largest g among those of key C^*, which is (a) and (b) for $j + 1$. Since the values of g increase by at least ε from each x_j to the next and are at most C^*, the sequence reaches a goal after at most C^*/ε steps; the selected nodes form a path from s_0, each the successor of the previous one on an optimal path, and it ends at the first goal selected. □

In words, with the ties in favor of the larger g the search dives along the last layer, and with an exact heuristic it expands only the $d + 1$ nodes of the path it returns, d being its number of edges; an informative heuristic, exact on part of the last layer, gives a similar dive on that part (an observation, not a theorem). Conversely, the opposite rule does the reverse: since $h \ge 0$, a goal t has the largest g of the last layer, $g(t) = C^*$, and with the ties in favor of the smaller g it is selected only when no other entry of key C^* is left in OPEN, i.e., after all the nodes of the last layer that are generated before it have been expanded. On the grid of Figure 2.6 the two rules expand 8 and 31 of the 31 cells of the last layer; on the domains with unit costs and few values of f, such as the sliding-tile puzzles, the last layer may hold a large part of the nodes, and the experiments of Burns et al. [4] report that the larger g is the better rule. Among the entries with the same f and g, the bucket queue takes the last one pushed (Lemma 3.4), which follows the most recent path, while the heaps take any of them; this last order differs among the three sets of §3.12, which therefore may expand different nodes of the last layer, and only of it.

With real costs, two values of g that are sums of the same edge costs in another order may differ in their last bits, and a node closed with the larger one would be reopened with the smaller, and its subtree expanded again, although the two paths have the same cost; therefore, in every search of the library a path counts as cheaper only by more than $\tau \max\{1, g(v)\}$, $\tau = 10^{-9}$ (`cheaper()` and `cost_slack()` of `Node.h:134-148`), so that the cost returned exceeds C^* by at most $\tau \max\{1, C^*\}$ per edge of the path; on the random grid 1 of 60 by 60 cells with 8 moves A* thus selects 252 nodes instead of 276, 24 of which were reopened by the rounding alone. The order of OPEN, instead, compares f and g exactly, since a comparison up to a tolerance is not transitive and would break the order of a heap; hence, the values of f equal but for the rounding are taken in the order of their last bits, and the ties among them are not broken by g.

3.6 CLOSED: a hash table with open addressing

CLOSED is a dictionary from the generated states to their nodes, with no removal, since a search never forgets a state. The default `NodeTable` is a hash table with *open addressing* and *linear probing*: its places are an array of $m = 2^k$ cells, each holding the pointer to a node (nil in an empty place) and the 64-bit hash of its state, and a state is looked for from its first place onward, along consecutive places taken modulo m, up to its place or an empty one (Figure 3.5). Two choices make a probe cheap. The hash is stored, and hence a probe compares two states only when their hashes are equal, and a growth moves the places without touching the nodes. Also, the

places are consecutive in memory, and a probe sequence reads one or two lines of cache instead of following the pointers of a list.

The first place of a hash x in a table of 2^k places is

$$\left\lfloor \frac{(x \cdot M) \bmod 2^{64}}{2^{64-k}} \right\rfloor, \qquad M = \left\lfloor \frac{2^{64}}{\phi} \right\rfloor, \qquad \phi = \frac{1+\sqrt{5}}{2},$$

i.e., the top k bits of the product of x by M, which is the multiplicative hashing of Knuth with the golden ratio (*Fibonacci hashing*, cf. section 6.4 of [157]). The low bits of x are often regular: in a worker of HDA*, for instance, all the states have the same x mod p, and a table that took the low k bits of x would use one place in p. The product, instead, spreads all the bits of x into its top bits, and the states of a worker fill its whole table.

Algorithm 3.5 (The lookup with insertion in CLOSED; $S[0..m-1]$ the places, n the nodes stored).

```
1  function FIND-OR-INSERT(s, x, make)            ▷ x the hash of s
2     i ← the top k bits of (x · M) mod 2^64      ▷ m = 2^k
3     while S[i].node ≠ nil do
4        if S[i].hash = x and the state of S[i].node = s then
5           return (S[i].node, false)
6        i ← (i + 1) mod m
7     u ← make();  S[i] ← (x, u);  n ← n + 1
8     if 10 n > 7 m then GROW
9     return (u, true)

1  procedure GROW
2     T ← S;  m ← 2 m;  k ← k + 1;  S ← m empty places
3     for each place (x, u) of T with u ≠ nil do
4        i ← the first place of x
5        while S[i].node ≠ nil do i ← (i + 1) mod m
6        S[i] ← (x, u)
```

The lookup without insertion, `find`, is lines 2 to 6 with nil returned at the first empty place, and `prefetch( x )` issues a prefetch of the place of line 2. Thus the table doubles when its load factor $\alpha = n/m$ would exceed $7/10$, and the insertion is done by the same probe that finds the state absent, with one inspection only.

Lemma 3.9 (Correctness of the linear probing). After each operation of Algorithm 3.5, every node stored is in a place reached from the first place of its hash along occupied places only, and the load factor is at most $7/10$. Hence FIND-OR-INSERT returns the node of s if s is stored, and otherwise inserts it, and `find` returns nil exactly for the states not stored.

Proof. A place, once occupied, is never emptied, since there is no removal; hence the places between the first place of a node and its place, all occupied when it was inserted, stay occupied. GROW reinserts every node by the same probe in an empty table, which gives the property again. The load factor is at most $7/10$ after line 8, and at most $7/20$ after a doubling. Since $\alpha < 1$, the table has an empty place, and the loop of line 3 ends. If s is stored, its place is on the probe sequence from its first place before any empty place, and the loop meets it (the hash compared at line 4 is the one stored with it); if s is not stored, the loop reaches an empty place. □

Lemma 3.10 (The size for n nodes). A table built with `slots_for( n )` places, the least power of 2, at least 16, with $10n \le 7m$, holds n nodes without growing.

Proof. The constructor takes the least power of 2 that is at least the given number (and at least 16), here `slots_for( n )` itself, and hence $7m \ge 10n \ge 10n'$ after each of the first n insertions, n' the nodes stored; the test of line 8 then fails for each of them. □

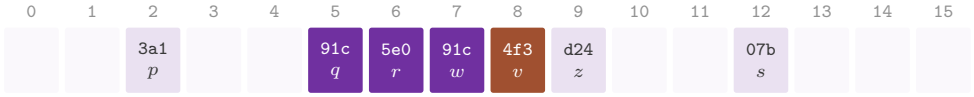
`SearchOptions::expected_nodes` gives the number n of nodes for which `AStar` builds its table with `slots_for( n )` places at the start of a search (§2.3), and by Lemma 3.10 a search that stores at most that many nodes never grows its table; otherwise the table starts at 1024 places. We recall the classical analysis of linear probing for the cost of a lookup.

Proposition 3.11 (The cost of CLOSED). With hash values independent and uniformly distributed, and a load factor $\alpha < 1$, a successful lookup inspects about $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ places on average and an unsuccessful one about $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$; with $\alpha \le 7/10$ these are at most about 2.2 and 6.1 places. The doublings cost $O(1)$ amortized time per insertion, besides the probes of the reinsertions, without any hypothesis on the hashes.

Proof. The two averages are the classical results on linear probing (section 6.4 of [157]), and at $\alpha = 7/10$ they are $\frac{1}{2}(1 + 10/3) \approx 2.17$ and $\frac{1}{2}(1 + 100/9) \approx 6.06$. A doubling from m to $2m$ places scans the m places, writes $2m$ empty ones and reinserts at most $\frac{7}{10}m + 1$ nodes, i.e., it costs $O(m)$ besides the probes of the reinsertions; it follows at least $\frac{7}{20}m$ insertions since the previous doubling, i.e., at least $\lfloor 7m/10 \rfloor - \lfloor 7m/20 \rfloor \geq 7m/20 - 1$ of them (and $\lfloor 7m/10 \rfloor - 1$ before the first one, from an empty table), to which its cost is charged, $O(1)$ each for $m \geq 16$. □

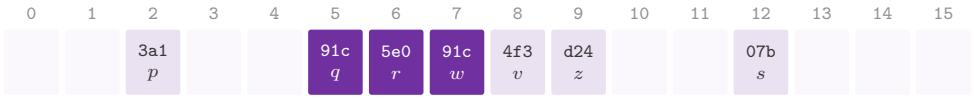
We remark that the hypothesis on the hashes is an assumption on the member `hash` of the domain, which the library does not check: a domain whose hashes collide often makes the probe sequences longer, and the averages above then do not hold. The unsuccessful lookups are those of the new states, i.e., one per node stored, and the successful ones those of the duplicates. A probe sequence of 6 places of 16 bytes spans 96 bytes, i.e., two or three lines of cache. In the set `std`, CLOSED is a `std::unordered_map` (a table of chains) whose key is the pair of the state and of its hash, the hash of the key being the one stored in it and the equality that of the states; the map thus does not evaluate the hash of the domain again, and `try_emplace` gives the insertion with one lookup. Its `prefetch` does nothing, since a table of chains cannot prefetch a bucket. In the set `boost`, CLOSED is `boos t::unordered_flat_map`, a table with open addressing, with the same key, reserved for 7/10 of its places as the default one; its `prefetch` does nothing either. We remark that when the default data structures are built with the headers of Boost (`ADS_WITH_BOOST`), the table of the set `ads` keeps the interface above, but its places are those of `boost::unordered_flat_set`: its load factor stays below 7/8, a table of `slots_for( n )` places still holds n nodes without growing, and its `prefetch` does nothing.

(a) `find_or_insert`(v), hash `4f3`: the first place is 5



places 5, 6, 7 hold other hashes, compared without reading their states; the empty place 8 receives v

(b) `find`(w), hash `91c`: the first place is 5



place 5 has the same hash but another state (one comparison of states); place 7 holds w

Figure 3.5. Linear probing in CLOSED, whose places hold a hash (here 3 hexadecimal digits) and a pointer to a node. (a) The insertion of v, whose first place is 5: the places 5, 6 and 7 hold other hashes, compared without reading their states, and v goes into the empty place 8. (b) The lookup of w, of the same first place: the place 5 has the same hash and another state, which costs one comparison of states, and the place 7 holds w.

3.7 The arena of the nodes

A node (`Node`, `Node.h:73-89`) has the following fields, in this order: the state; g, the cost of the best path found from s_0; h; the pointer to its parent on that path (nil at the root); the flag *closed*, set when the node is expanded with its current g and cleared when g improves; the flag *expanded*, set at the first expansion and never cleared, by which the re-expansions are counted (§2.3); the operator that generated the state, of one byte (`no_o p` at the root, used only by an `OperatorDomain`); and a 32-bit key of the state for its owner, used only by the distributions of the parallel searches that follow the moves (Chapter 9). The member `f()` returns $g + h$. With a state of one word and integer costs of 32 bits, a node takes 32 bytes: 8 for the state, 4 each for g and h, 8 for the parent, and 8 for the two flags, the operator and the key, which fit in what would otherwise be the padding after the flags. The infinite cost of a cost type, which stands for no incumbent and for the bound of a search not yet cut, is `infinite_cost()` (`Node.h`), i.e., the infinity of a floating-point type and the largest value of an integer one; the path to a goal is rebuilt by `trace_path`, which follows the parents from the goal and reverses the states.

The nodes are allocated from an *arena* (`NodePool`): a block of $B = 2^{16}$ nodes is taken from the system when the previous one is full, and a node is allocated by advancing an index in the block, without a call to the allocator (Figure 3.6).

Algorithm 3.6 (The allocation of a node from the arena).

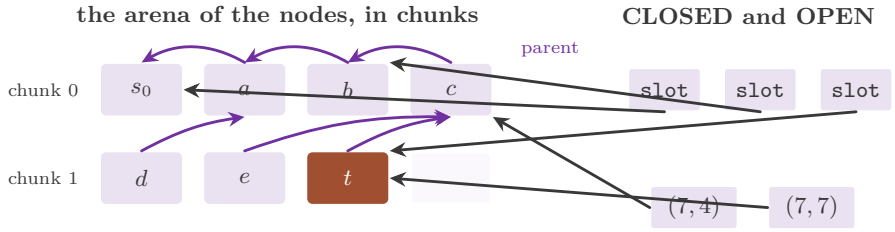
```
1  function ALLOCATE()
2     if there is no block or used = B then
3        append a new block of B nodes;  used ← 0
4     used ← used + 1
5     return the place used − 1 of the last block, value-initialized
```

Proposition 3.12 (The arena). The address of a node never changes until the arena is destroyed, a call of ALLOCATE costs $O(1)$ time besides the allocation of a block once in B calls, and N nodes take the memory of $\lceil N/B \rceil$ blocks, i.e., of at most $B - 1$ nodes more than N.

Proof. A block is never moved nor freed before the arena, and each node is built in place in a block (line 5); the other claims follow from line 2, which takes a new block only when the last one is full. □

The arena has some consequences. The pointers to a node (from CLOSED, from the entries of OPEN and from its children as their parent) stay valid until the end; hence CLOSED and OPEN can store pointers, and neither moves a node. Also, the nodes made one after another are next to each other in memory, and the children of a node, which are made together, often share lines of cache; furthermore, a node costs no header of the allocator. With nodes of 32 bytes a block is 2 MB, i.e., one huge page. The price is that the nodes are freed only all together, at the end of the search, and that the arena does not reuse the nodes of the states that are closed for good. In the set `std` the nodes are kept in a `std::deque`, whose `emplace_back` does not move the nodes already there either, and in the set `boost` in a `boost::object_pool`.

The blocks of the arena, the places of CLOSED and the array of a large heap of the default set are requested from the operating system as *huge pages* where possible: a request of at least 2 MB is aligned to 2 MB, rounded up to whole huge pages and, on Linux, advised to the kernel for the transparent huge pages, which the kernel may ignore; a smaller request goes to the standard allocator. A structure whose places are reached in no useful order, as those of CLOSED by hashes, then misses the translation lookaside buffer once per 2 MB instead of once per 4 KB page, and its memory is faulted in a huge page at a time.



the nodes never move: the pointers of CLOSED, of the entries of OPEN and of the parents stay valid; a full chunk is followed by a new one (here t is the first node of chunk 1 after e and d)

Figure 3.6. The arena of the nodes, in chunks: each node points to its parent (the arcs above), the places of CLOSED and the entries of OPEN point to the nodes, and no node ever moves; a chunk that is full is followed by a new one, from which the next node (t) is allocated.

3.8 Prefetching the lookups

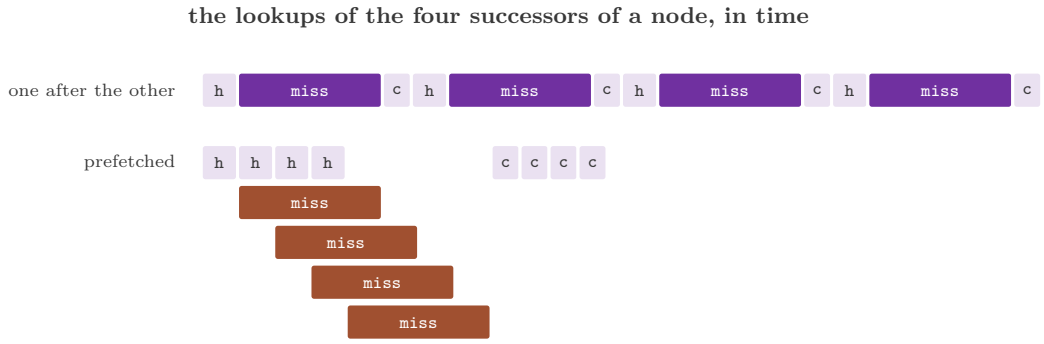
On a large search, CLOSED is much larger than the cache, and the place of a successor lies at an address that the hash makes random; almost all the lookups then miss the cache, and the time of an iteration is dominated by b misses. For this reason, the sequential A* splits the expansion of lines 9 to 16 of Algorithm 2.2 in two passes.

Algorithm 3.7 (The expansion of u in two passes).

```
1  procedure EXPAND(u)
2     K ← ∅                                        ▷ the successors of u
3     for each successor v of u do                 ▷ the first pass
4        x ← hash(v);  prefetch the first place of x in CLOSED
5        append (v, c(u, v), h(v), x) to K
6     for each (v, c, h(v), x) in K do           ▷ the second pass
7        (w, new) ← FIND-OR-INSERT(v, x, a node of v with g(u) + c)
8        if new then push w in OPEN
```

```
 9        else if g(u) + c < g(w) then
10           g(w) ← g(u) + c;  parent(w) ← u;  w is not closed
11           push w in OPEN
```

The first pass generates all the successors, computes their hashes and issues a prefetch of the first place of each hash into the cache, without waiting for it. The second does the lookups, by which time the lines are on their way or already there. A prefetch is a hint to the processor, which changes no value; hence Algorithm 3.7 makes the same lookups, insertions and pushes as lines 9 to 16 of Algorithm 2.2, in the same order. Its gain, on the other hand, is not a property that we can prove: on a processor that serves several misses at a time, the misses of the b lookups overlap (Figure 3.7), and the iteration waits for about one miss instead of b, while on one that serves a single miss at a time nothing is gained. Also, the parallel searches do the same with the batches of nodes they receive (Chapter 9), whose lookups are independent of each other. The price is the memory of K, i.e., b records, reused from one expansion to the next.



h: hash and prefetch of the place; miss: the line comes from memory; c: comparison and insertion. Prefetched, the four misses overlap, and the lookups wait for about one miss instead of four

Figure 3.7. The lookups of the four successors of a node in time: one after the other, each waits for its miss (above); prefetched, the four misses overlap after the four hashes, and the comparisons follow (below).

3.9 Perfect hashing

A *perfect hash* of the states is an injective map of the states to the integers, and a *minimal* one maps n states onto $[0, n)$. A domain with a perfect hash models the concept `PerfectHashDomain` (`Domain.h:230`), whose `perfect_hash( s )` the distribution HDA*[P] of Burns et al. [3] uses to assign the states to the workers (Chapter 9). On a grid of width W and height H, the cell (x, y), with $0 \le x < W$ and $0 \le y < H$, has the index $x + Wy$, and the state of the domain `Grid` is that index itself (Figure 3.8).

Lemma 3.13 (The perfect hash of a grid). The map $(x, y) \mapsto x + Wy$ is a bijection of the cells of a grid of width W and height H onto $[0, WH)$, i.e., a minimal perfect hash.

Proof. Since $0 \le x \le W - 1$, the index $i = x + Wy$ satisfies $Wy \le i \le Wy + W - 1$, and hence $y = \lfloor i/W \rfloor$ and $x = i \bmod W$ are determined by i, i.e., the map is injective. Its values lie in $[0, W(H-1) + W - 1] = [0, WH)$, and the WH cells are thus mapped injectively onto these WH integers. □

The explicit graphs use the index of the node, and the traveling salesperson, the vacuum world and the missionaries and cannibals pack their states into words that are their perfect hashes. On the permutations, the lexicographic ranking of §5.4 is a minimal perfect hash of the placements of a pattern, which the pattern databases use as the indices of their tables (the ranking of Myrvold and Ruskey [34] would serve as well, cf. §5.4).

With a minimal perfect hash and a known number of states, CLOSED could be a plain array indexed by it, with no probe at all; the library keeps the hash table for all the domains, which does not depend on the range of the hash nor on the number of states, and leaves the perfect hashes to the distributions and to the tables of the heuristics.

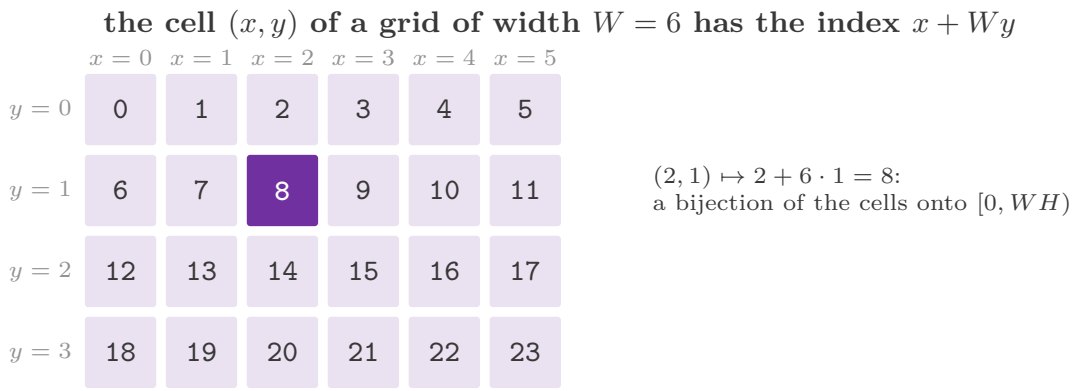


Figure 3.8. The perfect hash of a grid: the cell (x, y) of a grid of width $W = 6$ has the index $x + Wy$, a bijection of the cells onto $[0, WH)$.

3.10 Transposition tables

A *transposition table* is a CLOSED that may forget: it has a fixed number of places, and when it is full a new state replaces an old one by a replacement rule (e.g., the one with the smaller g, or the one reached less often), or the new state is not stored. Such a table gives the depth-first searches, which keep no CLOSED, part of the duplicate detection of a graph search within a bounded memory: a state found in the table with a g not larger is not searched again (that this keeps IDA* optimal is Proposition 4.10). With the *stochastic node caching* of Miura and Ishida [162], a state is stored with a probability p at each visit, the draws being independent; a state visited n times is then in the table with probability $1 - (1 - p)^n$, since it is missing only if all the n draws fail, and the states reached by many paths, which are those whose detection saves the most, are stored with a high probability.

> **Status — planned.** The library has no transposition table; its depth-first search, IDA*, detects only the move back to the parent (§2.9 and Chapter 4).

3.11 Immediate and delayed duplicate detection

The searches of Chapter 2 detect the duplicates *immediately*: each successor is looked up in CLOSED when it is generated, and it is dropped at once if its state is known with a g that is not larger. A search with *delayed* duplicate detection instead appends the successors to a buffer, or to a file on the disk, without looking them up, and removes the copies later, in a batch, by sorting or hashing the buffer and comparing it with the states already expanded (Figure 3.9). The batch replaces the random lookups by sequential passes, as the disk requires, and it lets CLOSED live outside the memory. Its price is that the copies are stored until the batch, and that the order of the expansions is that of the batches, not of the single nodes. The search of Chapter 4 (`DDDAStar`) keeps its buckets in files and detects the duplicates of a file at once, before expanding it, so that each state is expanded at most once (Theorem 4.15); the test `ddd` checks it, and checks that the records of a search that exhausts its buckets split, with the root, into the states stored and the copies dropped.

3.12 The sets of data structures

The arena, CLOSED and the OPEN lists of A* come from one of three sets, chosen when the library is built (`Structures.h`): the default one; that of the standard library, chosen by the macro `PBEFS_STRUCTURES_STD`; and that of Boost, chosen by `PBEFS_STRUCTURES_BOOST`. CMake defines the macro from its option `PBEFS_STRUCTURES`, whose values are `std`, `boost` and the name of the default set (the default; any other value stops the configuration), and the makefile from `PBeFSSTRUCT` (`std`, `boost`, or empty for the default set); defining more than one macro is an error. `Node.h`, `NodeTable.h`, `OpenList.h` and `ConcurrentClosed.h` then include the classes of the set from `structures/std/`, `structures/boost/` or the default directory, with the same interface (the adjacency of the explicit graphs has a version in `structures/boost/` and in the default directory, that of the grids only in the default one, and the other sets use the adjacency written in `ExplicitGraph.h` and `Grid.h`); the sets are the following.

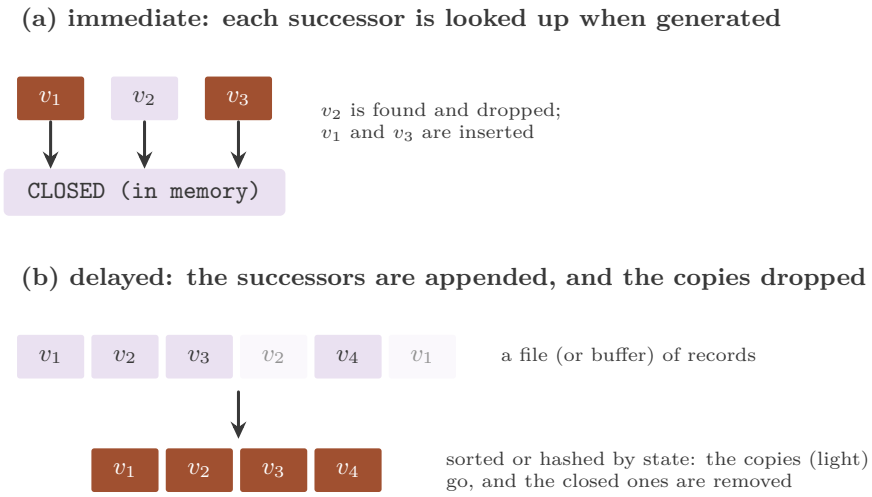


Figure 3.9. Immediate and delayed duplicate detection. (a) Each successor is looked up in CLOSED when it is generated. (b) The successors are appended to a buffer or a file, copies included, and the copies are removed later in a batch, together with the states already closed.

set	NodePool	NodeTable	HeapOpenList	BucketOpenList
default	arena, huge pages	linear probing	4-ary heap	two levels
std	std::deque	std::unordered_map	binary heap	binary heap
boost	object_pool	unordered_flat_map	4-ary heap	4-ary heap

The searches are the same with all the sets, and so is the order of their expansions, but for the ties of f and g in OPEN, which the sets break differently; by Proposition 3.7, with a consistent heuristic they expand the same nodes but in the last layer, and the costs returned are the same. A search is parametrized by its OPEN list at compile time, and the function `astar` chooses it at run time by an `OpenKind`; their declarations (`AStar.h`, here without the bodies) are

```
template< Domain D , class Open = std::conditional_t<
                      std::is_integral_v< cost_t< D > > ,
                      BucketOpenList< Node< state_t< D > , cost_t< D > > > ,
                      HeapOpenList< Node< state_t< D > , cost_t< D > > > > >
class AStar {
 public:
 explicit AStar( const D & domain , const SearchOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
astar( const D & domain , OpenKind open = OpenKind::automatic ,
       const SearchOptions & options = {} );
```

with the kinds

```
/// the kind of OPEN list a search uses
enum class OpenKind {
 automatic ,  ///< buckets for integer costs, heap otherwise
 heap ,       ///< HeapOpenList
 buckets ,    ///< BucketOpenList (integer costs only)
 heap_low_g   ///< HeapOpenList with the ties to the smaller g [see
              ///< LowGOpenList]
 };
```

Thus `AStar< D , Open >` takes `BucketOpenList` by default for integer costs and `HeapOpenList` otherwise; in `a star`, `automatic` and `buckets` give `BucketOpenList` for integer costs, `heap` gives `HeapOpenList`, and with costs that are not integers all three give `HeapOpenList` (a request of `buckets` is then silently served by the heap). The `SearchOptions` are those of Chapter 1, of which `expected_nodes` sizes CLOSED (Lemma 3.10). The arena and CLOSED have no parameter, being those of the set. Furthermore, the parallel searches use the same classes in each worker, and their shared structures (e.g., the CLOSED of SPA*) are described in Chapter 8. We keep the standard library and Boost to measure the searches against them, with the script of the next section.

An example

The following example, a case of the tester, solves an instance of the 15-puzzle with the two OPEN lists chosen at run time and with the heap chosen at compile time, the table of CLOSED being sized by `expected_nodes`; all the costs are those of the instance, and the automatic choice is the bucket queue, with the same expansions.

```
const FifteenPuzzle d( ki.board );
SearchOptions o;
o.expected_nodes = 1 << 20;                // CLOSED sized at the start
const auto b = astar( d , OpenKind::buckets , o );     // the buckets
const auto h = astar( d , OpenKind::heap , o );        // the heap
const auto a = astar( d , OpenKind::automatic , o );   // here the buckets
// the same choice at compile time, by the OPEN list of AStar
using NodeT = Node< state_t< FifteenPuzzle > , cost_t< FifteenPuzzle > >;
const auto e = AStar< FifteenPuzzle , HeapOpenList< NodeT > >( d , o )
               .solve();
```

3.13 Benchmark

Status — planned. The table of this section is not yet produced by the tables of the manual.

The script `tools/bench/structures.sh` runs `tools/AStarBench` from three builds of the library, one per set of data structures, on instances of growing size: the 15-puzzle in four bands of the instances of Korf [7], by the nodes that A* expands (about 10^5, 10^6, 10^7 and above $7 \cdot 10^7$), and random grids of 2000 to 20000 cells a side, with 35% of the cells blocked and 4 and 8 moves. Each instance is solved by the sequential A* (with both OPEN lists on the 15-puzzle) and by the parallel searches (by default HDA* with the Zobrist distribution, whose structures are those of each worker, and SPA*, whose CLOSED is shared) on 1, 2, 4, 8 and 16 threads, twice with each set, the sets taking turns so that a drift of the machine falls on all of them; each run is a process of its own, pinned to the cores and the memory of one node, and the two runs of a set are its A/A. As in §2.16, the clock is that of `solve()`, which does not include the reading of the instance nor the construction of the domain. Since the expansions are the same for the three sets but for the last layer, the comparison is one of time and of memory per node.

3.14 Notes and references

The bucket queue for integer keys is due to Dial [65], and its nested use with two levels, the ties broken in favor of the larger g and the pruning of the move back to the parent in a fast A* are studied by Burns et al. [4], who report the bucket queue to be faster than a binary heap on the sliding-tile puzzles (they update the position of a node in OPEN when its g improves, where the library uses the lazy deletion); the textbook of Edelkamp and Schrödl [10] treats the priority queues, the hash tables and the duplicate detection of the searches at length. The d-ary heaps go back to Johnson [158], the radix heap is due to Ahuja et al. [160], the linear probing and the multiplicative hashing are analyzed in section 6.4 of [157], and the arenas, i.e., the allocation by lifetime, are those of Hanson [159]. The delayed duplicate detection on the disk is that of Korf [40] and of the External A* of Edelkamp, Jabbar and Schrödl [39], and the stochastic node caching that of Miura and Ishida [162]. With respect to [4], the library adds the prefetch of the lookups, the huge pages, and the sparse lists of the large values of g in the bucket queue, which keep the queue usable on the grids and in the workers of HDA*; it does not reuse the nodes of the closed states, which the arena keeps until the end, and it has neither the radix heap nor the transposition tables.

3.15 Exercises

Exercise 3.1 (`ctest -R PBeFS.open`). Show that the order of `BucketOpenList` (least f, then largest g, then the last pushed) and that of `HeapOpenList` agree on the pairs (f, g) that they pop, and explain why the test compares the pops with an ordered multiset of the pairs $(f, -g)$ rather than of the entries.

Exercise 3.2 (`ctest -R PBeFS.open`). In a worker of HDA* the least f of OPEN may fall below the f of the last pop. Give a sequence of m pushes and pops on which, by Proposition 3.5 (i), the scan of the layers costs $\Theta(mf_{\max})$, and explain what the sparse lists of the values $g \geq 256$ save in the same situation.

Exercise 3.3 (`ctest -R PBeFS.astar`). On the 8-puzzle, break the ties of f by the least h and then by the largest h, and compare the nodes expanded; show by Proposition 3.7 that the two rules differ only in the last layer, and that the costs are equal, as the test checks for the two OPEN lists of the library.

Exercise 3.4 (`ctest -R PBeFS.mqa-thread`). Prove Lemma 3.6 also when $g + h$ reaches $2^{32} - 1$, and show that it fails when $g + h \geq 2^{32}$. Which representation of the key should a domain whose costs exceed that range use? (MultiQueue A*, whose test runs here, orders its queues by these keys.)

Exercise 3.5 (`ctest -R PBeFS.hda-thread`). Show, from Lemma 3.13, that the distribution HDA*[P], which takes the perfect hash modulo the number p of the workers, gives each worker the cells of whole columns when p divides W. The test runs HDA*[P] on the 15-puzzle, with the other distributions.

Exercise 3.6 (`ctest -R PBeFS.ddd`). Compare, on the 8-puzzle, the nodes expanded and generated by A* with immediate duplicate detection and by the search with delayed duplicate detection of Chapter 4; explain why the second expands each state at most once but may expand more states, and why, on a search that ends without a goal (as on the grid with the wall of the test), its records satisfy `generated` $+1 =$ `stored` $+$ `duplicates`.

Exercise 3.7 (`ctest -R PBeFS.astar`). With a load factor of at most 7/10, compute the memory of CLOSED per node stored, and that of the nodes in the arena, for the 15-puzzle (a state of 8 bytes and integer costs); how much memory does a search that stores 10^8 nodes need, without OPEN? How much does the power of 2 of `slots_for( n )` (Lemma 3.10) add to $10n/7$ places, at worst?

4. Memory-bounded and external search

[Source: include/PBeFS/astar/IDAStar.h (IDAStar, IDAOptions, idastar, cost_bounded_dfs), include/PBeFS/astar/Deepening.h (DeepeningOptions, CutoffHistogram), include/PBeFS/astar/BeamSearch.h (BeamSearch, BeamOptions, beam_search), include/PBeFS/astar/RBFS.h (RBFS, rbfs), include/PBeFS/astar/PartialExpansion.h (PEAStar, EPEAStar, PartialOptions, PartialExpansionBase, PartialDelta, PartialNoExtra, pea_star, epea_star), include/PBeFS/astar/FringeSearch.h (FringeSearch, fringe_search), include/PBeFS/astar/RealTimeSearch.h (RealTimeSearch, RealTimeOptions, lss_lrta_star), include/PBeFS/astar/HPAStar.h (HPAStar, HPAOptions), include/PBeFS/astar/BreadthFirst.h (BreadthFirst, breadth_first), include/PBeFS/astar/DDDAStar.h (DDDAStar, DDDOptions, DiskDomain, ddd_astar), include/PBeFS/astar/Domain.h (OperatorDomain, TreeOperatorDomain, PartialExpansionDomain, for_each_child, for_each_tree_child), include/PBeFS/astar/Solution.h (SearchOptions, Status); the tests idastar, bfs, classic, realtime, hpa, ddd and tiles24 of test/test.cpp]

We are concerned in this chapter with the searches that find a path when the nodes generated by A* do not fit in memory, or when the time of A* is not available. A* (Chapter 2) stores all the nodes that it generates, in OPEN or in CLOSED, and on the larger puzzles this is its real limit: the 15-puzzle has $16!/2 \approx 1.05 \cdot 10^{13}$ reachable states, and Korf and Schultze needed disks and a parallel breadth-first search to visit them all once [156]. One can give up part of what A* keeps in several ways, and the chapter is organized by them. A search may keep only a bounded number of nodes and accept to lose completeness (beam search, §4.2), keep only the current path and give up the detection of the duplicates (depth-first branch-and-bound and iterative deepening, §4.3 and §4.4), keep the current path with the values backed up along it (recursive best-first search, §4.5), keep everything but on the disk, detecting the duplicates in batches (§4.6), represent sets of states compactly as Boolean functions (§4.7), or prune the states that bring nothing new in terms of features (§4.8). Then come the searches that save the nodes that A* stores without need, i.e., the successors whose f exceeds C^* (partial expansion, §4.9), or the cost of OPEN itself (fringe search, §4.10); the bounds by which iterative deepening repeats fewer iterations (§4.11); the searches that give up the optimality to bound the time before a move, by a search of bounded size around the agent (§4.12) or by an abstraction of the map computed once (§4.13); and the two baselines of the library, breadth-first search and the cost-bounded depth-first search (§4.14). The library implements all of them but the depth-first branch-and-bound of a Domain, the transposition tables and the stochastic node caching of IDA* and the symbolic search, which are described with their theory in sections marked as planned. We use the notation of Chapter 1 and the results on A* of Chapter 2, in particular the invariant of OPEN (Lemma 2.7) and the monotonicity of f under a consistent heuristic (Lemma 2.12); $\varepsilon > 0$ is a lower bound on the costs of the edges whenever one is assumed, b the branching factor and d the number of edges of a path.

4.1 What limits A*

With an admissible heuristic A* expands (at least) every node with $f < C^*$ when h is consistent (Theorem 2.14), and it stores, besides them, all the successors that it has generated; the memory therefore grows as the number of the generated nodes, i.e., roughly b times the expansions. In the library each node holds the state, g, h, the parent and a few flags (§3.7), plus its place in CLOSED and its entries in OPEN; with a state of one or two words this is some tens of bytes per node, and a search that generates 10^{10} nodes needs a memory that few machines have. The time of A*, on the other hand, usually remains acceptable well after its memory is exhausted, since an expansion costs little more than the generation of its successors and a few lookups. Hence the methods of this chapter trade time for memory, and the table below summarizes what each one keeps and what it guarantees.

Method	Keeps	Optimal	Complete	In the library
A*	OPEN and CLOSED	yes	yes	`AStar`
beam search	CLOSED, B nodes per layer	no	no	`BeamSearch`
depth-first B&B	the path	finite tree	finite tree	trees only
DFID, IDA*	the path	yes	yes	`IDAStar`
RBFS	the path and the siblings	yes	yes	`RBFS`
external A*	OPEN and CLOSED on the disk	yes	yes	`DDDAStar`
symbolic search	sets of states as BDDs	yes	yes	planned
width-based search	tuples of features	no	no (IW(i))	`IteratedWidth`
PEA*, EPEA*	the nodes of $f \leq C^*$	yes	yes	`PEAStar`, `EPEAStar`
fringe search	CLOSED and a list	yes	yes	`FringeSearch`
IDA*-CR	the path	yes, or bounded	yes	`IDAStar`
LRTA*, LSS-LRTA*	the learned H	after trials	yes	`RealTimeSearch`
HPA*	an abstract graph	no	no	`HPAStar`
breadth-first search	CLOSED	unit costs	yes	`BreadthFirst`

The column "Optimal" assumes an admissible heuristic (and, for the external A* and EPEA*, a consistent one), the depth-first branch-and-bound of the library is that of the trees of Chapter 13, and the conditions under which each method terminates are given in its section; for IDA*-CR "bounded" is the bound on the suboptimality of Proposition 4.26 when the last iteration is not completed, for the real-time searches the optimality is that of the limit of the trials, and their completeness needs a goal reachable from every state (§4.12).

4.2 Beam search

A beam search bounds the memory by keeping, at each layer of a breadth-first search, only the B nodes that are best according to an evaluation, usually f or h; the others are dropped for good, i.e., never expanded, although the library keeps them in CLOSED, as in the beam search of Zhou and Hansen [117] (section 2), so that a later layer does not pay for them again.

Definition 4.1 (Beam search). Let $L_0 = \{s_0\}$. For $d \geq 0$, let S_d be the set of the successors of the nodes of L_d that are new or reached through a cheaper path, and let L_{d+1} be the B nodes of S_d with the least evaluation (all of them if they are at most B), the ties going to the node generated first. The search stops at the first goal generated, or when a layer is empty.

`BeamSearch` (`BeamSearch.h`) is the breadth-first beam search of Lowerre [171] in the form of [117], whose pseudocode is that of the header.

Algorithm 4.1 (Breadth-first beam search of width B).

```
1  procedure BEAM(s0, B)
2     layer ← ⟨s0⟩;  CLOSED ← {s0}
3     while layer ≠ ⟨⟩ do
4        next ← ⟨⟩
5        for each node n of layer, each successor c of n do
6           if c is a goal then return the path to c
7           if c ∉ CLOSED or it is reached through a cheaper path then
                CLOSED ← CLOSED ∪ {c};  append c to next
8        layer ← the B best nodes of next, by key and then by order
```

The key of line 8 is $f = g + h$ by default and h with `BeamOptions::greedy`; the order is that of the generation, kept by a stable sort. The goal is tested when a node is generated (line 6), and s_0 before the loop, since a node that is not kept would never be tested again; a successor of infinite h, i.e., a dead end, is not kept, and a node

reached through a cheaper path after its expansion is reopened, i.e., it goes back to the next layer with the new g (a node that was appended to `next` with a g that a later generation in the same layer lowered is appended once, with the lower one). The options are

```
struct BeamOptions : public SearchOptions {

 std::size_t width = 100;      ///< the nodes kept in a layer, B
 bool greedy = false;          ///< the key is h instead of f

 };  // end( struct( BeamOptions ) )
```

where a width of 0 is taken as 1, and the declarations of the search, quoted by hand, are

```
template< Domain D >
class BeamSearch {
 public:
 explicit BeamSearch( const D & domain , const BeamOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 [[nodiscard]] std::size_t layers( void ) const noexcept;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
beam_search( const D & domain , const BeamOptions & options = {} );
```

with `layers()` the layers expanded by the last search. Of the `SearchOptions`, `node_limit` bounds the nodes of CLOSED, tested after each layer, and `expected_nodes` sizes CLOSED; `stored` is the size of CLOSED, and `expanded` counts the nodes of the layers.

Proposition 4.2 (Memory and incompleteness of beam search). If the goal is generated in the layer d, Algorithm 4.1 holds at most $1 + dBb$ nodes in CLOSED, i.e., a number linear in the depth, and at most B nodes in a layer plus the Bb successors of the layer being expanded; it is not complete, and when it finds a goal the path need not be optimal. With an infinite width it is a breadth-first search, and the first goal generated is one of least depth.

Proof. A layer has at most B nodes and each has at most b successors, of which only those generated enter CLOSED; the layers expanded before the goal are d. For the incompleteness it suffices to drop, in some layer, the only node from which a goal is reachable, as in Figure 4.1; for the optimality, a goal at depth d may be found before a cheaper one at a larger depth, since the layers are taken by depth, whatever their cost. With $B = \infty$ no node is dropped, and line 8 keeps every node of `next`; by induction on d, the layer d then holds every state at the least depth d that is not a goal (a state is first generated from the layer before its least depth, and it may enter a later layer again only when its g falls), and the goals of the least depth D are generated by the expansion of the layer $D - 1$, before which no goal is generated. □

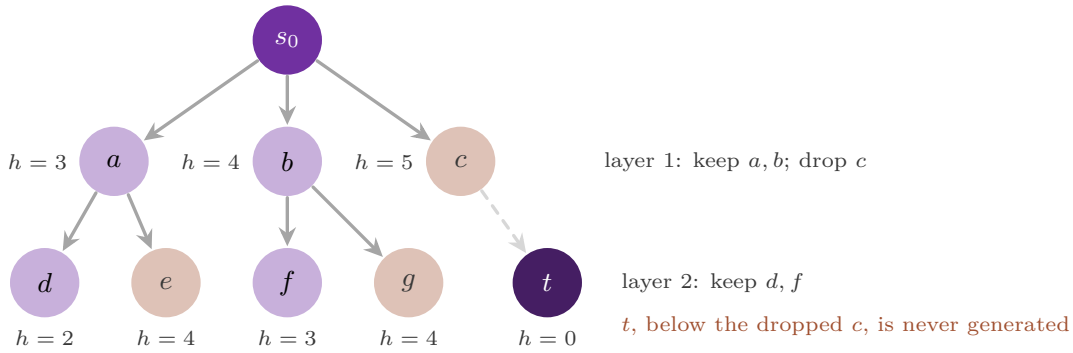


Figure 4.1. Breadth-first beam search of width $B = 2$ by h. In each layer the two nodes of least h are kept and the others are dropped (rust); the goal t, a successor of the dropped node c, is never generated, although it is the best node of the next layer.

Also, the width B trades the quality of the path for the memory, and the closed list of the library helps on graphs with many cycles, where the same state would otherwise fill several layers. We remark that the reopening

changes the parent of a node after its successors have been generated with the old g: the cost returned is the g of the goal when it was generated, whereas the path rebuilt from the parents at the end may be another one, with more edges and a smaller cost (e.g., if a, reached first by an edge of cost 10 from s_0, generates c and then gets the parent b at the cost 2, the goal generated from c in the next layer is returned with the path through b and the cost through the edge of cost 10). Edelkamp and Schrödl [10] describe the other variants (e.g., beams on f instead of layers), and the beam-stack search of Zhou and Hansen [117], which backtracks over the beams to become complete, is not in the library. The test `classic` checks that the beam of infinite width is optimal on 200 states of the 8-puzzle, whose costs are uniform, that a beam of 2000 nodes returns a valid path on instances of the 15-puzzle of Korf, and that a graph with no path is reported unsolvable.

4.3 Depth-first branch-and-bound

> **Status — planned.** The depth-first branch-and-bound of a `Domain` is not in the library; the branch-and-bound searches of Chapter 13 offer it on a tree given by its branching, with the depth-first selection.

Depth-first branch-and-bound (DFBnB) searches the tree of the paths from s_0 depth first, with the successors of a node taken by increasing h, and prunes each node whose f is not less than the cost U of the best goal found so far (the *incumbent*); the memory is that of the current path and of the successors not yet tried along it.

Algorithm 4.2 (Depth-first branch-and-bound).

```
1  procedure DFBNB(u, g)
2     if g + h(u) ≥ U then return                          ▷ pruned
3     if u is a goal then U ← g;  record the path;  return
4     for each successor v of u, by increasing h(v) do
5        DFBNB(v, g + c(u, v))

   U ← ∞;  DFBNB(s₀, 0);  return the recorded path, if U < ∞
```

Proposition 4.3 (Optimality and memory of DFBnB). On a finite tree of depth D, with h admissible, Algorithm 4.2 terminates, returns a path of cost C^* if a goal exists, and holds $O(bD)$ nodes at a time.

Proof. The tree is finite and each node is visited at most once, hence the search terminates. Let P be an optimal path; the nodes u of P have $g(u)+h(u) \le C^*$, by admissibility. While $U > C^*$, no node of P is pruned at line 2, and the depth-first search, which visits each node that it does not prune, reaches the goal of P and sets $U = C^*$; afterwards U never increases, and a goal recorded later would need $g < U$. Hence the recorded path costs C^*. The recursion has at most $D+1$ levels, each holding the at most b successors of its node sorted by h. □

Since the first goal is often found early and U only decreases, DFBnB is an "anytime" search, which can be stopped with the best path found so far; on a graph with cycles, however, the tree of its paths is infinite, and a depth bound (or an initial U) is needed for it to end. On a graph, moreover, DFBnB expands a state once per path that reaches it within the bound, the defect that the next section treats.

4.4 Iterative deepening

Depth-first iterative deepening (DFID, Korf [7]) runs a depth-first search bounded by the depth $1, 2, 3, \ldots$ until a goal is found; with unit costs the first goal found is at the least depth, as with breadth-first search, while the memory is that of the current path. IDA*, also of Korf [7], replaces the depth by $f = g + h$ and raises the bound, at each iteration, to the least f that exceeded the previous one; it is DFID when $h = 0$ and the costs are 1 (Corollary 4.6).

The interface

`IDAStar` (`IDAStar.h`) searches a `Domain` with IDA*; its declarations, without the bodies, are

```
struct IDAOptions : public SearchOptions , public DeepeningOptions {
 IDAOptions( void ) = default;
 IDAOptions( const SearchOptions & o );
 };
```

```
template< Domain D >
class IDAStar {
 public:
 explicit IDAStar( const D & domain , const IDAOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 [[nodiscard]] Solution< state_t< D > , cost_t< D > >
  solve( cost_t< D > bound ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
idastar( const D & domain , const IDAOptions & options = {} );

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
cost_bounded_dfs( const D & domain , cost_t< D > bound ,
                  const SearchOptions & options = {} );
```

and `idastar( domain , options )` is `IDAStar< D >( domain , options ).solve()`. The `Solution` and the `SearchOptions` are those of §1.5, and `IDAOptions` adds to them the `DeepeningOptions` of IDA*-CR, whose `growth` (0 by default) and `optimal` choose the bounds of §4.11; a `SearchOptions` converts to an `IDAOptions` with the classical bounds. The second `solve( bound )`, which `cost_bounded_dfs` calls, is one iteration with the given bound, the baseline of §4.14. Of the options, `node_limit` bounds the expansions (summed over the iterations), and the search stops with `Status::node_limit` and an empty path when it reaches it; `expected_nodes` is not used, since there is no CLOSED. With a weight $w > 1$ the bounds are on $f' = g + wh$, and the path costs at most wC^*, since every node of an optimal path has $f' \leq wC^*$ (Chapter 6); `drop_duplicates` does nothing, there being no duplicate detection, and the constructor throws `std::invalid_argument` if the options ask for an anytime search. The status is `solved` with the cost and the path of the goal, or `unsolvable` when an iteration cuts no node. In the counters of the `Solution`, `expanded` counts every expansion, those repeated in later iterations included, `generated` every successor that the domain passes (those cut by the bound included, and, in a domain that is not an `OperatorDomain`, the moves back to the parent too, which an `OperatorDomain` does not pass at all), and `stored` the largest number of states held on the path (Proposition 4.8); the field `bound` of the `Solution` is, with $w = 1$, the cost found, i.e., a proof of optimality (with $w > 1$, the cost over w), unless the bounds of IDA*-CR end the search with a path whose optimality they do not prove (Proposition 4.26). In particular, one obtains DFID by giving `IDAStar` a `Domain` whose heuristic returns 0.

IDA*

Each iteration of IDA* is a depth-first search from s_0 that cuts all the nodes with $f > \beta$, the bound β being $h(s_0)$ in the first iteration and, in each of the others, the least f that exceeded the bound of the previous one; the search ends when an iteration enters a goal, or with no goal reachable when no node was cut. Figure 4.2 shows the bounds as nested contours of f.

The implementation makes the test of the bound when a successor is generated, before entering it, and the goal test when a node is entered; we say that a node is *entered* when the depth-first search is called on it (and *expanded* when its successors are generated, which is the case of every node entered that is not a goal), and *cut* when it is generated with $f > \beta$ and not entered. Also, the successors are those of `for_each_tree_child` (`Domain.h`), passed one at a time, without the move back to the *parent on the path*: by its operator in an `OperatorDomain`, whose successor function skips the inverse of the operator that generated the state, and otherwise by comparing the successor with the state two places back on the path. In a `TreeOperatorDomain`, `for_each_tree_child` calls `for_each_tree_successor( s , h , op , f )` instead, which may also skip the successors of the pairs of moves (op, op_t) that the domain declares redundant, because another sequence of moves that it keeps reaches the same state at a cost not larger: two turns of the same face of Rubik's cube, or of two opposite faces in the other order, and two moves of the same disk of the Towers of Hanoi, as in Korf [59]. The contract of the concept is that every path is matched by a path with the same end, of a cost not larger, whose moves are all kept; the searches with CLOSED do not use it, since the path kept may go through a state closed with another operator, and the test `rubik` checks the nodes of the tree of the cube at the depths 1 to 5, i.e., 18, 243, 3240, 43254 and 577368, against table 1 of [59]. The following pseudocode is that of the header `IDAStar.h`, i.e., the code of `IDAStar::solve()` without the node limit, the counters and the bounds of IDA*-CR.

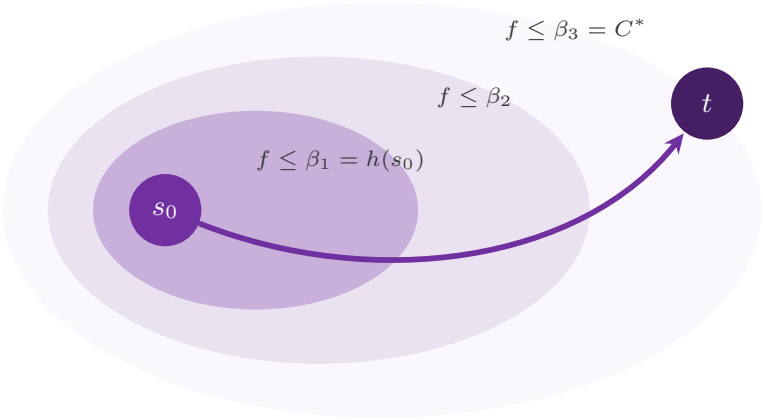


Figure 4.2. The bounds of IDA* as nested contours of f around s_0: the first iteration searches the paths whose nodes have $f \le \beta_1 = h(s_0)$, each later one the paths within the least f that the previous iteration cut, and the last one, with $\beta = C^*$, reaches the goal t along an optimal path.

Algorithm 4.3 (IDA*).

```
1  procedure IDA*(s₀)
2     β ← h(s₀)
3     loop
4        β' ← ∞
5        if DFS(s₀, 0, β) then return the path
6        if β' = ∞ then return unsolvable
7        β ← β'

1  function DFS(u, g, β)                    ▷ g + h(u) ≤ β
2     if u is a goal then return true
3     for each successor v of u but the parent of u do
4        g' ← g + c(u, v)
5        if g' + h(v) > β then β' ← min(β', g' + h(v)); continue
6        push v on the path
7        if DFS(v, g', β) then return true
8        pop v from the path
9     return false
```

When a goal is entered, line 7 returns at once, the remaining successors of each node on the path being skipped, and the path is left from s_0 to the goal. The node limit is tested when a node that is not a goal is entered, before its expansion; a goal is thus found even when the limit has just been reached. Figure 4.3 follows two iterations on a small tree. We now prove the properties of the algorithm (note that the proofs do not need a consistent heuristic), assuming an admissible heuristic, a finite branching factor and costs bounded below by some $\varepsilon > 0$.

Lemma 4.4 (The bounds of IDA*). If a goal is reachable, the bounds of the iterations increase strictly and none of them exceeds C^*; furthermore, an iteration with $\beta < C^*$ cuts at least one node with $f \le C^*$.

Proof. The first bound is $h(s_0) \le C^*$, by admissibility. Consider an iteration with bound $\beta < C^*$, and an optimal path $P = (s_0 = u_0, \dots, u_k)$. The depth-first search enters s_0 with $g = 0 = g^*(s_0)$. Suppose that it enters some u_j, $j < k$, with $g = g^*(u_j)$, along some path of that cost. Then it generates u_{j+1}, with $g = g^*(u_{j+1})$, unless u_{j+1} is the parent w of u_j on the path held. This is not the case: w has $g^*(w) \le g^*(u_j) - c(w, u_j) < g^*(u_j)$, whereas $g^*(u_{j+1}) = g^*(u_j) + c(u_j, u_{j+1}) > g^*(u_j)$, and hence $u_{j+1} \ne w$ (with an `OperatorDomain` the state skipped is w too, since the inverse of an operator leads back to the state it came from, cf. §1.5). Thus u_{j+1} is either cut or entered with $g = g^*(u_{j+1})$. The search cannot enter u_k, since $f(u_k) = C^* > \beta$, and an iteration that enters some other goal first ends the search (which is then not of the form considered here); also, an iteration that enters no goal visits the whole tree of its paths within the bound, so that the generation above does take place. Hence some u_{j+1} is cut, with $f(u_{j+1}) = g^*(u_{j+1}) + h(u_{j+1}) \le C^*$ by admissibility. The next bound is the least f cut, which exceeds β (only nodes with $f > \beta$ are cut) and does not exceed $f(u_{j+1}) \le C^*$. □

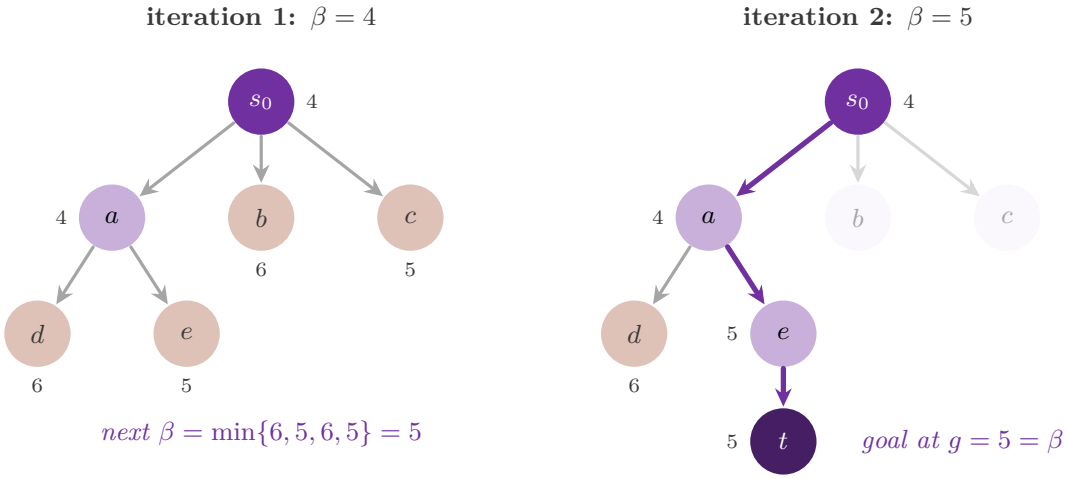


Figure 4.3. Two iterations of IDA*. With $\beta = 4$ (left) the search enters s_0 and a and cuts the four nodes in the rust tint, whose f is written by them; the next bound is the least of them, 5. With $\beta = 5$ (right) the search enters e and reaches the goal t with $g = 5 = \beta$; the nodes b and c are not generated again, since the iteration stops at the goal.

In a `TreeOperatorDomain` the move from u_j to u_{j+1} may be one that the domain skips after the move by which u_j was entered, and the proof is then applied to an optimal path P whose moves are all kept, which exists by the contract of the concept. The argument changes only in that the iteration follows P itself, i.e., it enters each u_j along the prefix of P, since an iteration that enters no goal visits every path of the tree of the kept moves whose nodes are within the bound; the first node of P above the bound is then generated and cut. The same holds for Theorem 4.5 and for the propositions below.

Theorem 4.5 (Optimality of IDA*). If h is admissible and a goal is reachable, Algorithm 4.3 returns a path of cost C^*.

Proof. A goal t is entered only within the bound, i.e., with $g(t) + h(t) \leq \beta$; by admissibility $h(t) = 0$, and by Lemma 4.4 $\beta \leq C^*$, so that $g(t) \leq C^*$, and hence $g(t) = C^*$, since $g(t)$ is the cost of a path. It remains to see that some iteration does enter a goal. By Lemma 4.4 the bounds increase strictly within the values of f not larger than C^* taken by the paths from s_0, of which there are finitely many (a path of cost at most C^* has at most C^*/ε edges, and the branching is finite), and an iteration with $\beta = C^*$ cannot cut a node of an optimal path, by the argument of the lemma, and therefore it enters the goal of one, if no goal was entered before. □

Corollary 4.6 (DFID). With $h = 0$ and unit costs, the bounds of Algorithm 4.3 are $0, 1, 2, \ldots$, the iteration of bound β is the depth-first search to the depth β, and the goal returned is at the least depth.

Proof. With $h = 0$ and unit costs, f is the depth; a node cut by the iteration of bound β is at depth $\beta + 1$, and hence each bound exceeds the previous one by 1 (an iteration that cuts no node ends the search). The least depth of a goal is C^*, and the claim follows from Theorem 4.5. □

Proposition 4.7 (Termination and number of iterations). (i) Each iteration ends, since it visits only paths of cost at most β, which have at most β/ε edges. (ii) With integer costs and a reachable goal, IDA* makes at most $C^* - h(s_0) + 1$ iterations; on the sliding-tile puzzles with the Manhattan distance, at most $(C^* - h(s_0))/2 + 1$. (iii) If no goal is reachable, IDA* reports it on a finite acyclic graph; on a graph with a reachable cycle that the parent pruning does not remove (e.g., a cycle of at least three states) it does not terminate, unless the node limit stops it.

Proof. (i) is the bound on the length of a path, together with the finite branching. (ii) By Lemma 4.4 the bounds are integers that increase strictly from $h(s_0)$ to at most C^*. On the puzzles a move changes g by 1 and the Manhattan distance by ± 1; hence, f changes by 0 or 2, and all the values of f have the parity of $h(s_0)$, so that the bound increases by at least 2. (iii) On a finite acyclic graph the paths from s_0 are finitely many; once the bound is at least the largest f on them, no node is cut, $\beta' = \infty$, and line 6 reports the instance unsolvable. On a graph with a reachable cycle that the parent pruning does not remove, the paths that follow the cycle again and again are not pruned and have unbounded cost; for any β, the first node with $f > \beta$ on such a path is generated from an entered node and cut, and hence $\beta' < \infty$ at every iteration. (A cycle u, v, u of two states is removed by

the parent pruning, so that, e.g., on a finite undirected tree without a goal the search does terminate.) □

Proposition 4.8 (The memory of IDA*). At any time, Algorithm 4.3 holds the path, of at most $\lfloor \beta/\varepsilon \rfloor + 1$ states, and one frame of the recursion per state of the path; the successors are not stored, since they are passed one at a time by the successor function of the domain (which may keep, in each frame, the state of its own enumeration). If a goal is reachable, the path never holds more than $\lfloor C^*/\varepsilon \rfloor + 1$ states; with unit costs and s_0 not a goal, the counter `stored` of a solved search is $C^* + 1$.

Proof. Every state on the path is entered, with $g \leq \beta$, and the path has at most β/ε edges; by Lemma 4.4 $\beta \leq C^*$. With unit costs a path of an iteration has at most C^* edges, and the path returned has exactly C^* (Theorem 4.5); `stored` is the largest length of the path, measured after each state pushed at line 6 of DFS, and hence it is $C^* + 1$ when at least one state is appended, i.e., when s_0 is not a goal. □

The test `idastar` checks `stored` $= C^* + 1$ on three instances of the 15-puzzle of Korf, the costs against the exact distances of 100 states of the 8-puzzle and against A* on small grids and on the map of Romania, the report of an acyclic graph as unsolvable, and the stop at 1000 expansions. In fact, the cost of the repetitions is usually rather small, as the next proposition shows, and the bound in (ii) of Proposition 4.7 is the one of Exercise 4.5.

Proposition 4.9 (The overhead of the iterations). Let N_k be the expansions of the iteration k, and let $N_{k+1} \geq b' N_k$ for some $b' > 1$ and all k. Then the expansions of the K iterations sum to at most $N_K \, b'/(b'-1)$.

Proof. $N_k \leq N_K / b'^{K-k}$, and therefore $\sum_{k=1}^{K} N_k \leq N_K \sum_{j \geq 0} b'^{-j} = N_K \, b'/(b'-1)$. □

On a tree of uniform branching b and unit costs, DFID has $b' = b$, and the overhead of the iterations is a factor of about $b/(b-1)$ over the last one (Korf [7]); IDA* has a similar behavior when the number of nodes within the bound grows geometrically with it, as it is usually observed on the sliding-tile puzzles (an empirical fact, which the hypothesis of the proposition only models). In the worst case, however, each iteration adds a single node, the K-th iteration expands K nodes, and the total is $K(K+1)/2$, i.e., quadratic in the expansions of A* (e.g., with real costs that are all different).

Duplicate paths and parent pruning

A depth-first search detects no duplicates: a state reached by several paths within the bound is expanded once per path in each iteration. On the sliding-tile puzzles the short cycles are rare, and IDA* with a pattern database is the method of Korf and Felner [22] for the 24-puzzle (cf. Chapter 5); on the grids, on the contrary, a cell is reached by very many paths (Figure 4.4). Korf and Schultze give an order of magnitude for the 15-puzzle: a depth-first search to the average depth of an optimal solution, 53 moves, with the branching factor 2.13, would generate about $2.5 \cdot 10^{17}$ nodes, against about 10^{13} states [156].



cell (i, j): $\binom{i+j}{i}$ shortest paths;
the cells with at least 10 are in the rust colour;
one iteration of IDA* that reaches the far corner
enters it 70 times along shortest paths alone

Figure 4.4. The number of shortest paths from the corner s_0 to each cell of a 5×5 grid with 4 moves, i.e., $\binom{i+j}{i}$ for the cell (i, j): a depth-first search without duplicate detection enters the far corner 70 times along the shortest paths only, and the counts grow exponentially with the side of the grid.

The library prunes only the move back to the parent on the path, as described above. The pruning removes no optimal path, since an optimal path with positive costs has no cycle (cf. Proposition 2.16), and the proof of Lemma 4.4 shows that it never skips the next state of an optimal path.

Transposition tables

Status — planned. `IDAStar` has no transposition table.

A transposition table (§3.10) keeps, in a memory of fixed size, some of the states visited in the current iteration with the least g at which each was entered; a state reached again with a g not smaller than the one in the table is not entered, since everything below it was searched with a larger remaining budget (Reinefeld and Marsland [161] describe this and further uses of the table across the iterations). The pruning changes the least f that is cut, which may skip some values of the bound; the next proposition shows that the optimality survives.

Proposition 4.10 (Transposition tables keep IDA* optimal). Let IDA* skip, within an iteration, any state v reached with $g \geq g_T(v)$, where $g_T(v)$ is the least g with which v was entered earlier in the same iteration (and is kept in the table). Then, with h admissible, the bounds remain at most C^*, and the path returned is optimal.

Proof. Let $P = (s_0 = u_0, \ldots, u_k)$ be an optimal path and consider an iteration that enters no goal; let j be the largest index such that u_j is entered, at some time of the iteration, with $g = g^*(u_j)$ ($j \geq 0$, since s_0 is entered with $g = 0$), and $j < k$. At that entry u_{j+1} is generated with $g^*(u_{j+1})$ (it is not the parent, as in Lemma 4.4). It is not entered with $g^*(u_{j+1})$, by the choice of j; nor is it skipped, since a skip needs an earlier entry with $g_T(u_{j+1}) \leq g^*(u_{j+1})$, i.e., with $g^*(u_{j+1})$, again excluded by the choice of j. Hence u_{j+1} is cut, with $\beta < f(u_{j+1}) \leq C^*$. Thus each iteration that ends without a goal has $\beta < C^*$ and gives a next bound in $(\beta, C^*]$, which shows that the bounds increase strictly and stay at most C^*; and an iteration with $\beta = C^*$ cannot cut u_{j+1}, so that it enters a goal. The rest of the proof of Theorem 4.5 applies unchanged. □

We remark that the pruning may make IDA* skip some intermediate values of the bound, never a value above C^*; with a table of bounded size, entries are replaced, which only reduces the pruning.

Stochastic node caching

Status — planned.

Miura and Ishida [162] keep in a cache of bounded size a random subset of the states, each generated state being stored with a probability p if it is not cached yet; the cache is then filled mostly by the states that are generated often, i.e., by those reached along many paths (§3.10). The cached states are then used for duplicate detection, as in a transposition table, and the proof of Proposition 4.10 applies unchanged, since it holds for any content of the table.

4.5 Recursive best-first search

Recursive best-first search (RBFS, Korf [116]) keeps, like IDA*, a single path, but with the siblings of its nodes, and it expands the nodes in best-first order. Each node n carries a stored value $F(n) \geq f(n)$, initially $f(n)$, the least f of the frontier below it when it was last explored; the search below a node is bounded by B, the least stored value of the alternatives found higher up, and when the subtree of a node exceeds its bound, the subtree is dropped and its least f is backed up into $F(n)$, so that the search can return to it later without starting from scratch (Figure 4.5). `RBFS` (`RBFS.h`) is the algorithm of [116] with the pseudocode of the header.

Algorithm 4.4 (RBFS; c_1 and c_2 are the two successors of least F, and $F(c_2) = \infty$ when n has one successor).

```
 1  function RBFS(n, F(n), B)
 2     if f(n) > B then return f(n)
 3     if n is a goal then stop with the path to n
 4     if n has no successor then return ∞
 5     for each successor c of n do
 6        F(c) ← max(F(n), f(c)) if f(n) < F(n), else f(c)
 7     sort the successors by F
 8     while F(c1) ≤ B and F(c1) < ∞ do      ▷ c1, c2: the two best
 9        F(c1) ← RBFS(c1, F(c1), min(B, F(c2)));  sort again
10     return F(c1)
11  RBFS(s0, f(s0), ∞)
```

Line 6 inherits the backed-up value of the parent: if $F(n) > f(n)$, the subtree of n was already explored up to $F(n)$, and its successors are given at least that value. In the code the successors of the node at the depth d of the recursion are kept in a vector of that depth, reused from one call to the next; they are those of `for_each_child`, without the move back to the parent in an `OperatorDomain` (but with it otherwise, since nothing compares

a successor with the path), and a successor of infinite h is not kept. After a call of line 9 the first successor is moved back to its place by swaps with the following ones, i.e., by one pass of insertion, the others being still sorted. The declarations, quoted by hand, are

```
template< Domain D >
class RBFS {
 public:
 explicit RBFS( const D & domain , const SearchOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void );
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
rbfs( const D & domain , const SearchOptions & options = {} );
```

where the constructor throws `std::invalid_argument` for a weight other than 1, `node_limit` bounds the expansions (the calls that reach line 5), the stop at the limit returns `Status::node_limit`, and `stored` is the largest number of successors held at a time on the path, i.e., the memory of Proposition 4.11; `bound` is the cost of the path found.

Proposition 4.11 (The memory and the optimality of RBFS). (i) At any time, Algorithm 4.4 holds the nodes of the recursion and their successors, i.e., at most b nodes per level of the recursion, which is $O(bD)$ for a recursion of depth D. (ii) Let h be admissible, the branching finite, the costs at least some $\varepsilon > 0$, and a goal reachable. Then every node that passes the test of line 2 has $f \le C^*$, the recursion has a depth of at most $C^*/\varepsilon + 1$, and Algorithm 4.4 terminates with a path of cost C^*.

Proof. (i) A call holds its node and the list of its successors with their stored values (lines 5 to 10), and nothing else survives a return.

(ii) Fix an optimal path $P = (s_0 = u_0, \dots, u_k)$ without a cycle (one exists, the costs being positive). We say that a node x of the recursion is *on* P if the path of the recursion from s_0 to x is a prefix of P, in which case its g is $g^*(x)$ and $f(x) \le C^*$ by admissibility. We first show that $F(x) \le C^*$ at any time for each successor x held that is on P. When $F(x)$ is set at line 6 it is $f(x)$ or $\max(F(n), f(x))$ for the parent n of x, which is on P too, and the claim follows by induction on the depth (at the root, $F(s_0) = f(s_0) \le C^*$). When it is set at line 9, it is the value returned by RBFS$(x, \dots)$, which is $f(x) \le C^*$ at line 2, or not ∞ at line 4, since the successor of x on P exists, is not the parent of x (P has no cycle) and has a finite h ($h \le h^* < \infty$), or the F of the best successor of x at line 10, which is at most the F of the successor of x on P, at most C^* by induction on the number of the assignments.

Now consider any call RBFS$(n, F(n), B)$ while no goal has been found, and let x be the deepest node of the recursion path from s_0 to n that is on P. If $x = n$, then $f(n) \le C^*$. Otherwise the successor y of x on P is held, at the depth of x, but it is not the successor c_1 that the recursion follows; since c_1 was chosen as the least by F and c_2 as the second, $F(c_2) \le F(y) \le C^*$, and every call below c_1 receives a bound $B \le F(c_2) \le C^*$, since the bound passed at line 9 only decreases with the depth. Hence a node that passes line 2 has $f \le C^*$ in both cases. Its g is then at most C^*, and its depth at most C^*/ε, which bounds the recursion; the nodes at these depths are finitely many, and so are the values of f that can be returned at line 2, 4 or 10, i.e., the values of f of these nodes and of their successors, and ∞.

A call of line 9 receives $F(c_1) \le B' = \min(B, F(c_2))$ and, if it does not find a goal, returns a value larger than B' or ∞ (by induction on the depth: line 2 returns $f > B'$, line 4 returns ∞, and line 10 returns an F that fails the test of line 8); hence each iteration of the loop of line 8 strictly raises $F(c_1)$ within the finite set of the values above, and the loop ends. Since the recursion has a bounded depth, every call terminates. The root call cannot end without a goal: its loop ends only with $F(c_1) = \infty$, whereas the successor of s_0 on P has $F \le C^*$. When a goal t is found, it passed line 2 with $g(t) = f(t) \le C^*$ ($h(t) = 0$ by admissibility), and $g(t)$ is the cost of a path; thus $g(t) = C^*$. □

Korf [116] also proves that RBFS with a consistent heuristic expands the new nodes in best-first order (his theorems 4.2 and 4.3), which we do not reproduce; we checked only what the proof above needs, i.e., the code of `RBFS.h`. Its weakness is the same as that of IDA*, i.e., the regeneration of the dropped subtrees, which may be frequent when the stored values of two siblings alternate as the least, and the absence of duplicate detection, by which a state reached by several paths is expanded once per path; on a grid with 4 moves the test `classic` runs it only on maps of 8×8 cells, and on the 15-puzzle on one instance of Korf, against the costs of A* and of the exact distances of 200 states of the 8-puzzle.

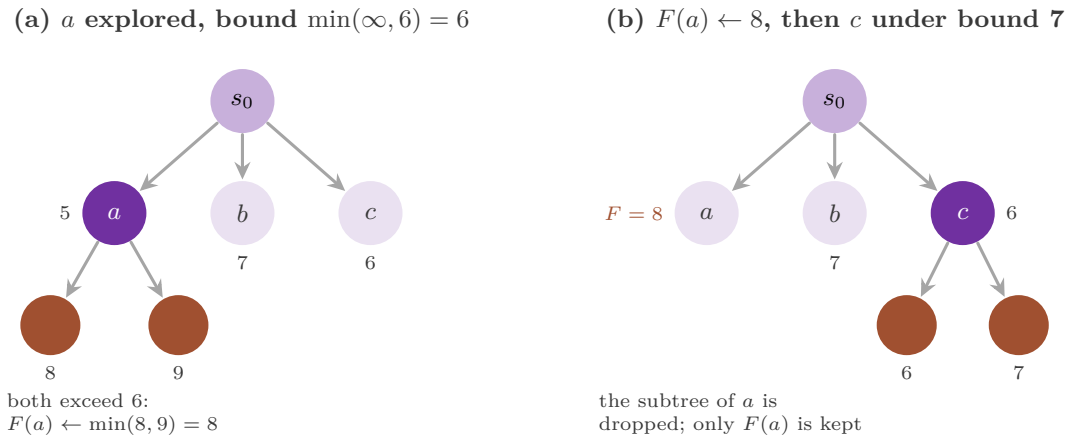


Figure 4.5. Two steps of RBFS. (a) The best child a (with $f = 5$) is explored under the bound 6, the value of its best sibling c; both successors of a exceed it, and the least, 8, becomes the stored value $F(a)$. (b) The subtree of a is dropped, and the search turns to c, under the bound 7 of the next best sibling b.

4.6 External search with delayed duplicate detection

IDA* saves the memory of CLOSED by giving up the detection of the duplicates; an external search keeps it, but on the disk, where the time of an access is many orders of magnitude larger than in memory when it is random, and only somewhat larger when it is sequential. Hence the duplicates are not looked up when a node is generated, but detected later and in batches, by scanning files: this is the *delayed duplicate detection* (DDD) of §3.11, whose hash-based form is due to Korf [40].

Breadth-first search on the disk

The simplest case is the breadth-first search with unit costs. On an undirected graph the duplicates of a new layer are found in very few layers, as we now show; on a directed graph they may be in any earlier layer.

Proposition 4.12 (Duplicates of a breadth-first layer). On an undirected graph with unit costs, let L_i be the states at distance i from s_0. A successor of a state of L_i is in L_{i-1}, L_i or L_{i+1}.

Proof. If v is adjacent to $u \in L_i$, then $\delta(v) \le \delta(u) + 1 = i + 1$ and, the graph being undirected, $\delta(u) \le \delta(v) + 1$, i.e., $\delta(v) \ge i - 1$, δ denoting the distance from s_0. □

Hence, when the layer L_{i+1} is generated from L_i, its duplicates are removed by comparing it with L_i and L_{i-1} only, and the older layers can be dropped (the frontier search of Korf et al., cf. [156]). The detection was first made by sorting the file of the new layer and merging it with the previous ones (sorting-based DDD); the hash-based DDD of Korf [40] avoids the sorting by two independent hash functions, a first one that splits the nodes into files (all the copies of a state are then in the same file), and a second one by which each file, loaded in a table in memory, drops its copies [156]. When the states have a perfect hash into a range that fits in memory at 2 bits per state, the two-bit breadth-first search of Korf [26] keeps instead, for each state, one of four values (not reached, in the current layer, in the next one, older) in an array of 2-bit entries; the library uses it to build the pattern databases (§5.5).

A* with delayed duplicate detection: the interface

`DDDAStar` is the External A* of Edelkamp, Jabbar and Schrödl [39], with the hash-based DDD of Korf [40] in place of the external sorts. It requires a `DiskDomain` (`DDDAStar.h:136`), i.e., a `Domain` with integer costs, since a bucket is a pair of values, and trivially copyable states, since they are written to the files as their bytes. Its declarations, without the bodies, are

```
template< DiskDomain D >
class DDDAStar {
 public:
 explicit DDDAStar( const D & domain , const SearchOptions & options = {} ,
                    DDDOptions ddd = {} );
```

```
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< DiskDomain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
ddd_astar( const D & domain , const SearchOptions & options = {} ,
           const DDDOptions & ddd = {} );
```

where `ddd_astar` is `DDDAStar< D >( domain , options , ddd ).solve()`, and the options of the disk are

```
struct DDDOptions {
 /// the files of a bucket, i.e., the parts by the hash of the state: the
 /// search holds in memory about the largest bucket over this number, with
 /// its copies, plus a buffer of buffer_bytes per file it writes and the
 /// buffer of the scans
 std::size_t partitions = 64;
 /// where the files go, in a directory of their own removed at the end;
 /// the temporary directory of the system if empty
 std::filesystem::path directory;
 /// the bytes of the buffer of each file being written, i.e., of each
 /// file of the buckets that the successors of a bucket go to
 std::size_t buffer_bytes = std::size_t( 1 ) << 16;
 };  // end( struct( DDDOptions ) )
```

i.e., the number P of files per bucket (`partitions`, 64 by default, and 0 is refused with `std::invalid_argument`, as is a weight other than 1 in the `SearchOptions`), the directory of the files (a directory of its own is made in it, or in the temporary directory of the system when it is empty, and removed with its files at the end, also when the search stops at the limit or throws), and the bytes of the buffer of each file being written (`buffer_bytes`, 2^{16}). Of the `SearchOptions`, `node_limit` bounds the states written to the closed files (the counter `stored`), checked after each file, and `expected_nodes` is not used. In the counters, `expanded` counts the states expanded, each once, `generated` the successors, `duplicates` the copies dropped, and `stored` the states written to the closed files, i.e., the distinct states closed.

The buckets and the files

The nodes are kept in *buckets* $B(g, h)$, one per pair of values, and each bucket is split into P *files* by the hash of the state, the file p holding the states v with $\text{hash}(v) \bmod P = p$; all the copies of a state with the same g are thus in the same file, and all those of the same state in the files of the same index p, since h is a function of the state. A record of a file is the state, the hash of the state from which it was generated (its *parent*) and, in an `OperatorDomain`, the operator that generated it. Each bucket has an *open* part, the records appended to it and not yet examined, and a *closed* part, the states of it that have been expanded. The buckets are expanded by increasing $f = g + h$ and, for equal f, by increasing g (i.e., the key of a bucket is the pair $(g + h, g)$, compared lexicographically, cf. Figure 4.6).

Algorithm 4.5 (DDD-A*, the pseudocode of the header `DDDAStar.h:23-36`).

```
 1  procedure DDD-A*(s0)
 2     OPEN(0, h(s0)) ← {s0}
 3     while some OPEN(g, h) is not empty do
 4        (g, h) ← the bucket of least g + h, and then of least g
 5        for p ← 0 to P − 1 do
 6           T ← the states of the file p of OPEN(g, h), without copies
 7           for each closed bucket (g', h) with g' < g do
 8              T ← T \ the file p of CLOSED(g', h)
 9           the file p of CLOSED(g, h) ← T
10           if T has a goal then return the path to it, of cost g
11           for each u in T and each successor v of u do
12              append (v, hash(u)) to the file hash(v) mod P of
                 OPEN(g + c(u, v), h(v))
13     return unsolvable
```

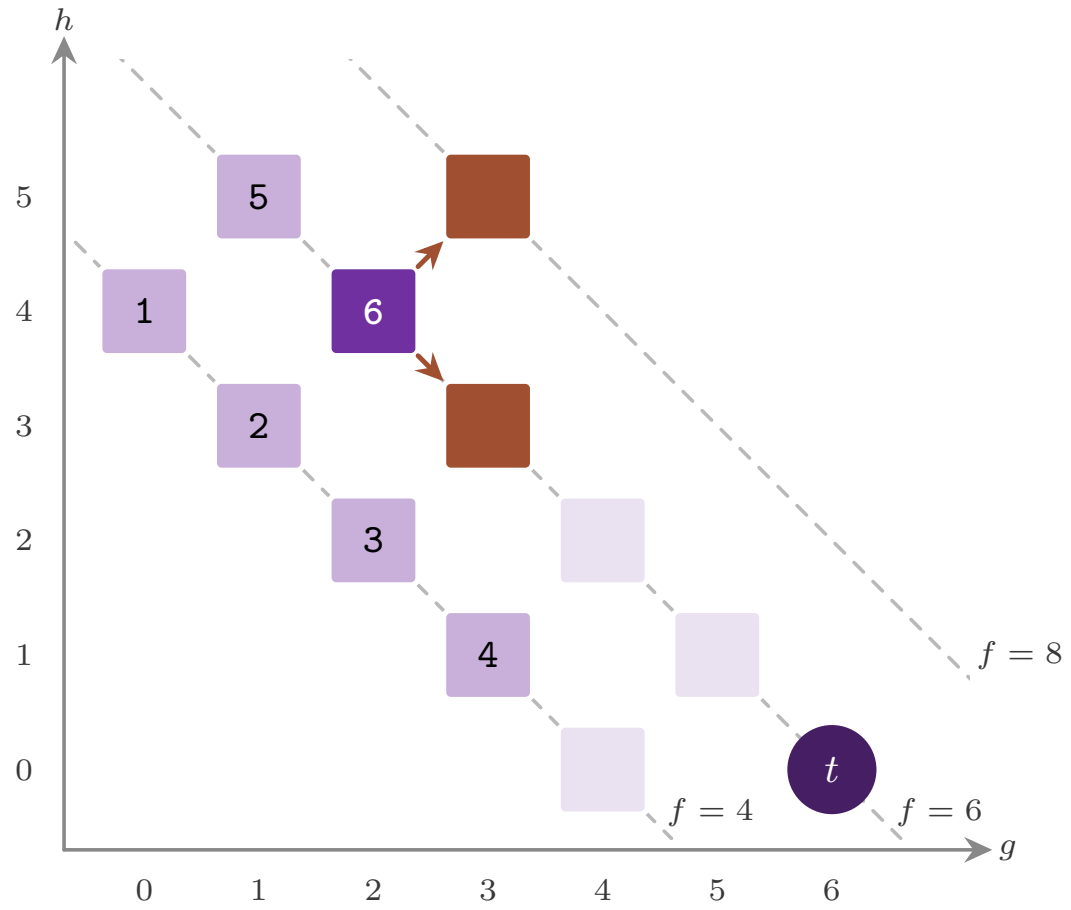


Figure 4.6. The buckets (g, h) of DDD-A* on a domain with unit costs in which h changes by 1 at each move, with the lines of equal f. The closed buckets carry their order of expansion (all those of $f = 4$ by increasing g, then $(1, 5)$); the current bucket, 6, sends its successors to $(3, 3)$, of the same f and a larger g, and to $(3, 5)$, of $f = 8$; the goal is in the bucket $(6, 0)$.

Lines 6 to 12 are the processing of one file, which the code does as follows: the open file is loaded in a hash table of the memory, which keeps the first record of each state; the closed files of the same h are scanned, and the states found in them are removed from the table; what is left is written as the closed file and expanded, each successor being appended through a buffer per file of the buckets it goes to (Figure 4.7). The expansion also checks the two hypotheses on which the correctness rests, and throws `std::logic_error` on an edge of a cost that is not positive or on a successor of a smaller f than its parent, i.e., on a heuristic that is not consistent.

Algorithm 4.6 (The file p of the bucket (g, h)).

```
 1  procedure FILE(g, h, p)
 2     T ← an empty hash table
 3     for each record (v, x, o) of the file p of OPEN(g, h) do
 4        if v ∈ T then count a duplicate else T[v] ← (x, o)
 5     delete the file p of OPEN(g, h)
 6     for each closed bucket (g', h) do                ▷ all have g' < g
 7        for each record (v, ·, ·) of the file p of CLOSED(g', h) do
 8           if v ∈ T then remove v from T;  count a duplicate
 9     if T = ∅ then return
10     write the records of T as the file p of CLOSED(g, h)
11     if some state of T is a goal then
12        return TRACE(its record, g), of cost g            ▷ Algorithm 4.7
13     for each record (u, x, o) of T do
14        for each successor v of u but the inverse of o do
15           if c(u, v) ≤ 0 or g + c(u, v) + h(v) < g + h then
16              throw logic_error
17           append (v, hash(u), the operator of v) to the buffer of
18              the file hash(v) mod P of OPEN(g + c(u, v), h(v))
```

The buffers of the files written by a bucket are flushed when all its P files have been processed, and the buckets that they fill are then added to the open ones; the node limit is tested after line 18 of each file.

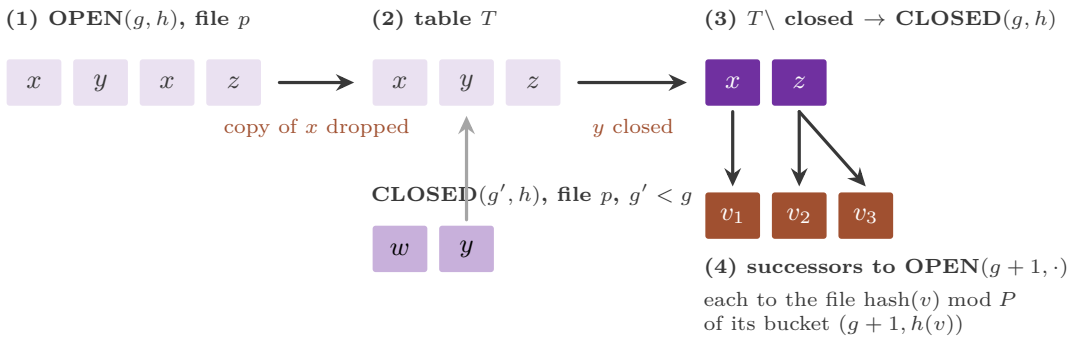


Figure 4.7. One file p of the bucket (g, h). (1) The open file, which may hold copies; (2) loaded in a table, which drops the copy of x; (3) cleared of the states of the closed files of the same h and a smaller g (here y), and written as the closed file p of (g, h); (4) expanded, each successor v appended to the file hash(v) mod P of its bucket.

At the end the path is traced back from the goal through the hashes of the parents.

Algorithm 4.7 (The path to the goal of the record x, closed with the cost g).

```
1  function TRACE(x, g)
2     π ← (the state of x)
3     while g ≠ 0 or the state of x ≠ s₀ do
4        p ← (the parent hash of x) mod P
5        y ← a record of the file p of some CLOSED(g', h'), g' < g,
6           whose state u has hash(u) = the parent hash of x and the
7           state of x as a successor of cost g − g'    ▷ else throw
8        x ← y;  g ← g';  prepend u to π
9     return π
```

Correctness

We now prove that the search is correct. Throughout, let h be consistent (Definition 2.2) and 0 on the goals (hence admissible, cf. §2.4), the costs positive integers, and the inverse of the operator of a state, in an `Operator Domain`, lead back to the state that generated it.

Lemma 4.13 (The order of the buckets). A successor of a state of the bucket (g, h) goes to a bucket of a larger key; hence no bucket receives a record after it has been expanded, and the buckets are expanded by increasing key.

Proof. A successor v of u has $g(v) = g+c(u, v) > g$ and, by consistency, $f(v) = g+c(u, v)+h(v) \geq g+h(u) = f(u)$; hence its key $(f(v), g(v))$ is larger than $(f(u), g)$. The search takes the open bucket of least key (line 4), and each bucket filled later has a larger key than the one being expanded. □

Lemma 4.14 (Each state is closed once, at its optimal g). Every state v reachable from s_0 whose bucket $(g^*(v), h(v))$ is expanded before the search stops is written to a closed file exactly once, in that bucket, and each other copy of it is dropped at line 4 or at line 8 of Algorithm 4.6; no state is written to two closed files.

Proof. Each copy of a state v carries the cost of a path to v, hence $g \geq g^*(v)$; all the copies of v have $h(v)$ and lie in the files of index hash(v) mod P; and by Lemma 4.13 the buckets are expanded by increasing key $(g + h, g)$, so that among the buckets of $h(v)$ the one of $g^*(v)$ comes first. We prove by induction on the order of the expansions that, when a bucket (g, h) is expanded, (a) every state v with $g^*(v) = g$ and $h(v) = h$ has a copy in it, and (b) a state of it is written to its closed file only if $g = g^*(v)$ and it was not written before. For (a), s_0 is put in $(0, h(s_0))$ by line 2 of Algorithm 4.5. Let $v \neq s_0$, and w its predecessor on an optimal path, so that $g^*(w) = g - c(w, v) < g$; the bucket $(g^*(w), h(w))$ has a smaller key (Lemma 4.13) and was expanded before, and by (a) and (b) for it w was written to its closed file there and expanded (the search did not stop, since it reached (g, h)). The successor that line 14 of Algorithm 4.6 skips is the state u from which the record kept for w was generated, closed with some g_u such that $g_u + c(u, w) = g^*(w)$; hence $g^*(u) \leq g_u < g^*(w) < g^*(v)$, and $u \neq v$. Thus the expansion of w appended v with $g^*(v)$ to (g, h). For (b), a copy of v with $g > g^*(v)$ is in a bucket that follows $(g^*(v), h)$, which by (a) held a copy of v and by (b) wrote v to its closed file of index hash(v) mod P (no earlier closed file of h could hold v, since it would have a smaller g); line 8 scans that file and removes v. A state with $g = g^*(v)$ is not in an earlier closed file, for the same reason, and line 4 keeps one record per state of a file. Hence each state whose bucket $(g^*(v), h(v))$ is expanded is written exactly once, there, and each other copy of it is dropped at line 4 or at line 8. □

Theorem 4.15 (Correctness of DDD-A*). If a goal is reachable, Algorithm 4.5 returns a path of cost C^* (when the node limit does not stop it first); otherwise, on a finite graph, it reports the instance unsolvable. In both cases it terminates, and each state is expanded at most once.

Proof. By Lemma 4.14 each state is written to a closed file at most once, and it is expanded only after that, i.e., at most once. Let a goal be reachable, and let t be an optimal one: t is closed in the bucket $(C^*, 0)$ if the search reaches it, and before it the open buckets are not exhausted, since by the induction of Lemma 4.14 the copies of the states of an optimal path reach their buckets. A goal closed earlier, in a bucket $(g, 0)$, has $f = g$, and by Lemma 4.13 its bucket precedes $(C^*, 0)$ only if $g \leq C^*$; since $g \geq C^*$ for a goal, the goal returned at line 10 has $g = C^*$. By Proposition 4.17 the path returned costs g. For the termination, the buckets with $f \leq C^*$ are finitely many (with integer costs, $g \leq C^*$ and $h \leq C^*$ in them). Each of them receives finitely many records: a record is a successor of a state closed in a bucket of smaller key, i.e., with $g \leq C^*$; these states have $g^* \leq C^*$, are finitely many (the branching being finite and the costs at least 1), and each is expanded once, with at most b successors. When no goal is reachable on a finite graph, the states are finitely many and each is expanded once, and hence the open buckets are eventually exhausted, and line 13 reports the instance unsolvable. □

The counters follow from Lemma 4.14: each record of an open file that is examined is either written to a closed file or dropped, and hence one has *generated* + 1 = *stored* + *duplicates* on a search that ends without a goal, the 1 being the root, and *expanded* = *stored* (in a search that stops at a goal the last file is closed but not expanded, and *expanded* ≤ *stored*). The test `ddd` checks the identity on a grid of 50×50 cells split by a wall, where the search closes the 1250 cells it can reach, the costs against the exact distances of 50 states of the 8-puzzle with 1, 7 and 64 partitions, against the optimal cost of an instance of the 15-puzzle of Korf, against A* on small grids, and on the map of Romania, the stop at the node limit, the refusal of an inconsistent heuristic and of no partition, and that no file is left on the disk.

Memory, files and the path

Proposition 4.16 (Memory and transfers of DDD-A*). At any time, the memory of Algorithm 4.5 holds (i) the hash table of one file, sized for its records, copies included, and the states left in it; (ii) a buffer of `buffer_bytes` bytes for each file written by the current bucket, i.e., at most P for each distinct pair $(g + c(u, v), h(v))$ of its successors; (iii) the buffer of the file being read; and (iv) the sets of the open and of the closed buckets, one entry per bucket. Each record of an open file is read once, and the closed file p of a bucket (g', h) is read once for each file p of a later bucket of the same h that is not empty.

Proof. The table of a file and the states written from it (lines 2 to 10 of Algorithm 4.6) are released when the next file is processed; the writers of a bucket live until its P files are processed; the scans read through a buffer; and line 6 of Algorithm 4.6 runs over all the closed buckets of the same h, all with a smaller g by Lemma 4.13. □

Hence the memory is about that of the largest file, i.e., about $1/P$ of the largest bucket when the hash spreads the states evenly (an assumption on the hash of the domain), plus the buffers; with the defaults, a bucket whose successors go to k buckets holds $64k$ buffers of 64 KB, i.e., $4k$ MB. The files of all the closed buckets are kept to the end, since a duplicate may be in any of those with the same h, and hence a closed file is read once for each later bucket of its h. Of course, on the undirected graphs with unit costs one could drop the older layers, as in Proposition 4.12, which the library does not do, since it assumes no locality of the domain. Note also that the order by increasing g within a layer of f, which Lemma 4.13 requires, is the opposite of the tie-breaking of A* (§2.8): the search expands every state of the last layer $f = C^*$ that it generates before the bucket of the goal, and it may expand many more nodes than A* when that layer is large.

Proposition 4.17 (The path). Under the hypotheses of Lemma 4.14, Algorithm 4.7 never throws, terminates, and returns a path from s_0 to the goal of cost g; each step reads at most all the closed files of one partition.

Proof. The record x kept for a state v closed with g was generated from a state w that was closed in a bucket $(g_w, h(w))$ with $g_w + c(w, v) = g$, and hence $g_w < g$, in the file hash(w) mod P, which is kept to the end; thus a record y as at lines 5 to 7 exists (it may be another state with the same hash, which is still a predecessor of v at the cost $g - g'$). The state u of y was closed with g', and the invariant of the loop is that π is a path from the state of x to the goal of cost $g_0 - g$, g_0 being the cost of the goal. Since g decreases strictly and is a nonnegative integer, the loop ends; it ends only with $g = 0$ and the state s_0, and then π has the cost g_0. The only state closed with $g = 0$ is s_0, since the costs are positive; hence the loop cannot reach $g = 0$ elsewhere. □

4.7 Symbolic search

Status — planned. The library has no symbolic search.

A symbolic search represents whole sets of states by their characteristic functions. With the states coded by n Boolean variables $x = (x_1, \ldots, x_n)$, a set S is the function $\chi_S(x)$, true exactly on the codes of the states of S, and the moves are a transition relation $Trans(x, x')$, true when the state x' is a successor of x. The image of a set is

$$Img(S)(x') = \exists x \left(\chi_S(x) \wedge Trans(x, x')\right),$$

renamed from x' to x; it is the set of the successors of the states of S, computed in one operation on the functions. The functions are represented by reduced ordered binary decision diagrams (BDDs), i.e., directed acyclic graphs in which each node tests a variable, in a fixed order, with two successors for its values, and the leaves are 0 and 1 (Figure 4.8). For a fixed order of the variables the reduced ordered BDD of a function is unique (Bryant [163]); hence, the equality of two sets is a comparison of two pointers when the BDDs share their nodes, and the operations $\wedge$, $\vee$, $\neg$ take a time polynomial in the sizes of the operands [163]; the size of a BDD, however, depends heavily on the order, and may be exponential in n.

Algorithm 4.8 (Symbolic breadth-first search).

```
1  procedure SYMBOLIC-BFS(s₀)
2     Reach ← {s₀};  Frontier ← {s₀};  d ← 0          ▷ as BDDs
3     loop
4        if Frontier ∩ Goals ≠ ∅ then return d
5        Frontier ← Img(Frontier) \ Reach
6        if Frontier = ∅ then return unsolvable
7        Reach ← Reach ∪ Frontier;  d ← d + 1
```

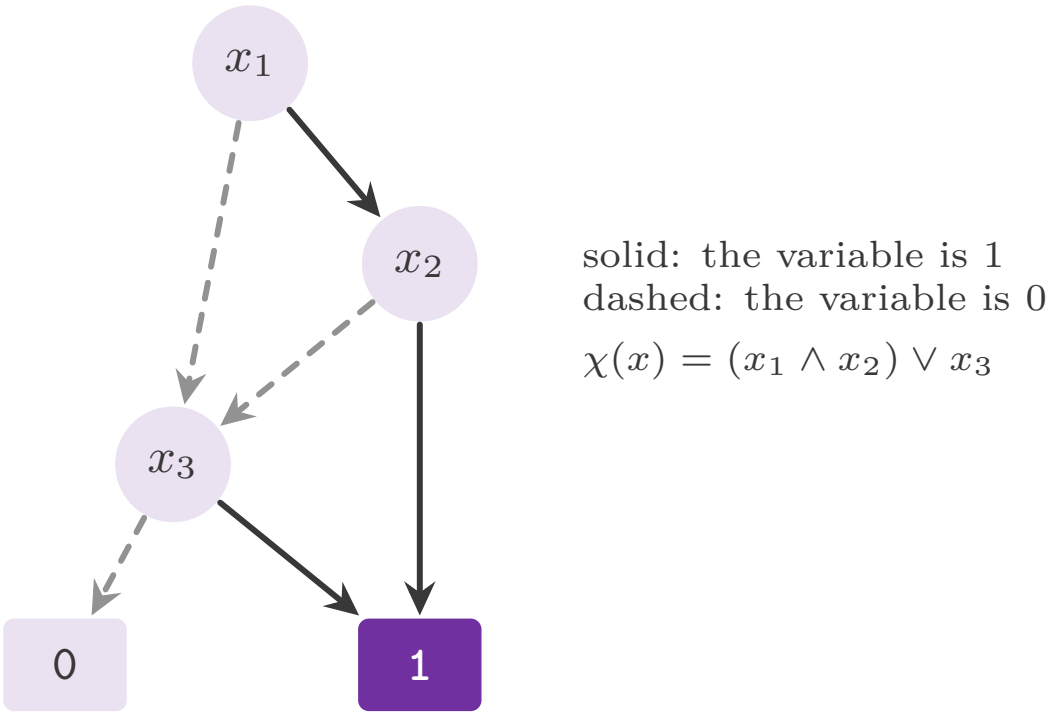


Figure 4.8. The reduced ordered BDD of $\chi(x) = (x_1 \wedge x_2) \vee x_3$ with the order $x_1 < x_2 < x_3$. A solid arc follows the value 1 of the variable of its node, a dashed one the value 0; the node x_3 is shared by the two paths that reach it.

Proposition 4.18 (Symbolic breadth-first search). At line 4 of the iteration d, Frontier is the set of the states at distance d from s_0 (with unit costs) and Reach the set of those at distance at most d; hence Algorithm 4.8 returns the least distance of a goal, and on a finite space it terminates.

Proof. By induction on d. For $d = 0$ both sets are $\{s_0\}$. If the claim holds for d, the successors of the states at distance d that are not at distance at most d are those at distance $d+1$, which is line 5, and line 7 adds them to Reach. The first d at which the frontier meets the goals is the least distance of a goal; on a finite space Reach grows strictly until the frontier is empty. □

One can rebuild the path backwards, by intersecting the preimage of the goal reached with the frontiers kept for the purpose. Symbolic A* keeps instead one BDD per bucket of (g, h), or of (f, h), and expands the buckets in the order of A*, as DDD-A* does with its files (cf. Edelkamp and Schrödl [10]).

4.8 Novelty and width-based search

Let a state be described by the values of m variables, and call an *atom* a pair (variable, value). The *novelty* of a newly generated state s is the least i such that some set of i atoms true in s is true in no state generated before s. The search IW(i) of Lipovetzky and Geffner [104] is a breadth-first search that prunes each generated state of novelty larger than i; its memory is that of the sets of at most i atoms seen so far, and the iterated width runs IW(1), IW(2), … until a goal is found.

Algorithm 4.9 (IW(i)).

```
1  procedure IW(s₀, i)
2     Seen ← the sets of at most i atoms true in s₀;  Q ← (s₀)
3     while Q is not empty do
4        u ← the first node of Q, removed
5        if u is a goal then return the path to u
6        for each successor v of u do
7           if some set of at most i atoms of v is not in Seen then
8              add those sets to Seen;  append v to Q
9     return failure
```

Proposition 4.19 (The size of IW(i)). With m variables of at most D values each, IW(i) appends at most $\sum_{j=1}^{i} \binom{m}{j} D^j = O\big((mD)^i\big)$ states to Q. Furthermore, the iterated width finds a goal, if one is reachable, at the latest with $i = m$.

Proof. A state appended at line 8 adds at least one set of at most i atoms to Seen, which never loses one; the sets of j atoms over distinct variables are at most $\binom{m}{j} D^j$. With $i = m$, a state not generated before has the set of all its m atoms unseen, since the m atoms determine the state, while a state generated before has all its sets in Seen; hence IW(m) is a breadth-first search with duplicate detection, which is complete on a finite space. □

The library implements IW(i) (`IteratedWidth`), the serialized width search SIW (`SerializedWidth`) and the best-first width search BFWS (`BestFirstWidth`) in `WidthSearch.h`, over the features of a `FeatureDomain` rather than the atoms of variables, with the novelty kept by a `NoveltyTable`; they are described in full, with their options and their tests, in §2.15, and here we only place them among the searches that bound what they keep. We remark that IW(i) for small i is neither complete nor optimal; its interest is that many classical planning problems with single atomic goals have a small width, and IW(1) or IW(2) solves them in time polynomial in m [104] (cf. Chapter 16).

4.9 Partial expansion

A* puts in OPEN every successor of a node that it expands, and on a domain with a large branching factor most of them have $f > C^*$, i.e., they are *surplus* nodes, which are stored and never expanded. Partial Expansion A* (PEA*, Yoshizumi, Miura and Ishida [42]) stores only the successors whose f is within a cutoff of the value of their parent, and keeps the parent in OPEN with the least f of the others; Enhanced Partial Expansion A* (EPEA*, Felner et al. [113] and Goldenberg et al. [114]) does not even generate the others, since the domain tells which successors have the value sought.

Both keep with a node n a stored value $F(n) \geq f(n)$, $f(n)$ when n gets a new g, by which OPEN is ordered, the ties going to the larger g. When n is selected, only its successors c with $f(c) \leq F(n) + C$ (PEA*, $C \geq 0$ being the cutoff) or with $f(c) = F(n)$ (EPEA*) are put in OPEN, and n goes back to OPEN with $F(n)$ the least f of the others, if any, or is closed otherwise. We write $\Delta f(c) = f(c) - f(n) = c(n,c) + h(c) - h(n)$ for a successor c of n; with a consistent heuristic $\Delta f \geq 0$ on every edge.

Algorithm 4.10 (PEA* and EPEA*, the pseudocode of the header `PartialExpansion.h`).

```
 1  procedure PEA*(s0)                          ▷ EPEA* at the lines marked E
 2     F(s0) ← f(s0);  push s0 in OPEN, by F and then by larger g
 3     while OPEN ≠ ∅ do
 4        n ← the first node of OPEN, removed;  if n is a goal then
             return the path to n
 5        generate the successors c of n          ▷ E: those with
             f(c) − f(n) = F(n) − f(n) only, by the operator selection
             function, which also gives the least larger Δf
 6        for each c with f(c) ≤ F(n) + C do
 7           if c is new or reached through a cheaper path then
                F(c) ← f(c);  push c in OPEN
 8        F' ← the least f(c) > F(n) + C of the others that are new or
             reached through a cheaper path (E: f(n) plus the least
             larger Δf of all the others), ∞ if none
 9        if F' < ∞ then F(n) ← F';  push n in OPEN again
10        else close n
```

The *operator selection function* of EPEA* is the method `for_each_successor_delta( s , h_s , op , delta , f )` of a `PartialExpansionDomain` (`Domain.h`), an `OperatorDomain` that calls `f( t , c , h_t , op_t )` for the successors t of s whose Δf is `delta` (with real costs, up to a slack such as that of `cost_slack()`), without the one that the inverse of `op` leads to, and returns the least Δf above `delta` of the others, or `infinite_cost()` if there is none. On the sliding-tile puzzles a move costs 1 and changes the Manhattan distance by ± 1, so that $\Delta f \in \{0, 2\}$ is read from the table of the distances without building the successor; on the grids it depends on the cost of the cell entered and on the octile distance (Chapter 14).

Line 8 of PEA* leaves out of F' the successors already stored with a g not larger than $g(n) + c(n,c)$ (`PartialOptions::skip_known`, true by default), which line 7 would reject at every later selection of n with the same $g(n)$; the comparisons of line 6 allow, with real costs, the slack $\tau \max\{1, |F(n)|\}$ of `cost_slack()`, τ being `PartialO`

`ptions::tolerance` (10^{-9} by default), so that two values of f that are equal but for the rounding of sums of the same costs in another order do not cost a selection each. EPEA* keeps with a node, with real costs, the Δf of its next selection, since $F(n) - f(n)$ need not be exact; with integer costs it computes it as $F(n) - f(n)$. The options of PEA* are

```
struct PartialOptions : public SearchOptions {

 /// the cutoff C of PEA*: the successors with \f$ f \leq F( n ) + C \f$
 /// enter OPEN at a selection of n
 double cutoff = 0;

 /// leave out of the next \f$ F( n ) \f$ the successors already stored
 /// with a g not larger than the one through n
 bool skip_known = true;

 /// the relative tolerance of the comparisons of f with
 /// \f$ F( n ) + C \f$, for real costs [see cost_slack()]
 double tolerance = real_cost_tolerance;

 };  // end( struct( PartialOptions ) )
```

and the declarations, quoted by hand, are

```
template< Domain D >
class PEAStar : public PartialExpansionBase< D , PEAStar< D > > {
 public:
 explicit PEAStar( const D & domain , const PartialOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< PartialExpansionDomain D >
class EPEAStar : public PartialExpansionBase< D , EPEAStar< D > ,
                                              PartialExtra< cost_t< D > > > {
 public:
 explicit EPEAStar( const D & domain , const SearchOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
pea_star( const D & domain , const PartialOptions & options = {} );

template< PartialExpansionDomain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
epea_star( const D & domain , const SearchOptions & options = {} );
```

The two searches share the class template `PartialExpansionBase< D , Derived , Extra >`, which holds the nodes (with F and the value of type `Extra` that the derived class keeps, value-initialized when a node gets a new g), CLOSED, OPEN (a d-ary heap of entries (F, g, n), an entry being stale when its node is closed or has another g or another F) and the loop of lines 3 to 10, and calls the method `expand( n , run )` of `Derived` for lines 5 to 8, i.e., the static polymorphism of the curiously recurring template. `Extra` is `PartialNoExtra`, an empty struct, for PEA* and for EPEA* with integer costs, and `PartialDelta< Cost >`, which holds the Δf of the next selection, for EPEA* with real costs (the alias `PartialExtra< Cost >` chooses between them). Both constructors throw `std::invalid_argument` for a weight other than 1; `node_limit` bounds the stored nodes, `expanded` counts the selections and `reexpanded` those of a node selected before, `generated` counts every successor generated by PEA* but only those passed by the operator selection function for EPEA*, and `duplicates` counts, for PEA*, also the successors left out of F' as known.

We now prove that the skip of the known successors loses nothing, and then the optimality of both searches; the argument adapts the invariant of OPEN of A* (Lemma 2.7) to the stored values.

Lemma 4.20 (The skip of the known successors). Let v be a successor of n left out of F' at a selection of n by `skip_known`. Then every later generation of v from n with the same $g(n)$ is rejected as a duplicate at line 7,

and every selection that the skip removes is one that would put no node in OPEN through v.

Proof. The stored $g(v)$ never increases, and v is put in OPEN through n only if $g(n) + c(n, v) < g(v)$; since v was left out with $g(v) \leq g(n) + c(n, v)$, as long as $g(n)$ does not change v is rejected at every later selection of n. A smaller $g(n)$, on the other hand, puts n back in OPEN with $F(n) = f(n)$, and then all its successors are examined again with the new $g(n)$. □

Theorem 4.21 (Optimality of PEA* and EPEA*). Let h be admissible, and for EPEA* also consistent (it suffices that $\Delta f \geq 0$ on every edge). Then, as long as the search runs, OPEN holds a non-stale entry with $F \leq C^*$ (up to the slack of the real costs), and the first goal selected has $g = C^*$; this holds for any cutoff $C \geq 0$, with or without `skip_known`.

Proof. Let P be an optimal path to a goal t. If all its nodes are stored with their optimal g, then t is in OPEN with $F(t) = f(t) = C^*$, since a goal is never expanded (its selection ends the search). Otherwise, let u be the first node of P whose successor v on P is not stored with $g(v) = g^*(v)$; then u is stored with $g(u) = g^*(u)$, being s_0 or the successor of a node of P before it. At no selection of u with this g has v been put in OPEN through u, which would give $g(v) \leq g^*(u) + c(u, v) = g^*(v)$, nor left out of F' by `skip_known`, which needs a stored $g(v) \leq g^*(v)$, nor rejected at line 7, for the same reason.

For PEA*, at each such selection v was therefore one of the others of line 8, with $f(v) > F(u) + C$, and F' was at most $f(v)$; hence u is not closed, and $F(u)$ is either $f(u) \leq C^*$ (before its first selection with this g) or at most $f(v) = g^*(v) + h(v) \leq C^*$. For EPEA*, let δ be the Δf of a selection of u, 0 at the first one. If $\Delta f(v) = \delta$, the operator selection function would pass v, which then would be stored with $g^*(v)$; thus $\Delta f(v) \neq \delta$. Since $\delta = 0 \leq \Delta f(v)$ at the first selection, and each later δ is the least Δf above the one before among the successors not passed, v among them, by induction $\delta < \Delta f(v)$ at every selection, and the next one is at most $\Delta f(v)$; hence u is not closed, and $F(u) = f(u) + \delta \leq f(u) + \Delta f(v) = f(v) \leq C^*$.

In both cases the entry pushed with the current $F(u)$ and $g(u)$ is not stale, and OPEN holds an entry with $F \leq C^*$. The entry selected has the least F; when it is a goal t', its F is $f(t') = g(t')$, since a goal is pushed only by line 7 and never selected twice, and therefore $g(t') \leq C^*$; finally $g(t') \geq C^*$, being the cost of a path. With real costs the comparisons of line 6 and the grouping of Δf by the domain only widen the set of the successors put in OPEN by the slack, which the argument absorbs as it absorbs C. □

The consistency matters to EPEA* only: with a $\Delta f < 0$ on an edge, the first selection asks for $\Delta f = 0$ and the next ones for larger values, so that the successor of that edge is never generated through it, which the pseudocode of the header does not exclude and the library does not check. PEA*, on the contrary, puts in OPEN every successor of $f \leq F(n) + C$, including those of a smaller f.

Proposition 4.22 (What partial expansion stores and generates). Under the hypotheses of Theorem 4.21, every node put in OPEN by PEA* before the end has $f \leq C^* + C$ (up to the slack), and every node put in OPEN by EPEA* has $f \leq C^*$; hence the surplus nodes, of $f > C^* + C$ and of $f > C^*$ respectively, are never stored. A selection of EPEA* generates only nodes that it puts in OPEN or finds to be duplicates, and each of them at most once per selection.

Proof. By Theorem 4.21 each entry selected before the end has $F \leq C^*$, and line 6 puts in OPEN only successors with $f \leq F + C$; EPEA* puts in OPEN only successors with $f = F$. The last claim is the definition of the operator selection function, which passes each successor whose Δf is δ once, and nothing else. □

The price is time: PEA* generates all the successors of a node at each of its selections, and a node with m distinct values of f among its successors may be selected m times; EPEA* saves the generations, but it cannot leave out the known successors without generating them, so that it makes the selections of PEA* without the skip while generating fewer nodes. We remark that the lemma does not bound the number of the selections of PEA* with the skip by that of PEA* without it, since the order of the ties of OPEN may change; the test `clas sic` checks that it is not larger on the grids of 60×60 cells of three random maps with 4 and 8 moves, that neither is larger than the selections of the code before the skip (466, 750 and 574 with 4 moves, 752, 2167 and 1495 with 8), and that EPEA* generates fewer nodes than PEA*; on instances of the 15-puzzle of Korf, EPEA* generates and stores fewer nodes than A*, and PEA* stores fewer. Figure 4.9 follows the selections of a node.

4.10 Fringe search

IDA* starts each iteration from the root and regenerates every node of the iterations before; fringe search, of Björnsson, Enzenberger, Holte and Schaeffer [115], keeps instead the frontier of IDA*, the *fringe*, in a list L, so that an iteration starts from the nodes where the one before stopped, and the g of every state seen in a cache C,

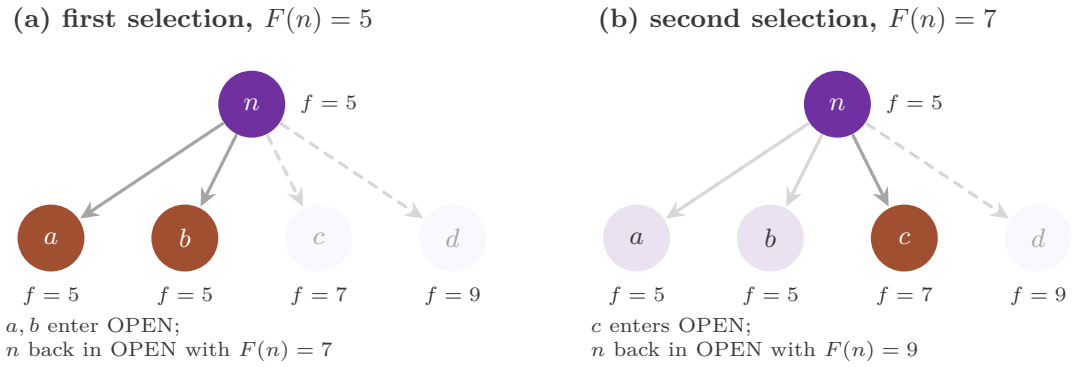


Figure 4.9. Partial expansion with $C = 0$. The node n, with $f(n) = 5$, has successors of $f = 5, 5, 7, 9$. (a) At its first selection, with $F(n) = 5$, only the two of $f = 5$ enter OPEN, and n goes back with $F(n) = 7$, the least f of the others. (b) At the second selection the successor of $f = 7$ enters OPEN, and n goes back with $F(n) = 9$; the successor of $f = 9$ is never stored if $C^* < 9$. EPEA* generates at each selection only the successors drawn in rust.

so that a state is not expanded again through a path that is not cheaper. Its memory is that of A*, while its time per node is smaller, since the list costs $O(1)$ per operation instead of the $O(\log n)$ of a heap, at the price of visiting the nodes of the fringe above the threshold at each iteration.

Algorithm 4.11 (Fringe search, the pseudocode of the header `FringeSearch.h`).

```
 1  procedure FRINGE(s0)
 2     L ← ⟨s0⟩;  C[s0] ← (0, nil);  θ ← h(s0)
 3     while L ≠ ⟨⟩ do
 4        θ' ← ∞
 5        for each node n of L, from the first to the last do
 6           if g(n) + h(n) > θ then θ' ← min(θ', g(n) + h(n));  continue
 7           if n is a goal then return the path to n
 8           for each successor c of n do
 9              g' ← g(n) + c(n, c)
10              if c ∈ C and g' ≥ g(c) then continue
11              remove c from L if it is there;  C[c] ← (g', n)
12              insert c in L just after n
13           remove n from L
14        θ ← θ'
```

The list is doubly linked through the nodes of the cache, with a sentinel, and a flag tells whether a node is in it; the successors are those of `for_each_child`, inserted after n in the order of their generation, so that they are visited right after their parent, which is the depth-first order of IDA*. The declarations, quoted by hand, are

```
template< Domain D >
class FringeSearch {
 public:
 explicit FringeSearch( const D & domain ,
                        const SearchOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void );
 [[nodiscard]] std::size_t iterations( void ) const noexcept;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
fringe_search( const D & domain , const SearchOptions & options = {} );
```

where the constructor throws `std::invalid_argument` for a weight other than 1, `iterations()` gives the passes of the last search, `node_limit` bounds the nodes of the cache, tested after each expansion, `expanded` counts the expansions (line 8), `reexpanded` those of a state expanded before, and `stored` the states of the cache.

Theorem 4.23 (Optimality and termination of fringe search). Let h be admissible and finite on the states that

the search meets. (i) At the start of each pass the threshold satisfies $\theta \leq C^*$, the thresholds increase strictly, and the first goal found has $g = C^*$. (ii) On a finite graph Algorithm 4.11 terminates, and it returns `unsolvable` if no goal is reachable.

Proof. We first show an invariant: a state of C that has not been expanded with its current g is in L. Indeed, a state gets a new g only at line 11, after which line 12 inserts it in L, and it leaves L only at line 11, to be inserted again, or at line 13, after its expansion. Let P be an optimal path whose goal has not been returned, and u the first node of P that has not been expanded with $g = g^*(u)$. Then $u = s_0$, which is in L with $g = 0$, or the predecessor p of u on P was expanded with $g^*(p)$, which gave u the value $g^*(u)$ at line 11 or found it there already (line 10); since g cannot fall below g^*, it has not changed since, and by the invariant u is in L with $g(u) = g^*(u)$, i.e., $f(u) \leq C^*$.

(i) Consider a pass that returns no goal. Every node in L at its end was visited by the pass, since a node is always inserted just after the node being expanded, i.e., ahead of the scan; and at its last visit it was skipped at line 6 with its current g, since a change of g after the visit would have moved it ahead of the scan again, and an expansion would have removed it. Hence u was skipped with $f(u) > \theta$, and $\theta' \leq f(u) \leq C^*$, while $\theta' > \theta$ since only the nodes above θ contribute to it. The first threshold is $h(s_0) \leq C^*$, and by induction $\theta \leq C^*$ at each pass. A goal t is returned only within the threshold, with $h(t) = 0$, and thus $g(t) \leq \theta \leq C^*$, while $g(t) \geq C^*$, being the cost of a path. (ii) On a finite graph the values of g of a state are finitely many and strictly decreasing (the argument of Theorem 2.11), each insertion in L is paid by one of them, and a pass ends; the thresholds are values of f of these pairs, finitely many, and increase strictly. If no goal is reachable, every state of C is eventually expanded with its last g, and L empties. □

On an infinite graph with finite branching and costs at least $\varepsilon > 0$, the argument of (ii) of Theorem 2.11 applies in the same way, since each node expanded has $g \leq \theta \leq C^*$. We remark that the cost returned is the g of the goal, and that the path is that of its parents: with an admissible h both are C^*, since the path rebuilt costs at most $g(t)$ and at least C^*. The test `classic` checks the costs on 200 states of the 8-puzzle, on instances of the 15-puzzle of Korf and on grids of 50×50 cells with 8 moves, the node limit, and a graph with no path. The finiteness of h is needed by the code, and not only by the proof: a dead end whose h is `infinite_cost()` makes $g + h$ overflow with integer costs (`FringeSearch.h:155`), whereas `BeamSearch` and `RBFS` drop such a successor when it is generated.

4.11 The bounds of iterative deepening

An iteration of IDA* of bound β_k prunes the nodes of $f > \beta_k$, and the classical next bound is the least of their f, ℓ_k, so that each iteration adds at least one value of f. When the values of f are many and close, e.g., with costs that are not integers, or on the grids, whose number of nodes grows slowly with the bound, the iterations are as many as the values of f below C^*, and the search expands the nodes of the first ones again at each of them, i.e., quadratically many times in the worst case of Proposition 4.9. IDA*-CR, of Sarkar, Chakrabarti, Ghose and De Sarkar [132], chooses instead a bound beyond which enough pruned nodes lie, so that the iterations grow geometrically, and pays for it with an iteration whose bound may exceed C^*.

The options are those of `DeepeningOptions` (`Deepening.h`), which `IDAOptions` inherits and which the iterative-deepening searches of the parallel chapters share:

```
struct DeepeningOptions {

 /// the growth \f$ r \f$ of IDA*-CR: if larger than 1, the next bound is
 /// the one of CutoffHistogram, otherwise the least f pruned
 double growth = 0;

 /// true to complete the iteration of the first goal as a branch-and-bound
 /// when its bound is above the least f pruned by the iteration before, so
 /// that the path is optimal; false to stop at the first goal, whose cost
 /// is then within \f$ \beta_{k+1} - \ell_k \f$ of the optimum
 bool optimal = true;

 };  // end( struct( DeepeningOptions ) )
```

With the growth \$r = \$ `growth` > 1, the next bound is chosen by a `CutoffHistogram< Cost >`, in which each iteration records the f of the nodes that it prunes; its interface, quoted by hand without the bodies, is

```
template< class Cost >
class CutoffHistogram {
 public:
 void reset( Cost b , bool buckets ) noexcept;
 void restart( void ) noexcept;
 void add( Cost f ) noexcept;
 void merge( const CutoffHistogram & o ) noexcept;
 [[nodiscard]] Cost low( void ) const noexcept;
 Cost next( double growth , std::uint64_t expanded );
 };
```

where `reset( b , buckets )` empties it for the iteration of bound b (counting the f in buckets only if `buckets`), `restart()` forgets the last step, `add( f )` records a node of $f > b$, `merge( o )` adds the nodes of another histogram of the same iteration (that of another worker, in a parallel search), `low()` is ℓ_k, and `next( r , E_k )` is the next bound for an iteration of E_k expansions. The buckets are those of the binary exponent of the excess $f - \beta_k$, the excesses in $[2^{x-1}, 2^x)$ in the bucket x, so that the histogram has a fixed size whatever the costs (2100 buckets, one per exponent of a `double`), and its buckets are fine near the bound and coarse far from it. The rule of `next` is the following. If $r \leq 1$, or no node was pruned, the next bound is ℓ_k (the infinite cost if none). Otherwise let x^* be the first bucket such that the buckets up to x^* hold at least $(r-1)E_k$ nodes, or the last nonempty one if all of them hold fewer, and let e be the largest f in the buckets up to x^*. If all the buckets hold fewer than $(r-1)E_k$ nodes, $E_k < rE_{k-1}$ and the last step $\beta_k - \beta_{k-1}$ is positive, e is raised to at least $\beta_k + 2(\beta_k - \beta_{k-1})$, i.e., the step doubles, which is what a grid needs, where the children of the nodes of the last layer of f lie just above it and each layer adds few nodes. In all cases $\beta_{k+1} = \max(\ell_k, e)$.

Proposition 4.24 (The bounds of IDA*-CR). The bounds chosen by `CutoffHistogram::next` satisfy $\beta_k < \ell_k \leq \beta_{k+1}$; hence they increase strictly, and every iteration enters, as long as no goal is entered, the nodes of the iteration before and at least one node more. If $r > 1$, an iteration $k+1$ that enters no goal, whose bound was chosen with at least $(r-1)E_k$ pruned nodes in the buckets taken, makes at least rE_k expansions; if all the iterations but the last are of this kind, the expansions of the K iterations sum to at most $E_K\, r/(r-1)$.

Proof. A pruned node has $f > \beta_k$, hence $\ell_k > \beta_k$, and $\beta_{k+1} = \max(\ell_k, e) \geq \ell_k$. An iteration that enters no goal visits every path of its tree whose nodes are within its bound, and the tree of the bound β_{k+1} contains that of β_k; thus the iteration $k+1$ enters every node entered by the iteration k (each expanded there, no goal having been entered) and every node pruned by the iteration k with $f \leq \beta_{k+1}$, among which the one of $f = \ell_k$. A bucket holds the excesses of a range of binary exponents, and the ranges grow with the index; hence the nodes counted in the buckets up to x^* have $f \leq e \leq \beta_{k+1}$, and there are at least $(r-1)E_k$ of them, counted once per generation, i.e., once per node of the tree. Each of them is entered, and expanded since it is not a goal, besides the E_k nodes of the iteration k; thus $E_{k+1} \geq rE_k$. The sum follows from Proposition 4.9 with $b' = r$. □

Proposition 4.25 (No bound above C^* without a reason). If h is admissible and a goal is reachable, then $\ell_k \leq C^*$ for every iteration k that enters no goal; hence an iteration whose bound is ℓ_k, as every bound of the classical rule is, finds a goal only of cost at most C^*, i.e., of cost C^*.

Proof. This is the argument of Lemma 4.4: an iteration that enters no goal cuts a node u_{j+1} of an optimal path, with $f(u_{j+1}) \leq C^*$, and $\ell_k \leq f(u_{j+1})$. A goal entered by an iteration of bound β has $g = f \leq \beta$. □

Proposition 4.26 (The cost of the first path of IDA*-CR, and the branch-and-bound). Let h be admissible, $w = 1$, and let the iteration $k+1$ be the first to enter a goal, with the bound $\beta_{k+1} > \ell_k$. (i) If `optimal` is false, the search returns the first goal entered, of cost c with

$$c - C^* \leq \beta_{k+1} - \ell_k,$$

and `Solution::bound` is $\min(c, \ell_k)$, a lower bound on C^*. (ii) If `optimal` is true (the default), the iteration $k+1$ goes on as a depth-first branch-and-bound, i.e., at a goal of cost g smaller than the incumbent z it records the path and continues, and it does not enter a successor of $f \geq z$; at its end the incumbent is C^*, and `Solution::bound` is C^*. If $\beta_{k+1} = \ell_k$, the first goal entered is returned, and it is optimal by Proposition 4.25.

Proof. (i) The iteration k entered no goal, and by Proposition 4.25 $C^* \geq \ell_k$; the goal entered has $c = g = f \leq \beta_{k+1}$; subtracting gives the bound, and ℓ_k is the lower bound that the code returns as `key_lower_bound( low )`, the minimum with c being harmless. (ii) Let P be an optimal path whose moves are all kept. Every node of P has $f \leq C^* \leq c \leq \beta_{k+1}$, c being the cost of the first goal entered, and it is not pruned by the incumbent z while $z > C^*$, since it has $f \leq C^* < z$; as in Lemma 4.4, the iteration follows P to its goal unless the incumbent is already C^*, and then records C^*. The incumbent never falls below C^*, being the cost of a path. □

The rule of doubling the step has no such guarantee on the number of iterations, which it only makes grow geometrically in the bound; where the iterations grow by r anyway, as on the sliding-tile puzzles, the bounds stay the classical ones and nothing is lost. We remark that the code returns the incumbent of the branch-and-bound also when the node limit stops the iteration before its end, with `Solution::bound` equal to its cost, which (ii) does not cover: the cost is then only an upper bound on C^* (`IDAStar.h:210-217`, the test of the incumbent preceding that of the limit). The test `idastar` checks IDA*-CR with $r = 2$ on grids with 8 moves and costs from 1 to 3, where the classical bounds are many, and on two instances of the 15-puzzle of Korf, against A*, and that the first path of the variant with `optimal` false is not cheaper than the optimum.

4.12 Real-time search

The searches above find a whole path before the first move. An agent that must move within a fixed time, e.g., a unit of a game on a map, interleaves instead a search of bounded size around its current state with its moves, and it learns on the way a heuristic H that makes its later decisions better: this is the agent-centered real-time search of LRTA* (Korf [118]) and of its generalization LSS-LRTA* (Koenig and Sun [119]), both in `RealTimeSearch.h`. From the state s of the agent, an A* ordered by $g + H$ (and then by the larger g, and first in first out) expands at most L states, the *lookahead*, stopping earlier if a goal is first in OPEN; then the values H of the states that it expanded are raised by a Dijkstra search backward from its OPEN, i.e., to the least $c(u, v) + H(v)$ over the paths to OPEN within the local space, and the agent moves along the path to the state of least $g + H$ in OPEN.

Algorithm 4.12 (LSS-LRTA*, the pseudocode of the header).

```
 1  procedure LSS-LRTA*(s₀, L)
 2     s ← s₀;  H(u) ← h(u) for each state u not yet met
 3     while s is not a goal do
 4        A* from s on g + H, at most L expansions, stopped when a goal
             is the first of OPEN;  CLOSED_s and OPEN_s its lists
 5        H(u) ← ∞ for each u ∈ CLOSED_s
 6        while some u ∈ CLOSED_s is not yet taken do     ▷ Dijkstra
 7           v ← the state of least H in OPEN_s ∪ CLOSED_s not yet taken,
                 whose H is then final
 8           for each u ∈ CLOSED_s with v a successor of u do
 9              H(u) ← min(H(u), c(u, v) + H(v))
10        move the agent along the path to the first state of OPEN_s
```

With $L = 1$ this is LRTA*, whose update is $H(s) = \min_t c(s, t) + H(t)$ over the successors t of s and whose move is to the best successor. The options are

```
struct RealTimeOptions : public SearchOptions {

 /// the expansions L of the local search (1: LRTA*)
 std::size_t lookahead = 1;

 /// the trials, each from the initial state with the H learned so far
 std::size_t trials = 1;

 /// the trials go on, up to trials, until one leaves H unchanged
 bool until_convergence = false;

 /// the most moves of a trial (0: no limit)
 std::size_t max_moves = 0;

 };  // end( struct( RealTimeOptions ) )
```

where a lookahead or a number of trials of 0 is taken as 1, and the declarations, quoted by hand, are

```
template< Domain D >
class RealTimeSearch {
 public:
 explicit RealTimeSearch( const D & domain ,
```

```
                        const RealTimeOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void );
 [[nodiscard]] const std::vector< cost_t< D > > & trials( void ) const
  noexcept;
 [[nodiscard]] cost_t< D > learned( const state_t< D > & s ) const;
 };

template< Domain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
lss_lrta_star( const D & domain , const RealTimeOptions & options = {} );
```

A *trial* is a run of Algorithm 4.12 from s_0 with the H learned so far; `solve()` runs up to `trials` of them, stopping early, with `until_convergence`, at the first trial that changes no value of H, or at the first trial that fails, and returns the path of the last one, which may visit a state more than once, with its cost, the sum of the costs of the moves. `trials()` gives the cost of each trial (the infinite cost for one that failed), and `learned ( s )` the value of H at s, $h(s)$ if it was never updated; H is kept in a hash table across the trials and across the calls of `solve()`, whose size is the counter `stored`. A trial fails with `Status::node_limit` after `max_moves` moves (0 for no limit), and with `Status::unsolvable` when the local A* empties its OPEN, i.e., at a dead end from which no state outside the local space is reachable. The local A* does not reopen a closed state, which only changes the path that the agent follows, not the update below; the heuristic passed with each successor is not used, H being read from the table or from `heuristic( t )`.

Lemma 4.27 (The update of LSS-LRTA*). After lines 5 to 9, for each $u \in \text{CLOSED}_s$,

$$H(u) = \min \Big\{ \sum_{i=1}^{m} c(x_{i-1}, x_i) + H(x_m) \Big\},$$

the minimum being over the paths $u = x_0, x_1, \ldots, x_m$ with $x_0, \ldots, x_{m-1} \in \text{CLOSED}_s$ and $x_m \in \text{OPEN}_s$, and $H(x_m)$ the value before the update (∞ if there is no such path); the values outside CLOSED_s do not change.

Proof. Every successor of a state of CLOSED_s has been generated, and it is in CLOSED_s or in OPEN_s. Lines 5 to 9 are the algorithm of Dijkstra on the reversed edges of the local space, from the sources OPEN_s, whose distances are their values of H, which never change; with nonnegative costs it takes the states by nondecreasing distance, and the distance of each closed state when it is taken is the least cost of a path to a source, plus the value of the source. The code takes the open states with their H, pushes a closed state each time its H falls, drops the entries that are no longer current, and stops when every closed state has been taken or the queue is empty, the states left at ∞ being those with no path to OPEN_s. □

Theorem 4.28 (The learned heuristic stays admissible). If h is admissible, then $H(u) \le h^*(u)$ for every state u at any time of any trial; in particular, the cost of the first trial is not smaller than C^*, and `learned( s )` is a lower bound on $h^*(s)$.

Proof. By induction on the updates, $H = h$ being admissible at the start. Let $u \in \text{CLOSED}_s$ with $h^*(u) < \infty$, and let Q be an optimal path from u to a goal. No goal is in CLOSED_s, since the local A* stops when a goal is the first of OPEN, before expanding it; let x_m be the first state of Q that is not in CLOSED_s, which is then in OPEN_s, being a successor of a closed state. By Lemma 4.27 and by the induction, $H(u) \le c(Q[u, x_m]) + H(x_m) \le c(Q[u, x_m]) + h^*(x_m) = h^*(u)$, $Q[u, x_m]$ being the prefix of Q to x_m and the last equality its optimality. The states outside CLOSED_s keep their values. The cost of a trial is that of a path from s_0 to a goal, and hence at least C^*, whatever H. □

What the library does not prove is the convergence. Koenig and Sun [119] state (their theorem 3, after Korf [118] for LRTA*) that on a finite graph with positive costs, where a goal is reachable from every state, each trial reaches a goal, and the trials repeated with the H learned converge to an optimal path, after which H no longer changes along it; we report this as stated in the header and in the papers, without having checked their proofs. The test `realtime` checks it on random grids of 30×30 cells with 4 moves and of 40×40 cells with 8 moves and costs from 1 to 3, for the lookaheads 1, 8, 32 and 64: the trials, up to 2000 or 3000 with `until_convergence`, end with a path of the cost of A*, the cost of the first trial is not smaller, and $H \le h^*$ on every state, h^* being computed by a search of Dijkstra backward from the goal (Theorem 4.28). Figure 4.10 shows one update.

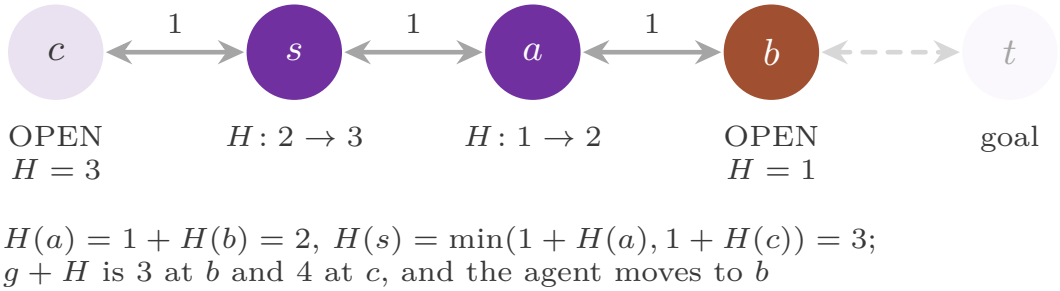


Figure 4.10. One step of LSS-LRTA* with $L = 2$ on a path of unit costs whose goal is t, with an admissible h. The local A* from s expands s and a (violet), and its OPEN holds b and c with their H; the update of Lemma 4.27 sets $H(a) = 1 + H(b) = 2$ and $H(s) = \min(1 + H(a), 1 + H(c)) = 3$, and the agent moves to b, the state of OPEN of least $g + H$ (rust).

4.13 Hierarchical path-finding

On a map that is searched many times, with different starts and goals, one can pay once for an abstraction of the map and then answer each query with a search of the abstract graph and of two small parts of the map. This is Hierarchical Path-Finding A* (HPA*) of Botea, Müller and Schaeffer [120], which `HPAStar< Moves >` (`HPAStar.h`) implements with one level of abstraction on a `GridMap` with 4 or 8 moves. The map is split in *clusters* of $C \times C$ cells. Along the border of two adjacent clusters, every maximal run of pairs of free cells facing each other is an *entrance*, which gets a *transition*, i.e., a pair of abstract nodes, one on each side, joined by the move between them, in its middle if the run is shorter than the split length, and one at each end otherwise. Inside a cluster, its abstract nodes are joined by the shortest paths that stay in the cluster, computed once by the algorithm of Dijkstra and kept with the edges. A query inserts the start s and the goal t into the abstract graph, joined to the abstract nodes of their clusters by the shortest paths inside them, searches it with A* on the distance of the Grid, and refines the abstract path by concatenating the paths kept; when s and t are in the same cluster, the shortest path inside it is a candidate too, and the abstract search stops as soon as the node selected has a g not smaller than that candidate. The path is then *smoothed* as in [120]: from each of its cells, a straight walk in each of the directions of the moves that meets a later cell of the path replaces the piece between them if it costs less, so that the detours through the transitions are cut. The options are

```
struct HPAOptions {

 int cluster = 10;            ///< the side C of a cluster
 int split = 6;               ///< the entrances at least this long get two
                              ///< transitions
 bool smooth = true;          ///< the path is smoothed [see HPAStar]

 };  // end( struct( HPAOptions ) )
```

and the declarations, quoted by hand, are

```
template< int Moves = 4 >
class HPAStar {
 public:
 explicit HPAStar( const GridMap & map , const HPAOptions & options = {} );
 [[nodiscard]] std::size_t nodes( void ) const noexcept;
 [[nodiscard]] std::size_t edges( void ) const noexcept;
 [[nodiscard]] std::uint64_t preprocessing( void ) const noexcept;
 [[nodiscard]] Solution< typename Grid< Moves >::state_type ,
                         typename Grid< Moves >::cost_type >
  solve( int sx , int sy , int gx , int gy ) const;
 };
```

where a state is the index of a cell and the cost is an integer with 4 moves and a `double` with 8. The constructor builds the abstract graph, and throws `std::invalid_argument` if the map has no free cell or the side of a cluster is not positive; `nodes()` and `edges()` give its size and `preprocessing()` the expansions of its searches

of Dijkstra. The query `solve( sx , sy , gx , gy )` throws `std::invalid_argument` if the start or the goal is blocked, and returns the path with the counters of the insertion of s and t and of the abstract search in `expanded` and `generated`.

Proposition 4.29 (What HPA* returns). (i) Before the smoothing, the path of a query is a cheapest one among the *abstract* paths, i.e., the concatenations of a shortest path inside the cluster of s from s to one of its abstract nodes, of abstract edges, and of a shortest path inside the cluster of t from one of its abstract nodes to t, together with the shortest path inside the cluster when s and t share it. (ii) The smoothing does not increase the cost, and the cost returned is that of the path returned, which is a path of the grid; hence it is at least C^*. (iii) The search is not complete: a query may fail although t is reachable from s.

Proof. (i) The distance of the Grid (Manhattan or octile, times the least cost of a free cell) is consistent on the grid (Chapter 14), and an abstract edge costs as a path of the grid between its ends; hence the heuristic is consistent on the abstract graph, by the triangle inequality along that path, and the abstract A* selects the goal with the least cost of an abstract path (Theorem 2.13). When it stops early, the node selected has $f \geq g \geq$ the candidate, and every abstract path not yet found costs at least the least f of OPEN, so that the candidate is not worse. (ii) A piece is replaced only by a walk of moves that the grid allows (a blocked cell or a cut corner ends the walk) and only if the walk costs less; the cost is then summed again over the moves of the final path. (iii) A transition is placed in the middle or at the ends of an entrance, and when a cluster is disconnected inside, a cell of an entrance may be reachable within its cluster while the cell of its transition is not; then the abstract graph lacks a connection that the map has, as in the original method. □

The path is therefore not optimal in general, since it goes through the transitions; the test `hpa` checks on four random maps of 64×64 cells, with clusters of 8 and 16 cells, that the paths are valid paths of the grid and never cheaper than those of A*, that at least 90% of the queries that A* solves are solved, and that the worst ratio to the cost of A* is below 1.5 with 8 moves, together with a query from the goal and one inside a single cluster, which is optimal. The query costs a search of the abstract graph and of two clusters instead of a search of the map, which is the point of the method.

4.14 Baselines: breadth-first and cost-bounded depth-first search

Two searches of the library serve as baselines, against which the others are measured. `BreadthFirst` (`BreadthFirst.h`) is the breadth-first search of a `Domain` with duplicate detection, which ignores both the costs and the heuristic, with the goal test at the generation.

Algorithm 4.13 (Breadth-first search, the pseudocode of the header).

```
 1  procedure BFS(s0)
 2     if s0 is a goal then return the path (s0)
 3     CLOSED ← {s0};  L ← (s0)
 4     while L ≠ ∅ do
 5        L' ← ()
 6        for each u of L, in order, do
 7           for each successor v ∉ CLOSED of u do
 8              add v to CLOSED, with parent(v) ← u
 9              if v is a goal then return the path to v
10              append v to L'
11        L ← L'
12     return unsolvable
```

The layers L and L' are two vectors of nodes and CLOSED is a `NodeTable`; the heuristic is computed by the domain only because the successors come with it. The declarations, quoted by hand, are

```
template< Domain D >
class BreadthFirst {
 public:
 explicit BreadthFirst( const D & domain ,
                        const SearchOptions & options = {} );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };

template< Domain D >
```

```
[[nodiscard]] Solution< state_t< D > , cost_t< D > >
breadth_first( const D & domain , const SearchOptions & options = {} );
```

with `node_limit` on the nodes of CLOSED, tested after each expansion, and `expected_nodes` sizing CLOSED. The search stops at the first goal generated, but the domain gives all the successors of its parent, and those that follow the goal are counted in `generated` without being looked up or stored.

Proposition 4.30 (Breadth-first search). Algorithm 4.13 returns a path with the least number of edges to a goal, of cost the sum of its edges, which is the least cost when all the edges cost the same; on a finite graph it terminates, and returns `unsolvable` if no goal is reachable.

Proof. By induction on d, the layer L of the iteration d holds exactly the states at the least depth d that are not goals, and CLOSED those at depth at most d and, at the end of the iteration, those at depth $d+1$ generated so far: a state at least depth $d+1$ is a successor of a state at depth d, which is in L, and it is not in CLOSED before, since it is not at a smaller depth. The first goal generated is therefore at the least depth, and its path through the parents has that number of edges, the parents being set once. On a finite graph each state enters a layer at most once. □

The test `bfs` checks the costs on 21 states of the 8-puzzle, against their exact distances, the path of fewest edges of the map of Romania from Arad (Arad, Sibiu, Fagaras, Bucharest, of cost 450, not the cheapest one, of cost 418), the proof that a grid split by a wall is unsolvable, and the edge cases of A* (a start that is a goal, a start one move away, and the node limit). The *cost-bounded depth-first search* `cost_bounded_dfs( domain , bound , options )` is one iteration of IDA* with the bound β given, i.e., the first call of line 5 of Algorithm 4.3 with that β and no other (it calls `IDAStar::solve( bound )`, with the bound on f' if the options have a weight).

Proposition 4.31 (The cost-bounded depth-first search). Let the costs be at least some $\varepsilon > 0$ and the branching finite. The search terminates, and it returns a path whose nodes have $f \le \beta$, hence of cost at most β, if it finds one; if h is admissible, it finds one whenever $\beta \ge C^*$, and it returns `unsolvable` whenever $\beta < C^*$.

Proof. The iteration visits only paths whose nodes have $g \le f \le \beta$, i.e., with at most β/ε edges, finitely many. The goal entered has $g = f \le \beta$ (its h being 0 if h is admissible, and in any case $g \le g + h \le \beta$). If $\beta \ge C^*$, an optimal path whose moves are all kept has every node within the bound, and the iteration, which visits every path of its tree within the bound unless it enters a goal before, enters its goal or another one. If $\beta < C^*$, a goal entered would have $g \le \beta < C^*$, which no path has. □

The path need not be optimal when $\beta > C^*$: the test `bfs` checks that the search at the cost of two instances of the 15-puzzle of Korf finds a path of that cost and one less finds none, that on the map of Romania the bound 450 gives a path of cost between 418 and 450 and the bound 417 none, and the bound 0 at a goal and one move away.

4.15 Benchmark

Our benchmark of this chapter compares the searches that are optimal with an admissible heuristic and keep less than A*, or spend less per node, with A* on the same instances: IDA* and IDA*-CR, RBFS, PEA* and EPEA*, fringe search and DDD-A* on the 100 instances of the 15-puzzle of Korf [7] with the Manhattan distance, and IDA*-CR, PEA*, EPEA* and fringe search on the grids of [15], where the values of f are many and the iterations of IDA* are a burden. The measures are those of §2.16: each instance is solved in a process of its own, pinned to one core, with the clock of the search only, and the counters of the `Solution` are machine-independent.

Table 4.1. The memory-bounded searches against A*: for each search and set of instances, the instances solved, the seconds summed over the instances (each the median of its repetitions), the nodes expanded and the nodes stored summed over the instances, and the millions of expansions per second.

search	instances	solved	seconds	expanded	stored	expansions per second (millions)

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

4.16 An example

The following example, a case of the tester, solves an instance of the 15-puzzle of Korf with IDA*, whose memory is the path, and with DDD-A* on 16 files per bucket and small buffers in a directory of its own; it checks the costs, the counter `stored` of IDA* (Proposition 4.8), that DDD-A* expands each state at most once (Theorem 4.15), that no file is left behind, and that IDA* stops at a limit of 1000 expansions.

```
const FifteenPuzzle d( ki.board );
const auto i = idastar( d );                  // memory: the path only
DDDOptions x;
x.partitions = 16;              // files per bucket (default 64)
x.buffer_bytes = 1 << 12;       // buffer of each file written (2^16)
x.directory = dir.path();       // if empty, the temporary of the system
const auto e = ddd_astar( d , SearchOptions{} , x );
// the node limit bounds the expansions of IDA*, the states closed by
// DDD-A*
SearchOptions lim;
lim.node_limit = 1000;
const auto l = idastar( d , lim );
```

4.17 Notes and references

DFID and IDA* are due to Korf [7], who proved that DFID is asymptotically optimal among the brute-force tree searches in time and space, and IDA* with a disjoint pattern database is the search of Korf and Felner [22] on the 24-puzzle; RBFS is also due to Korf [116], the transposition tables in iterative deepening were studied by Reinefeld and Marsland [161], and the stochastic node caching by Miura and Ishida [162]. The External A* is that of Edelkamp, Jabbar and Schrödl [39], the hash-based DDD that of Korf [40], applied by Korf and Schultze to the complete breadth-first search of the 15-puzzle [156], and the two-bit breadth-first search that of Korf [26]. The BDDs are due to Bryant [163], and the symbolic search with them and the beam search are surveyed by Edelkamp and Schrödl [10], the breadth-first beam search going back to Lowerre [171] and the beam that keeps its dropped nodes in CLOSED, with its complete variant by backtracking, to Zhou and Hansen [117]; the width-based search is due to Lipovetzky and Geffner [104]. Partial expansion is due to Yoshizumi, Miura and Ishida [42], its enhanced form with the operator selection function to Felner et al. [113] and Goldenberg et al. [114], fringe search to Björnsson, Enzenberger, Holte and Schaeffer [115], the controlled bounds of IDA*-CR to Sarkar, Chakrabarti, Ghose and De Sarkar [132], LRTA* to Korf [118], LSS-LRTA*, with the proofs of admissibility and convergence that we cite in §4.12, to Koenig and Sun [119], and HPA* to Botea, Müller and Schaeffer [120]; the moves that a tree search of Rubik's cube need not try are those of Korf [59].

As to where the library stands, the following facts are stated from the code. `IDAStar` is the algorithm of [7] with the test of the bound at the generation and the parent pruning of §2.9, and it runs with the pattern databases of Chapter 5 on the 24-puzzle (the README reports, for 50 ends of random walks of 150 moves, 49 instances solved within 10^{10} expansions each, a count that does not depend on the machine). `DDDAStar` uses the hash-based detection rather than the sorting of [39], but it keeps all the closed files and does not implement the frontier search, nor a parallel expansion of the files as in [156]; its order by increasing g within a layer may cost many more expansions than A* in the last layer. `BeamSearch` keeps the dropped nodes in CLOSED, which bounds its memory only by the depth, and has no beam stack; `RBFS` and `FringeSearch` follow the methods of [116] and [115], the latter with the cache of g and the insertion of the children after their parent that make it the depth-first order of IDA*; `PEAStar` adds to [42] the skip of the known successors of Lemma 4.20 and the slack of the real costs, and `EPEAStar` relies on an operator selection function written for each domain, which the sliding-tile puzzles and the grids provide; `IDAStar` takes the bounds of IDA*-CR from a histogram of binary exponents, adds the doubling of the step of §4.11 for the iterations that grow slowly, and completes the iteration of the first goal as a branch-and-bound; `RealTimeSearch` runs the trials of LRTA* and LSS-LRTA* on any `Domain`, with the update by the algorithm of Dijkstra of [119]; `HPAStar` has one level of abstraction and the smoothing of [120], and works on the grids only. The depth-first branch-and-bound of a `Domain`, the transposition tables and the stochastic node caching of IDA*, and the symbolic search are not in the library.

4.18 Exercises

Exercise 4.1 (discussion, no test). On a grid with 8 moves, when any path to the goal will do and its length does not matter, is depth-first branch-and-bound a good choice? Compare it with a greedy best-first search by h and with A*, and consider what happens to the incumbent U and to the paths that reach the same cell.

Exercise 4.2 (`ctest -R PBeFS.idastar`). IDA* usually expands more nodes per second than A* on the 15-puzzle. Explain why from Algorithm 4.3 and the structures of Chapter 3 (what each expansion touches in memory), and use the counters `expanded` and `stored` of the `Solution` of both on the instances of the test to compare the work and the memory.

Exercise 4.3 (discussion, no test). With a branching factor of 4 and a number of nodes that grows by that factor per level, an external search that can store 1000 times more nodes than A* reaches a depth larger by how much? Show that the answer is $\log_4 1000$, i.e., about 5 levels, and discuss what this means for `DDDAStar`.

Exercise 4.4 (`ctest -R PBeFS.tiles24`). Which of the searches of this chapter would you choose for problems whose solutions are deeper still, if the time were unlimited but the memory were not? Argue from the memory of each (Propositions 4.8, 4.11 and 4.16), and look at how the test runs IDA* with a pattern database on the 24-puzzle.

Exercise 4.5 (`ctest -R PBeFS.idastar`). Prove the bound $(C^* - h(s_0))/2 + 1$ on the iterations of IDA* on the sliding-tile puzzles with the Manhattan distance (Proposition 4.7) in detail, i.e., prove that a move changes the Manhattan distance by exactly 1, and check on the instances of the test that `stored` equals $C^* + 1$ (Proposition 4.8).

Exercise 4.6 (`ctest -R PBeFS.idastar`). Show that IDA* reports the acyclic graph of the test unsolvable, and build a graph of three states with a cycle and no goal on which it does not terminate without the node limit; what is the sequence of its bounds?

Exercise 4.7 (`ctest -R PBeFS.ddd`). Prove in detail the identity $\mathit{generated} + 1 = \mathit{stored} + \mathit{duplicates}$ for a search of `DDDAStar` that ends without a goal, and that `expanded` = `stored` = 1250 on the grid with the wall of the test.

Exercise 4.8 (`ctest -R PBeFS.ddd`). Build a graph with an admissible but inconsistent heuristic on which the order of the buckets of DDD-A* would close a state with a g larger than its optimal one, and explain why `DDDAStar` refuses such a heuristic rather than reopening the state (the test runs it on the inconsistent graph of the test `reopen`).

Exercise 4.9 (discussion, no test). Give an example in which the transposition table of Proposition 4.10 makes IDA* skip a value of the bound, and show that the search still returns an optimal path; how many iterations does it save on your example?

Exercise 4.10 (`ctest -R PBeFS.ddd`). `IDAStar` bounds with `node_limit` the expansions, while `DDDAStar` bounds the states written to the closed files. Explain why each choice measures the resource that the search consumes, and check how the two tests stop the searches at their limits.

Exercise 4.11 (`ctest -R PBeFS.classic`). Run PEA* with $C = 0$ by hand on a node whose successors have $f = 5, 5, 7, 9$, as in Figure 4.9, and count its selections; then do the same with EPEA* and count the successors generated. Show that with $C = 2$ the node is selected twice, and explain why the test checks that PEA* with $C = 2$ is still optimal on 200 states of the 8-puzzle (Theorem 4.21).

Exercise 4.12 (`ctest -R PBeFS.classic`). Build a graph with an admissible heuristic and an edge of $\Delta f < 0$ on the only optimal path, and show that EPEA* with an operator selection function that follows its definition misses that path, whereas PEA* does not. Which hypothesis of Theorem 4.21 fails, and why do the domains of the test (the puzzles with the Manhattan distance, the grids with the octile distance) satisfy it?

Exercise 4.13 (`ctest -R PBeFS.classic`). Prove that fringe search and IDA* have the same thresholds on a tree, i.e., when no state is reached by two paths, and that on a graph the cache of fringe search may only skip expansions. Then show why the test runs RBFS on grids of only 8×8 cells, by counting the paths of Figure 4.4 that RBFS follows below a bound.

Exercise 4.14 (`ctest -R PBeFS.classic`). In the proof of Proposition 4.11, where is the absence of a cycle on the optimal path used? Give a domain with a cycle of cost 0 on which the proof would fail as written, and repair it by choosing the path.

Exercise 4.15 (`ctest -R PBeFS.classic`). Prove that the beam search of infinite width returns the cost of the shortest path on the 8-puzzle, whose costs are uniform, and give a graph with nonuniform costs on which the cost returned differs from the cost of the path rebuilt from the parents (cf. the remark after Proposition 4.2).

Exercise 4.16 (`ctest -R PBeFS.idastar`). On a grid with 8 moves, whose diagonal moves cost $\sqrt{2}$ times the straight ones, show that the classical bounds of IDA* may be as many as the values of f below C^*, and that the bounds of IDA*-CR with $r = 2$ are fewer. With `optimal` false, compute the bound of Proposition 4.26 on an example, and check that the test compares the first path with the optimum.

Exercise 4.17 (`ctest -R PBeFS.idastar`). Show that the branch-and-bound of Proposition 4.26 (ii) ends with the optimum even if its iteration has a bound much larger than C^*, and estimate how many nodes it enters in that case against an iteration of bound C^*. Why does the code not prove the optimality when the node limit stops that iteration?

Exercise 4.18 (`ctest -R PBeFS.realtime`). Run LRTA* by hand on a path of four states with unit costs and $h = 0$, and list the values of H after each move of the first trial and of the second; show that they stay below h^* (Theorem 4.28) and that the second trial costs the optimum. The test checks the convergence and the admissibility on random grids for the lookaheads 1 to 64.

Exercise 4.19 (`ctest -R PBeFS.realtime`). Prove that with a lookahead L at least the number of the states reachable from s_0, the first trial of LSS-LRTA* is an A* from s_0 followed by the moves of an optimal path, and therefore costs C^* when h is consistent.

Exercise 4.20 (`ctest -R PBeFS.hpa`). Draw a cluster whose free cells form two parts, such that the cell of the transition of an entrance is in one part and the start in the other, and show that HPA* fails on a query that A* solves (Proposition 4.29 (iii)). How does the test allow for such failures?

Exercise 4.21 (`ctest -R PBeFS.hpa`). Show that a query whose start and goal are in the same cluster of an empty map is answered at the optimal cost, as the test checks for the cells $(1, 1)$ and $(4, 5)$ of a map of 20×20 cells (cost 7), and give an example in which the abstract path is cheaper than the path inside the cluster.

Exercise 4.22 (`ctest -R PBeFS.bfs`). On the map of Romania, show that breadth-first search returns Arad, Sibiu, Fagaras, Bucharest, of cost 450, while the cost-bounded depth-first search with the bound 450 may return a path of cost between 418 and 450 (Proposition 4.31); which path does it return with the straight-line distance and the order of the roads of the test?

Exercise 4.23 (`ctest -R PBeFS.rubik`). Prove that skipping, after a turn of a face of Rubik's cube, the turns of the same face and, after a turn of the second face of a pair of opposite faces, those of the first, keeps a path of the same end and of the same length for every path, as `TreeOperatorDomain` requires, and derive the counts 18 and 243 of the first two depths of the tree that the test checks.

5. Heuristics and pattern databases

[Source: include/PBeFS/astar/PatternDatabase.h (BlankRule, PDBOptions, TilePattern, the encodings, PatternTable, AdditivePDB, SlidingTilePDB), include/PBeFS/astar/SlidingTile.h, include/PBeFS/astar/Grid.h, include/PBeFS/astar/TravelingSalesman.h, include/PBeFS/astar/SequenceScore.h, include/PBeFS/astar/AbstractionPDB.h (AbstractSpace, abstract_distances, AbstractionTable), Permutation.h (the rankings), PermutationPDB.h, Pancake.h, TopSpin.h, RubiksCube.h, TowersOfHanoi.h; the tests pdb, pdb-search, tiles24 and grid of test/test.cpp, and permutation, abstraction, pancake, topspin, rubik, rubik-pdb and hanoi of test/permutation.cpp; tools/PDBBench.cpp and tools/bench/pdb.sh]

We are concerned in this chapter with the heuristic h that a domain gives to the searches, i.e., with how admissible and consistent heuristics are obtained (the definitions and their consequences on A* are those of §2.4 to §2.7). Most of the chapter is devoted to the pattern databases of the sliding-tile puzzles, which store the exact distances of an abstraction of the puzzle in a table: the databases of Culberson and Schaeffer [21], summed over a partition of the tiles as in the disjoint databases of Korf and Felner [22] and of Felner, Korf and Hanan [23]. We first derive the heuristics of the domains of the library from relaxations and abstractions. Then we define the abstraction of a pattern under the two rules of the blank of the library, rank its states and build its table by a backward breadth-first search. We prove that the sum of the tables of a partition and its maximum with the reflection about the main diagonal are admissible, and we establish when they are consistent; then we give the encodings of the entries, from one byte down to one bit, with the argument by which the relative encodings recover the exact value. Then come the files of the tables, their sizes and a guide to the searches whose heuristic does not help. Sections 5.13 to 5.16 leave the puzzles of tiles: they build by layers the table of any abstraction whose states are ranked, rank the k-permutations in three ways, define the pattern databases of the permutation puzzles (the pancakes, TopSpin and Rubik's cube) with the conditions under which their maximum and their sum are admissible and consistent, and prove the properties of the heuristics of these puzzles and of the additive tables of the Towers of Hanoi. The benchmark, the notes and the exercises of both parts close the chapter (sections 5.17 to 5.19).

5.1 Relaxations and abstractions

A heuristic is usually the exact cost of a simpler problem. In a *relaxation*, the problem is given more edges or cheaper ones, and each path of the original graph is still a path, of no larger cost; hence the optimal cost of the relaxation from u is a lower bound on $h^*(u)$. For instance, the Manhattan distance of the sliding-tile puzzles is the optimal cost of the relaxation in which a tile may move onto an occupied cell. An *abstraction* maps the states to fewer abstract states and keeps the edges between their images; the distances of the abstract graph, which may be small enough to be computed once and stored, are again lower bounds. We state the property in general, since the pattern databases of the next sections are abstractions of this kind.

Definition 5.1 (Abstraction). Let $G = (V, E)$ be the graph of a domain, with costs c and goals $T \subseteq V$. An *abstraction* is a map $\psi : V \to V'$ onto the states of a graph $G' = (V', E')$ with costs c' and goals T', such that (i) $\psi(t) \in T'$ for each $t \in T$, and (ii) for each edge $(u, v) \in E$, either $\psi(u) = \psi(v)$, or $(\psi(u), \psi(v)) \in E'$ with $c'(\psi(u), \psi(v)) \leq c(u, v)$. The *abstraction heuristic* is $h_\psi(u) = d'(\psi(u))$, the least cost in G' of a path from $\psi(u)$ to a goal of T'.

Proposition 5.2 (Abstraction heuristics are consistent). The heuristic h_ψ of an abstraction is consistent, and hence admissible.

Proof. At a goal t, $\psi(t) \in T'$ and $h_\psi(t) = 0$. For an edge (u, v) with $\psi(u) = \psi(v)$, $h_\psi(u) = h_\psi(v) \leq c(u, v) + h_\psi(v)$; otherwise, a path of G' from $\psi(v)$ to a goal, preceded by the edge $(\psi(u), \psi(v))$, gives $h_\psi(u) \leq c'(\psi(u), \psi(v)) + h_\psi(v) \leq c(u, v) + h_\psi(v)$. Admissibility follows from Lemma 2.4. □

A relaxation is the special case in which ψ is the identity and G' has the same states, more edges or cheaper ones, and the same goals; its heuristic is thus consistent too.

Of course, the more states ψ merges, the smaller G' and the (typically) weaker the bound, and the art of an abstraction is to keep the part of the state that makes the problem hard. Two abstractions combine by the maximum, which keeps both properties (Lemma 2.5); they combine by the sum only when they do not count the same cost twice, which is the subject of §5.6.

5.2 The heuristics of the domains of the library

Each domain of Chapter 14 comes with one of the following heuristics, most of which are the exact costs of a relaxation.

- *Sliding-tile puzzles* (`SlidingTile.h`): the Manhattan distance, the sum over the tiles of the distance, in moves of a rook of one cell, between the cell of the tile and its goal cell. A move changes the cell of one tile by one unit, and hence the distance by exactly one, at the cost of one. Thus the heuristic is consistent, it has the parity of the true distance, and the domain computes it incrementally, from the value of the parent passed to `for_each_successor`. The number of the tiles out of their goal cell (the heuristic h_1 of the example `8Puzzle`) is also consistent, and weaker.
- *Grids* (`Grid.h`): with 4 moves, the Manhattan distance between the cell and the goal; with 8 moves, the *octile* distance $\max(\Delta x, \Delta y) + (\sqrt{2} - 1)\min(\Delta x, \Delta y)$, the cost of the path with as many diagonal moves as possible. Both are multiplied by the least cost of a cell (`Grid.h:851-854`), since a move pays the cost of the cell it leaves, times $\sqrt{2}$ on a diagonal. Both are consistent by Proposition 5.2, being the exact distances of the relaxation in which no cell is a wall, each cell costs the least cost $c_{\min}$ and every diagonal move is allowed: each move of the grid is a move of the relaxation, at a cost not smaller. In the relaxation, a path from (x, y) to (u, v) with a diagonal moves needs at least $\Delta x - a$ and $\Delta y - a$ straight moves along the two axes if $a \le \mu = \min(\Delta x, \Delta y)$, and at least $\max(\Delta x, \Delta y) - a$ if $a > \mu$; its cost over $c_{\min}$ is thus at least $\Delta x + \Delta y - (2 - \sqrt{2})a$ in the first case and $\max(\Delta x, \Delta y) + (\sqrt{2} - 1)a$ in the second, both at least $\max(\Delta x, \Delta y) + (\sqrt{2} - 1)\mu$, the cost of the path with μ diagonals. With the life costs of the boards of Burns et al., the heuristic is instead the exact cost of a relaxation in closed form (Proposition 14.32 and Proposition 14.33).
- *Traveling salesperson* (`TravelingSalesman.h`): the cost of a minimum spanning tree of the cities still to be visited, together with the current one and the first (cf. exercise 3.30 of [13]). The rest of the tour is a path through all these cities, and a path costs at least as much as a spanning tree of its cities; hence the heuristic is admissible. It is also consistent when the distances are symmetric. Let U be the set of the unvisited cities with the current one a and the first; a move to an unvisited city k, of cost $c(a, k)$, leaves the set $U \setminus \{a\}$ (or U itself if a is the first city), and a spanning tree of that set plus the edge (a, k) is a spanning tree of U, so that $h(s) \le c(a, k) + h(t)$; the last move, back to the first city, costs exactly the spanning tree of the two cities left, and the goal has $h = 0$ (the full proof, and the role of the symmetry, are in Proposition 14.23).
- *Explicit graphs* (`ExplicitGraph.h`): the heuristic given node by node by `set_heuristic`, e.g., the straight-line distance of the map of Romania [13], which is consistent when each road is at least as long as the segment between its ends.
- *Permutation puzzles, Towers of Hanoi, road graphs and navigation*: the gaps of the pancakes and the breakpoints of TopSpin, which are not the costs of a relaxation but are consistent by the arguments of §5.16, the tables of §5.15 and §5.16 (Rubik's cube and the Towers of Hanoi have no other heuristic), the great-circle distance scaled to the least ratio of the cost of a road to its length on the road graphs of DIMACS (Proposition 14.30), and the Euclidean distance to the disk of the goal for the navigation of a robot (Proposition 14.31).

On a grid with 8 moves the Manhattan distance is *not* admissible, whatever the cost of a diagonal move between 1 and $\sqrt{2}$. For instance, three diagonal moves of unit cells cost $3\sqrt{2} \approx 4.24$, while the Manhattan distance of their ends is 6 (Figure 5.1); the Euclidean distance is admissible there, and consistent, but it is weaker than the octile distance, which is the distance of the relaxation without walls. Finally, the sequence score of Nilsson (`SequenceScore.h`) adds to the Manhattan distance a term that counts the tiles out of sequence; it is not admissible, and A* then usually expands fewer nodes, but it may return a path that is not optimal (§2.4).

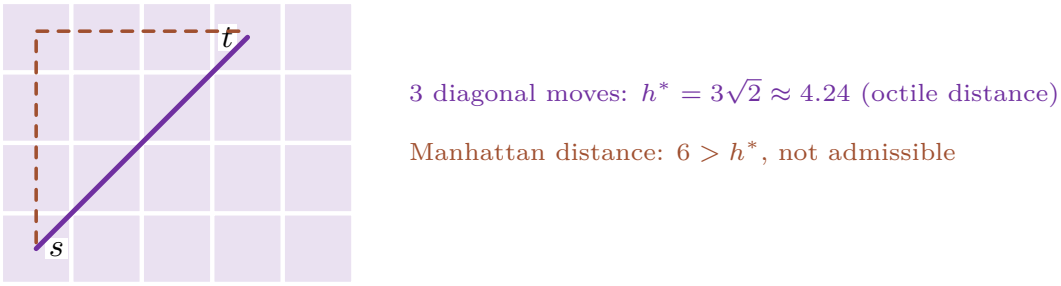


Figure 5.1. On a grid with 8 moves, a path of three diagonal moves (violet) costs $3\sqrt{2} \approx 4.24$, which the octile distance gives, while the Manhattan distance of its ends, the cost of the path with 4 moves (dashed), is 6: the Manhattan distance overestimates, and it is not admissible with 8 moves.

5.3 The abstraction of a pattern

In a sliding-tile puzzle, the board has $n = WH$ cells, and a state s places the tiles $1, \dots, n-1$ and the blank on distinct cells; a move exchanges the blank with a tile in an adjacent cell, at cost 1. A *pattern* is a set $\pi = \{t_1, \dots, t_k\}$ of tiles, and the *placement* of π in s is the vector $p = \psi_\pi(s) = (q_{t_1}, \dots, q_{t_k})$ of the cells of its tiles, the other tiles being "don't care" (Figure 5.2). Thus, the placements are the k-permutations of the n cells, i.e., $N = n!/(n-k)!$ of them, and the *goal placement* is the placement of the goal. In this chapter π denotes a pattern, as in the literature of the pattern databases and in the comments of `PatternDatabase.h`, and not a path as in Table 1.2 of §1.4. In the library, `TilePattern` (`PatternDatabase.h:243`) takes a pattern as the list of its tiles, in a chosen order, and the board of the goal. Its constructor refuses a goal that is not a permutation of the cells (`PatternDatabase.h:289`), an empty pattern or one of more than $n-2$ tiles (`:292`), and a tile outside $1, \dots, n-1$ (the blank, in particular) or a tile given twice (`:300`).

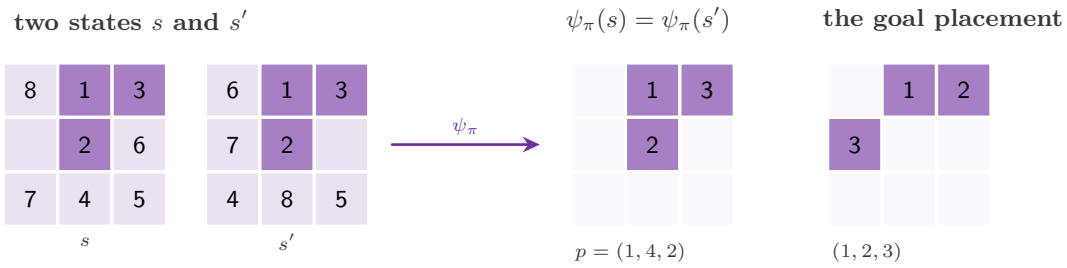


Figure 5.2. The abstraction of the pattern $\pi = \{1, 2, 3\}$ of the 8-puzzle: the states s and s', which differ in the other tiles and in the blank, have the same placement $p = (1, 4, 2)$ of the tiles of π (the cells numbered row by row from 0); the table of π holds the distance from each placement to the goal placement.

The *abstract space* of π is a graph on the placements whose edges move one tile of π to an adjacent cell, at cost 1. As Figure 5.3 shows, the library builds it under two rules of the blank (`BlankRule`, `PatternDatabase.h:204`).

- With the blank *ignored*, an abstract state is a placement, and each move of a tile of π to an adjacent cell free of the other tiles of π is allowed, as if the blank were wherever needed. The table holds the distance $d_\pi(p)$ from p to the goal placement in this graph, which is undirected and has unit costs.
- With the blank *minimized*, an abstract state is a placement together with the cell b of the blank, a move of the blank into a cell that no tile of π holds costs 0 (it moves a tile out of the pattern), and a move of a tile of π into the cell of the blank costs 1. The blank thus reaches at no cost each cell of its *region*, i.e., of its connected component among the cells free of π, and the table holds the least distance over the cells of the blank, $d_\pi(p) = \min_b d_\pi(p, b)$; this is the rule of the disjoint databases of [22].

In both cases a move of the puzzle that moves a tile of π is an edge of cost 1 of the abstract space. A move of any other tile changes no placement (with the blank ignored) or is an edge of cost 0 (with the blank minimized); hence ψ_π, with the cell of the blank added in the second case, is an abstraction in the sense of Definition 5.1. We now compare the two rules.

Lemma 5.3 (The minimized blank dominates). For each placement p, $d_\pi(p)$ with the blank minimized is at least $d_\pi(p)$ with the blank ignored.

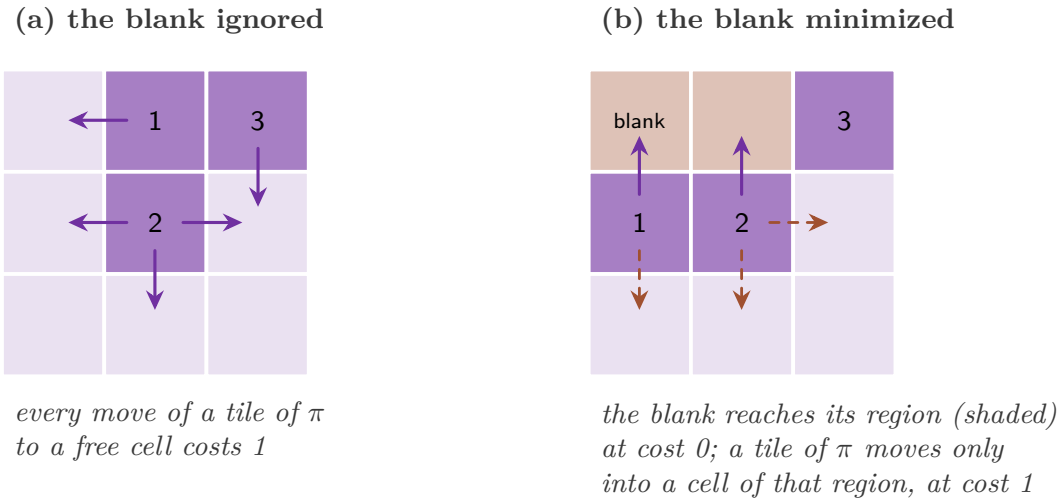


Figure 5.3. The moves of the abstraction of $\pi = \{1, 2, 3\}$. (a) With the blank ignored, a tile of π moves to any adjacent cell free of the other tiles of π. (b) With the blank minimized, the blank (top left) reaches its region (shaded) at no cost, and a tile of π moves only into a cell of that region; the moves in rust would need the blank in the other region, which it cannot reach without moving a tile of π.

Proof. A path of the minimized space from (p, b) to the goal, deprived of its moves of the blank, is a sequence of moves of the tiles of π, each into a cell free of the other tiles of π, i.e., a path of the space with the blank ignored, of the same cost. Hence the distance with the blank ignored is at most $d_\pi(p, b)$ for each b, and at most their minimum. □

Note that the test `pdb` checks the values of the tables, under both rules, against a breadth-first search of the abstraction written apart from the library, on each placement of patterns of 1 to 7 tiles of the 8-puzzle and with two goals; it does not compare the two rules with each other, and Lemma 5.3 rests on its proof.

5.4 Ranking the placements

A table is an array indexed by the placements, which need a *perfect ranking*, i.e., a bijection rank from the placements onto $[0, N)$, computed at each lookup. We rank a placement as a k-permutation in lexicographic order: the digit of the i-th tile is the number d_i of the cells below p_i that the tiles before it do not hold, and

$$\operatorname{rank}(p) = \sum_{i=1}^{k} d_i \prod_{j=i+1}^{k} (n - j + 1),$$

which is the number $d_1 d_2 \cdots d_k$ written in the mixed radix $(n, n-1, \ldots, n-k+1)$, the i-th digit d_i being in $[0, n-i+1)$ since p_i has at most $n-i+1$ free cells below it, itself excluded (Figure 5.4).

`TilePattern::rank` and `TilePattern::unrank` compute the map and its inverse by Algorithm 5.1, in which the set U of the cells already used, and the set F of the free ones, are words of $n \le 32$ bits. Hence the count of line 4 is one population count of U restricted to the cells below p_i, and the selection of line 13 clears the d_i lowest bits of F, one at a time, and takes the lowest one left.

Algorithm 5.1 (RANK and UNRANK, the lexicographic rank of a placement of k tiles on n cells).

```
1  function RANK(p)                ▷ p = (p_1, ..., p_k), distinct cells
2     r ← 0;  U ← ∅                  ▷ U: the cells of the tiles before
3     for i ← 1 to k do
4        d_i ← p_i − |{c ∈ U : c < p_i}|
5        r ← r · (n − i + 1) + d_i
6        U ← U ∪ {p_i}
7     return r
8  function UNRANK(r)              ▷ 0 ≤ r < N
```

```
 9     for i ← k down to 1 do
10        d_i ← r mod (n − i + 1);  r ← r div (n − i + 1)
11     F ← {0, 1, ..., n − 1}       ▷ the free cells
12     for i ← 1 to k do
13        p_i ← the (d_i + 1)-th least cell of F;  F ← F \ {p_i}
14     return p
```

Lemma 5.4 (The rank is a bijection). (i) The map rank is a bijection from the k-permutations of n cells onto $[0, n!/(n-k)!)$, RANK computes it in $O(k)$ operations on words of n bits, and UNRANK computes its inverse in $O(k + \sum_i d_i) = O(k\,n)$ of them. (ii) Two ranks r and r' with $\lfloor r/(n-k+1) \rfloor = \lfloor r'/(n-k+1) \rfloor$, i.e., in the same *block* of $n-k+1$ consecutive ranks, are the ranks of placements that differ only in the cell of the last tile.

Proof. (i) The cells below p_i are p_i, of which $|\{c \in U : c < p_i\}|$ are used, and hence line 4 computes the digit d_i; lines 3 to 6 evaluate the mixed radix sum by the rule of Horner. The mixed radix representation is a bijection from the vectors of digits with $d_i \in [0, n-i+1)$ onto $[0, \prod_i (n-i+1)) = [0, n!/(n-k)!)$, and lines 9 and 10 compute its inverse. The map from a placement to its digits is a bijection too: given the cells $p_1, \ldots, p_{i-1}$, the cell p_i is the (d_i+1)-th of the $n-i+1$ cells that they leave free, in increasing order, which is line 13, so that p is rebuilt from its digits one cell after the other. Each iteration of lines 3 to 6 takes $O(1)$ operations, and that of lines 12 and 13 takes $O(1+d_i)$, one per bit cleared. (ii) The digit of the last tile is $d_k = r \bmod (n-k+1)$, and the other digits are those of $\lfloor r/(n-k+1) \rfloor$; two ranks of the same block thus have the same digits $d_1, \ldots, d_{k-1}$, hence by (i) the same cells of the first $k-1$ tiles. □

Thus, the entries of one block are the placements that differ in the cell of the last tile, which the max-compression of §5.9 exploits. The linear-time ranking of Myrvold and Ruskey [34] would do as well, and it does without the population counts (it serves the permutation puzzles, §5.14). The lexicographic one keeps close placements in close entries, as (ii) shows, and its cost of k population counts is small against the cache miss that the lookup of a large table pays anyway (a judgement on the hardware, not a theorem).

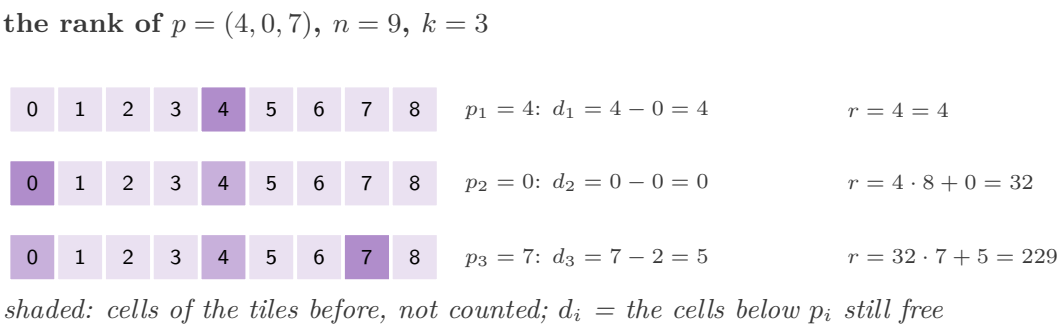


Figure 5.4. The rank of the placement $p = (4, 0, 7)$ of three tiles on 9 cells: each row is the board before the tile i is placed, with the cells of the tiles before it shaded; the digit d_i is the number of the free cells below p_i, and the rank is accumulated in the mixed radix $(9, 8, 7)$, giving $(4 \cdot 8 + 0) \cdot 7 + 5 = 229$.

5.5 Building a table

Each table is filled by a breadth-first search of the abstract space, backward from the goal placement and by layers: at the layer d each entry of value d is expanded, and its neighbours not seen yet get the value $d+1$; the search ends when a layer adds no entry. Since the abstract space with the blank ignored is undirected, the backward distances are the forward ones. We give the pseudocode of the Doxygen comment of `PatternDatabase.h`.

Algorithm 5.2 (BUILD-IGNORED, the table of a pattern with the blank ignored).

```
1  procedure BUILD-IGNORED(π)              ▷ the blank ignored
2     for each r ∈ [0, N) do h[r] ← ∞          ▷ N = n! / (n − k)!
3     h[rank(goal(π))] ← 0;  d ← 0
4     loop
5        new ← 0
```

```
 6        for each r with h[r] = d do          ▷ by blocks, in parallel
 7           p ← unrank(r)
 8           for each tile i of π, each cell c next to p_i, c ∉ p do
 9              r' ← rank(p with p_i ← c)
10              if h[r'] = ∞ then              ▷ by compare-and-swap
11                 h[r'] ← d + 1;  new ← new + 1
12        if new = 0 then return h
13        d ← d + 1
```

Figure 5.5 follows Algorithm 5.2 on the board of 3×2 cells with the pattern $\{1, 2\}$, whose 30 placements are laid out by rank as a 6×5 matrix: the layer 0 is the goal placement, the layer 1 its three neighbours, and the table is complete after the layer 5.

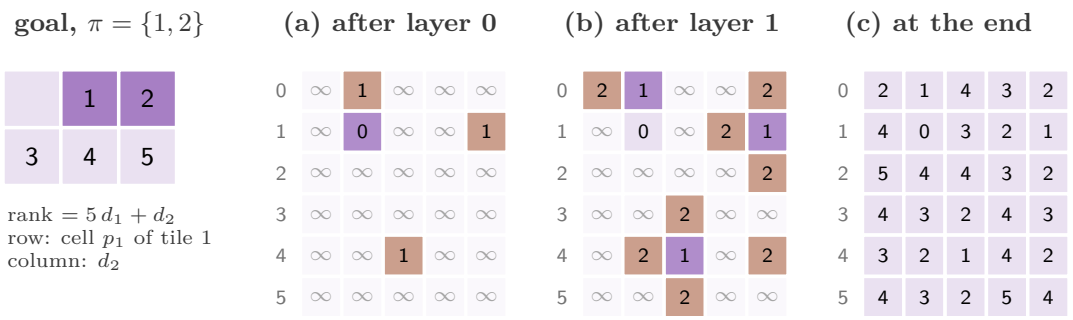


Figure 5.5. Algorithm 5.2 on the board of 3×2 cells with $\pi = \{1, 2\}$: the 30 entries, in the order of their ranks (the row is the cell of the tile 1, the column the digit of the tile 2), are ∞ until reached. (a) The layer 0 (violet) is expanded and its neighbours get 1 (rust); (b) the layer 1 is expanded and gives the layer 2; (c) the table at the end, whose largest distance is 5.

Each layer is scanned in blocks of 2^{14} consecutive ranks, which the workers of a runtime take from a shared counter, each block at a time. The entries take one byte each during the construction, with the value 255 standing for ∞, and an entry is claimed by an atomic compare-and-swap from 255 to $d+1$ (line 10): of the workers that try to claim the same entry in a layer, exactly one succeeds and counts it in `new`, and the others fail and count nothing. A layer d with $d+1 \geq 255$ throws `std::logic_error` (`PatternDatabase.h:1507`), and a placement still at 255 at the end makes `PatternTable::distances` throw too (`PatternDatabase.h:1137`). A pattern of $k = n-1$ tiles, for which the abstraction would be the puzzle itself and only half of the placements would be reachable (Exercise 5.3), is refused by the constructor of `TilePattern`; for $k \leq n-2$ we do not prove that each placement is reachable, and the throw is the check of it. Also, the test that throws is made at the start of each layer d with $d+1 \geq 255$, before the entries of that layer are scanned; hence a table whose largest distance is exactly 254 is refused too (no table of the library comes near that value).

With the blank minimized the states are the pairs (p, b), with b coded by its index among the $n-k$ cells free of the pattern. Each state takes 2 bits (unseen, current, next, old), as in the two-bit breadth-first search of Korf [26]. A state is expanded once per region, from the least cell of the region, and its successors are marked together with all the cells of the region of the blank after the move, which realizes the moves of cost 0 without a second queue.

Algorithm 5.3 (BUILD-MINIMIZED, the table of a pattern with the blank minimized).

```
 1  procedure BUILD-MINIMIZED(π)              ▷ the blank minimized
 2     S[r, b] ← unseen for each r and each cell b ∉ unrank(r)
 3     S[rank(goal(π)), b] ← current for each b ∉ goal(π)
 4     h[r] ← ∞ for each r;  h[rank(goal(π))] ← 0;  d ← 0
 5     loop
 6        for each (r, b) with S[r, b] = current do    ▷ in parallel
 7           p ← unrank(r);  R ← region(p, b)  ▷ the cells b reaches
 8           if b ≠ min R then continue      ▷ one cell per region
 9           for each c ∈ R, each tile i of π with p_i next to c do
10              p' ← p with p_i ← c;  r' ← rank(p')
11              for each b' ∈ region(p', old p_i) do
```

```
12                if S[r', b'] = unseen then S[r', b'] ← next
13             if h[r'] = ∞ then h[r'] ← d + 1
14       if no state is next then return h
15       S ← S with current → old and next → current;  d ← d + 1
```

Line 3 starts from the goal placement with the blank anywhere, since the backward search looks for the states from which the goal placement is reached, with the blank wherever it ends. Line 11 marks the whole region of the blank after the move, which contains the cell left by the tile, since the states of a region are at the same distance (the blank moves within it at no cost), and line 13 writes the table of the placements. Both marks are made by a compare-and-swap, as in Algorithm 5.2. The patterns of a partition are built one after the other, each with all the workers (`PDBOptions::threads`, all the cores if 0). We now prove that the two procedures compute the tables, whatever the interleaving of the workers, and give their cost.

Theorem 5.5 (The construction of the tables). Let D be the largest distance in the table. (i) Algorithm 5.2, with the layers scanned by any number of workers in any interleaving, gives each entry the distance d_π of its placement from the goal placement in the space with the blank ignored. (ii) Algorithm 5.3, in the same conditions, gives each entry $\min_b d_\pi(p, b)$, the least distance in the space with the blank minimized over the cells b of the blank. (iii) Algorithm 5.2 takes $O(N(D + k\,n))$ operations and N bytes; Algorithm 5.3 takes $O(N(n-k)(D + k\,n))$ operations, N bytes for the table and $N(n-k)/4$ bytes for the states.

Proof. (i) We show by induction on d that at the start of the layer d the entries of value at most d are the placements at distance at most d, with their distances, and the others are ∞; it holds for $d = 0$ by line 3. During the layer d, the workers read the entries equal to d and write only the value $d+1$ into entries equal to ∞. Hence the set of the entries equal to d does not change during the layer, whatever the interleaving; each neighbour of such an entry that was ∞ at the start of the layer is set to $d+1$ by exactly one compare-and-swap; and no other entry changes. The placements so set are those at distance $d+1$: a placement at distance $d+1$ has a neighbour at distance d (the space being undirected, the next placement of a shortest path to the goal), and a neighbour of a placement at distance d is at distance at most $d+1$, hence exactly $d+1$ if it was ∞. The search ends at the first layer that sets no entry, when every placement reachable from the goal has its distance. (ii) The moves of the space with the blank minimized are reversible at the same cost: a tile of π moved from a into the blank b moves back from b into the blank, now in a, and a move of the blank inside a region is undone by the opposite move. The space is thus undirected, the distance $d_\pi(p, b)$ is constant over the cells b of a region, and the states at distance $d+1$ are those that are not at distance at most d and are reached from a state at distance d by one move of cost 1 followed by moves of cost 0 (a shortest path from such a state to the goal begins with moves of cost 0 inside its region and then makes a move of cost 1, read backward). Lines 7 to 12 enumerate, from the least cell of each region of the layer d, every cell c of the region and every tile of π next to it, which are all the moves of cost 1 out of the states of the region, and line 11 marks the whole region of the new blank. By the induction of (i), with the 2 bits in place of the entries, the states marked *current* at the start of the layer d are those at distance d, the *old* ones those at a smaller distance, and the marks of the layer are those at distance $d+1$. Line 13 writes $d+1$ into $h[r']$ only if no state of the placement r' has been reached before, i.e., at the least d at which some state (r', b') is at distance d, which is $\min_b d_\pi(p, b)$. (iii) Each layer scans the N entries, or the $N(n-k)/32$ words of the states, and there are $D+2$ layers. With the blank ignored, each placement is expanded once, at the layer of its distance, by one unrank of $O(k\,n)$ operations and at most $4k$ moves with as many ranks of $O(k)$ operations, i.e., $O(k\,n + k^2) = O(k\,n)$ operations by Lemma 5.4. With the blank minimized, each state is current in one layer only and costs an unrank, of $O(k\,n)$ operations, and the computation of its region, of $O(n)$ operations, since the region grows by one step of adjacency per operation on its word; the least state of a region also makes, for each of its cells, the at most 4 tests of a neighbour, and for each of the at most $4k$ pairs of a cell of the region and a tile next to it a rank, a region and the marks of that region, i.e., $O(k + n)$ operations each. The memory is that of the arrays. □

The test `pdb` checks (i) and (ii) against a breadth-first search of the abstraction written apart from the library, on 1 and on 3 workers. The bounds of (iii) are linear in N for a fixed size of the pattern and a fixed largest distance D.

5.6 Additive pattern databases

Clearly, the table of one pattern is a rather weak heuristic, since it ignores the other tiles. Taking the maximum of the tables of several patterns is admissible (Lemma 2.5), but it is not much stronger than the best of them; the sum is much stronger, and it is admissible when the patterns are disjoint and each table counts only the

moves of its own tiles. Let $\pi_1, \ldots, \pi_m$ be disjoint patterns (a partition of the tiles, or of a subset of them; Figure 5.6), and let

$$h(s) = \sum_{j=1}^{m} d_{\pi_j}(\psi_{\pi_j}(s)).$$

Theorem 5.6 (Admissibility of the additive heuristic). If the patterns are disjoint and none holds the blank, h is admissible, with either rule of the blank, and $h = 0$ at the goal.

Proof. Consider a path P from s to the goal, and let $c_j(P)$ be the number of its moves that move a tile of π_j. The projection of P on π_j (with the cell of the blank, under the minimized rule) is a path of the abstract space from $\psi_{\pi_j}(s)$ to the goal placement whose cost is $c_j(P)$, since the moves of the other tiles cost nothing there. Hence $d_{\pi_j}(\psi_{\pi_j}(s)) \le c_j(P)$ (with the minimized rule, d_{π_j} is the minimum over the cells of the blank, and the projection starts from one of them). Each move of the puzzle moves one tile, which belongs to at most one pattern; thus $\sum_j c_j(P) \le c(P)$, and summing over j gives $h(s) \le c(P)$ for each path, i.e., $h(s) \le h^*(s)$. At the goal each placement is the goal placement, of distance 0. □

This is the argument of Korf and Felner [22], which Yang et al. [28] cast for any family of abstractions whose costs split the cost of each edge. This argument needs the patterns to be disjoint and the abstract spaces to charge only the moves of their tiles. For this reason the blank belongs to no pattern: a table that charged the moves of the blank would count each move twice, once for the blank and once for the tile. `AdditivePDB` (`PatternDatabase.h:1733`) refuses a tile in two patterns, and `TilePattern` refuses the blank. In fact, the Manhattan distance is the additive database of the $n-1$ patterns of one tile each, and a partition into larger patterns dominates it (Proposition 5.14). The partitions of Figure 5.6 are given by the functions `partition_15_7_8()`, `partition_15_5_5_5()`, `partition_15_6_6_3()` and `partition_24_6_6_6_6()` of `PatternDatabase.h`.

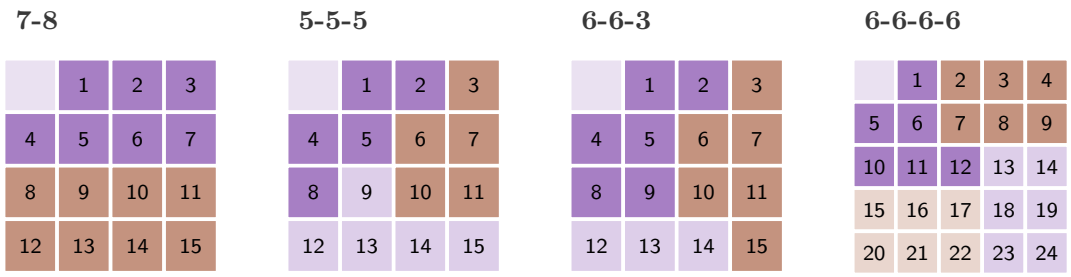


Figure 5.6. The partitions of the tiles of the library, each pattern in a colour: 7-8 of the 15-puzzle [22], 5-5-5 and 6-6-3 of the 15-puzzle [23], and 6-6-6-6 of the 24-puzzle [22]; the blank, in the first cell of the goal, belongs to no pattern.

The heuristic of a state s is computed from the array q of the cells of all the tiles, $q[t]$ being the cell of the tile t and $q[0]$ that of the blank. Algorithm 5.4 gives the lookup H of `AdditivePDB::heuristic` (the maximum with the reflection of §5.8, when it is used), the value VALUE of an entry for each encoding of §5.9, in which $m_j(p)$ is the Manhattan distance of the tiles of π_j read from a table of $k_j n$ bytes and e_j the stored excess, and the heuristic CHILD-H of a successor, which `SlidingTilePDB` computes for each successor that it generates; DELTA and the residues res_j belong to the relative encodings of §5.9.

Algorithm 5.4 (The lookups of an additive pattern database).

```
 1  function H(q)
 2     a ← SUM(q)
 3     if not reflected then return a
 4     for each tile t, the blank included, do q'[σ(t)] ← T(q[t])
 5     return max(a, SUM(q'))              ▷ q': the reflection of s
 6  function SUM(q)
 7     v ← 0
 8     for each pattern π_j do
 9        p ← (q[t] for the tiles t of π_j, in their order)
10        v ← v + VALUE_j(RANK_j(p), m_j(p))
11     return v
12  function VALUE_j(r, m)                 ▷ by the encoding of the table
```

```
13     byte: d_j[r];  nibble: m + 2 e_j[r];  min: m + 2 e_j[r div c]
14     sparse: m if B_j[r] = 0, else m + 2 x_j[rank₁(B_j, r)]
15   function CHILD-H(q, h, x, b)   ▷ the tile x moves into the blank cell b
16     if the encoding is relative then return h + DELTA(q, x, b)
17     a ← q[x];  q[x] ← b;  q[0] ← a
18     v ← H(q);  q[x] ← a;  q[0] ← b
19     return v
20   function DELTA(q, x, b)
21     if x belongs to no pattern then return 0
22     j ← the pattern of x
23     y ← res_j(q);  y' ← res_j(q with q[x] ← b)
24     return ((y' − y + 1) mod M) − 1
```

The symbols of Algorithm 5.4 that the later sections define are the following: T and σ of line 4 are the transposition of the cells and the renaming of the tiles of the reflection (§5.8); in line 13, d_j is a table of bytes, e_j a table of stored excesses and c the group of `MinNibbleEntries`; in line 14, B_j is the bit vector of `SparseEntries`, $\mathrm{rank}_1(B_j, r)$ the number of its ones before the position r and x_j the array of its nonzero excesses (§5.9); and M in line 24 is 3 or 4 (Lemma 5.10). Line 17 makes q the array of the successor (the tile x in the cell b, the blank in the cell that x leaves), and line 18 restores it, so that the same array serves all the successors of s. Thus, `SlidingTilePDB` (`PatternDatabase.h:1935`) is the domain `SlidingTile` whose heuristic is that of an `AdditivePDB`, and each search of the library runs on it unchanged; it calls H, or EXACT of §5.9 with a relative encoding, only for the initial state, and CHILD-H for each successor. Proposition 5.13 gives the cost of these functions.

The options of the construction, and the classes that a user declares, are the following; the first excerpt is from the header, the constructors are quoted by hand, since in the header they come with their bodies.

```
enum class BlankRule : std::uint8_t {
 ignored ,   ///< the blank is not in the abstract state: consistent tables
 minimized   ///< the blank is, and the table keeps the least distance
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// a partition of the tiles into disjoint patterns

using Partition = std::vector< std::vector< int > >;

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// the options of the construction of an AdditivePDB

struct PDBOptions {
 BlankRule blank = BlankRule::ignored;     ///< the rule of the blank
 bool reflect = true;          ///< also the reflection, where it is valid
 unsigned compress = 1;        ///< ranks per entry of MinNibbleEntries
 unsigned threads = 0;         ///< workers of the construction (0: all)
 std::string cache;            ///< directory of the tables ("": none)
 };
```

```
template< int W , int H , class Enc = NibbleEntries >
class AdditivePDB {
 public:
 AdditivePDB( const Partition & partition ,
              const board_type & goal = Pattern::korf_goal() ,
              const PDBOptions & o = {} );
 [[nodiscard]] int heuristic( const std::uint8_t * q ) const;
 [[nodiscard]] bool reflected( void ) const;
 };

template< int W , int H , class Enc = NibbleEntries >
class SlidingTilePDB : public SlidingTile< W , H > {
 public:
 SlidingTilePDB( const board_type & start ,
                 const AdditivePDB< W , H , Enc > & pdb );
```

```
};
```

The constructor of `AdditivePDB` builds the table of each pattern, or takes it from the directory `cache` (§5.10), with `threads` workers, and `reflect` asks for the reflection where it is valid; `compress` is the group c of `MinNibbleEntries`. A second constructor takes the tables already built, and `make()` builds them on another runtime of the workers (§1.6). `heuristic` requires an absolute encoding, `exact` (H with EXACT in place of VALUE) and `delta` (DELTA) a relative one. The `SlidingTilePDB` keeps a reference to the `AdditivePDB`, which must outlive it, and it refuses a relative encoding with the reflection (§5.9). The following test case builds the partition 4-4 of the 8-puzzle with two encodings and searches with both.

```
PDBOptions o;
o.blank = BlankRule::ignored;   // consistent tables (the default)
o.reflect = true;               // also the reflection about the diagonal
o.threads = 2;                  // workers of the construction
// a directory in o.cache: the tables built once, then read from there
const Partition part{ { 1 , 2 , 3 , 4 } , { 5 , 6 , 7 , 8 } };
const AdditivePDB< 3 , 3 > pdb( part , EightPuzzle::korf_goal() , o );
const auto board = EightPuzzle::parse( "7 2 4 5 0 6 8 3 1" );
const SlidingTilePDB< 3 , 3 > d( board , pdb );
const auto s = astar( d );      // s.cost and s.stats.expanded
// the residues modulo 3 need the tables without the reflection
o.reflect = false;
const AdditivePDB< 3 , 3 , Mod3Entries > pdb3( part ,
                                               EightPuzzle::korf_goal() ,
                                               o );
const SlidingTilePDB< 3 , 3 , Mod3Entries > d3( board , pdb3 );
const auto s3 = astar( d3 );
```

5.7 Consistency

With the blank ignored the additive heuristic is consistent, and the proof gives a finer property of the tables, on which the encodings of §5.9 rest. Let $m_\pi(p)$ be the Manhattan distance of the tiles of π in the placement p.

Lemma 5.7 (Parity and excess). With either rule of the blank, $d_\pi(p) \geq m_\pi(p)$ and $d_\pi(p) \equiv m_\pi(p) \pmod 2$ for each placement p. With the blank ignored, two adjacent placements p and p' have $|d_\pi(p) - d_\pi(p')| = 1$.

Proof. A move of cost 1 moves one tile of π by one cell, and hence changes m_π by exactly one unit, while a move of cost 0 (of the blank, with the minimized rule) does not change it. Along a path of least cost from p to the goal placement, where $m_\pi = 0$, the value of m_π thus goes to 0 by $d_\pi(p)$ steps of ± 1, which gives both claims of the first sentence. With the blank ignored, the abstract space is undirected with unit costs, so that $|d_\pi(p) - d_\pi(p')| \leq 1$ for adjacent placements, and the parity of d_π, which is that of m_π, differs between them, since $m_\pi(p)$ and $m_\pi(p')$ differ by one; hence the difference is not zero. □

Theorem 5.8 (Consistency with the blank ignored). With the blank ignored, the additive heuristic of disjoint patterns is consistent.

Proof. A move of the puzzle moves one tile; if it belongs to no pattern, no placement changes and h does not change; otherwise only the placement of its pattern changes, to an adjacent one, and by Lemma 5.7 h changes by one. In either case $h(s) \leq 1 + h(s')$ for each move (s, s'), and $h = 0$ at the goal by Theorem 5.6. □

With the blank minimized, the same move may change the value of a table by more than one. In fact, the minimum over the cells of the blank forgets in which region the blank actually is, and the region that the true blank occupies after the move may be far from the one that realized the minimum before it. Hence, the sum is admissible but usually not consistent. On the 8-puzzle with the goal of Korf and the partition 4-4, the two ends of 18840 of its 241920 moves differ by more than one without the reflection, and of 16986 with it, as a computation of the tables written apart from the library gives (the test `pdb` prints the same counts as ordered pairs of a state and a successor, i.e., twice these numbers), and A* stays optimal by reopening the closed nodes (§2.7 and [31]). In exchange, the heuristic is larger: by Lemma 5.3 each table with the blank minimized is at least the one with the blank ignored, which in turn is at least the Manhattan distance of its tiles (Lemma 5.7); `PDBBench --mode table` reports the mean of each heuristic on 10^6 random states (column `mean_h`), against that of the Manhattan distance (column `mean_manhattan`). By default, `PDBOptions` has the blank ignored, which all

the encodings accept and which keeps the heuristic consistent; the sequential A* then never expands a node twice. One chooses the blank minimized explicitly, as in the example of §5.6.

5.8 The reflection about the main diagonal

Let the board be square, and let the goal have the blank on the main diagonal, as the goal of Korf (the blank in the first cell) does. The transposition T of the cells, $T(\text{row } i, \text{column } j) = (\text{row } j, \text{column } i)$, maps adjacent cells to adjacent cells. Let $\sigma(t)$ be the tile that the goal holds in the cell $T(c_t^*)$, with c_t^* the goal cell of t, and let $\phi(s)$ be the state that holds the tile $\sigma(t)$ in the cell $T(q_t)$, for each tile t of s (the blank included, Figure 5.7).

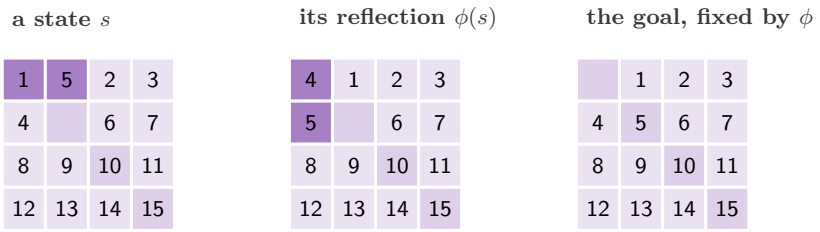
Lemma 5.9 (The reflection is an automorphism). The map ϕ is a bijection of the states, maps the moves of the puzzle to moves, and fixes the goal; hence $h^*(\phi(s)) = h^*(s)$ for each state s.

Proof. Since T and σ are bijections, so is ϕ. The blank is fixed by σ, since $T(c_0^*) = c_0^*$ when the goal cell of the blank is on the diagonal. A move of s exchanges the blank, in the cell b, with the tile x in an adjacent cell a; in $\phi(s)$ the blank is in $T(b)$ and $\sigma(x)$ in $T(a)$, which is adjacent to $T(b)$, and exchanging them gives ϕ of the successor. The goal holds t in c_t^*, and ϕ puts in $T(c_t^*)$ the tile $\sigma(t)$, which is by definition the tile of the goal in $T(c_t^*)$, i.e., it gives the goal back. Hence ϕ maps the paths from s to the goal onto the paths from $\phi(s)$ to the goal, of the same costs. □

Thus, the sum of the same tables on $\phi(s)$ is another admissible heuristic of s, since $h(\phi(s)) \le h^*(\phi(s)) = h^*(s)$, and it is consistent if h is, since ϕ maps moves to moves; the library takes

$$h_\phi(s) = \max\{h(s), h(\phi(s))\},$$

which is the reflection of [22], admissible and, with the blank ignored, consistent by Lemma 2.5. The reflection costs a second lookup per table; it is used only when it is valid, i.e., on a square board whose goal has the blank on the diagonal, and when `PDBOptions::reflect` asks for it (`AdditivePDB::reflected()` tells whether it is used). The constructor of `AdditivePDB` computes the arrays T and σ once, in $O(n)$ time, and line 4 of Algorithm 5.4 builds the cells of $\phi(s)$ from them: the tile $\sigma(t)$ of $\phi(s)$ is in the cell $T(q_t)$.



the tile t in the cell c goes to the transposed cell $T(c)$ and is renamed $\sigma(t)$, the tile of the goal in $T(c_t^*)$: here $1 \mapsto 4$ and $5 \mapsto 5$; the shaded cells are the main diagonal, which holds the blank of the goal

Figure 5.7. The reflection about the main diagonal (the shaded cells) on the 15-puzzle with the goal of Korf (right): the state s (left), two moves from the goal, has the tiles 1 and 5 out of place; its reflection $\phi(s)$ (middle) holds $\sigma(1) = 4$ in the cell transposed of that of 1 and $\sigma(5) = 5$ in the cell transposed of that of 5, and it is also two moves from the goal.

The *dual lookup* of Zahavi et al. [29], which looks up the placement of the inverse permutation of the state, is another admissible lookup of the same tables; it is not consistent, and it pays off with the bidirectional pathmax of IDA*, which the library does not have.

5.9 The encodings of the entries

A table of the partition 7-8 of the 15-puzzle has more than half a billion entries (§5.11), and the memory of the entries decides both whether the tables fit and how often a lookup misses the cache. Let $e_\pi(p) = (d_\pi(p) - m_\pi(p))/2$ be the *excess* of the distance over the Manhattan distance of the tiles of the pattern; by Lemma 5.7 it is a nonnegative integer, typically much smaller than the distance (cf. Figure 5.8). At the lookup, the Manhattan distance of the tiles is computed from a table of kn bytes, and the entries store the excess, or less. We list the encodings of the library below (Figure 5.8).

- `ByteEntries` (`PatternDatabase.h:526`) stores d_π in 8 bits per entry.

- `NibbleEntries` (`PatternDatabase.h:591`, the default) stores e_π in 4 bits per entry, saturated at 15; the value is $m_\pi + 2e_\pi$. A saturated entry gives a smaller value, which is still admissible, and `clamped()` counts them (the test `pdb` checks that the tables of the 8-puzzle have none).
- `Mod3Entries` (`PatternDatabase.h:679`) stores d_π mod 3, five residues in a byte as the digits of a number in base 3 ($3^5 = 243 \leq 256$), i.e., $8/5 = 1.6$ bits per entry, after Breyer and Korf [25].
- `ParityBitEntries` (`PatternDatabase.h:758`) stores $\lfloor d_\pi/2 \rfloor$ mod 2, one bit per entry: with the parity of d_π, which is that of m_π, it gives d_π mod 4, as in the 1-bit encoding of Clausecker and Reinefeld [27].
- `MinNibbleEntries` (`PatternDatabase.h:838`) stores, for each group of c consecutive ranks (c a power of 2, `PDBOptions::compress`), the least excess of the group in 4 bits, i.e., $4/c$ bits per entry: the lossy max-compression of Felner et al. [24], which here keeps the Manhattan distance of each placement exact.
- `SparseEntries` (`PatternDatabase.h:940`) stores a bit per entry that records whether its excess is nonzero, with a directory of rank and select of about 0.4 bits per entry, and the nonzero excesses in 4 bits each, in the order of their ranks; it is smaller than `NibbleEntries` when fewer than about 65% of the excesses are nonzero, at the price of a rank per lookup.

rank	0	1	2	3	4	5	6	7	8	9	10	11
d	2	1	4	3	2	4	0	3	2	1	5	4
m	2	1	4	3	2	2	0	3	2	1	3	2
$e = (d - m)/2$	0	0	0	0	0	1	0	0	0	0	1	1
byte: d	2	1	4	3	2	4	0	3	2	1	5	4
nibble: e	0	0	0	0	0	1	0	0	0	0	1	1
mod 3: $d \bmod 3$	2	1	1	0	2	1	0	0	2	1	2	1
bit: $\lfloor d/2 \rfloor \bmod 2$	1	0	0	1	1	0	0	1	1	0	0	0
min, $c = 2$: $\min e$	0		0		0		0		0		1	
value: $m + 2 \min e$	2	1	4	3	2	2	0	3	2	1	5	4

Figure 5.8. The encodings of the entries of ranks 0 to 11 of the table of Figure 5.5: the distance d, the Manhattan distance m of the tiles of the pattern and the excess e; the entries stored in a byte, in a nibble, modulo 3 and in one bit; the least excess of each group of $c = 2$ ranks and the value it gives, in rust where it is smaller than d (the rank 5, of distance 4, gets 2).

The values of `ByteEntries`, of `NibbleEntries` when no entry saturates, and of `SparseEntries` (which saturates in the same way) when no entry saturates, are the distances themselves, so that they are admissible and consistent as in §5.6 and §5.7. Each of the other three needs an argument.

The relative encodings

In `Mod3Entries` and `ParityBitEntries`, the residues do not give the value of a placement, but they give the change of the value along a move.

Lemma 5.10 (The change from the residues). With the blank ignored, let p and p' be adjacent placements and r, r' the residues of their distances modulo $M \in \{3, 4\}$. Then

$$d_\pi(p') - d_\pi(p) = \big((r' - r + 1) \bmod M\big) - 1.$$

Proof. By Lemma 5.7 the difference δ is $+1$ or -1. If $\delta = 1$, then $r' - r \equiv 1$ and the right-hand side is $2 - 1 = 1$; if $\delta = -1$, then $r' - r \equiv -1$, $(r' - r + 1) \bmod M = 0$, and the right-hand side is -1. Both cases are told apart since $1 \not\equiv -1 \pmod M$ for $M \geq 3$. □

A move of the puzzle moves the tile of one pattern only, so that the heuristic of a successor is that of its parent plus the change of one table. This is DELTA in Algorithm 5.4 (`AdditivePDB::delta()`), whose line 24 is the

formula of Lemma 5.10, and CHILD-H calls it for each successor; A* thus never needs the absolute value but for the initial state. That one is found from the residues only, by a descent to the goal (Figure 5.9).

Theorem 5.11 (The value from the residues). With the blank ignored, starting from a placement p and moving each time to a neighbour whose residue is that of the current placement minus one modulo M, one reaches the goal placement in exactly $d_\pi(p)$ steps; such a neighbour exists at each step before the goal.

Proof. Let q be the current placement, not the goal. The abstract space is undirected with unit costs, and the next placement on a path of least cost from q to the goal is a neighbour of distance $d_\pi(q) - 1$, whose residue is $r - 1 \bmod M$. Each neighbour has distance $d_\pi(q) \pm 1$ (Lemma 5.7), and those of distance $d_\pi(q) + 1$ have residue $r + 1 \bmod M \neq r - 1 \bmod M$; hence each neighbour with the residue sought is one step closer, and the descent reaches the goal placement, the only one of distance 0, after exactly $d_\pi(p)$ steps. □

The descent is Algorithm 5.5 (`PatternTable::exact()`), which throws `std::logic_error` if it finds no neighbour with the residue sought, a case that Theorem 5.11 excludes for the tables that the library builds.

Algorithm 5.5 (EXACT, the value of a placement from the residues alone; $M = 3$ for `Mod3Entries`, 4 for `ParityBitEntries`).

```
1  function EXACT_j(p)
2     v ← 0
3     while p ≠ goal(π_j) do
4        y ← (res_j(p) − 1) mod M
5        p ← a neighbour p' of p, i.e., p with one tile moved to an
6            adjacent cell free of the others, with res_j(p') = y
7        v ← v + 1
8     return v
```

Each step tries at most $4k$ neighbours, i.e., $4k$ ranks and lookups; the value of the initial state thus costs $O(k\, d_\pi(p))$ lookups per table (Proposition 5.13).

We remark that the argument needs adjacent placements to differ by exactly one, i.e., the blank ignored (the constructor of `PatternTable` refuses a relative encoding with the blank minimized, `PatternDatabase.h:1110`), and the heuristic to be the sum of the tables, since from the maximum of two sums the change of one table cannot be told: this is why `SlidingTilePDB` refuses a relative encoding with the reflection (`PatternDatabase.h:1958`).

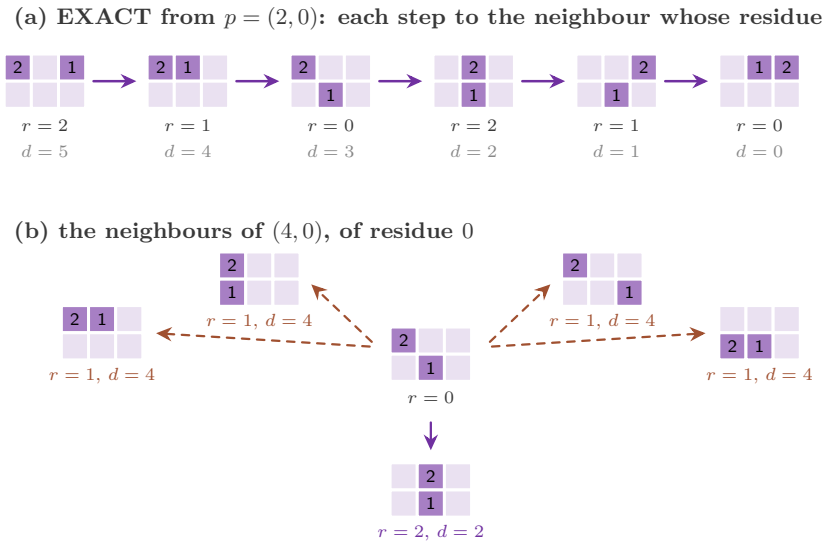


Figure 5.9. The value of a placement from its residues modulo 3, on the table of Figure 5.5. (a) From $p = (2, 0)$, each step goes to a neighbour whose residue is one less, and the goal placement is reached in 5 steps, i.e., $d = 5$. (b) The neighbours of $(4, 0)$, of residue 0 and distance 3: four of them have residue 1, i.e., distance 4, and one has residue 2, i.e., distance 2, which is the one taken.

The max-compression

In `MinNibbleEntries`, the ranks are grouped by c consecutive ones, $G(p)$ being the group of the rank of p, and the value of p is $m_\pi(p) + 2\min_{p' \in G(p)} e_\pi(p')$, the least excess being saturated at 15.

Proposition 5.12 (The max-compression). For each placement p, $m_\pi(p) \leq m_\pi(p) + 2\min_{p' \in G(p)} e_\pi(p') \leq d_\pi(p)$; hence the sum of the max-compressed tables of disjoint patterns is admissible and at least the Manhattan distance of the tiles of the patterns. It is not consistent in general.

Proof. The excesses are nonnegative (Lemma 5.7), which gives the first inequality, and the least excess of the group is at most $e_\pi(p)$, while the saturation can only lower it, which gives the second, since $m_\pi(p) + 2e_\pi(p) = d_\pi(p)$. The sum is then at most the sum of the distances, which is admissible by Theorem 5.6. For the last claim, on the table of Figure 5.5 with $c = 2$ the adjacent placements $(1, 0)$ and $(2, 0)$, of ranks 5 and 10, have the values 2 and 5 (Exercise 5.10), and a heuristic consistent along a move of cost 1 changes by at most 1. □

A* may thus reopen nodes on these tables. Felner et al. [24] store the least distance of the group instead of the least excess. Neither value dominates the other in general (the placement of least excess may have a larger Manhattan distance than p, or a smaller one), but only the second is always at least the Manhattan distance of the pattern, by Proposition 5.12; that it loses less on the tables of the sliding-tile puzzles is a motivation, not a theorem. The groups are aligned to c and the blocks of Lemma 5.4 (ii) to $n - k + 1$, so that a group with $c \leq n - k + 1$ meets at most two blocks and holds placements that differ mostly in the cell of the last tile.

The cost of the lookups and the size of the patterns

Proposition 5.13 (The cost of a lookup). Let the partition have m patterns of $k_1, \ldots, k_m$ tiles, $\sum_j k_j \leq n - 1$. With an absolute encoding, H of Algorithm 5.4 takes $O(n)$ operations and m accesses to the tables, twice as many with the reflection, plus one rank of the directory per table with `SparseEntries`; CHILD-H takes the same. With a relative encoding, CHILD-H takes $O(k_j)$ operations and 2 accesses to the table of the tile moved, and EXACT of a placement p of π_j takes $O(k_j^2\, d_{\pi_j}(p))$ operations and at most $4k_j\, d_{\pi_j}(p) + 1$ accesses.

Proof. SUM gathers k_j cells and computes a rank and a Manhattan distance in $O(k_j)$ operations per pattern (Lemma 5.4), with one access to the entries, and line 4 of Algorithm 5.4 takes $O(n)$; VALUE takes $O(1)$ operations but for the rank of `SparseEntries`. CHILD-H changes two cells of q and calls H, or DELTA, which computes two residues of one table. By Theorem 5.11, EXACT makes $d_{\pi_j}(p)$ steps, each of which tries at most $4k_j$ moves, each with a rank of $O(k_j)$ operations and one access. □

Proposition 5.14 (Larger patterns). (i) The additive heuristic of the $n - 1$ patterns of one tile, with either rule of the blank, is the Manhattan distance. (ii) If π and π' are disjoint, then $d_{\pi\cup\pi'}(p, p') \geq d_\pi(p) + d_{\pi'}(p')$ for each placement (p, p') of $\pi \cup \pi'$, with either rule; hence merging two patterns of a partition never decreases the additive heuristic.

Proof. (i) With the blank ignored the abstract space of one tile is the board, in which the distance between two cells is their Manhattan distance. With the blank minimized, the blank reaches every cell but that of the tile at no cost, since the board less one cell is connected on the boards of at least 2×2 cells; the tile thus moves to any adjacent cell at cost 1, and the distance is again the Manhattan distance. (ii) A path of the space of $\pi \cup \pi'$ moves one tile per move of cost 1, each time into a cell free of the tiles of $\pi \cup \pi'$ (and with the blank minimized, into the region of the blank); its moves of the tiles of π are then moves of the space of π, into a cell free of the tiles of π (and of the region of the blank of that space, which contains the region in the larger space), and its other moves change no placement of π, or are moves of the blank of cost 0 in the space of π. Its cost is thus at least the sum of the costs of its projections on π and on π', which are paths to the goal placements, and at least $d_\pi(p) + d_{\pi'}(p')$. □

By (i), (ii) and Theorem 2.15, a partition into patterns larger than single tiles dominates the Manhattan distance, which is the reason for the patterns; between consistent heuristics, Theorem 2.15 bounds the expansions when the domination is strict on the states that are not goals, and a mere domination bounds them only below the last layer of f.

5.10 Files and the cache of the tables

Building the largest tables (the 24-puzzle, the partition 7-8 of the 15-puzzle) is far more expensive than a search on them; hence the tables are built once and saved. `PatternTable::save()` (`PatternDatabase.h:1288`) writes a table, in little-endian order, with a header (the magic `PBeFSPDB`, the version of the format, the board, the rule of the blank, the code of the encoding and its parameter, the tiles and the goal), the number of the entries,

the length and the bytes of the payload, and a checksum of everything before it. The file is written under a temporary name in the same directory and then renamed; since a rename within a directory replaces the name at once on the file systems of POSIX, a reader never sees a file half written. Conversely, `load()` rejects a file that is truncated, of another version, whose checksum differs, or that holds another table (another pattern, goal, rule or encoding), by throwing `std::runtime_error`. With a cache directory (`PDBOptions::cache`), `PatternTable::cached()` (`PatternDatabase.h:1379`) runs Algorithm 5.6, in which the *file* of a table is the name that `file_name()` derives from the board, the rule, the encoding, its parameter, the tiles and a checksum of the goal.

Algorithm 5.6 (CACHED, a table from a cache directory).

```
1  function CACHED(dir, π, rule, Enc, c)
2     if the file of (π, rule, Enc, c) is in dir and loads then
3        return it
4     if the file of (π, rule, byte, 1) is in dir and loads then
5        d ← its distances
6     else d ← BUILD(π, rule);  save d as (π, rule, byte, 1)
7     t ← the encoding Enc of d with the parameter c
8     if (Enc, c) ≠ (byte, 1) then save t as (π, rule, Enc, c)
9     return t
```

Here BUILD is Algorithm 5.2 or 5.3. A table returned by CACHED is the encoding of the distances of Theorem 5.5, up to the corruptions of a file that its checksum does not detect: a checksum of 64 bits detects any change of a file with a probability close to 1 but not with certainty, and this is the only claim of this section that we do not prove. The test `pdb` checks the round trip, the rejection of a truncated, a corrupted and a foreign file, and the reuse of the cache.

5.11 Sizes

The table of a pattern of k tiles has $N = n!/(n-k)!$ entries. The partitions of the library (7-8 and 6-6-6-6 of [22], 6-6-3 and 5-5-5 of [23], on the 15-puzzle or the 24-puzzle as the number in parentheses indicates) give the following sizes, computed from N and from the bits per entry of each encoding. In the last column we give the max-compression by $c = 8$ for 7-8 and by $c = 16$ for 5-5-5, the groups of the benchmark of §5.17. We remark that the entries are summed over the patterns, e.g., $57657600 + 518918400$ for the patterns of 7 and 8 tiles of the partition 7-8.

partition	entries	8 bits	4 bits	mod 3	1 bit	min
7-8 (15)	576576000	576 MB	288 MB	115 MB	72 MB	36 MB
6-6-3 (15)	11534880	11.5 MB	5.8 MB	2.3 MB	1.4 MB	
5-5-5 (15)	1572480	1.6 MB	786 KB	314 KB	197 KB	49 KB
6-6-6-6 (24)	510048000	510 MB	255 MB	102 MB	64 MB	

A table with the blank minimized has the same size (cf. §5.5), but its construction needs $N(n-k)/4$ more bytes for the states of Algorithm 5.3, i.e., about 1 GB for the pattern of 8 tiles of the 15-puzzle.

5.12 When the heuristic does not help

Recall that the time of a search is the number of its expansions times the time of an expansion (§2.10). A search that is too slow is thus too slow for one of these two reasons, and the counters of the `SearchStats` usually indicate which one, and what one may try.

- *Reexpansions.* If `reexpanded` is positive, the heuristic is not consistent (Theorem 2.13); a heuristic meant to be consistent then has an error, and an inconsistent one (e.g., the blank minimized) may cost more in reexpansions than it saves in expansions. On a domain whose states can all be enumerated, the tests of the library check the admissibility and the consistency of a heuristic on each state and each move (the 8-puzzle has $9!/2 = 181440$ states, `check_heuristic8` in `test/test.cpp`). The same check on a small instance of a new domain is the quickest diagnosis.
- *Expansions.* The number of expansions is fixed by the heuristic up to the ties (Theorem 2.14); a stronger admissible heuristic (larger patterns, the reflection, the blank minimized) reduces it, and the mean heuristic

over a sample of states, which `PDBBench --mode table` reports, is a first estimate of the gain. If even a heuristic with a large weight does not solve the instance (a weighted A*, which the library obtains by a wrapper of the domain, §2.2), the instance is probably very hard for any optimal search. If an instance seems never to be solved, one should check that it is solvable at all: `SlidingTile` refuses a start whose permutation has the wrong parity (`SlidingTile.h:242`), while on a grid A* proves the absence of a path only after exhausting the component of the start.

- *Time per expansion.* The rate of expansions (`expanded / seconds`) falls when the heuristic is costly: a lookup of a table much larger than the cache misses on most expansions, and a smaller encoding (§5.9) may then be faster even with more expansions. The data structures of Chapter 3 are the other cost.
- *Memory.* If the search stops with `Status::node_limit`, or the memory runs out, `stored` measures the nodes held. Furthermore, the searches of Chapter 4 trade time for memory, and IDA* with a pattern database is the method of [22] for the 24-puzzle (the test `tiles24` runs it with patterns of 3 tiles).

5.13 The pattern databases of any abstraction

The tables of the previous sections rest on the structure of the sliding-tile puzzles: the rank of a placement, the moves of a tile and the Manhattan distance of a pattern are those of the board. The header `AbstractionPDB.h` gives instead the construction for any abstraction whose states are ranked, i.e., indexed by the integers of $[0, N)$, and whose edges have small integer costs; the permutation puzzles of §5.15 and the Towers of Hanoi of §5.16 are built on it. Such an abstraction is described by the concept `AbstractSpace`, in which `size()` is N, `for_each_goal( f )` calls `f( r )` for each abstract goal r, and `for_each_predecessor( r , f )` calls `f( r' , c )` for each edge (r', r) of cost $c \in \{1, \ldots, 254\}$; both are `const` and may be called by many threads at once.

```
template< class S >
concept AbstractSpace = requires( const S & s , std::uint64_t r ) {
 { s.size() } -> std::convertible_to< std::uint64_t >;
 s.for_each_goal( detail::rank_sink{} );
 s.for_each_predecessor( r , detail::rank_cost_sink{} );
 };
```

The edges are meant to be those of an abstraction of the moves of a domain in the sense of Definition 5.1, so that the distance of the image of a state is at most the cost of each of its paths to a goal, and its table is a consistent heuristic by Proposition 5.2; the concept cannot check this, which is the responsibility of whoever writes the space. The table holds, for each abstract state r, the least cost $\delta(r)$ of a path from r to an abstract goal, which `abstract_distances( space , threads )` computes on `threads` workers of an executor (all the cores if 0) by Algorithm 5.7, the pseudocode of the Doxygen comment of the header. It is the algorithm of Dial [65], whose buckets are here not lists but the layers of the table itself: the entries of value d are final when the layer d is scanned, since every edge costs at least 1, and each of them lowers the value of its predecessors to $d + c$. The table is cut into blocks of 2^{14} entries, each with the least value $\mathrm{lo}[B]$ not yet scanned, so that a layer skips the blocks that hold none of its entries, which are most of them in the first and in the last layers.

Algorithm 5.7 (DISTANCES, the table of an abstract space by layers).

```
 1  procedure DISTANCES(S)
 2     h[r] ← ∞ for each r ∈ [0, N);  h[g] ← 0 for each goal g
 3     lo[B] ← least h of the block B;  next[B] ← ∞
 4     d ← 0
 5     while d < ∞ do
 6        for each block B with lo[B] ≤ d do          ▷ in parallel
 7           m ← ∞
 8           for each r ∈ B do
 9              if h[r] = d then
10                 for each predecessor r' of r, by an edge of cost c, do
11                    if d + c < h[r'] then          ▷ by compare-and-swap
12                       h[r'] ← d + c
                         next[B(r')] ← min(next[B(r')], d + c)
13              else if d < h[r] < ∞ then m ← min(m, h[r])
14           lo[B] ← m
15        lo[B] ← min(lo[B], next[B]);  next[B] ← ∞ for each block B
16        d ← min_B lo[B]
```

The entries take one byte each, the value 255 standing for ∞; the blocks are handed to the workers by a shared counter, one at a time, and lines 11 and 12 are a loop of compare-and-swap on the entry, which ends as soon as the entry holds at most $d + c$, followed, when the entry was lowered, by a loop of compare-and-swap that lowers next of its block. The writes of a layer go to next and not to lo, which the workers read during the layer: a block scanned at line 14 forgets the entries lowered after it read them, and line 15, which runs after the workers of the layer have joined, brings them back. With unit costs Algorithm 5.7 is a breadth-first search by layers, as Algorithm 5.2; with integer costs it is a uniform-cost search, backward from the goals. Two checks are made at the end of each layer: if some relaxation of the layer had $d + c \geq 255$ (whether or not it would have lowered the entry), `abstract_distances` throws `std::logic_error` (`AbstractionPDB.h:218`), and it throws as well if the least value of line 16 is not above d, which only an edge of cost 0, excluded by the concept, can cause (`AbstractionPDB.h:226`).

Theorem 5.15 (The construction by layers). Let the edges of an `AbstractSpace` have the costs of $\{1, \ldots, 254\}$, let $\delta(r)$ be the least cost of a path from r to a goal (∞ if there is none), and let $D_0 < D_1 < \cdots < D_L$ be the distinct finite values of δ. (i) If no edge (r', r) of cost c has $\delta(r) + c \geq 255$ with $\delta(r)$ finite, Algorithm 5.7, on any number of workers and in any interleaving, runs its loop with $d = D_0, D_1, \ldots, D_L$ in this order, scans at the layer D_k exactly the blocks that hold an entry at distance D_k, and returns $h[r] = \delta(r)$ for each r with $\delta(r)$ finite and $h[r] = 255$ for the others. (ii) Otherwise it throws `std::logic_error`, at the end of the first layer D_k for which such an edge enters an entry at distance D_k. (iii) Let β_k be the number of the blocks that hold an entry at distance D_k, $n_b = \lceil N/2^{14} \rceil$ the number of the blocks, and N_f the number of the entries of finite distance. The construction makes N_f calls of `for_each_predecessor`, reads at most $2^{14} \sum_k \beta_k \leq 2^{14} \min\{(L+1)\, n_b, N_f\}$ entries, and takes $O((L+1)\, n_b)$ operations more for lines 6 and 15; it takes $N + 2n_b$ bytes.

Proof. Every value written at line 12 is $d + c$, with $d = h[r]$ the cost of a path from r to a goal (by induction on the writes) and (r', r) an edge of cost c; hence $h[r'] \geq \delta(r')$ holds at all times, and an entry of infinite distance is never written. We show by induction on k that the k-th iteration of the loop starts with $d = D_k$ and with (a) $h[r] = \delta(r)$ for each r with $\delta(r) < D_k$, which was scanned at the layer $\delta(r)$; (b) $h[r] = D_k$ if and only if $\delta(r) = D_k$; and (c) lo$[B]$ equal to the least value in $(D_{k-1}, 255)$ of the entries of B (in $[0, 255)$ for $k = 0$), 255 if there is none. For $k = 0$, the goals are the entries of value $0 = D_0$, the others are 255, and line 3 gives (c). Assume (a) to (c) at the start of the layer $d = D_k$. The workers write only values $d + c > d$, and only to lower an entry, so that the set of the entries of value d does not change during the layer, and by (b) it is the set of the entries at distance D_k; by (c), the blocks with lo$[B] \leq d$ are exactly those that hold one of them, and each block is taken by one worker, so that each such entry is scanned exactly once. The loop of compare-and-swap of a relaxation ends with $h[r'] \leq d + c$; hence, after the layer, $h[r'] \leq D_k + c$ for each edge (r', r) of cost c into an entry r at distance D_k. Now (a) holds for $k + 1$, since the entries at distance less than D_{k+1} are those at distance at most D_k. For (b), let $\delta(r) = D_{k+1}$, and let q be the next state on a path of least cost from r to a goal, by an edge of cost $c \geq 1$; then $\delta(q) = D_{k+1} - c \leq D_k$, q has been scanned with its exact value, and $h[r] \leq \delta(q) + c = \delta(r)$, i.e., $h[r] = D_{k+1}$ since $h[r] \geq \delta(r)$; an entry at a larger distance has $h[r] \geq \delta(r) > D_{k+1}$, and one at a smaller distance has $h[r] \leq D_k$. For (c), consider a block B after line 15 and an entry e of B whose current value v is in $(D_k, 255)$. If e was last written during the layer, v was also offered to next$[B]$ by the compare-and-swap that follows the write; otherwise, v is the value that the scan of B read at line 13, if B was scanned, or a value that e had at the start of the layer, which by (c) is at least the old lo$[B]$, if it was not (such a block holds no entry of value D_k). Hence the new lo$[B]$ is at most v. Conversely, each term of the minimum of line 15 is a value that an entry of B held during the layer, in $(D_k, 255)$, and the current value of that entry is not larger and still above D_k, since entries are only lowered, to values above D_k. Thus (c) holds for $k + 1$, and line 16 sets d to the least value above D_k in the table, which by (b) is D_{k+1}; after D_L no value is in $(D_L, 255)$, line 16 sets $d = 255$, and the loop ends. The end of each layer is a join of the workers, which orders the writes of a layer before the reads of the next one; within a layer the entries and next are accessed by atomic operations, and lo$[B]$ only by the worker that took B. (ii) By the induction, the entries scanned at the layer D_k are those at distance D_k, and the check of `AbstractionPDB.h:193` is made for each edge entering them, before the comparison with $h[r']$; the first layer at which an edge with $D_k + c \geq 255$ enters a scanned entry raises the flag, and the throw follows the join. If some entry had a finite distance above 254, a path of least cost from it would contain an edge (r', r) with $\delta(r) \leq 254 < \delta(r') = \delta(r) + c$, and the throw would occur. (iii) Each entry of finite distance is scanned once, with one call of `for_each_predecessor`, and a scanned block costs 2^{14} reads; a block is scanned at most once per layer and only at the layers of the distances of its entries; hence $\sum_k \beta_k \leq (L+1)\, n_b$, and $\sum_k \beta_k \leq N_f$, since each pair of a block and of a layer at which it is scanned holds an entry of its own. Lines 6 and 15 visit the n_b blocks once per layer. The memory is the byte per entry and the two bytes of lo and next per block. □

Note that the condition of (i) is slightly stronger than "every distance is at most 254": with unit costs and a

space whose edges come in both directions, as all those of the library, a table whose largest distance is 254 is refused, exactly as in §5.5. Also, the bound (iii) is better than the $O((D+1)N)$ reads of Algorithm 5.2 when the entries of each distance are concentrated in few blocks, which the lexicographic ranking favours (Lemma 5.4 (ii)); with the ranking of Myrvold and Ruskey of §5.14, whose ranks have no locality, it degrades to the scan of every block at every layer. The test `abstraction` checks (i) against the algorithm of Dijkstra on a space of 10^5 states with the costs 1 to 2 and 1 to 4, with chords between r and $2r$ and 7 unreachable states, on 3 workers, and (ii) on a path of 300 states.

The distances are stored by the class template `AbstractionTable< B >`, $B \in \{4, 8\}$ bits per entry, whose constructor takes the vector of `abstract_distances`; a value above $2^B - 1$ is stored as $2^B - 1$, so that with $B = 4$ an entry of 255 becomes 15. Its lookups are `operator[]( r )`, and `size()`, `bytes()` and `histogram()` (the number of the entries of each value) describe it. The saturation keeps both properties of the table.

Lemma 5.16 (Saturation). If h is admissible, or consistent, with nonnegative costs, so is $\min\{h, M\}$ for each constant $M \geq 0$.

Proof. Admissibility is immediate, since $\min\{h, M\} \leq h$. For consistency, let (u, v) be an edge of cost $c \geq 0$. If $h(v) \geq M$, then $\min\{h(u), M\} \leq M \leq c + \min\{h(v), M\}$; otherwise $\min\{h(u), M\} \leq h(u) \leq c + h(v) = c + \min\{h(v), M\}$. Both values are 0 at a goal. □

A table is saved and loaded with a key of 64 bits that its owner computes from the abstraction (the goal, the pattern, the ranking and B, for the tables of §5.15 and §5.16). `save( path , key )` (`AbstractionPDB.h:310`) writes, in little-endian order, the magic `PBeFSABS`, the version of the format (`file_version`, 1), B, the key, the number of the entries, the words of 64 bits of the packed entries, and a checksum of everything before it. The checksum chains the mixing function of SplitMix64 (§1.5) over the words of the file, and, as in §5.10, the file is written under a temporary name in the same directory and then renamed. `load( path , key , n )` throws `std::runtime_error` if the file cannot be read, has a wrong length, fails the checksum, or holds another table (another magic, version, B, key or number of entries). Finally, `cached( dir , name , key , n , make )` returns the table of the file `dir/name` if it exists and loads. Otherwise it builds the table by `make()` (which returns the distances, typically by `abstract_distances`), creates the directory if needed, and saves the table there; with `dir` empty it only builds it. A file that does not load is thus built again and overwritten. The only claim of this paragraph that we do not prove is, as in §5.10, that a checksum of 64 bits detects every corruption. The declarations are the following, quoted by hand since they come with their bodies.

```
template< WorkerExecutor Executor = ThreadPool , AbstractSpace S >
[[nodiscard]] std::vector< std::uint8_t > abstract_distances(
 const S & space , unsigned threads = 0 );

template< unsigned B = 8 >
class AbstractionTable {
 public:
 static constexpr unsigned bits = B;
 static constexpr std::uint32_t file_version = 1;
 AbstractionTable( void ) = default;
 explicit AbstractionTable( const std::vector< std::uint8_t > & h );
 [[nodiscard]] unsigned operator[]( std::uint64_t r ) const noexcept;
 [[nodiscard]] std::uint64_t size( void ) const noexcept;
 [[nodiscard]] std::size_t bytes( void ) const noexcept;
 [[nodiscard]] std::vector< std::uint64_t > histogram( void ) const;
 void save( const std::string & path , std::uint64_t key ) const;
 [[nodiscard]] static AbstractionTable load( const std::string & path ,
                                             std::uint64_t key ,
                                             std::uint64_t n );
 template< class Make >
 [[nodiscard]] static AbstractionTable cached( const std::string & dir ,
                                               const std::string & name ,
                                               std::uint64_t key ,
                                               std::uint64_t n ,
                                               Make && make );
 [[nodiscard]] static std::uint64_t checksum( const std::uint8_t * b ,
                                              std::size_t n ) noexcept;
 };
```

Here `WorkerExecutor` is the concept of the executors of the workers and `ThreadPool` the default one (§1.6). We remark that the construction holds at the same time the N bytes of the distances and the $NB/8$ bytes of the packed table, into which the constructor of `AbstractionTable` copies them; this peak, and not the table, is what decides whether a table can be built on a machine.

5.14 Ranking the k-permutations

The abstract states of the permutation puzzles are the k-permutations of n items, i.e., the sequences $a = (a_0, \ldots, a_{k-1})$ of k distinct values of $[0, n)$, a_i being the position of the i-th object of a pattern; there are $N = n!/(n-k)!$ of them, which `k_permutations( n , k )` computes (0 if $k > n$ or if N does not fit in 64 bits). The header `Permutation.h` gives three bijections between them and $[0, N)$, each a model of the concept `Ranking`: `R( n , k )` builds the ranking, `size()` is N, `rank( a )` is the rank of the k values at `a`, and `unrank( r , a )` writes the values of the rank r into `a`; both use no state but the stack, so that many threads may call them at once. Each ranking has a `name` ("lex", "tree" and "mr"), which enters the names of the files of the tables, and its constructor throws `std::invalid_argument` if n exceeds its limit, if $k > n$, or if N does not fit in 64 bits (for the full permutations, if $n > 20$, since $20! < 2^{64} < 21!$).

```
template< class R >
concept Ranking = std::constructible_from< R , unsigned , unsigned > &&
 requires( const R & r , const std::uint8_t * a , std::uint8_t * b ,
           std::uint64_t x ) {
  { r.size() } -> std::convertible_to< std::uint64_t >;
  { r.rank( a ) } -> std::convertible_to< std::uint64_t >;
  r.unrank( x , b );
  };
```

- `LexRanking`, for $n \le 64$, is the lexicographic rank of §5.4 with the indices from 0: the digits are $d_i = a_i - |\{j < i : a_j < a_i\}| \in [0, n-i)$, the rank is $\sum_i d_i \prod_{j>i}(n-j)$, and the set of the values already seen is a word of 64 bits, so that Algorithm 5.1 applies as it is: each digit is a population count, and each inverse digit clears the d_i lowest bits of the word of the free values and takes the lowest one left. By Lemma 5.4, its rank takes $O(k)$ operations and its unrank $O(k + \sum_i d_i) = O(k\,n)$, and two ranks of a block of $n - k + 1$ consecutive ranks differ only in the last item.
- `TreeRanking`, for $n \le 256$, gives the same lexicographic rank by a complete binary tree of counters over the values, after Bonet [62], Algorithm 5.8.
- `MyrvoldRuskeyRanking`, for $n \le 256$, is the linear-time ranking of Myrvold and Ruskey [34] restricted to k items, Algorithm 5.9.

The header has no pseudocode for the last two; Algorithms 5.8 and 5.9 transcribe the formulas of its comments. In Algorithm 5.8 the tree has $2^L \ge n$ leaves, L being the least such integer, the node $v \ge 1$ has the children $2v$ and $2v+1$, and the leaf of the value x is $2^L + x$; the counters are a local array of 512 entries.

Algorithm 5.8 (RANK and UNRANK by a tree of counters).

```
 1  function RANK(a)
 2     c[v] ← 0 for each node v;  r ← 0
 3     for i ← 0 to k − 1 do
 4        v ← 2^L + a_i;  below ← 0
 5        while v > 1 do
 6           if v is a right child then below ← below + c[v − 1]
 7           c[v] ← c[v] + 1;  v ← ⌊v / 2⌋
 8        r ← r · (n − i) + (a_i − below)
 9     return r
10  function UNRANK(r)
11     for i ← k − 1 down to 0 do
12        d_i ← r mod (n − i);  r ← r div (n − i)
13     c[2^L + x] ← [x < n] for each x;  c[v] ← c[2v] + c[2v + 1] for v < 2^L
14     for i ← 0 to k − 1 do
15        v ← 1;  x ← d_i
16        while v < 2^L do
17           c[v] ← c[v] − 1;  v ← 2v
18           if x ≥ c[v] then x ← x − c[v];  v ← v + 1
```

```
19        c[v] ← 0;  a_i ← v − 2^L
20    return a
```

Lemma 5.17 (The tree of counters). Algorithm 5.8 computes the rank of `LexRanking` and its inverse. Its rank takes $O(kL) = O(k \log n)$ operations besides the clearing of line 2, and its unrank $O(2^L + kL) = O(n + k \log n)$, the term 2^L being the construction of the tree of line 13 at each call.

Proof. In RANK, $c[v]$ counts, for each node $v \geq 2$, the values of $a_0, \ldots, a_{i-1}$ in the leaves below v, since line 7 adds 1 to every node on the path of a_i but the root. The left siblings of the nodes on the path from the leaf of a_i to the root are the roots of disjoint subtrees whose leaves are exactly the values smaller than a_i; hence line 6 sums to $|\{j < i : a_j < a_i\}|$, and line 8 adds the digit d_i of the lexicographic rank by the rule of Horner. In UNRANK, $c[v]$ counts the values below v not yet taken, by line 13 at the start. By induction on the depth, if $x < c[v]$ then the $(x+1)$-th free value below v is the $(x+1)$-th below its left child when $x < c[2v]$, and the $(x - c[2v] + 1)$-th below its right child otherwise, which is line 18; hence the loop of lines 16 to 18 reaches the leaf of the $(d_i + 1)$-th free value, which is line 13 of Algorithm 5.1, and lines 17 and 19 remove it from the counts on its path. The bounds follow from the L levels of each path and from the $2^{L+1} - 1$ nodes of line 13. □

The counters of RANK never exceed $2^{L-1} \leq 128$, the number of the leaves below a child of the root, and are bytes; those of UNRANK reach $n \leq 256$ at the root, and have 16 bits. We remark that the comment of the header gives $O(k \log n)$ for both directions, which leaves out the $O(n)$ of line 13; with n up to 256 and k often close to n the difference is a constant.

Algorithm 5.9 (RANK and UNRANK of Myrvold and Ruskey on k items).

```
 1  function UNRANK(r)
 2     π ← (0, 1, ..., n − 1)
 3     for j ← 0 to k − 1 do
 4        e_j ← r mod (n − j);  r ← r div (n − j)
 5        exchange π[n − 1 − j] and π[e_j]
 6        a_j ← π[n − 1 − j]
 7     return a
 8  function RANK(a)
 9     π ← (0, 1, ..., n − 1);  π^{-1} ← π;  r ← 0;  m ← 1
10     for j ← 0 to k − 1 do
11        e_j ← π^{-1}[a_j]
12        r ← r + m · e_j;  m ← m · (n − j)
13        exchange π[n − 1 − j] and π[e_j], and update π^{-1}
14     return r
```

Thus the rank is $\sum_j e_j \prod_{l<j} (n-l)$, with $e_j \in [0, n-j)$ the position exchanged with $n-1-j$ at the step j; for $k = n$, UNRANK runs the steps of the function unrank1 of [34], $\pi[i-1] \leftrightarrow \pi[r \bmod i]$ followed by $r \leftarrow \lfloor r/i \rfloor$ for $i = n, n-1, \ldots$, and RANK is its function rank1. Figure 5.10 follows UNRANK on an example.

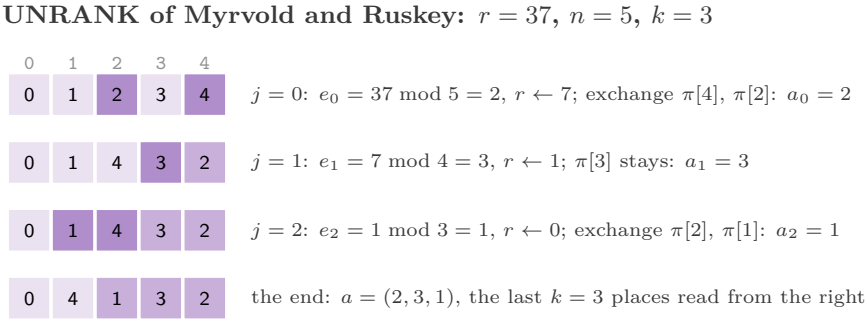


Figure 5.10. UNRANK of Algorithm 5.9 for $r = 37$, $n = 5$ and $k = 3$: each row is the array π before the step j, with the two places that the step exchanges in violet and the places already read in the darker tint; the digits $e_0 = 2$, $e_1 = 3$ and $e_2 = 1$ give $a = (2, 3, 1)$, and RANK of $(2, 3, 1)$ gives back $2 + 3 \cdot 5 + 1 \cdot 5 \cdot 4 = 37$.

Lemma 5.18 (The ranking of Myrvold and Ruskey). Algorithm 5.9 is a bijection between $[0, N)$ and the

k-permutations of n items, with RANK its inverse; both take $O(n+k)$ operations.

Proof. Let π_j be the array before the step j, in UNRANK or in RANK. The step l moves a value into the position $n-1-l$, and the later steps exchange only positions below it; hence the positions $n-j,\dots,n-1$ of π_j hold $a_{j-1},\dots,a_0$, and the positions $0,\dots,n-1-j$ hold the other $n-j$ values. In UNRANK, $e_j \le n-1-j$, and line 6 gives $a_j = \pi_j[e_j]$, a value not among $a_0,\dots,a_{j-1}$: the output is a k-permutation. In RANK on a k-permutation a, the array π_j depends only on $e_0,\dots,e_{j-1}$, as in UNRANK, and a_j, not among $a_0,\dots,a_{j-1}$, lies in a position $e_j = \pi_j^{-1}[a_j] \le n-1-j$; thus the digits are in range, and UNRANK applied to them makes the same exchanges and outputs $\pi_j[e_j] = a_j$ at each step. Hence UNRANK(RANK(a)) $= a$ for each a, and UNRANK, which maps the N vectors of digits (in bijection with $[0,N)$ by the mixed radix of lines 4 and 12) onto the N k-permutations, is surjective between two sets of the same size, i.e., a bijection whose inverse is RANK. Each step takes $O(1)$ operations, and line 2 or 9 takes $O(n)$. □

The three rankings thus differ in the order of the ranks and in the cost. The lexicographic order keeps close abstract states in close entries, which serves the blocks of Algorithm 5.7 and of the max-compression; the tree of counters extends it beyond 64 items; the ranking of Myrvold and Ruskey has no locality, and its cost does not depend on the digits. The test `permutation` checks that each ranking is a bijection, and that `LexRanking` and `TreeRanking` enumerate the k-permutations in lexicographic order, for $n = 1, 4, 7$ and every $k \le n$, that the two lexicographic rankings agree on 64 items, and that `MyrvoldRuskeyRanking` and `TreeRanking` are inverse on 200 items; which of the three is fastest in a lookup is a matter of the hardware, not a theorem. Last, `bytes_hash( b , n )` is the hash of the states of the permutation puzzles, which are arrays of bytes: from the seed `0x9E377 9B97F4A7C15` exclusive-or n, it applies the mixing function of SplitMix64 to the exclusive or of the hash and of each word of 8 bytes of the array, the last one padded with zeros. It is not injective when $n > 8$, and the searches need it only as a hash.

5.15 The pattern databases of the permutation puzzles

A *permutation puzzle* has groups of positions, each holding as many objects as positions, and an object of a group of modulus m also has an orientation in $[0,m)$. A move is, for each group, a permutation of the positions with a turn per position: the object that comes to the position i is the one that was at from$[i]$, and its orientation grows by turn$[i]$ modulo m. The pancakes, the burned pancakes ($m=2$), TopSpin, and the corners ($m=3$) and the edges ($m=2$) of Rubik's cube are such puzzles (Chapter 14); the sliding-tile puzzles are not, since the effect of a move depends on where the blank is, and they keep the tables of the previous sections. In the library, `PermutationGroup` and `PermutationMove` describe the groups and the moves, and the concept `PermutationDomain` a `Domain` that is a permutation puzzle: `permutation_groups()` and `permutation_moves()` give the groups and the moves, every move of the domain being one of them, `permutation_of( s , g , obj , ori )` writes the object at each position of the group g in the state s and its orientation, and `for_each_goal_permutation( g , f )` calls `f( obj , ori )` for the group g of each goal (of each of them, if they are several).

```
struct PermutationGroup {
 unsigned size = 0;          ///< the positions, and the objects
 unsigned modulus = 1;       ///< the orientations of an object (1: none)
 bool zero_sum = false;      ///< the orientations sum to 0 modulo modulus
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// a move of a permutation puzzle, by group

struct PermutationMove {
 /// from[ g ][ i ]: the position whose object comes to i
 std::vector< std::vector< std::uint8_t > > from;
 /// turn[ g ][ i ]: what its orientation grows by
 std::vector< std::vector< std::uint8_t > > turn;
 unsigned cost = 1;          ///< the cost, 1 to 254
 };
```

```
template< class D >
concept PermutationDomain = Domain< D > &&
 requires( const D & d , const typename D::state_type & s , unsigned g ,
           std::uint8_t * o , std::uint8_t * r ) {
```

```
{ d.permutation_groups() }
 -> std::convertible_to< std::vector< PermutationGroup > >;
{ d.permutation_moves() }
 -> std::convertible_to< std::vector< PermutationMove > >;
d.permutation_of( s , g , o , r );
d.for_each_goal_permutation( g , detail::permutation_sink{} );
};
```

A *pattern* is a set of k objects of one group, all the others being "don't care" as in Culberson and Schaeffer [21]: its abstract state is the sequence p of the positions of its k objects, a k-permutation of the n positions of the group, with their k orientations o, and its rank is

$$\operatorname{rank}(p, o) = \operatorname{rank}_R(p)\, m^{k'} + \sum_{j<k'} o_j\, m^j,$$

where rank_R is a `Ranking` and $k' = k$, or $k' = k-1$ when the pattern holds the whole group and the orientations of the group sum to 0 modulo m in every state (`PermutationGroup::zero_sum`), the last orientation being then implied by the others. The abstract space has thus $N = m^{k'}\, n!/(n-k)!$ states, e.g., the $8! \cdot 3^7$ of the corners of Rubik's cube in Korf [59]. `PermutationSpace< R >` is this space, a model of `AbstractSpace`: its goals are the images of the goals of the puzzle, and the predecessor of (p, o) by a move is the abstract state whose object j is at $\text{from}[p_j]$ with the orientation $o_j - \text{turn}[p_j]$ modulo m, the move being skipped when it changes no position and no orientation of the pattern (it is then a loop of the abstract space). Its constructor refuses (by `std::invalid_argument`) a group out of range, of more than 256 positions or of a modulus of 0 or more than 256, an empty pattern, an object out of the group or given twice, a move whose permutation or turns of the group do not have n entries or whose cost is not in $\{1, \ldots, 254\}$, a goal of another size, and a space of more than 2^{64} states; the ranking refuses, in turn, a group beyond its limit (64 positions for `LexRanking`). It does not check that each move permutes the positions, nor that its turns keep the sum of the orientations when the last one is implied; these are properties of the puzzle that the following statement assumes.

Proposition 5.19 (The tables of the patterns are consistent). Let the moves of a `PermutationDomain` be permutations of the positions, each with the cost of the corresponding move of the domain, and, when the group of a pattern has `zero_sum`, keep the sum of the orientations. Then the map ψ of each state to the abstract state of a pattern is an abstraction in the sense of Definition 5.1; the table of the pattern built by Algorithm 5.7, stored in B bits, is a consistent heuristic; and so is the maximum of the tables of any set of patterns, and its maximum with any other consistent heuristic.

Proof. Let a move take the state s to t. An object x of the pattern at the position q with the orientation u in s is, in t, at the position p with $\text{from}[p] = q$ (the move being a permutation), with the orientation $o = u + \text{turn}[p]$ modulo m. Hence $\psi(s) = (q, u)$ is the predecessor of $\psi(t) = (p, o)$ that `for_each_predecessor` generates for this move, by an edge of the cost of the move, unless the move changes no position and no orientation of the pattern, in which case $\psi(s) = \psi(t)$; with an implied orientation, the rank uses only the first $k-1$ orientations, which determine the last one in both states. The goals of the puzzle are mapped to the abstract goals by construction. Hence ψ satisfies (i) and (ii) of Definition 5.1, and by Theorem 5.15 the table holds the distances of the abstract space, whose heuristic is consistent by Proposition 5.2; the saturation to $2^B - 1$ keeps it so (Lemma 5.16), and the maxima are consistent by Lemma 2.5. □

The tables add up only under the condition of §5.6, stated here for any puzzle.

Proposition 5.20 (The sum of the tables). If no move moves (changes the position or the orientation of) objects of two of the patterns, of the same group or of different ones, the sum of their tables is admissible and consistent.

Proof. Let P be a path from s to a goal, and $c_j(P)$ the cost of its moves that move an object of the pattern π_j. The image of P in the abstract space of π_j, without its loops, is a path from $\psi_j(s)$ to an abstract goal of cost $c_j(P)$, so that the table of π_j is at most $c_j(P)$; since each move counts in at most one $c_j(P)$, $\sum_j c_j(P) \le c(P)$. Along a move (s, t) of cost c, only the abstract state of one pattern may change, by an edge of cost c, and its table changes by at most c by Proposition 5.19, while the others do not change; at a goal every table is 0. □

The condition holds for no puzzle of Chapter 14 but the Towers of Hanoi, whose moves move one disk (§5.16): a flip of k pancakes, a turn of TopSpin and a quarter turn of the cube move several objects, which may belong to different patterns. In the library, the class template `PermutationPDB< R , B >` is the table of one pattern: its constructor takes the domain, the group, the objects of the pattern, the number of workers and a cache

directory, builds the `PermutationSpace` from the domain, and obtains the table by `AbstractionTable< B >::cached` with `abstract_distances` on the workers of an executor (PermutationPDB.h:389). The key of the file is the checksum of the groups of the puzzle, of the costs, permutations and turns of its moves in the group, of the goals of the group, of the group, of the objects of the pattern, of the name of the ranking and of B. The file is named `perm-` followed by the name of the ranking, B and the key in hexadecimal, so that two patterns, puzzles or rankings never share a file, up to the collisions of the checksum. `value( loc , ori )` looks up the abstract state whose object x is at `loc[x]` with the orientation `ori[x]`, and `value( d , s )` that of a state of the domain, by `permutation_of`. `PermutationHeuristic< D , R , B >` is the maximum of several tables, or their sum when `additive` is true, which the constructor does not check against Proposition 5.20; it reads the state once per group and looks up each table of that group. Last, `PDBDomain< D , H >` is the domain `D` with the heuristic $\max\{h_D(s), H(s)\}$ for a callable `H` of the states (such as a `PermutationHeuristic` or the `Hanoi` PDB of §5.16): it forwards the members of `D`, and `OperatorDomain`, `TreeOperatorDomain`, `FeatureDomain`, `IncrementalFeatureDomain` and `PerfectHashDomain` when `D` models them, passes to the expansions of `D` the heuristic of `D` itself (so that an incremental heuristic of `D` stays right), and computes `H` anew for each successor. The declarations are the following, quoted by hand.

```
template< Ranking R = LexRanking >
class PermutationSpace {
 public:
 PermutationSpace( const std::vector< PermutationGroup > & groups ,
                   const std::vector< PermutationMove > & moves , unsigned g ,
                   const std::vector< std::uint8_t > & objects ,
                   const std::vector< std::pair< std::vector< std::uint8_t > ,
                                                 std::vector< std::uint8_t >
                                                 > > & goals );
 [[nodiscard]] std::uint64_t size( void ) const noexcept;
 [[nodiscard]] std::uint64_t rank( const std::uint8_t * p ,
                                   const std::uint8_t * o ) const noexcept;
 [[nodiscard]] std::uint64_t rank_objects( const std::uint8_t * loc ,
                                           const std::uint8_t * ori ) const
  noexcept;
 void unrank( std::uint64_t r , std::uint8_t * p , std::uint8_t * o ) const
  noexcept;
 };

template< Ranking R = LexRanking , unsigned B = 8 >
class PermutationPDB {
 public:
 template< PermutationDomain D , WorkerExecutor Executor = ThreadPool >
 PermutationPDB( const D & d , unsigned g ,
                 const std::vector< std::uint8_t > & objects ,
                 unsigned threads = 0 , const std::string & cache = "" ,
                 Executor * = nullptr );
 [[nodiscard]] unsigned value( const std::uint8_t * loc ,
                               const std::uint8_t * ori ) const noexcept;
 template< PermutationDomain D >
 [[nodiscard]] unsigned value( const D & d ,
                               const typename D::state_type & s ) const;
 };

template< PermutationDomain D , Ranking R = LexRanking , unsigned B = 8 >
class PermutationHeuristic {
 public:
 using PDB = PermutationPDB< R , B >;
 PermutationHeuristic( const D & d , std::vector< PDB > pdbs ,
                       bool additive = false );
 [[nodiscard]] unsigned operator()( const typename D::state_type & s ) const;
 };

template< Domain D , class H >
class PDBDomain {
 public:
 PDBDomain( const D & d , const H & h );
```

```
};
```

The domain and the tables given to `PermutationHeuristic` and to `PDBDomain` must outlive them. A lookup costs `permutation_of`, i.e., $O(n)$ operations for the group, and, for each table of the group, the gathering of the k objects, one rank of §5.14 and one access to the table; the access, to a table of up to tens of megabytes, is usually the cost that counts.

5.16 The heuristics of the permutation puzzles

Each permutation puzzle of Chapter 14 comes with a heuristic of its own, computed from the state in time linear in its size, and the tables of §5.15 are combined with it by `PDBDomain`; Rubik's cube and the Towers of Hanoi have the heuristic 0, and their heuristics are the tables alone. In this section a stack, a ring or a cube is the sequence of its objects by position, and we prove the properties of each heuristic.

The gaps of the pancakes

A stack of N pancakes of sizes $0, \dots, N-1$ is a permutation p, $p[0]$ being the top, and the flip of $k \geq 2$ reverses $p[0..k-1]$ at cost 1; the goal is the identity. A *gap* is a pair of adjacent pancakes whose sizes differ by more than one, or the pair of the bottom pancake and of the plate, of size N, when the bottom pancake is not $N-1$; the gap heuristic of Helmert [60] is the number of the gaps. In the variant GAP-x of Holte et al. [61] the gaps between two pancakes one of which is among the x smallest are not counted, while the gap with the plate always is; it weakens the heuristic in a controlled way, by which heuristics of graded strength are obtained for the evaluation of the searches. `PancakePuzzle< N >` takes x as the second argument of its constructor (0 for the gap heuristic itself).

Proposition 5.21 (The gaps). For every $x \geq 0$, GAP-x is consistent, 0 at the goal, and the heuristic of the successor by the flip of k is $h - \mathrm{gap}(p[k-1], b) + \mathrm{gap}(p[0], b)$, with b the pancake $p[k]$ or the plate if $k = N$; for $x = 0$ it is 0 only at the goal.

Proof. The flip of k reverses the block $p[0..k-1]$, so that the adjacent pairs inside the block are the same unordered pairs after the flip, and the pairs below the block do not change; only the pair at the boundary, $(p[k-1], b)$, becomes $(p[0], b)$, which gives the formula. Whether a pair is a gap depends only on the two sizes as an unordered pair (their difference and the smaller one), except for the plate, which is always the lower element of its pair; hence h changes by at most 1 along a move of cost 1, and it is consistent. The identity has no gap. Conversely, let $x = 0$ and let a stack have no gap; then $p[N-1] = N-1$, and if $p[j] = j$ for each $j > i$, the pancake $p[i]$ is one of $0, \dots, i$ and differs from $p[i+1] = i+1$ by at most one, i.e., $p[i] = i$. By induction the stack is the identity. □

For $x > 0$ a stack other than the goal may have no gap that counts, e.g., $(0, 2, 1, 3, 4, 5, 6)$ under GAP-2, so that GAP-x is weaker than GAP-0 by construction (each gap it counts is a gap). In the burned variant, each pancake also has a side burned, a flip turns every pancake that it moves, and the goal is the identity with every burned side down. A pair of adjacent pancakes, a above b, is *good* if $b = a+1$ and neither is turned, or if $a = b+1$ and both are turned (the pair is then as after a flip of a block that holds it), and the bottom pancake is good with the plate if it is $N-1$ and not turned; the heuristic of `BurnedPancakePuzzle< N >` is the number of the pairs that are not good.

Proposition 5.22 (The burned gaps). The heuristic of the burned pancakes is consistent and 0 exactly at the goal, and the heuristic of the successor by the flip of $k \geq 1$ is $h - \mathrm{bad}(p[k-1], b) + \mathrm{bad}(\bar{p}[0], b)$, with $\bar{p}[0]$ the top pancake turned and b as in Proposition 5.21.

Proof. A flip of k reverses the block and turns its pancakes; a pair a above b inside the block becomes the pair $\bar{b}$ above $\bar{a}$. If the pair was good with neither turned and $b = a+1$, the new pair has both turned and its upper element exceeds the lower by one, so it is good; symmetrically for a good pair with both turned; and a pair with exactly one pancake turned, or whose sizes are not consecutive in the required order, stays bad, since the map of a pair to its image is an involution that preserves the two kinds. Hence only the boundary pair changes, which gives the formula (with $k = 1$ the flip only turns the top pancake), and the heuristic changes by at most 1 along a move of cost 1; the goal has only good pairs. Conversely, if every pair is good, the bottom pancake is $N-1$, not turned, and if the pancakes below the position i are $i+1, \dots, N-1$, none turned, the pair of $p[i]$ and $i+1$ is good with $i+1$ not turned, so that $p[i]$ is not turned and equals i; by induction from the bottom the stack is the goal. □

The breakpoints of TopSpin

A state of TopSpin is a ring of N tokens, and the move i reverses the K tokens at the positions $i, \dots, i+K-1$ modulo N; the goals are the N rotations of the identity. A *breakpoint* is a pair of adjacent positions, the position $N-1$ being adjacent to 0, whose tokens are not consecutive modulo N in either order, and the heuristic of `TopSpin< N , K >` is $\lceil b/2 \rceil$ for b breakpoints. TopSpin is the permutation puzzle of the experiments of Felner et al. [167] on the dual lookups and of Holte et al. [30] on the maximum of several databases, whose tables the library builds by §5.15.

Proposition 5.23 (The breakpoints). The heuristic $\lceil b/2 \rceil$ of TopSpin is consistent and 0 at the goals.

Proof. The move i reverses the tokens of the turnstile, so that the adjacent pairs inside it are the same unordered pairs after the move, and being consecutive modulo N in either order is a property of the unordered pair; the pairs outside it do not change. Only the two pairs at the ends of the turnstile, between the positions $i-1$ and i and between $i+K-1$ and $i+K$, may change (when $K=N$ they are the same pair, whose two tokens are both in the turnstile and simply exchange their positions, so that nothing changes). Hence b changes by at most 2, and $\lceil b/2 \rceil$ by at most 1, along a move of cost 1. A rotation of the identity has the tokens t and $t+1$ modulo N in adjacent positions, and no breakpoint. □

The converse fails: the ring in decreasing order has no breakpoint and is not a goal, so that the heuristic is not 0 exactly at the goals; this does not affect admissibility, which follows from consistency.

The tables of Korf for Rubik's cube

`RubikPDBHeuristic` is the heuristic of Korf [59]: the maximum of three tables, one of the 8 corners and two of two sets of 6 of the 12 edges, with 4 bits per entry, a `PermutationHeuristic< RubiksCube , LexRanking , 4 >`. The pattern of the corners is their whole group, whose orientations sum to 0 modulo 3, and its table has $8! \cdot 3^7 = 88179840$ entries; each table of edges has $12!/6! \cdot 2^6 = 42577920$ entries; the three take 86667840 bytes, about 83 MiB. The sets of edges are UR, UF, UL, UB, FR, FL and DR, DF, DL, DB, BL, BR unless given to the constructor, two sets that the half turn about the axis of the faces L and R maps into each other, so that their tables have the same distribution of values. The sum of the tables is not admissible, since a quarter turn moves 4 corners and 4 edges (Proposition 5.20 does not apply), and the heuristic is their maximum, consistent by Proposition 5.19. Since the largest values are 11 for the corners and 10 for the edges, which are below 15, the entries of 4 bits are exact and no saturation occurs. The test `rubik-pdb` checks, in an optimized build only (in the others it only reports that it was skipped), the number of the entries of each table, its largest value and its mean, 8.764 for the corners and 7.668 for the edges as in [59], within $5 \cdot 10^{-4}$; according to the comment of the header, the split into the first and the last six edges of the order of the positions gives the means 7.619 and 7.626 instead. The test `rubik` checks smaller tables (4 corners and two sets of 4 edges) by IDA* and the parallel IDA* against IDA* without tables on random walks of 5 moves.

```
class RubikPDBHeuristic
 : public PermutationHeuristic< RubiksCube , LexRanking , 4 > {
 public:
 explicit RubikPDBHeuristic( const RubiksCube & d , unsigned threads = 0 ,
                             const std::string & cache = "" ,
                             const std::vector< std::uint8_t > & edges_a = {
                              0 , 1 , 2 , 3 , 8 , 9 } ,
                             const std::vector< std::uint8_t > & edges_b = {
                              4 , 5 , 6 , 7 , 10 , 11 } );
 };
```

The additive tables of the Towers of Hanoi

The Towers of Hanoi with four pegs of `TowersOfHanoi< D >` place D disks on the pegs 0 to 3, the peg of the disk d in the bits $2d$ and $2d+1$ of a word, and a move takes the top disk of a peg to a peg whose top disk is larger, or which is empty, at cost 1. The abstraction of the range of the disks $l, \dots, l+k-1$, `HanoiSpace( l o , k , goal )`, keeps only them: its 4^k states are the $2k$ bits of their pegs (the rank of the image of a state is a shift and a mask), its goal is the image of the goal of the puzzle, and its moves are those of the top disks among the range, each its own inverse, so that the predecessors of a state are its successors; it admits $1 \le k \le 16$ and $l+k \le 32$. The top disk of each peg is found by `hanoi_tops( x , k , top )`, which writes in `top[p]` the smallest of the k disks of the word x on the peg p, or k if there is none. Let $y = x \oplus (p \cdot \texttt{0x5555\dots5})$; a field

of two bits of x equals p if and only if both bits of y in it are 0, i.e., if and only if the bit of $\neg(y \mid (y \gg 1))$ at its low position is 1. A mask keeps the low bits of the fields of the k disks, and the disk is the number of the trailing zeros of the result divided by 2. The disks on a peg being stacked by size, the smallest one is the top one, and the four pegs take $O(1)$ operations on the word each.

Proposition 5.24 (The ranges of disks). `HanoiSpace` is an abstraction of `TowersOfHanoi` in the sense of Definition 5.1, the table of the range of all the disks is the exact distance to the goal, and the sum of the tables of disjoint ranges is admissible and consistent.

Proof. A move of a disk d of the range from the peg a to the peg b requires d to be the top disk of a and the top disk of b, if any, to be larger, among all the disks; then the same holds among the disks of the range, and the move is a move of the abstraction, of the same cost. A move of a disk out of the range leaves the image unchanged. The goal is mapped to the abstract goal, which gives Definition 5.1. With the range of all the disks the abstraction is the puzzle itself, and its distances are exact by Theorem 5.15. Each move moves one disk, which belongs to one range at most, and the argument of Proposition 5.20 gives the sum. □

`HanoiPDB( ks , goal , threads , cache )` is the sum of the tables of the consecutive ranges of the sizes `ks`, from the smallest disk, which are meant to sum to the number of the disks of the puzzle (the constructor does not check it); each table is an `AbstractionTable< 8 >`, built by Algorithm 5.7 or read from the cache directory, where it is named after the first disk, the size of the range and the rank of the goal, with a key that mixes the three. The heuristic of Siag et al. [57] for their instances of 20 disks is the sum of the tables of the 4 smallest and of the 16 largest disks, i.e., of $4^4 = 256$ and $4^{16} = 2^{32}$ entries, the latter of 4 GiB at a byte per entry, plus as much during its construction (§5.13). The test `hanoi` checks, on the 4^8 states of 8 disks and a random goal, that the table of all the disks is the exact distance, that the tables of 3 and 5 disks are admissible and consistent, and A* and the parallel IDA* with them on 16 instances.

```
inline void hanoi_tops( std::uint64_t x , unsigned k , unsigned top[ 4 ] )
 noexcept;

class HanoiSpace {
 public:
 HanoiSpace( unsigned lo , unsigned k , std::uint64_t goal );
 [[nodiscard]] std::uint64_t rank( std::uint64_t s ) const noexcept;
 [[nodiscard]] std::uint64_t size( void ) const noexcept;
 [[nodiscard]] unsigned lo( void ) const noexcept;
 [[nodiscard]] unsigned k( void ) const noexcept;
 };

class HanoiPDB {
 public:
 HanoiPDB( const std::vector< unsigned > & ks , std::uint64_t goal ,
           unsigned threads = 0 , const std::string & cache = "" );
 [[nodiscard]] unsigned operator()( std::uint64_t s ) const noexcept;
 [[nodiscard]] const std::vector< AbstractionTable< 8 > > & tables( void )
  const noexcept;
 };
```

5.17 Benchmark

Our benchmark of the heuristics is the sequential A* on the 100 instances of the 15-puzzle of Korf [7], with the Manhattan distance and with the pattern databases, run by `tools/PDBBench` from `tools/bench/pdb.sh`. First, the script builds (or loads from its cache directory) the tables of the partitions 5-5-5, 6-6-3 and 7-8, with the blank ignored and, for 5-5-5 and 7-8, minimized; the searches use the encodings in a nibble, in one bit (without the reflection), max-compressed (by 16 for 5-5-5, by 8 for 7-8), and by rank and select when it is available. Then it solves each instance with each heuristic, twice, running up to `JOBS` searches at the same time (4 by default). The costs are integers and exact, i.e., each heuristic is admissible and each instance is solved to its optimal cost. The machine and the compiler are those stated in the note of the table. The clock is the `Stopwatch` of `AStar::solve()`, as in §2.16: it does not include the construction or the loading of the tables, which precede the search, nor the start of the process and the reading of the instances.

Table 5.1. A* with the Manhattan distance and with the pattern databases on the 100 instances of Korf: for each heuristic (the partition and its encoding), the instances solved, the nodes expanded summed over the

instances, and the seconds summed over the instances, each instance the median of its repetitions.

heuristic	solved	expanded	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`*; no number is given until then.*

The numbers of expansions do not depend on the machine, and those of the README of the repository (section "Pattern databases", from the same script) are the following, summed over the 100 instances, with the ratio to the Manhattan distance.

heuristic	blank	entries	memory	expanded	ratio
Manhattan				1557138490	1
5-5-5, max-compressed by 16	ignored	min	49 KB	681766045	2.3
5-5-5, no reflection	ignored	1 bit	197 KB	369972440	4.2
5-5-5	ignored	4 bits	786 KB	263234565	5.9
6-6-3	ignored	4 bits	5.8 MB	167667391	9.3
7-8, no reflection	ignored	1 bit	72 MB	137386424	11.3
7-8, max-compressed by 8	ignored	min	36 MB	119574805	13.0
7-8	ignored	4 bits	288 MB	43587167	35.7
5-5-5	minimized	4 bits	786 KB	59968561	26.0
7-8	minimized	4 bits	288 MB	2019506	771

The rule of the blank weighs more than the size of the patterns there: with the blank minimized the partition 7-8 expands about 20000 nodes per instance, 21 times fewer than with the blank ignored, at the price of about 6000 reexpansions over the 100 instances. On the 24-puzzle the same script solves with IDA* and the partition 6-6-6-6 the ends of 50 random walks of 150 moves of the blank from the goal, with at most 10^{10} expansions each; the README reports 49 of them solved, with optimal costs from 50 to 84, while the 50 instances used by Jinnai and Fukunaga [5] are beyond such a limit.

The tables of §5.13 to §5.16 are measured by `AStarBench` on the permutation puzzles and on the Towers of Hanoi (`--domain pancake`, `burned`, `topspin`, `rubik` and `hanoi`, with the tables of `--pdb`, built on `--threads` workers or read from the directory of `--cache`), with the heuristic of the domain alone and with the tables. The clock of a search does not include the construction of the tables, which is measured apart, by the same program with an empty cache.

Table 5.2. The pattern databases of the permutation puzzles and of the Towers of Hanoi: for each puzzle, search and heuristic, the instances, the entries and the bytes of the tables, the seconds of their construction, and the instances solved, the nodes expanded and the seconds of the searches, summed over the instances.

puzzle	search	heuristic	instances	entries	MB	seconds to build	solved	expanded	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`*; no number is given until then.*

5.18 Notes and references

Pattern databases were introduced by Culberson and Schaeffer [21] for the 15-puzzle, with patterns that count each move. Korf and Felner [22] made them additive by counting only the moves of the tiles of the pattern, with

the partitions 7-8 and 6-6-6-6 and the reflection, and Felner, Korf and Hanan [23] studied the static and dynamic partitions, among them 5-5-5 and 6-6-3. Yang et al. [28] gave the general theory of the additive abstractions, and Holte et al. [30] showed that the maximum over several smaller databases can beat one larger database. Zahavi et al. [29] introduced the dual lookups, which Felner et al. [31] exploit with the bidirectional pathmax. Felner et al. [24] compressed the tables by the maximum (lossy) and without loss when the merged entries are known to differ by little. Breyer and Korf [25] stored the distances modulo 3, and Clausecker and Reinefeld [27] in one bit with the zero-aware encoding. Samadi et al. [33] learned a smaller model of the entries, and Nagahashi and Takahashi [32] built the largest tables on clusters, by partitioning the ranks among the processes as the blocks of the library are partitioned among the threads. The ranking of Myrvold and Ruskey [34] is the linear-time alternative to the lexicographic one, and the two-bit breadth-first search of Korf [26] is the model of the construction with the blank minimized. The library differs from these works in storing the excess over the Manhattan distance rather than the distance. According to the documentation of the repository, this makes 4 bits per entry enough for the tables of the library, and it keeps the Manhattan distance exact in the max-compression. Yet, the library has neither the dual lookups nor the lossless compression, and the searches that use the tables best on the 24-puzzle, i.e., IDA* with the bidirectional pathmax [31], are beyond its IDA*, which has no pathmax (Chapter 4).

The construction of §5.13 is the algorithm of Dial [65] with the table itself as its buckets; the lexicographic ranking by a tree of counters is that of Bonet [62], and the ranking of Myrvold and Ruskey [34] is restricted to k items by stopping their unranking after k steps. The pancake problem is due to Dweighter [63], the gap heuristic to Helmert [60], and its variant GAP-x to Holte et al. [61]; TopSpin is the permutation puzzle of the experiments of Felner et al. [167] and of Holte et al. [30]. The three tables of Rubik's cube, in half a byte per entry, are those with which Korf [59] solved optimally random instances of the cube, and the additive tables of ranges of disks of the Towers of Hanoi with four pegs follow Felner, Korf and Hanan [23], with the split of Siag et al. [57] for 20 disks. We remark that the permutation puzzles are those on which the dual lookups of [29] and [167] apply most directly, and that the tables of §5.15 have neither the dual lookups nor the compressions of §5.9, whose relative encodings rest on a Manhattan distance that these puzzles do not have.

5.19 Exercises

Exercise 5.1 (`ctest -R PBeFS.grid`). On a grid with 8 moves, where a move pays the cost of the cell it leaves and $\sqrt{2}$ times that on a diagonal, prove that the octile distance times the least cost of a cell is consistent, that the Euclidean distance times the same cost is consistent and dominated by it, and that the Manhattan distance is not admissible (Figure 5.1). The test checks A* with the octile distance against the costs of Dijkstra.

Exercise 5.2 (`ctest -R PBeFS.pdb`). Compute by hand, by Algorithm 5.1, the rank of the placement $(2, 7, 1)$ of three tiles on 9 cells, and unrank 229 back to $(4, 0, 7)$ (Figure 5.4). How many ranks does a block of Lemma 5.4 (ii) hold for the pattern of 8 tiles of the 15-puzzle, and how many blocks does a group of `MinNibbleEntries` with $c = 8$ meet at most? The test checks that unrank and rank are inverse on every placement of the patterns of the 8-puzzle.

Exercise 5.3 (`ctest -R PBeFS.pdb`). Prove that with $k = n - 1$ tiles (all the tiles in the pattern) both rules of the blank give the puzzle itself, and deduce from the parity of Chapter 14 that only half of the placements are reachable, which is why the constructor of `TilePattern` refuses such a pattern. Give a placement of a pattern of 2 tiles of the 8-puzzle for which the two rules give different distances. The test checks the tables of both rules against a breadth-first search written apart from the library, on each placement of patterns of 1 to 7 tiles of the 8-puzzle.

Exercise 5.4 (`ctest -R PBeFS.pdb`). Find a move of the 8-puzzle along which the sum of the tables of the partition 4-4 with the blank minimized decreases by more than one, and explain it by the regions of the blank. Why does the proof of Theorem 5.8 not apply? The test counts these moves.

Exercise 5.5 (`ctest -R PBeFS.pdb`). Prove that the reflection of Lemma 5.9 is not an automorphism when the goal has the blank off the diagonal, by an example on the 8-puzzle, and check that the library then does not use it. Show that $\phi(\phi(s)) = s$.

Exercise 5.6 (`ctest -R PBeFS.pdb`). Prove Lemma 5.10 for $M = 4$ with the bit of `ParityBitEntries` and the parity of m_π, and show by an example that the residues modulo 2 would not suffice. Why does a relative encoding with the reflection fail? The test checks that the value of each state of the 8-puzzle is recovered from the residues only, and that of each successor from its parent.

Exercise 5.7 (`ctest -R PBeFS.pdb`). Show by an example that the sum of the tables of two patterns that share a tile, or of a pattern that holds the blank, may exceed h^*; the test checks that the library refuses both.

Exercise 5.8 (`ctest -R PBeFS.pdb-search`). Show that the domination of Proposition 5.14 (ii) may be strict by a placement of the patterns $\{1\}$ and $\{2\}$ of the 8-puzzle in which the two tiles block each other. By Theorem 2.15, what can be said of the expansions of A* with the partition 4-4 of the 8-puzzle against those with the Manhattan distance, and why only below the last layer of f? The test checks that they are fewer on its instances.

Exercise 5.9 (`ctest -R PBeFS.tiles24`). The test builds the additive database of the 24-puzzle with the 8 patterns of 3 consecutive tiles, checks that it dominates the Manhattan distance on random walks of 40 moves, and solves them with IDA*. Compute the number of the entries of these tables, and compare it with that of the partition 6-6-6-6 (§5.11).

Exercise 5.10 (`ctest -R PBeFS.pdb`). Prove that the value of `MinNibbleEntries` is admissible, and give two adjacent placements of the table of Figure 5.5 whose values with $c = 2$ differ by 3. The test checks that the max-compressed tables never exceed the exact ones, that `SparseEntries` gives the same values as `NibbleEntries`, and that a corrupted file is rejected.

Exercise 5.11 (`ctest -R PBeFS.abstraction`). Run Algorithm 5.7 by hand on the space of the states $0, \dots, n-1$ with the goal 0 and the edges between r and $r+1$ of cost $1 + (r \bmod c)$, for $c = 2$ and $n = 8$: list the layers D_k and the entries lowered at each of them. Then show, by Theorem 5.15 (ii), that the path of 300 states with unit costs is refused, and that so is a path of 255 states, whose largest distance is 254. The test checks the distances against the algorithm of Dijkstra on a space of 10^5 states with chords and unreachable states, and the refusal of the path of 300.

Exercise 5.12 (`ctest -R PBeFS.abstraction`). Prove that the bound of Theorem 5.15 (iii) on the entries read is attained, up to a constant, by a ranking that puts the entries of each distance in every block, and give the bound for a ranking that puts them in consecutive ranks. Show by an example that the saturation of Lemma 5.16 may make two adjacent abstract states, whose distances differ, take the same value. The test checks that the tables of 4 bits are the tables of 8 bits saturated at 15, their save and load, the refusal of another key, and that a table of the cache is built once.

Exercise 5.13 (`ctest -R PBeFS.permutation`). Unrank 2 by Algorithm 5.9 with $n = 4$ and $k = 2$, compare the result with the lexicographic rank of the same 2-permutation, and conclude that the ranks of Myrvold and Ruskey are not in lexicographic order. Then rank $(4, 0, 7)$ on 9 items by Algorithm 5.8, and check that it gives 229 as in Figure 5.4. The test checks the three rankings as bijections for $n = 1, 4, 7$ and every k, the lexicographic order of the first two, and the agreement of the lexicographic rankings on 64 items.

Exercise 5.14 (`ctest -R PBeFS.pancake`). Prove Proposition 5.21 in detail, and give a stack of 7 pancakes, other than the goal, on which GAP-2 is 0. Is GAP-0 an abstraction heuristic in the sense of Definition 5.1? The test checks on the 5040 stacks of 7 pancakes that GAP-0 and GAP-2 are consistent and that the domain computes them incrementally, and A* and IDA* against the exact distances.

Exercise 5.15 (`ctest -R PBeFS.pancake`). Compute the number of the entries of the table of 3 of the 5 burned pancakes, and show that its abstract space is connected. Then show that the maximum of the gap heuristic and of a table is consistent, while the sum of the tables of $\{0, 1, 2, 3\}$ and of $\{3, 4, 5, 6\}$ of 7 pancakes (which share an object) or of $\{0, 1, 2\}$ and $\{3, 4, 5\}$ (which do not) may exceed h^* (consider the stack in decreasing order), since Proposition 5.20 applies to neither. The test checks the tables of 4 of 7 pancakes, with two rankings, and of 3 of 5 burned ones against the exact distances, and the consistency of the maximum.

Exercise 5.16 (`ctest -R PBeFS.topspin`). Prove Proposition 5.23, and show that the ring in decreasing order has no breakpoint; is it reachable in TopSpin with 8 tokens and a turnstile of 4? Explain why the table of a pattern of TopSpin is built from the N goals at once, and what its value would be if only the identity were a goal. The test checks the breakpoints and a table of 5 tokens on the 40320 rings of TopSpin with 8 tokens, and A* and IDA* with them.

Exercise 5.17 (`ctest -R PBeFS.rubik-pdb`). Derive the numbers of the entries of the three tables of Korf, and show by the state after one quarter turn from the goal that the sum of the table of the corners and of a table of the edges is not admissible. Why does the half turn about the axis of L and R give the two tables of the edges the same mean? The test checks the sizes, the largest values and the means of the tables against [59], in an optimized build.

Exercise 5.18 (`ctest -R PBeFS.hanoi`). Prove that `hanoi_tops` returns the top disk of each peg, and Proposition 5.24. Show that the sum of the tables of two ranges of disks that are not consecutive (e.g., the disks 0 and 2 in one table and 1 in the other) is still admissible. The test checks, on the 4^8 states of 8 disks, that the table of all the disks is exact and that the tables of 3 and 5 disks are additive and consistent.

6. Bounded-suboptimal, anytime and incremental search

[Source: include/PBeFS/astar/Weight.h (Weight, UnitWeight, finish_solution, settle_improvements), the fields weight, drop_duplicates and anytime of SearchOptions and the fields bound and improvements of Solution (Solution.h), AnytimeAStar.h, OptimisticSearch.h, ImprovedOptimisticSearch.h, FocalSearch.h, PotentialSearch.h, BatchAStar.h, LPAStar.h, HDLPAStar.h, DynamicGraph.h, the events of Domain.h and of Grid.h, and the weighted forms of the searches of AStar.h, IDAStar.h, SPAStar.h, MQAStar.h, HDAStar.h, HDAStarMPI.h, FarmAStar.h and SafePBNF.h]

> **Status — beta.** The code that this chapter describes is new: it is covered by the cases weighted, anytime, focal, lpa, batch, bounded-* and hdlpa-* of the tests, but it has not been through a release yet, and the benchmark of section 6.10 is still to be run.

We are concerned in this chapter with three families of searches that relax one of the assumptions of A* (Chapter 2). The first gives up the optimality of the path for a bound on its cost: a search is *bounded-suboptimal* if, given a weight $w \geq 1$, it returns a path that costs at most w times the optimum C^*, usually after far fewer expansions than A* needs to prove a path optimal (the classical member of the family is the weighted A* of Pohl [46], which orders OPEN by $g + wh$). The second family, the *anytime* searches, returns a first path quickly and then improves it, together with a lower bound on C^*, until the two meet or the search is stopped; the ratio of the incumbent to the lower bound bounds the suboptimality at every moment. The third family gives up the assumption that the graph is fixed: an *incremental* search, after the costs of some edges have changed, repairs its previous search instead of starting anew, which is the setting of the replanning of a robot that discovers its terrain as it moves [54], [143].

The three families share the machinery of this chapter, i.e., the class Weight, which states once the rules by which a weighted search orders OPEN, prunes with the incumbent and reopens a closed node, and the fields of Solution that report the lower bound and the sequence of the paths found; and the same rules make the parallel searches of Chapter 8, Chapter 9 and Chapter 11 bounded-suboptimal and anytime. We remark that the classical argument for weighted A* relies on the order of the expansions by $g + wh$, which no parallel search follows; the rules of Weight are therefore stated, and proved, for any order of expansion.

The structure of the chapter is as follows. Section 6.1 states the problem and the interface common to the searches. Section 6.2 gives the class Weight and proves that its rules keep the bound wC^* in any order of expansion, with and without the dropping of duplicates, and section 6.3 applies them to the sequential A* and IDA*. Section 6.4 describes the weighted and anytime parallel searches. Section 6.5 is devoted to the anytime searches AWA* and ARA* and to the anytime restarting PBNF, section 6.6 to the searches that follow a second order besides f (the optimistic searches, the focal searches A^*_ϵ and EES and the dynamic potential search), and section 6.7 to the batch weighted A*. Sections 6.8 and 6.9 are concerned with the incremental searches, LPA* and D* Lite and their hash-distributed version, whose correctness we prove in full. Sections 6.10 to 6.12 give the benchmark, the references and the exercises. The notation is that of Chapter 1 and Chapter 2: $g^*(u)$ is the cost of a cheapest path from s_0 to u, C^* the cost of an optimal path to a goal (∞ if there is none), $f = g + h$, and z the *incumbent*, i.e., the cost of the best path found so far (∞ if none).

6.1 The problem and the interface

We consider the problem of Chapter 1: a Domain gives the initial state s_0, the successors of a state with the costs of the edges, the goal test and a heuristic h, which is admissible unless we say otherwise (and then $h(t) = 0$

on every goal t). The costs are nonnegative, and positive where a statement needs it.

Definition 6.1 (Bounded suboptimality). Let $w \geq 1$. A search is *w-admissible* if, whenever a goal is reachable from s_0, it returns a path to a goal of cost at most wC^*. The *key* of a node n in a weighted search is $f'(n) = g(n) + wh(n)$, and a lower bound L on C^* is *proven* by a search if $L \leq C^*$ follows from what the search has done; the ratio c/L of the cost c of a path to a proven lower bound bounds the suboptimality of that path.

Since h is admissible, $f'(n) = g(n) + wh(n) \leq w(g(n) + h(n)) = wf(n)$ for every node, with equality when $h(n) = 0$, e.g., on a goal; this inequality is the one that all the proofs of this chapter use. The weight is given to a search by the options that all the searches share (`Solution.h`),

```
struct SearchOptions {

 /// stop with Status::node_limit after storing this many nodes (0 = no
 /// limit); IDAStar bounds the expansions instead and DDDAStar the states
 /// written to its closed files, and in a parallel search the limit holds
 /// for each worker
 std::size_t node_limit = 0;

 /// the number of nodes the search is expected to store (0 = unknown): the
 /// closed list is sized for them at the start, instead of being doubled
 /// (and rehashed) as it fills, by the searches with a closed list of their
 /// own that read it (AStar, the bidirectional and the greedy searches and
 /// most of the others)
 std::size_t expected_nodes = 0;

 /// the weight \f$ w \geq 1 \f$ of the heuristic: OPEN is ordered by
 /// \f$ f' = g + w h \f$, and the path found costs at most \f$ w C^* \f$
 /// [see Weight]; 1 is A*; the greedy searches take any \f$ w \geq 0 \f$,
 /// infinite (h alone) by default [see GreedyOptions]
 double weight = 1;

 /// with a weight above 1, a closed node reached through a cheaper path is
 /// reopened only if the saving exceeds what the weight allows, instead of
 /// always, which keeps the bound \f$ w C^* \f$ with a consistent
 /// heuristic [see Weight::improves]
 bool drop_duplicates = false;

 /// with a weight above 1, the search goes on after the first goal, the
 /// incumbent pruning the nodes by \f$ f \f$ instead of \f$ f' \f$, until
 /// it proves the incumbent optimal, recording each better path found [see
 /// Solution::improvements]
 bool anytime = false;
 /// if true, the sequential A* keeps the cheapest goal generated so far
 /// and stops as soon as no entry of OPEN has a smaller f [see AStar], and
 /// the greedy searches with eager evaluation return a goal when it is
 /// generated [see GreedyOptions]; by default the goal is tested only when
 /// a node is selected
 bool goal_at_generation = false;

 };  // end( struct( SearchOptions ) )
```

whose fields of this chapter are the following:

- `weight`, the weight $w \geq 1$ (1 by default, i.e., A*); the searches that take it throw `std::invalid_argumen t` if it is below 1, and those that cannot use it (`LPAStar`, `HDLPAStar`) if it is not 1. The greedy searches take any $w \geq 0$, with a meaning of their own (Chapter 4);
- `drop_duplicates`, which, with $w > 1$, keeps closed a node reached through a cheaper path unless the saving exceeds what the weight allows (section 6.2);
- `anytime`, which, with $w > 1$, makes a search go on after its first goal until it proves the incumbent optimal, recording each path that lowers it; `Weight` refuses `anytime` together with `drop_duplicates`, since a duplicate dropped may hold the only optimal path;
- `goal_at_generation`, which makes the sequential A* keep the cheapest goal generated and stop as soon as no entry of OPEN has a smaller key (section 6.3).

The searches return a `Solution` (`Solution.h`), whose fields of this chapter are

```
struct Improvement {
 Cost cost;                              ///< the cost of the path
 Cost bound;                             ///< the lower bound on C*
 std::uint64_t expanded;                 ///< the expansions so far
 double seconds;                         ///< the time since the start
 };

Status status = Status::unsolvable;      ///< how the search ended
Cost cost{};                             ///< the cost of the path
Cost bound{};                            ///< a lower bound on C*
std::vector< State > path;               ///< the states along the path
std::vector< Improvement > improvements;  ///< the paths of an anytime
                                           ///< search, by time
SearchStats stats;                       ///< the counters of the search
```

The field `bound` is a proven lower bound on C^*: the cost itself when the path is optimal, and otherwise the bound that the weight gives (Corollary 6.6) or a better one that the search has proven, so that `cost / bound` bounds the suboptimality of the path. The field `improvements` lists the paths of an anytime search in the order in which they were found, each with its cost, the lower bound known at that moment, the expansions done so far (by the worker that found it, in a parallel search) and the time since the start; the costs decrease strictly, and the last one is the cost of the path returned. A search that reaches its node limit returns `Status::node_limit`; an anytime search then keeps the best path found, with the last lower bound it knows, while the other searches return no path. The counters of `SearchStats` that this chapter adds are `goals_generated` (the goals that improved the incumbent of `goal_at_generation` when generated), `iterations` (the searches over the whole graph, e.g., the iterations of ARA*), and the counters of the incremental searches, which we give in sections 6.8 and 6.9.

6.2 The weight and the rules of a weighted search

A weighted search orders OPEN by $f' = g + wh$ in place of $f = g + h$, and the incumbent z prunes the nodes that cannot lead to a path the search still looks for. Three rules decide what a search does with a node, and the class template `Weight< Cost >` (`Weight.h`) states each of them once, so that every search of the library applies the same ones: the *end*, i.e., the condition under which no node of OPEN can lead to a path that the search still wants; the *pruning*, i.e., the condition under which the incumbent prunes a node; and the *reopening*, i.e., the condition under which a node reached through a cheaper path takes the new path. The rules depend on the mode of the search, as follows.

mode	key	end when no node of OPEN has	pruned if	a closed node is reopened if
A* ($w = 1$)	f	$f < z$	$f \geq z$	$g' < g(v)$
weighted	f'	$f' < z$	$wf \geq z$	$g' < g(v)$
duplicate dropping	f'	$f' < z$	$f' \geq z$	$g(v) - g' > (w-1)(h(u) - h(v))$
anytime	f'	(OPEN empty)	$f \geq z$	$g' < g(v)$

Here v is reached from u through a path of cost g'. Note that the end is tested on f' and the pruning on $wf \geq f'$: a node may stay in OPEN and yet never be expanded, which is why a worker of HDA* drops such a node when it reaches the top of its OPEN, rather than stopping there. With $w = 1$ the rules of the three weighted modes are those of the optimal searches of Chapter 8, and `Weight` turns the modes off (a search with the weight 1 is never anytime and drops no duplicate).

The interface of Weight

`Weight` is built from the weight and the two modes, or from the `SearchOptions`; we quote its declarations (the bodies are in `Weight.h:104-302`):

```
template< class Cost > class Weight {
 public:
  explicit Weight( double w = 1 , bool anytime = false , bool drop = false );
  explicit Weight( const SearchOptions & o );
  Cost key( Cost g , Cost h ) const noexcept;          // b g + a h, or g + w h
  Cost key_weighted( Cost g , Cost h ) const noexcept; // key() with w > 1
  Cost scale( Cost z ) const noexcept;                 // b z, infinity kept
  bool live( Cost k , Cost z ) const noexcept;         // k < b z, unless anytime
  bool prunes( Cost g , Cost h , Cost z ) const noexcept;
  bool prunes_weighted( Cost g , Cost h , Cost z ) const noexcept;
  template< class HU > bool improves( Cost g_old , Cost g , bool closed ,
                                      HU && h_u , Cost h_v ) const;
  Cost times( Cost f ) const noexcept;                 // floor( w f )
  Cost proven( Cost c ) const noexcept;
  Cost lower_bound( Cost c ) const noexcept;           // ceil( b c / a )
  Cost key_lower_bound( Cost k ) const noexcept;       // ceil( k / a )
  double value( void ) const noexcept;
  bool unit( void ) const noexcept;
  bool anytime( void ) const noexcept;
  bool drops( void ) const noexcept;
  };
```

With real costs, `key( g , h )` is $g + wh$ and the other members compare the values that their names say. With integer costs, a key $g + wh$ would have to be rounded, and a rounding could change the order of two nodes or the outcome of a test; the weight is therefore the exact fraction $w = a/b$ with the least $b \leq 10^4$ such that wb is an integer (to a relative tolerance of 10^{-9}), or $\lfloor 10^4 w \rfloor / 10^4$ if there is none (`Weight.h:114-125`), and the key of a node is the integer $bg + ah$. The weight that the search uses is `value()` $= a/b$, which is the one given when it is a fraction with a denominator up to 10^4 (e.g., 1.5 is 3/2 and 1.1 is 11/10), and otherwise slightly smaller than it, so that the bound wC^* of the given weight still holds. The keys have to fit in the cost type, which the library does not check.

Lemma 6.2 (The integer keys). Let the costs be integers, $w = a/b$, and let f', z and the costs be finite. Then (i) `key( g , h )` $= bf'$, so that the order of the keys is the order of f'; (ii) `live( k , z )` holds if and only if $f' < z$, `k` being the key of the node, unless the search is anytime, when it holds for every finite key; (iii) `prunes( g , h , z )` holds if and only if $f \geq z$ in the modes A* and anytime, $f' \geq z$ with duplicate dropping and $wf \geq z$ otherwise; (iv) `improves( g_old , g , closed , h_u , h_v )` holds if and only if $g < g_{\text{old}}$ and, when the node is closed and duplicates are dropped, $g_{\text{old}} - g > (w-1)(h(u) - h(v))$; (v) for an integer f', $f' \leq wf$ if and only if $f' \leq$ `times( f )`; (vi) if $c \leq wC^*$ then $C^* \geq$ `lower_bound( c )`; and (vii) if the key of some node of every optimal path, reached with $g = g^*$, is at least k, then $C^* \geq$ `key_lower_bound( k )`.

Proof. (i) $bg + ah = b(g + (a/b)h) = bf'$, and $b > 0$. (ii) `scale( z )` is bz, and $bf' < bz$ if and only if $f' < z$; an anytime search tests only that the key is finite (`Weight.h:173-175`). (iii) Without duplicate dropping the test is $a(g + h) \geq bz$ in 64-bit arithmetic, i.e., $(a/b)f \geq z$; with it, the key is compared with bz as in (ii); the other modes test $g + h \geq z$ (`Weight.h:181-202`). (iv) The test $b(g_{\text{old}} - g) > (a - b)(h(u) - h(v))$ is the stated one multiplied by $b > 0$, and the heuristic of u is asked for only in this case (`Weight.h:211-223`), since the parent may be far away (another worker, in HDA* over MPI). (v) `times( f )` is $\lfloor af/b \rfloor$, and an integer is at most wf if and only if it is at most its floor. (vi) $c \leq wC^* = aC^*/b$ gives $C^* \geq bc/a$, and since C^* is an integer, $C^* \geq \lceil bc/a \rceil$. (vii) A node of an optimal path with $g = g^*$ has the key $bg^* + ah \leq a(g^* + h) \leq aC^*$, since $b \leq a$ and h is admissible; hence $aC^* \geq k$, and $C^* \geq \lceil k/a \rceil$ as in (vi). □

The searches are written once over the rules, as templates of the weight: `UnitWeight< Cost >` (`Weight.h:316-346`) has the interface of `Weight` with the weight 1 known at compile time, its members being `static constexpr` and its rules those of A*, and each search runs its loop over `UnitWeight` when `unit()` holds and over `Weight` otherwise (e.g., `AStar.h:180-183`). Hence, the weight costs (almost) nothing to the optimal searches, which compile to the code they would have without it; the members `key_weighted()` and `prunes_weighted()` serve the same purpose inside `Weight`, keeping out of line what only a weight above 1 needs.

The bound in any order of expansion

The classical argument for weighted A* (Theorem 6.7 below) relies on the order of the expansions by f': when the first goal is selected, its key is the least of OPEN. A parallel search does not expand in this order, and its

first goal may be much worse than wC^*; we therefore state the rules in a form that holds in any order, with the notion of the entries in flight of §8.3 (an entry (u, γ) is in flight if it is in an OPEN list, held by a worker that is processing it, or in a message still to be received). A node is *pruned* when it is dropped without being expanded.

Proposition 6.3 (Weighted search in any order). Let h be admissible and $w \geq 1$, and let a search, sequential or parallel, be well-formed in the sense of §8.3, except that (b) and the selection may prune a node n when $wf(n) \geq z'$ for a value z' that z had at some earlier instant. If at an instant τ no entry in flight that is not stale has $f' < z$, then $z \leq wC^*$ at τ.

Proof. If no goal is reachable there is nothing to prove, C^* being infinite. Otherwise suppose that $z > wC^*$ at τ, and let $P = (s_0, \ldots, s_k)$ be an optimal path to a goal s_k with no goal before s_k, as in the proof of Theorem 8.6. Since the search reopens every node reached through a cheaper path, the remark of that proof holds unchanged: $g(u) \geq g^*(u)$ at every instant, the entry $(s_j, g^*(s_j))$ is created at most once, and once created it becomes stale only when this very entry is selected. Call the expansion of s_j *done* if a worker has selected the entry $(s_j, g^*(s_j))$ and finished processing it before τ, and let j be the least index whose expansion is not done ($j \leq k$, since a done expansion of s_k would have set $z \leq C^*$). If $j > 0$, the entry of s_{j-1} was selected and processed: had it been pruned, $z \leq z' \leq wf(s_{j-1}) = w(g^*(s_{j-1}) + h(s_{j-1})) \leq wC^*$ at τ, since z never increases; hence it was expanded, and s_j was either pruned, with $z \leq w(g^*(s_j) + h(s_j)) \leq wC^*$ in the same way, or given $g(s_j) = g^*(s_j)$ and the entry $(s_j, g^*(s_j))$. If $j = 0$, the entry $(s_0, 0)$ is the one created at the start. In every case not yet excluded this entry exists, its expansion is not done, and it is therefore in flight at τ and not stale, with the key

$$f'(s_j) = g^*(s_j) + wh(s_j) \leq w(g^*(s_j) + h(s_j)) \leq wC^* < z,$$

which contradicts the hypothesis. □

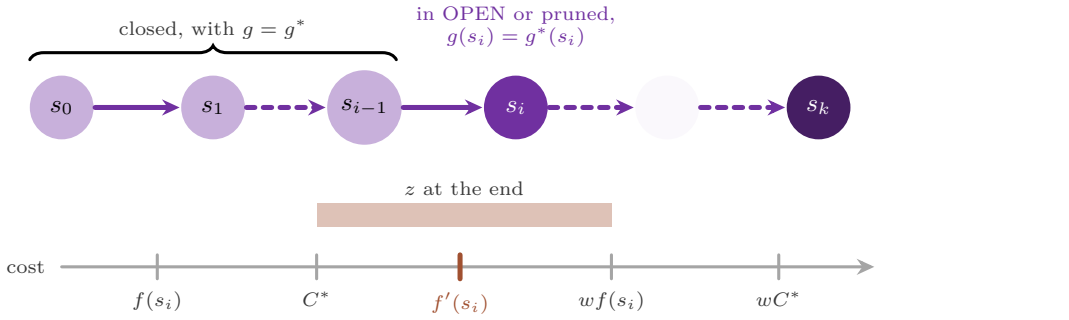


Figure 6.1. The bound wC^* of Proposition 6.3 drawn on an optimal path: the nodes before s_i have been expanded with their optimal g, and s_i, the first node that has not, is in flight with $g = g^*(s_i)$ or has been pruned. On the axis of the costs, the incumbent at the end lies between C^* and $wf(s_i) \leq wC^*$: below $f'(s_i)$ if s_i is still in OPEN, and below $wf(s_i)$ if it was pruned.

Figure 6.1 draws the proof. This is, in substance, Theorem 4 and Corollary 2 of Burns et al. ([3], section 5.1), which justify the pruning of weighted PBNF; we state it with the condition of the end, the one that a parallel search tests, and with the entries in flight, so that it covers at once all the searches of Chapter 8, Chapter 9 and Chapter 11. With $w = 1$ it is Theorem 8.6.

Duplicate dropping

A weighted search reaches a closed node through a cheaper path much more often than A* does, and reopening it costs expansions that rarely change the path found; the *duplicate dropping* variants keep such a node closed. In a sequential weighted A* with a consistent heuristic this keeps the bound (Theorem 1 of Likhachev, Gordon and Thrun [48]), but the argument relies on the order of the expansions, and a parallel search may close a node with a g much larger than g^* before the cheaper path arrives. The rule of `Weight` reopens a closed node v, reached from u with $g' < g(v)$, only when the saving exceeds a budget that the weight allows,

$$g(v) - g' > (w - 1)\big(h(u) - h(v)\big),$$

and it prunes by $f' \geq z$ instead of $wf \geq z$; with $w = 1$ it reopens each node reached through a cheaper path.

Proposition 6.4 (Duplicate dropping). Let h be consistent and $w \geq 1$, and let a search be as in Proposition

6.3, except that it reopens a closed node only as the rule above says and prunes a node n when $f'(n) \geq z'$. If at an instant τ no entry in flight that is not stale has $f' < z$, then $z \leq wC^*$ at τ, in any order of expansion.

Proof. Let $P = (s_0, \ldots, s_k)$ be as in the proof of Proposition 6.3, suppose $z > wC^*$ at τ, and write $D_i = \sum_{l<i} \max(0, h(s_l) - h(s_{l+1}))$ for the sum of the decreases of h along the prefix of P to s_i. By the consistency of h, each decrease is at most the cost of its edge, and hence $D_i \leq g^*(s_i)$. We first remark that a closed node keeps its g until it is reopened (the rule changes g only together with the reopening), and that g never increases. Let j be the least index such that s_j is not closed at τ with its last expansion finished, or $j = k+1$ if there is none. We claim that for every $i \leq \min(j, k)$, from the last expansion of s_{i-1} on (from the start, for $i = 0$),

$$g(s_i) \leq g^*(s_i) + (w-1)D_i,$$

unless $z \leq wC^*$. For $i = 0$, $g(s_0) = 0$. For $0 < i \leq \min(j, k)$, s_{i-1} is closed with its last expansion done, with a value $g(s_{i-1})$ that satisfies the claim, and that expansion offered $g' = g(s_{i-1}) + c(s_{i-1}, s_i)$ to s_i. Either it pruned s_i, with

$$z \leq g' + wh(s_i) \leq g^*(s_i) + (w-1)D_{i-1} + wh(s_i),$$

or s_i took the new path, or s_i already had $g(s_i) \leq g'$, or the saving $g(s_i) - g'$ that s_i refused is positive and at most the budget $(w-1)(h(s_{i-1}) - h(s_i))$; in the last three cases $g(s_i) \leq g' + (w-1)\max(0, h(s_{i-1}) - h(s_i)) \leq g^*(s_i) + (w-1)D_i$, and $g(s_i)$ only decreases afterwards. In the case of the pruning, $D_{i-1} \leq g^*(s_{i-1}) \leq g^*(s_i)$ gives $z \leq w(g^*(s_i) + h(s_i)) \leq wC^*$, against the assumption. If $j = k+1$, the goal s_k has been expanded, and the claim with $D_k \leq g^*(s_k) = C^*$ gives $z \leq g(s_k) \leq wC^*$, a contradiction. Otherwise the claim holds for s_j, which (as in Proposition 6.3) has an entry with its current g that is in flight and not stale, or was dropped at its selection by $f'(s_j) \geq z'$; in both cases

$$z \leq f'(s_j) = g(s_j) + wh(s_j) \leq g^*(s_j) + (w-1)g^*(s_j) + wh(s_j) = w\big(g^*(s_j) + h(s_j)\big) \leq wC^*,$$

a contradiction. □

The proof needs the consistency of h only to bound the sum D_j of the decreases of h along the prefix of the optimal path by $g^*(s_j)$; with an inconsistent heuristic this sum may be as large as j times C^*, and the bound may fail, as the following example shows.

Example 6.5 (Duplicate dropping with an inconsistent heuristic). Let $L \geq 1$ and $w = 2$, and consider the graph with the edges $s_0 \to a$, $a \to v$, $v \to b$ and $b \to x$ of cost 1, $x \to t$ of cost L, and the two shortcuts $s_0 \to v$ of cost $5 + L$ and $s_0 \to x$ of cost $8 + 2L$, the goal being t, and hence $C^* = 4 + L$ along s_0, a, v, b, x, t. The heuristic $h(a) = 3 + L$, $h(b) = 1 + L$ and $h = 0$ elsewhere is admissible (it is exact on a and b) but not consistent. Weighted A* with duplicate dropping expands s_0, then v with $g = 5 + L$ (key $5 + L$, below the key $7 + 2L$ of a), then a, which offers $g' = 2$ to the closed v: the saving $3 + L$ does not exceed the budget $(w-1)(h(a) - h(v)) = 3 + L$, and v keeps its path. Then x is expanded with key $8 + 2L$, before b, whose key is $6 + L + 2(1 + L) = 8 + 3L$, and the goal t is selected with $g = 8 + 3L > wC^* = 8 + 2L$ (it ties with b, and the ties go to the larger g; if b were expanded first, it would offer $7 + L$ to the closed x, a saving of $1 + L$ that the budget $1 + L$ refuses again, and the result would be the same).

We do not know of a published rule that keeps the bound wC^* with duplicate dropping and an inconsistent heuristic in any order of expansion; with such a heuristic the option `drop_duplicates` should not be used (the test `weighted` runs it on one inconsistent instance, that of the test `reopen` of Chapter 2, on which the bound happens to hold). With a consistent heuristic one has $h(u) - h(v) \leq c(u, v)$, and the budget is at most $(w-1)c(u, v)$, the one of the weak duplicate dropping of Burns et al. ([3], section 5.2, Theorem 5), which keeps the old path of v when $g(v) \leq g(u) + wc(u, v)$, i.e., when the saving is at most $(w-1)c(u, v)$; hence the rule of `Weight` reopens fewer nodes, and it needs only the heuristic of the parent, which each node carries (HDA* over MPI computes it again from the state of the parent, and only when a closed node improves). The weighted A* that never reopens a closed node (cf. [49], p. 357) is the special case of an infinite budget, which keeps the bound with a consistent heuristic in the order of f' (by the theorem of Likhachev et al. cited above), though possibly not in an arbitrary order. In both cases the least f of OPEN is no longer a lower bound on C^* (cf. Corollary 3 of [3]), since a node of an optimal path may stay closed with $g > g^*$; the bound that the search reports is then the one of the weight.

Corollary 6.6 (The proven bounds). A search of Proposition 6.3 or Proposition 6.4 that ends with the incumbent c proves the lower bound `proven( c )` on C^*, i.e., `lower_bound( c )` $= \lceil bc/a \rceil$ with integer costs and c/w with real ones, or c itself if it is anytime.

Proof. By the propositions, $c \leq wC^*$, and (vi) of Lemma 6.2 gives the bound. An anytime search prunes by $f \geq z$ and ends only when no entry is in flight, which is the hypothesis of Theorem 8.6, and hence $c = C^*$. □

Two functions of `Weight.h` fill a `Solution` from the best goal of a search in the same way for all the searches. `finish_solution( sol , goal , limit_hit , h0 , w , d )` sets `Status::node_limit` if the node limit was hit, and returns without a path unless the search is anytime; otherwise it sets `Status::solved` (or keeps `node_limit` for an anytime search, whose bound is then $h(s_0)$, the only one that a parallel search knows without the least f of every worker), the bound `proven( g )`, the path traced from the goal and the cost. The cost is the least of g of the goal and the cost of the path with the current parents (`path_cost()` of `Domain.h`), which may be smaller when the path to the goal has been improved after the goal was recorded; `settle_improvements( sol )` then appends to `improvements` an entry with that cost if it is below the last one recorded, so that the last improvement is always the cost returned.

6.3 Weighted A* and weighted IDA*

Weighted A*

`AStar` (Chapter 2) with a weight $w > 1$ is weighted A*: the keys of lines 13 and 16 are those of `Weight`, the test of line 14 is that of `Weight::improves()`, and with `SearchOptions::goal_at_generation` the search keeps the goal $\hat{u}$ of least g among those generated with $h = 0$ and stops at line 4' as soon as no entry of OPEN has a key below $g(\hat{u})$ on the scale of the keys (`AStar.h:216-226`). The pseudocode is that of `AStar.h`, with the keys f':

Algorithm 6.1 (Weighted A*).

```
 1  procedure A*(s0)
 2     CLOSED ← {s0}, with g(s0) ← 0 and parent(s0) ← nil
 3     OPEN ← {(f(s0), g(s0), s0)}
 4'    if û exists and no entry of OPEN has f < g(û) then
          return the path to û, of cost g(û)
 4     while OPEN ≠ ∅ do
 5        (f, g, u) ← the first entry of OPEN; remove it from OPEN
 6        if u is closed or g ≠ g(u) then continue      ▷ a stale entry
 7        close u
 8        if u is a goal then return the path to u, of cost g(u)
 9        for each successor v of u do
10           g' ← g(u) + c(u, v)
11           if v ∉ CLOSED then
12              add v to CLOSED, with g(v) ← g' and parent(v) ← u
13              push (g' + h(v), g', v) in OPEN
14           else if g' < g(v) then          ▷ a cheaper path: reopen v
15              g(v) ← g';  parent(v) ← u;  v is not closed
16              push (g' + h(v), g', v) in OPEN
17     return unsolvable
```

Here, with a weight, f stands for $f' = g + wh$ at lines 3, 4', 13 and 16, and line 14 is the test of reopening of section 6.2; line 4' is executed before line 5 at each iteration of the loop, when $\hat{u}$ exists, after the stale entries at the top of OPEN have been dropped. `AStar` refuses the anytime search, which is `AnytimeAStar` (section 6.5), by throwing `std::invalid_argument` from its constructor.

Theorem 6.7 (Weighted A*). Let h be admissible, $w \geq 1$, and let a goal be reachable. Then `AStar` with the weight w returns a path of cost at most wC^*, with and without `goal_at_generation`, and with `drop_duplicates` if h is consistent; its `bound` is `lower_bound()` of that cost.

Proof. Consider an iteration of the loop before the search returns. Without duplicate dropping, Lemma 2.7 holds unchanged (its proof does not depend on the keys) and gives, on an optimal path to a goal not yet closed with $g = C^*$, a node u' with $g(u') = g^*(u')$ and a valid entry in OPEN of key $f'(u') \leq w(g^*(u') + h(u')) \leq wC^*$. With duplicate dropping and a consistent h, the claim in the proof of Proposition 6.4, applied to the first node s_j of the optimal path that is not closed, gives a valid entry of key $f'(s_j) \leq wC^*$ (or a closed goal, and the search would have returned). Hence, the least key of OPEN is at most wC^* at each iteration. A goal t selected at line 5 has $h(t) = 0$, and its key $g(t)$ is the least of OPEN, so that $g(t) \leq wC^*$; at line 4', $g(\hat{u})$ is at most the least key of OPEN, i.e., at most wC^*, since the test failed for no smaller value. The search terminates as A* does (Theorem 2.11), and the bound is (vi) of Lemma 6.2. □

The order of the ties matters more with a weight than without, since many nodes share a key $g + wh$ while

their f differ. The OPEN lists of the library break the ties of the key in favour of the larger g (Chapter 3); `LowGOpenList< Base >` (`OpenList.h`) wraps a `HeapOpenList` and breaks them in favour of the smaller g, i.e., among the nodes of equal $f' = g + wh$, of the smaller f, since $f = f' - (w-1)h$ and, among the entries of equal key, the one of smaller g has the larger h. It pushes each entry with the g mirrored, $M - g$ with integer costs (M half the largest value of the type) and $-g$ with real ones, both exact, and mirrors it back in the entries it returns, so that the test of staleness is unchanged. On grids this order usually expands fewer nodes and finds cheaper paths, while the larger g is better on the sliding-tile puzzles; it is chosen by `OpenKind::heap_low_g` in `astar()`, `anytime_astar()` and `optimistic_search()`. A `BucketOpenList` cannot be its base, since it keeps its layers dense in g.

Weighted IDA*

`IDAStar` (Chapter 4) with a weight puts its thresholds on f': each iteration explores the paths whose nodes have a key at most the bound, the first bound being the key of s_0 and the next one the least key cut, and the root is searched only if its own key is within the bound (`IDAStar.h:185-234`). The option `drop_duplicates` does nothing, IDA* having no duplicate detection, and the constructor refuses `anytime`. With `IDAOptions::growth` above 1, the bounds are those of IDA*-CR of Sarkar et al. [132], chosen by the `CutoffHistogram` of `Deepening.h` from the histogram of the keys cut (the `DeepeningOptions` of Chapter 4), and the iteration that finds the first goal with a bound above the least key cut before is completed as a depth-first branch-and-bound if `IDAOptions::optimal` (the default), the incumbent z pruning the nodes of $wf \geq z$.

Proposition 6.8 (Weighted IDA*). Let h be admissible, $w \geq 1$, the costs positive, and let a goal be reachable. (i) With the classical bounds, `IDAStar` returns a path of cost at most wC^*, with `bound` equal to `lower_bound()` of its cost. (ii) With the bounds of IDA*-CR and `optimal`, the same holds. (iii) With the bounds of IDA*-CR and not `optimal`, the path need not be within wC^*, and `bound` is $\min(c, \lceil \ell/a \rceil)$ (ℓ/w with real costs), c being the cost of the path and ℓ the least key cut by the iteration before.

Proof. Every node of an optimal path P, reached along P, has the key $bg^* + ah \leq aC^*$ (as in (vii) of Lemma 6.2); write $K =$ max of the keys along P, and hence $K \leq aC^*$. (i) Let β_k be the bound of the iteration k. If the iteration k finds no goal, some node of P has a key above β_k (otherwise P would be explored and its goal found), the first such node is generated and cut, and therefore $\beta_{k+1} \leq K$. Hence, either a goal is found in an iteration whose bound is at most K, or the bounds stay at most K and increase strictly, which, the keys being of finitely many values below K when the costs are positive (integers, or bounded away from zero), cannot go on for ever. A goal t found with the bound $\beta \leq K$ has $b\,g(t) \leq \beta \leq aC^*$, i.e., $g(t) \leq wC^*$. (ii) The iteration of the branch-and-bound explores every path whose keys are at most its bound β, pruning the nodes of $wf \geq z$. If $K \leq \beta$, either P is explored and its goal found, and hence $z \leq C^*$, or a node n of P is pruned, with $z \leq wf(n) \leq wC^*$. If $K > \beta$, the goal t of the incumbent has $b\,g(t) \leq \beta < K \leq aC^*$. In both cases $z \leq wC^*$. (iii) Every path to a goal has a node of key at least ℓ, since the iteration before explored all the paths whose keys are at most its bound and found no goal; (vii) of Lemma 6.2 gives $C^* \geq \lceil \ell/a \rceil$, and $c \geq C^*$. The path itself may cost up to the new bound, which IDA*-CR may set far above K (cf. the bound $\beta_{k+1} - \ell_k$ of `Deepening.h` on the excess of the first path). □

With a weight, an iteration of IDA* cuts the nodes whose key exceeds the bound and therefore usually explores far fewer nodes than one on f with the same bound; however, since the keys $g + wh$ of the nodes are more spread out than their f, the classical bounds advance by smaller steps relative to the size of the iterations, which is one more reason to use the bounds of IDA*-CR with a weight. The test `weighted` runs the weighted IDA* with $w \in \{1, 1.5, 2, 5\}$ on the 8-puzzle, Romania and the inconsistent instance of the test `reopen` (and on instances of Korf with $w \geq 2$).

6.4 The weighted and anytime parallel searches

The parallel searches of Chapter 8, Chapter 9 and Chapter 11 read the weight and the two modes of their options (`ParallelOptions` and `PBNFOptions` derive from `SearchOptions`), build a `Weight` from them, and apply its rules at the lines of their pseudocode where the optimal search tests f against z; the order of the expansions changes, and the end of the search is detected exactly as before, since the rules change only which entries count as live. In detail:

- `SPAStar` (§8.4) orders its OPEN by f', tests at line 7 for a node of $f' < z$, does not expand at line 14 a node that z prunes, and prunes at lines 20 and 22 by the rule of `Weight` (`SPAStar.h`);
- `MQAStar` (§8.6) drops at line 12 an entry popped with $f' \geq z$ or pruned by z, and expands as `SPAStar` (`MQAStar.h`);

- `FarmAStar` (§8.7) changes the tests of the master at lines 5, 8, 10 and 12 and the reopening of INSERT (`FarmAStar.h`);
- `HDAStar` (§9.2) orders each OPEN by f', asks at line 6 for a node of $f' < z$, drops at line 7 the nodes that z prunes (the loop of `HDAStar.h:403-424`), and prunes at lines 12 and 20 by $wf \geq z$ (by $f' \geq z$ with duplicate dropping, whose test of reopening replaces that of line 23); `HDAStarMPI` (§9.6) changes lines 8 and 9 of its pseudocode in the same way, computing the heuristic of the parent again from its state when the test of reopening needs it;
- `SafePBNF` with a weight is the weighted PBNF (wPBNF) of section 6 of Burns et al. [3]: the OPEN lists, the values $f(b)$ of the nblocks and the free list are ordered by f', line 8 empties OPEN(b) when its first node has $f' \geq z$ and skips a node that z prunes, line 11 prunes the successors, line 12 reopens by the rule of `Weight`, and lines 28, 30 and 60 compare $f(b)$, now the least f' of the nblock, with z (`SafePBNF.h`, and Algorithm 11.1 of §11.5);
- `PASE` (§8.8) does not use `Weight`: its weight orders OPEN by $g + wh$ (wPA*SE), and its suboptimality is that of the inflation $\varepsilon_p \geq w$ of its rule of independence, which the constructor raises to the weight; with integer costs the weight is the fraction $a/256$ with a rounded up, not that of `Weight`.

In the anytime mode, the same searches go on after the first goal: the incumbent prunes only by $f \geq z$, every node of OPEN is live (in SafePBNF, every nblock with a node stays free), and the search ends when no entry is left in flight, i.e., with an optimal incumbent; each goal that lowers z is appended to `improvements` under the lock of the incumbent, with the bound $h(s_0)$, since no worker knows the least f of the others. These are the anytime weighted HDA* and PBNF of Burns et al. ([3], section 7), and the anytime SPA*, MQA* and farm, which follow them.

Theorem 6.9 (The weighted parallel searches). Let h be admissible, $w \geq 1$, and let a goal be reachable. When `SPAStar`, `MQAStar`, `FarmAStar`, `HDAStar`, `HDAStarMPI` and `SafePBNF` end without reaching their node limit, (i) with the weight w the path costs at most wC^* and `bound` is `lower_bound()` of its cost; (ii) the same holds with `drop_duplicates` if h is consistent; and (iii) in the anytime mode the path is optimal, `bound` is its cost, and the costs of `improvements` decrease to it.

Proof. Each search is well-formed in the sense of §8.3 up to the pruning, as the proofs of its chapter show (§8.4 to §8.7, §9.3, §11.6), and its pruning is that of `Weight`, i.e., by $wf \geq z'$, by $f' \geq z'$ with duplicate dropping and by $f \geq z'$ in the anytime mode, z' being the value of the incumbent that the worker read. Each search stops at an instant at which no entry in flight that is not stale is live, which those proofs establish with "$f < z$" replaced by the test `live()` of `Weight`, since they do not use what the test is: the counter of MQA*, the lock of SPA* and of the graph of SafePBNF, the master of the farm, and the mailbox of HDA* detect that no live entry exists, whatever the test. Then (i) is Proposition 6.3, (ii) is Proposition 6.4, and in (iii) every entry is live, so that the search stops only when none is in flight and Theorem 8.6 gives $z = C^*$. The bounds are those of Corollary 6.6, set by `finish_solution()`, and the improvements decrease since each is recorded under the lock only when it lowers z; `settle_improvements()` appends the cost of the path when it is below the last one. □

We remark that the first path of a parallel search need not be within wC^*, nor its intermediate paths in the anytime mode, which carry only the bound $h(s_0)$: the guarantee is on the path returned at the end. The test `bounded-thread` (and `bounded-openmp`, `bounded-fastflow`) runs the weighted, duplicate-dropping and anytime forms of these searches with 1, 3 and 8 workers and $w \in \{1.5, 3\}$ on two instances of Korf, a grid with 8 moves and Romania, and the anytime restarting PBNF of section 6.5; the case `mpi` does the same for `HDAStarMPI`.

6.5 Anytime search

An anytime search returns a first path quickly and then improves it, each path being reported with the lower bound on C^* known when it is found, so that the search can be stopped at any time with a path and a certified ratio (Figure 6.2). The sequential anytime searches are in `AnytimeAStar.h`:

```
struct AnytimeOptions : public SearchOptions {

 /// the weights of the iterations of ARA*, decreasing, the iterations
 /// after the last one having the weight 1; empty for AWA*, which keeps the
 /// weight of the SearchOptions for the whole search
 std::vector< double > schedule;

 };  // end( struct( AnytimeOptions ) )
```

```
template< Domain D , class Open = /* buckets or heap, as AStar */ >
class AnytimeAStar {
 public:
  explicit AnytimeAStar( const D & domain , const AnytimeOptions & options
                         = {} );
  [[nodiscard]] SolutionT solve( void ) const;
  };
template< Domain D > [[nodiscard]] Solution< state_t< D > , cost_t< D > >
 anytime_astar( const D & domain , const AnytimeOptions & options = {} ,
                OpenKind open = OpenKind::automatic );
```

With an empty `schedule` the search is Anytime Weighted A* (AWA*) of Hansen and Zhou [47] with the weight of the options; with a schedule it is Anytime Repairing A* (ARA*) of Likhachev, Gordon and Thrun [48]. The options `drop_duplicates` and `anytime` are not read: ARA* has its own rule for the closed nodes, and AWA* reopens them to reach the optimum. The constructor throws `std::invalid_argument` if a weight is below 1 or the schedule does not decrease strictly. `anytime_astar()` chooses the OPEN list as `astar()` does, `OpenKind::heap_low_g` giving the `LowGOpenList` of section 6.3.

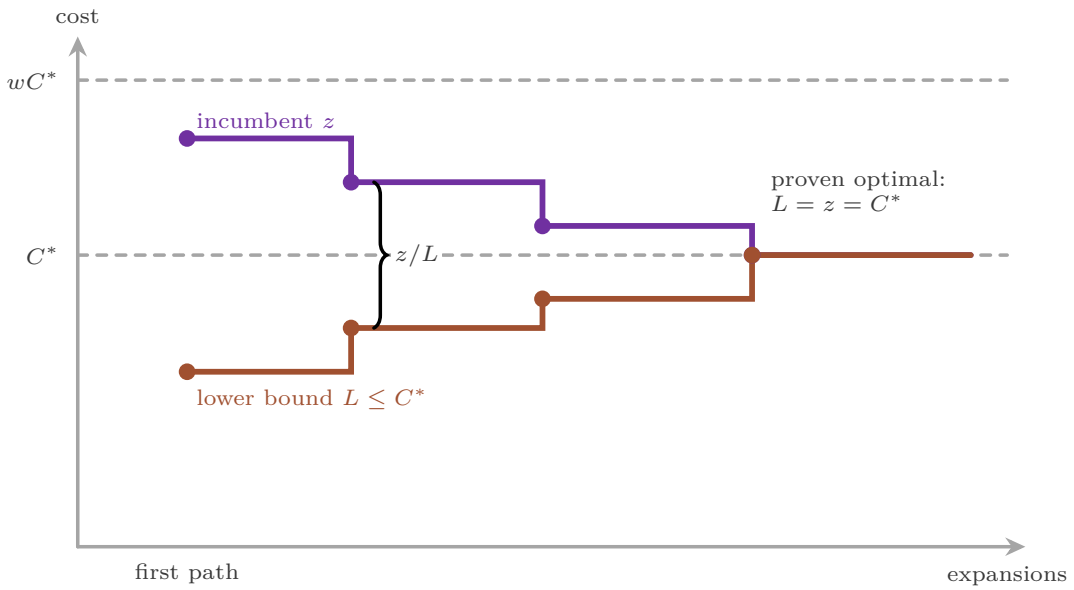


Figure 6.2. The paths of an anytime search over its expansions: the incumbent z (primary colour) falls as a staircase from the first path, which costs at most wC^*, while the lower bound $L \le C^*$ (accent colour) rises; at each moment z/L bounds the suboptimality of the incumbent, and the search ends when the two meet at C^*.

Anytime weighted A*

AWA* is weighted A* that does not stop at the first goal: the incumbent prunes by $f \ge z$, and the search ends when OPEN is empty. A second OPEN list FQ, ordered by f, holds the same nodes, its entries becoming stale with those of OPEN, and its least f is the lower bound reported with each path (`AnytimeAStar.h`):

Algorithm 6.2 (AWA*).

```
 1  procedure AWA*(s₀, w)
 2     z ← ∞;  push s₀ in OPEN (by f' = g + w h) and in FQ (by f)
 3     while OPEN ≠ ∅ do
 4        u ← the first node of OPEN, removed (stale entries dropped)
 5        if f(u) ≥ z then continue                          ▷ pruned
 6        close u
 7        if u is a goal then
 8           z ← g(u);  record (z, min(z, min f of FQ));  continue
 9        for each successor v of u with g(u) + c(u, v) + h(v) < z do
10           insert v as A* does (lines 10 to 16 of AStar.h), in OPEN
```

```
                and in FQ
11     return the path of cost z, optimal
```

The goal is tested when it is selected, as in `AStar`, whereas Algorithm 1 of Hansen and Zhou tests it when it is generated and does not put it in OPEN; we keep the test at the selection, so that the first goal of AWA* is the goal of weighted A*.

Theorem 6.10 (AWA*). Let h be admissible, $w \geq 1$ and the graph finite. Then AWA* (i) records first a path of cost at most wC^*, (ii) records with each path a lower bound $\min(z, \min_{\mathrm{FQ}} f) \leq C^*$, (iii) records paths of strictly decreasing costs, and (iv) ends with $z = C^*$, $\texttt{bound} = C^*$.

Proof. (i) Until the first goal, $z = \infty$, nothing is pruned, and the search is weighted A*, whose first goal costs at most wC^* by Theorem 6.7. (ii) Before a goal of cost C^* is selected, Lemma 2.7 (whose proof holds with the pruning, since a node of an optimal path with $g = g^*$ has $f \leq C^* < z$ and is never pruned while $z > C^*$) gives a node u' of an optimal path with $g(u') = g^*(u')$ and a valid entry in OPEN, hence in FQ, of $f \leq C^*$; if $z = C^*$ the bound is z. (iii) A goal selected has $f = g < z$ by line 5. (iv) The search is well-formed with the pruning by $f \geq z$ and ends when OPEN is empty; Theorem 8.6 with one worker gives $z = C^*$, and it ends on a finite graph since each node is reopened only when its g decreases, to one of finitely many values. □

The test `anytime` checks (i) to (iv) with $w \in \{1.5, 3\}$ on the 8-puzzle, three instances of Korf, three grids with 4 and 8 moves, Romania and the inconsistent instance of `reopen`. The second list doubles the entries; a min-max heap would hold both orders in one, at the price of a slower OPEN.

Anytime repairing A*

ARA* runs weighted A* with a decreasing schedule $w_0 > w_1 > \cdots \geq 1$ and reuses the work of each iteration in the next. Within an iteration no node is expanded twice: a closed node whose g decreases goes to the list INCONS instead of OPEN, and at the end of the iteration OPEN ∪ INCONS, ordered by the next weight, is the OPEN of the next one, CLOSED being emptied; the goal is tested when it is generated, and z is its g (`AnytimeAStar.h`):

Algorithm 6.3 (ARA*).

```
 1  procedure IMPROVEPATH(w)
 2     while OPEN has a node of f' < z do
 3        u ← the first node of OPEN, removed (stale entries dropped)
 4        if f(u) ≥ z then continue                          ▷ pruned
 5        close u
 6        for each successor v of u with g' = g(u) + c(u, v), g' + h(v) < z
 7           if v is new or g' < g(v) then
 8              g(v) ← g';  parent(v) ← u
 9              if v is a goal then z ← g', recording v
10              else if v is closed then add v to INCONS
11              else push (g' + w h(v), g', v) in OPEN
12  procedure ARA*(s0, w0, w1, ...)
13     z ← ∞;  OPEN ← {s0};  i ← 0
14     loop
15        IMPROVEPATH(w_i)
16        L ← min(z, min f over OPEN ∪ INCONS)
17        if z decreased in IMPROVEPATH then record (z, L)
18        if L ≥ z or OPEN ∪ INCONS = ∅ then return
19        i ← i + 1 (w_i = 1 once the schedule ends)
20        OPEN ← OPEN ∪ INCONS, by g + w_i h;  INCONS ← ∅;  CLOSED ← ∅
```

Since the OPEN lists of the library have no decrease-key, OPEN ∪ INCONS is formed at line 16 by emptying OPEN, whose live entries are then pushed again with the keys of the next weight, so that no second queue is needed (Figure 6.3); the nodes closed in the iteration are kept in a list, so that line 20 opens them in time proportional to their number. A root that is a goal ends the search at once with $z = 0$.

Lemma 6.11 (The bound of ARA*). Let h be admissible. At line 16, $L \leq C^*$.

Proof. If no goal is reachable there is nothing to prove. Otherwise let $P = (s_0, \ldots, s_k)$ be an optimal path to a goal, and suppose that $z > C^*$ (otherwise $L \leq z \leq C^*$). Note that g never increases and is always the cost of a

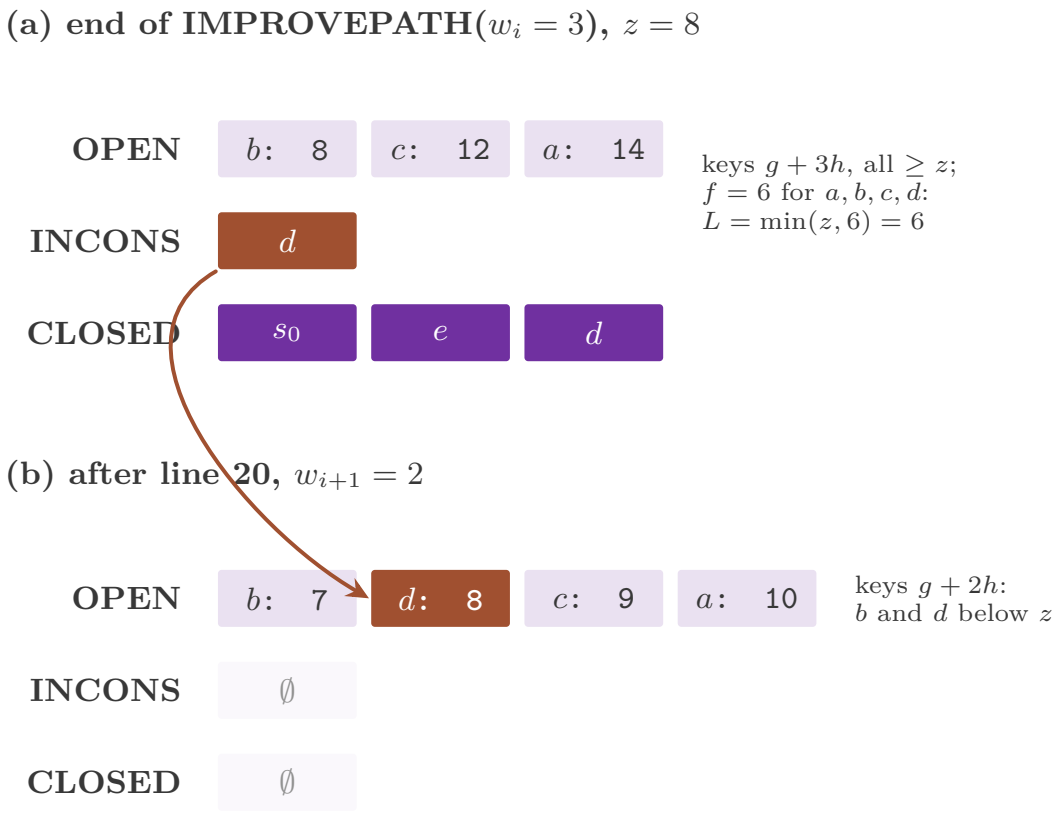


Figure 6.3. The lists of ARA* around line 20, with the nodes a, b, c, d of $(g, h) = (2, 4)$, $(5, 1)$, $(3, 3)$, $(4, 2)$ and the incumbent $z = 8$. (a) At the end of IMPROVEPATH(3) every key $g + 3h$ of OPEN is at least z, the closed node d, whose g decreased after its expansion, is in INCONS, and every node has $f = 6$, and hence $L = 6$ and $z/L = 4/3$. (b) Line 20 merges INCONS into OPEN with the keys $g + 2h$ of the next weight and empties INCONS and CLOSED; b and d are now below z.

path, so that once a node has $g = g^*$ it keeps it. Let j be the least index such that s_j has never been expanded with $g(s_j) = g^*(s_j)$ ($j \le k$, since the goal is never expanded). If $j = 0$, s_0 is in OPEN with $g = 0$ (it is expanded in the first iteration otherwise, the root not being a goal). If $j > 0$, the expansion of s_{j-1} with $g^*(s_{j-1})$ offered $g^*(s_j)$ to s_j, which was not pruned (the test of line 6 would give $z \le g^*(s_j) + h(s_j) \le C^*$, and z only decreases) and is not a goal before s_k; hence s_j got $g = g^*(s_j)$ then or had it already, and when it got it, it was put in OPEN or, being closed, in INCONS (or it is the goal s_k, and $z \le C^*$). Since then, s_j has not been expanded; an entry of OPEN with $g^*(s_j)$ is removed only by its selection, which either expands it (excluded) or prunes it by $f(s_j) \ge z$ (excluded, as above); and line 20 moves OPEN and INCONS together into the next OPEN. Hence, at line 16, s_j is in OPEN $\cup$ INCONS with $f(s_j) = g^*(s_j) + h(s_j) \le C^*$, and $L \le C^*$. □

Theorem 6.12 (ARA*). Let h be admissible and the graph finite. Then ARA* terminates, each path is recorded with a bound $L \le C^*$, the costs recorded decrease strictly, and the search ends with $z = C^*$ and `bound` $= C^*$. If h is consistent, the path of the iteration with the weight w_i costs at most $w_i C^*$.

Proof. The bound is Lemma 6.11, and line 17 records only a z that decreased. At line 18, $L \ge z$ gives $z \le L \le C^*$, i.e., $z = C^*$; an empty OPEN $\cup$ INCONS gives $L = z$ and the same conclusion (or no goal, $z = \infty = C^*$). Each iteration ends, since it expands each node at most once (a closed node goes to INCONS, not to OPEN). Once the schedule is exhausted, the iterations have the weight 1; an iteration that does not end the search leaves a node of $f < z$ in OPEN $\cup$ INCONS, which the next iteration, with the weight 1, either expands or prunes, and every expansion of the next iterations either lowers some g (finitely often on a finite graph, each g being the cost of one of finitely many simple paths) or empties OPEN; hence the iterations are finitely many. The last claim is Theorem 1 of Likhachev et al. [48], which holds for each call of IMPROVEPATH with a consistent heuristic, by the order of the expansions. □

The ratio z/L of line 17 is the suboptimality ϵ' that Likhachev et al. report with each path ([48], section 3); we state its bound with any admissible heuristic, while the bound $w_i C^*$ of each iteration needs a consistent one. The test `anytime` runs ARA* with the schedule $(w, 1 + (w-1)/2, 1 + (w-1)/4)$ on the instances of AWA* and checks that the first path is within wC^* and the last one optimal, that both searches keep the best path found at the node limit, and that a schedule that increases is refused.

Anytime restarting PBNF

With a nonempty `PBNFOptions::schedule` (the options of Chapter 11), `SafePBNF` is the Anytime Restarting PBNF (ARPBNF) of Burns et al. ([3], section 7.1), i.e., ARA* on the nblocks: each iteration is a wPBNF with the weight of the schedule, which prunes by $f \ge z$ only and keeps the nodes of $f' \ge z$ in OPEN for the next iterations, a closed node whose g decreases goes to the list INCONS of its nblock, and the iteration ends at line 32 of Algorithm 11.2 (§11.5), when no nblock has a node of $f' < z$ and none is in use. Then the worker that finds no nblock in use, with the graph locked, orders OPEN $\cup$ INCONS of every nblock by the next weight, empties the CLOSED lists, records the incumbent with the bound $\min(z, \min f)$ over OPEN $\cup$ INCONS, and frees the nblocks again, until the bound reaches z (`SafePBNF.h:524-570`). The weights after the schedule are 1, and the constructor refuses a schedule that does not decrease strictly or has a weight below 1. In the paper the lists are ordered again at each new incumbent, with all the workers stopped by a protocol of their own; we do it at the end of an iteration, when no worker holds an nblock and the lock of the graph suffices. Burns et al. report that ARPBNF improves its paths more smoothly than the anytime weighted PBNF, but more slowly ([3], section 8.4).

Proposition 6.13 (ARPBNF). Let h be admissible and the graph finite. Then ARPBNF ends with $z = C^*$, each path being recorded with a bound at most C^*; at the node limit it keeps the best path found, with the last bound recorded.

Proof. At the end of an iteration no entry is in flight, every worker being outside an nblock, and the argument of Lemma 6.11 applies to the union of the lists of all the nblocks, with the pruning by $f \ge z$ of lines 8 and 11; the end and the termination follow as in Theorem 6.12, each iteration being a wPBNF that ends by Theorem 11.13 and stops as in the proof of Theorem 11.12 (with the rules of `Weight`, as in Theorem 6.9). At the node limit the search returns the path of its incumbent with the bound $\min(L, z)$ of the last iteration ended (`SafePBNF.h:330-337`). □

6.6 Searches with a second order

Weighted A* uses the weight both to choose the node to expand and to bound the cost, and it is therefore often more cautious than the bound requires (its paths usually cost much less than wC^*). The searches of this section

decouple the two: they keep a lower bound on C^*, usually the least f of OPEN, and choose the node to expand by another order, as long as the bound allows. All of them reopen the closed nodes reached through a cheaper path, so that the least f of OPEN is a lower bound on C^* by Lemma 2.7 (with any admissible heuristic), and all of them read the weight of the options as the bound w and ignore `anytime` and `drop_duplicates`.

The optimistic search

The optimistic search of Thayer and Ruml [49] searches with an optimistic weight w_o larger than the bound w (by default $2w - 1$, as in the paper), and then expands by f only as much as it needs to prove that the path found is within the bound (`OptimisticSearch.h`):

```
struct OptimisticOptions : public SearchOptions {

 /// the optimistic weight \f$ w_o \geq 1 \f$; 0 stands for \f$ 2 w - 1 \f$
 double optimism = 0;

 };  // end( struct( OptimisticOptions ) )
```

```
template< Domain D , class Open = /* buckets or heap, as AStar */ >
class OptimisticSearch {
 public:
  explicit OptimisticSearch( const D & domain ,
                             const OptimisticOptions & options = {} );
  [[nodiscard]] SolutionT solve( void ) const;
  };
template< Domain D > [[nodiscard]] Solution< state_t< D > , cost_t< D > >
 optimistic_search( const D & domain , const OptimisticOptions & options = {} ,
                    OpenKind open = OpenKind::automatic );
```

The search keeps the same nodes in OPEN, ordered by $f'_o = g + w_o h$, and in FQ, ordered by f:

Algorithm 6.4 (The optimistic search).

```
 1  procedure OPTIMISTIC(s₀, w, w_o)
 2     z ← ∞;  push s₀ in OPEN (by f'_o) and in FQ (by f)
 3     while FQ ≠ ∅ and w · (the least f of FQ) < z do
 4        if z = ∞ or OPEN has a node of f'_o < z then
 5           u ← the first node of OPEN, removed
 6        else u ← the first node of FQ, removed          ▷ the cleanup
 7        if f(u) ≥ z then continue                        ▷ pruned
 8        close u
 9        if u is a goal then z ← g(u), recording u;  continue
10        for each successor v of u with g(u) + c(u, v) + h(v) < z do
11           insert v as A* does, in OPEN and in FQ
12     return the path of cost z
```

The stale entries are dropped as they surface in either list, a node removed from one of them being closed, and therefore stale, in the other.

Proposition 6.14 (The optimistic search). Let h be admissible, $1 \leq w$ and $1 \leq w_o$, and let a goal be reachable on a finite graph. Then the optimistic search returns a path of cost at most wC^*, with `bound` $= \min(z, \min_{\mathrm{FQ}} f) \leq C^*$, whatever w_o; each path that lowers z is recorded in `improvements` with the least f of FQ at that moment.

Proof. As in the proof of Theorem 6.10, while $z > C^*$ a node of an optimal path with $g = g^*$ and $f \leq C^*$ has a valid entry in FQ (it is never pruned, since $f \leq C^* < z$, and it leaves FQ only by its expansion, which puts the next node of the path in its place), so that the least f of FQ is at most C^*; if $z \leq C^*$ the bound is z. The loop ends at line 3 either when FQ is empty, and then no node of $f < z$ is left and $z = C^*$ by Theorem 8.6 with one worker, or when $w \min_{\mathrm{FQ}} f \geq z$, which gives $z \leq w \min_{\mathrm{FQ}} f \leq wC^*$. It ends, since every iteration either removes a stale entry or closes a node with a new value of g, which happens finitely often. The order of line 4 changes neither argument, and w_o decides only how soon the bound is reached. □

The test of line 3 is `prunes( f , 0 , z )` of a `Weight` without modes, i.e., $w \min f \geq z$ exactly with integer costs. If the first path is already within the bound when it is found, the search ends without any cleanup; if not, the cleanup expands the nodes of least f until the least f of FQ rises to z/w, possibly finding cheaper goals on the way (the "optimistic" bet of the paper is that it seldom needs many).

The improved optimistic search

The Improved Optimistic Search (IOS) of Chen, Sturtevant, Doyle and Ruml [50] adds two ideas to the optimistic search: a lower bound gathered by the first search, and the *solution update*, which lowers the incumbent when a cheaper path to a node of the incumbent path is found, without expanding that path again. In `ImprovedOptimisticSearch.h`, with the options of `OptimisticSearch`, z the cost of the first path P, r the largest saving found on a node of P and L the lower bound:

Algorithm 6.5 (IOS).

```
 1  procedure IOS(s₀, w, w_o)
 2     L ← 0;  weighted A* by g + w_o h, no closed node reopened, each
           node selected raising L to (g + w_o h) / w_o, until a goal is
           selected:  z ← its g, P ← its path
 3     every node is closed;  r ← 0;  push s₀ in OPEN by f
 4     while OPEN ≠ ∅ and z − r > w L do
 5        u ← the first node of OPEN by f, removed;  L ← max(L, f(u))
 6        close u;  if u is a goal then z ← g(u);  r ← 0;  P ← its path;
              break
 7        for each successor v of u, with g' = g(u) + c(u, v) do
 8           if v ∈ P and g' < its g on P then r ← max(r, its g on P − g')
 9           if v was not reached since line 3 then
10              g(v) ← g';  parent(v) ← u;  push v in OPEN      ▷ reopened
11           else insert v as A* does
12     return P improved by r, of cost z − r
```

The path of line 12 is the one to the node of P with the saving r, followed by the rest of P, which is the solution-update policy of the paper; if no saving was found, P itself is returned, since the second search may have given its nodes other parents. Algorithm 1 of the paper goes on instead from the OPEN of the first search, ordered by f, with no reopening, and proves the bound also by its least f; we restart from the root because, when the closed nodes are not reopened, a node of an optimal path may stay with $g > g^*$, and the least f of OPEN is then not a lower bound on C^* (cf. Corollary 3 of [3]). The two phases are not interleaved, as in the configuration that the paper recommends.

Proposition 6.15 (IOS). Let h be consistent, $w, w_o \geq 1$, and let a goal be reachable on a finite graph. Then IOS returns a path of cost at most wC^*, with `bound` $= \min(L, z - r) \leq C^*$.

Proof. The first search is a weighted A* with the weight w_o and no reopening; with a consistent heuristic every node it selects before its goal has $g + w_o h \leq w_o C^*$ (Theorem 1 of Likhachev et al. [48], the first iteration of ARA*), so that the L of line 2 is at most C^*; with integer costs it is the key divided by a and rounded up, $\lceil (bg + ah)/a \rceil$, still at most C^*. The second search is an A* from the root with reopening: line 9 treats a node of the first search as new the first time it is reached, and line 11 reopens it afterwards, so that Lemma 2.8 applies, every node it selects before a goal has $f \leq C^*$, and the L of line 5 stays at most C^*; a goal selected at line 6 is optimal. The value $z - r$ is the cost of a path (the path to the node v of P with the saving r, of cost g', and the rest of P, of cost $z - g_P(v)$). The loop ends by line 6, by an empty OPEN (and then the A* has expanded every node of $f < z - r$, so that $z - r = C^*$), or by $z - r \leq wL \leq wC^*$. □

The consistency is needed by the first phase only. The test `focal` runs IOS only on the instances with a consistent heuristic, among them three states of the 8-puzzle whose first path the second search reaches anew, giving its nodes other parents.

The focal searches

The focal search A^*_ϵ of Pearl and Kim [51] expands, among the nodes of OPEN with $f \leq w f_{\min}$ (the list FOCAL), the one that a second order prefers, $f_{\min}$ being the least f of OPEN. Explicit Estimation Search (EES) of Thayer and Ruml [52] orders the focal list by an inadmissible estimate $\hat{d}$ of the distance to a goal in edges, built from estimates corrected online, and chooses among three lists. Both are in `FocalSearch.h`:

```
enum class FocalOrder {
 distance ,    ///< the distance d of a DistanceDomain, h otherwise
 heuristic ,   ///< h
 optimistic ,  ///< \f$ g + ( 2 w - 1 ) h \f$
 explicit_estimation   ///< EES, with \f$ \hat d \f$ and three lists
 };
```

```
struct FocalOptions : public SearchOptions {

 FocalOrder order = FocalOrder::distance;     ///< the order of FOCAL

 };  // end( struct( FocalOptions ) )
```

```
template< Domain D > class FocalSearch {
 public:
  explicit FocalSearch( const D & domain , const FocalOptions & options = {} );
  [[nodiscard]] SolutionT solve( void ) const;
  };
```

The order `distance` is the distance d of a `DistanceDomain` (`Domain.h`), i.e., the estimate `distance( s )` of the number of edges to a goal, which need not be admissible and is h when every edge costs 1; on another domain it is h. The order `heuristic` is h, the order `optimistic` is $g + (2w-1)h$, and `explicit_estimation` is EES. In the focal search, with k the second order:

Algorithm 6.6 (The focal search A^*_ϵ).

```
 1  procedure FOCAL(s₀, w)
 2     OPEN ← {s₀};  B ← w f(s₀);  FOCAL ← {s₀}
 3     while OPEN ≠ ∅ do
 4        B' ← w f_min
 5        if B' > B then move to FOCAL the nodes of OPEN with B < f ≤ B'
 6        B ← B'
 7        u ← the first node of FOCAL by k, removed
 8        if f(u) > B then put u back among the nodes not in FOCAL;
             continue
 9        close u;  if u is a goal then return its path
10        for each successor v of u, inserted as A* does, put v also in
             FOCAL if f(v) ≤ B
```

OPEN is kept by f, with the nodes not in FOCAL also in a list PENDING by f, from which line 5 takes them in order; the three lists are `LazyQueues`, heaps of nodes by a key of any type (then by the larger g) that drop their stale entries lazily as the OPEN lists do, each entry carrying a second value read back with it (`FocalSearch.h`). Line 8 handles a $f_{\min}$ that decreases, which only an inconsistent heuristic causes (a node reopened with a smaller f). With integer costs, $B = \lfloor w f_{\min} \rfloor$ (`times()` of `Weight`), which compares exactly by (v) of Lemma 6.2.

EES keeps OPEN by $\hat f = g + \hat h$, FOCAL with the nodes of $\hat f \le w \hat f_{\min}$ by $\hat d$, and CLEANUP by f. The estimates are corrected online by the mean one-step errors $\bar\epsilon_h$ and $\bar\epsilon_d$, measured at each expansion on the best child (that of least $c + h$): $\epsilon_h = c(u,v) + h(v) - h(u)$, $\epsilon_d = 1 + d(v) - d(u)$, $\hat d = d/(1-\bar\epsilon_d)$ and $\hat h = h + \bar\epsilon_h \hat d$ (section 3 of [52]), with the values when the node is inserted; $\bar\epsilon_d$ is clamped to $[0, 1-10^{-6}]$ and $\bar\epsilon_h$ to nonnegative values (`FocalSearch.h:410-420`). FOCAL is kept as in the focal search, on $\hat f$ in place of f, and the node to expand is chosen by

Algorithm 6.7 (EES: the choice of the node).

```
 1  procedure SELECTNODE()
 2     b_d ← the first node of FOCAL;  b_f^ ← that of OPEN;
       b_f ← that of CLEANUP
 3     if f(b_d) ≤ w f(b_f) then return b_d
 4     if f(b_f) ≤ w f(b_f) then return b_f^
 5     return b_f
```

Proposition 6.16 (The focal searches). Let h be admissible, $w \geq 1$, and let a goal be reachable on a finite graph. Then `FocalSearch` returns a path of cost at most wC^* with every order, with `bound` $= \min(f_{\min}, g)$ at the selection of the goal, whatever the distance of a `DistanceDomain`.

Proof. By the reopening, the least f of OPEN (or of CLEANUP, in EES) is at most C^* at each iteration (Lemma 2.7). In the focal search, every node of OPEN with $f \leq B$ is in FOCAL, including the node of $f_{\min}$ (the nodes of PENDING have $f > B$, since line 5 moves those below B when B rises, and line 8 puts back only those above it when B falls), so that FOCAL is not empty at line 7, and a node expanded at line 9 has $f \leq B = wf_{\min}$; a goal t has $h(t) = 0$, hence $g(t) = f(t) \leq wf_{\min} \leq wC^*$. In EES the corrections are nonnegative ($\bar{\epsilon}_h$ is taken as 0 when negative, and $\hat{d} \geq 0$), and hence $\hat{f} \geq f \geq g$ for every node, whatever the errors; a goal returned at line 3 or 4 has $g \leq \hat{f} \leq wf(b_f) \leq wC^*$, and one returned at line 5 has the least f of CLEANUP, i.e., it is optimal. The search ends, since each iteration closes a node with a new value of g or moves an entry between the lists a bounded number of times. □

The estimates of EES change only the order, never the bound; on a goal $h = 0$ and, if the distance of the domain is 0 there too, $\hat{f} = g$, as in the paper. The orders of `FocalSearch` that ignore the cost (d and h) are those of the original A^*_ϵ, and they may expand many more nodes than weighted A* when the nodes of small d are expensive, which is the weakness of A^*_ϵ that motivated EES ([52], section 2).

The dynamic potential search

The Dynamic Potential Search (DPS) of Gilon, Felner and Stern [53] has no parameter but the bound: it expands the node of OPEN of largest potential

$$\mathrm{ud}(n) = \frac{wf_{\min} - g(n)}{h(n)},$$

i.e., the node most likely to lead to a path of cost at most $wf_{\min}$ if the relative error of h is uniform. A node of $h(n) = 0$ is given $h = 10^{-30}$, so that it comes before all the others if $g(n) < wf_{\min}$ and after all of them if $g(n) > wf_{\min}$, and among the nodes of $h = 0$ the one of larger $wf_{\min} - g(n)$ comes first (`PotentialSearch.h`):

```
template< Domain D > class PotentialSearch {
 public:
  explicit PotentialSearch( const D & domain ,
                            const SearchOptions & options = {} );
  [[nodiscard]] SolutionT solve( void ) const;
  };
```

Algorithm 6.8 (DPS).

```
1  procedure DPS(s0, w)
2     OPEN ← {s0};  F ← f(s0)
3     while OPEN ≠ ∅ do
4        if f_min ≠ F then F ← f_min;  order OPEN again by ud
5        u ← the node of OPEN of largest ud, removed
6        close u;  if u is a goal then return its path
7        insert the successors of u as A* does, by ud
```

OPEN is kept twice, by f for $f_{\min}$ and by ud in a heap rebuilt at line 4 from its live entries, both `LazyQueues`.

Proposition 6.17 (DPS). Let h be admissible, $w \geq 1$, and let a goal be reachable on a finite graph. Then DPS returns a path of cost at most $\max(wf_{\min}, C^*) \leq wC^*$, $f_{\min}$ being taken at the selection of the goal, with `boun d` $= \min(f_{\min}, g)$, provided that the factor 10^{30} separates the potentials of the nodes of $h = 0$ from the others (which holds unless the costs or h are below 10^{-30} relative to $wf_{\min}$).

Proof. By the reopening, OPEN holds a node n of an optimal path with $g(n) = g^*(n)$, and $f_{\min} \leq f(n) \leq C^*$. Let t be the goal selected. If $g(t) \leq wf_{\min}$ we are done. Otherwise its potential is negative and, $h(t)$ being 0, below that of every node of $h > 0$; since t has the largest potential, every node of OPEN has $h = 0$, in particular n, and the order among the nodes of $h = 0$ gives $wf_{\min} - g(n) \leq wf_{\min} - g(t)$, i.e., $g(t) \leq g^*(n) = g^*(n) + h(n) \leq C^*$. □

The paper gives every node of $h = 0$ an infinite potential (footnotes 4 and 6 of [53]), which would expand first a goal of $g > wf_{\min}$; the order above keeps the bound in this case too. The rebuild of line 4 costs $O(|\mathrm{OPEN}| \log |\mathrm{OPEN}|)$ once for each value of $f_{\min}$, which with integer costs and a consistent heuristic are at

most $C^* - h(s_0) + 1$; the paper keeps OPEN in buckets of equal g and h and orders only the buckets (section 4.2), which we have not implemented.

6.7 Batch weighted A*

When the heuristic is a learned function evaluated on an accelerator, its cost per state falls steeply with the number of states evaluated at once, and a search that computes it one state at a time wastes the accelerator. The batch weighted A* by which Agostinelli et al. [126] search with a learned heuristic takes the K best nodes of OPEN at each cycle, expands them all, and computes the heuristic of their new successors in one call; it orders OPEN by $g + wh$ (written there $\lambda g + h$, i.e., $w = 1/\lambda$). In `BatchAStar.h`:

```
struct BatchOptions : public SearchOptions {

 /// the nodes expanded at each cycle, 1 if 0
 std::size_t batch = 1;

 };  // end( struct( BatchOptions ) )
```

```
template< Domain D , class Open = /* buckets or heap, as AStar */ >
class BatchAStar {
 public:
  explicit BatchAStar( const D & domain , const BatchOptions & options = {} );
  [[nodiscard]] SolutionT solve( void ) const;
  };
template< Domain D > [[nodiscard]] Solution< state_t< D > , cost_t< D > >
 batch_astar( const D & domain , const BatchOptions & options = {} );
```

A `batch` of 0 is taken as 1, and a weight below 1 as 1. The heuristic is computed in batches when the domain models the concept `BatchHeuristicDomain` of `Domain.h`, i.e., provides `heuristic_batch( s , h )`, which writes in `h[ i ]` the heuristic of the state `s[ i ]` for two spans of equal size; the search then ignores the h that `for_each_successor` passes with each successor, which may be 0. A node keeps wh in its field `h`, rounded down with integer costs, so that the OPEN lists of A* order by $g + wh$ unchanged:

Algorithm 6.9 (Batch weighted A*).

```
 1  procedure BATCHA*(s₀, K, w)
 2     CLOSED ← {s₀};  OPEN ← {(g(s₀) + w h(s₀), s₀)};  z ← ∞
 3     while OPEN has a node of f < z do
 4        B ← the (at most) K first nodes of OPEN of f < z, removed
 5        N ← ∅
 6        for each u ∈ B, closed, do
 7           if u is a goal then z ← min(z, g(u)), recording u;  continue
 8           for each successor v of u, g' ← g(u) + c(u, v) do
 9              if v ∉ CLOSED then add v to CLOSED and to N
10              else if g' < g(v) then v is not closed
11              else continue
12              g(v) ← g';  parent(v) ← u
13        compute h(v) for all the v ∈ N at once
14        push in OPEN the nodes of lines 9 and 10 of f < z
15     return the path to the goal of cost z, if z < ∞
```

Here f stands for $g + wh$. Line 13 calls `heuristic_batch()` once per cycle, on the new states only (a duplicate keeps the h it has); on another domain the h passed with the successors is used, the search asking the domain for the true heuristic of the parent when $w > 1$, since the field `h` holds wh. Line 14 pushes the nodes in the order in which A* would have pushed them.

Proposition 6.18 (Batch weighted A*). Let h be admissible, $w \geq 1$, $K \geq 1$, and let a goal be reachable on a finite graph. Then `BatchAStar` returns a path of cost at most wC^*, optimal if $w = 1$, whatever K; with $K = 1$ it is A* with the heuristic $\lfloor wh \rfloor$ (or wh), and with $w = 1$ and $K = 1$ it expands the nodes of `AStar`, in the same order.

Proof. The search is well-formed in the sense of §8.3 with one worker, the batch being the entries held: it reopens the nodes reached through a cheaper path (line 10), records every goal it selects, and prunes a node when its key $g + wh$, which is at most wf, is not below z (lines 3, 4 and 14). It stops when no node of OPEN has a key below z and no batch is held, and the argument of Proposition 6.3 applies, with the key of the node of the optimal path at most wC^* (and $\lfloor wh \rfloor \leq wh$); with $w = 1$ it is Theorem 8.6. With $K = 1$ each cycle expands the first node of OPEN, and a goal selected sets z to its key, which no node of OPEN is below, so that the loop ends: this is A* with the heuristic of the keys, line 14 pushing the successors in the order of `AStar`. □

With $K > 1$ a goal may be selected together with nodes of smaller key, whose expansions may still lead to a cheaper goal, which is why a goal does not stop the search at once; a larger K expands more nodes, those of the last batch beyond the optimum, but calls the heuristic K times less often. The search leaves `Solution::bound` at 0 (it proves no bound) and records no improvement. The test `batch` checks that $K = 1$ expands the nodes of A* on easy instances of Korf, that $K \in \{8, 64, 512\}$ keeps the optimal cost, that $K = 64$ with $w = 2$ stays within $2C^*$, and, on a domain of the test that models `BatchHeuristicDomain`, that the heuristic is computed exactly once per state stored, in at most one call per 32 expansions with $K = 64$.

6.8 Incremental search

The changes of a domain

An incremental search keeps its work between two searches on a domain whose edges and goals change in between. The domain has to give the predecessors of a state, i.e., to model the concept `PredecessorDomain` of `Domain.h` (`for_each_predecessor( s , f )` calls `f( t , c )` for every state t with an edge (t, s) of cost c, the edges being those that `for_each_successor()` gives at the same time), and the changes are told to the search as events:

```
template< class State , class Cost > struct EdgeCostChanged {
  State u;  State v;  Cost from;  Cost to;      // the edge ( u , v ), costs
  };
template< class State > struct GoalChanged {
  State t;  bool added;                         // t became a goal, or not
  };
enum class CostChanges { increase_only , decrease_only , both };
template< class D > concept EventDomain;          // PredecessorDomain, and
                                                  // d.drain_events( f )
template< class D > constexpr CostChanges cost_changes_of( void ) noexcept;
```

An absent edge costs ∞ (`infinite_cost()`), so that an edge that appears is a change from ∞ and one that disappears a change to ∞, and a state blocked or freed is the change of each edge into it and out of it (as in Ramalingam and Reps [144]). The old cost is in the event because the domain no longer knows it, and because the repair treats a decrease in $O(1)$ and an increase only when the edge was the one of the parent (see below). An `EventDomain` gives, by `drain_events( f )`, the events since the last call, an `EdgeCostChanged` or a `GoalChanged` each, and forgets them; the events of one edge are coalesced into one, from the cost before the first change to the cost after the last, and those that change nothing are dropped. A domain may declare the static member `cost_changes`, of type `CostChanges`, which `cost_changes_of< D >()` returns (`both` if there is none), so that an incremental search compiles out the repair of the direction that never happens and refuses the changes of that direction. The heuristic must stay consistent for every cost that the edges take.

Two domains of the library are event domains. `DynamicGraph< Cost >` (`DynamicGraph.h`) is a directed graph on the nodes $0, \ldots, n-1$ whose arcs change their costs (`set_cost( e , c )`, the arcs being numbered in the order of `add_edge()`), whose nodes are blocked and freed (`set_blocked()`, an arc with a blocked end being absent) and whose goals change (`set_goal()`); the heuristic is the Manhattan distance to the nearest goal times `set_scale()` if the nodes have positions (`set_position()`), or a value per node given by `set_heuristic()`, and since an incremental search computes the h of a state once, a set of goals that changes under the Manhattan distances needs the fixed set of `set_heuristic_goals()`, the union of the goals of every search, and hence h stays consistent and 0 on the goals of each of them. `grid( w , h , c )` builds a four-connected gridworld, and `gridworld( w , block , gen )` and `perturb( frac , blocks , gen )` the gridworlds of the experiments of Koenig, Likhachev and Furcy ([54], section 7) and their changes: 51×51 cells, the arcs of cost 1 or 2 drawn uniformly and a fraction of them redrawn after each search (their figure 8), or the cells blocked with probability 0.2 and some of them blocked and freed after each search (their figure 9). The grids of `Grid.h` (Chapter 14) are

event domains as well: after a change of the cost of a cell by `GridMap::set_cost()`, `Grid::cell_changed( x , y , old )` patches the moves around the cell and records the change of each move between two of the 3×3 cells around it, the only ones that the cell can change (its own, those into it and the diagonals that cut it), and `drain_events()` hands them to the search; the heuristic must stay admissible as for `Grid::refresh()`, i.e., the new costs of the free cells must not go below the smallest cost of the map when the grid was built.

Lifelong planning A*

Lifelong Planning A* (LPA*) of Koenig, Likhachev and Furcy [54] keeps, besides g, the one-step lookahead

$$\mathrm{rhs}(s) = \min_{s' \in \mathrm{pred}(s)} g(s') + c(s', s)$$

(0 at the start s_0), and expands only the states where the two differ. After a change of some costs only the heads of the edges changed see their lookahead change, and the next search repairs what the change has made inconsistent.

Definition 6.19 (Consistency, keys and the invariants of LPA*). A state s is *locally consistent* if $g(s) = \mathrm{rhs}(s)$, *overconsistent* if $g(s) > \mathrm{rhs}(s)$ and *underconsistent* if $g(s) < \mathrm{rhs}(s)$. Its *key* is

$$k(s) = \big[\min(g(s), \mathrm{rhs}(s)) + h(s) + k_m;\ \min(g(s), \mathrm{rhs}(s))\big],$$

compared lexicographically, k_m being 0 until the target of D* Lite moves (below). The goals are joined to a virtual target t^* by edges of cost 0, and $B = \min_t m(t)$ over the goals t, where $m(t) = \min(g(t), \mathrm{rhs}(t))$ (or the last value of it that the owner of t has told, in section 6.9), so that the key of t^* is $K^* = [B + k_m; B]$. We say that the invariant (R) holds if $\mathrm{rhs}(s_0) = 0$ and, for every other state v (those never generated included), $\mathrm{rhs}(v) = \min_w g(w) + c(w, v)$ over its predecessors, attained by its parent $\mathrm{par}(v)$ when finite; and that (Q) holds if every inconsistent state has $k(v) \geq K^*$ and every inconsistent goal has $k(v) > K^*$.

The goals are several in general, and joining them to t^* makes the test of the end compare the least key of the queue with B, whatever their number. A goal at the key of t^* must be repaired too, since the edge to t^* costs 0, or the cost returned may be below the optimum. The pseudocode is that of `LPAStar.h`:

Algorithm 6.10 (LPA*).

```
 1  procedure UPDATEVERTEX(u)
 2     if u ≠ s0 then rhs(u) ← min over the predecessors s' of u of
          g(s') + c(s', u), parent(u) ← the s' of the minimum
 3     U ← U \ {u};  if g(u) ≠ rhs(u) then insert u in U with k(u)
 4  procedure COMPUTESHORTESTPATH()
 5     while the least key of U < [B + k_m; B] or an inconsistent goal t
          has min(g(t), rhs(t)) ≤ B do
 6        u ← that state, removed from U;  if its key in U is below
          k(u) then insert u with k(u) and go back to line 5
 7        if g(u) > rhs(u) then                          ▷ overconsistent
 8           g(u) ← rhs(u)
 9           for each successor v ≠ s0 of u do
10              if g(u) + c(u, v) < rhs(v) then
11                 rhs(v) ← g(u) + c(u, v);  parent(v) ← u
12              fix the place of v in U, as at line 3
13        else                                            ▷ underconsistent
14           g(u) ← ∞
15           for each v ∈ {u} ∪ the successors of u with parent(v) = u
                or v = u do UPDATEVERTEX(v)
16  procedure EDGECHANGED(u, v, from, to)
17     if to < from and v ≠ s0 and g(u) + to < rhs(v) then
          rhs(v) ← g(u) + to;  parent(v) ← u;  fix the place of v in U
18     if to > from and parent(v) = u then UPDATEVERTEX(v)
19  procedure LPA*()
20     rhs(s0) ← 0;  U ← {s0};  COMPUTESHORTESTPATH()
21     forever: wait for edge costs to change;  for each edge (u, v)
          changed, EDGECHANGED(u, v, from, to);  COMPUTESHORTESTPATH()
```

This is the optimized version of figure 7 of the paper (section 6), with lines 10 and 11 in place of a full UPDATEVERTEX for an overconsistent state, line 15 recomputing rhs only where the parent was u, and lines

17 and 18 those of its lines {33'} to {44'}, which need the old cost of the edge: a decrease costs $O(1)$, an increase a scan of the predecessors only where the edge was the one of the parent. The states are created when they are first reached, with $g = \text{rhs} = \infty$, or when a change names them: a state never reached has no predecessor of finite g, so that its rhs is ∞ as if it had been created, until an edge into it appears, and a decrease into a state never reached creates it when $g(u)$ is finite. The goals of finite g are in an indexed heap by g, whose least key is B, and the inconsistent goals in another by $\min(g, \text{rhs})$, so that the test of line 5 costs $O(1)$ whatever the number of goals; the two heaps are kept by `set_heap_key( h , id , k )`, which inserts or updates the entry `id` (growing the ids if needed), and `erase_heap_key( h , id )`, and the costs are added by `add_cost( a , b )`, which is ∞ if either term is. The queue U has no decrease-key: a state moved or removed leaves its entry, which is stale when the state is no longer in U or when its key in U is no longer the last one it was pushed with; the latter is kept in the state, so that line 6, that of D* Lite (figure 5 of the version of [143] in IEEE T-RO, lines {13'‘} and {14''}), finds an entry whose key the growth of k_m and the change of the heuristic have raised, and pushes it again.

The interface is that of a search that is called many times (`LPAStar.h`):

```
template< PredecessorDomain D , template< class , class > class Queue = LPAHeap >
class LPAStar {
 public:
  explicit LPAStar( const D & domain , const SearchOptions & options = {} );
  [[nodiscard]] SolutionT solve( void );
  void edge_changed( const State & u , const State & v , Cost from , Cost to );
  template< class E > void take_events( E & d ) requires EventDomain< E >;
  void update( const State & v );
  void goal_changed( const State & t , bool added );
  void shift_keys( Cost delta );
  };
```

The first `solve()` is a search from scratch; then the caller changes the costs in the domain (which `LPAStar` reads and does not copy), reports each change by `edge_changed( u , v , from , to )` (lines 16 to 18), or all of them by `take_events()` on an `EventDomain`, and each goal added or removed by `goal_changed()`, and the next `solve()` repairs the search and returns the shortest path with the new costs, with `bound` equal to its cost. The changes of one edge are told once, from the cost before the first to the cost after the last, after all of them are made in the domain; `update( v )`, which recomputes $\text{rhs}(v)$ over all the predecessors, serves when the old costs are not known (e.g., after `Grid::refresh()`, on the 3×3 cells around each cell changed). The start is the initial state of the domain when the `LPAStar` is built, and the constructor refuses a weight other than 1. The counters of a `Solution` are those of its `solve()` and of the changes told before it, except `stored`, which counts all the states created; `reexpanded` and `under` count the underconsistent expansions (line 13), `rescans` the calls of UPDATEVERTEX, and `delta` the states whose g differs from the one before the search. The queue `Queue e` is `LPAHeap` (the default), a 4-ary heap of the pairs of an `LPAKey` $[k_1; k_2]$ (compared lexicographically) and a state, or, for integer costs, `LPABuckets`, a bucket queue of the least k_1 and, among those, of the least k_2, which is the order of Theorem 6 of [54], the last state pushed coming first among equal keys; its least first key is found by a scan that only moves forward between two pushes of a smaller one, so that the pushes below the last key extracted, which the decreases of an incremental search make, cost nothing more.

The correctness of LPA* follows from a static statement, which we state in the form in which section 6.9 needs it too.

Theorem 6.20 (Exactness at quiescence). Let the costs be at least $c_{\min} > 0$, the graph locally finite and h consistent with $h(t) = 0$ on the goals, and let (R) and (Q) hold. Then B is the least g of the goals and the cost $d(t^*)$ of a shortest path from s_0 to a goal, and the parents from a goal t with $g(t) = B$ give a shortest path (if B is finite; if it is infinite, no goal is reachable).

Proof. We take $k_m = 0$, the general case shifting both sides of every comparison of keys by k_m, and write $K^* = [B; B]$. If B is finite, let $\hat{t}$ be a goal with $m(\hat{t}) = B$. If $\hat{t}$ were inconsistent, then $k(\hat{t}) = [B; B] = K^*$, since $h(\hat{t}) = 0$, against (Q); hence $g(\hat{t}) = B$, and $g(t) \geq m(t) \geq B$ for every goal, and hence B is the least g of the goals. (a) $B \geq d(t^*)$. Suppose that $B < d(t^*)$; then $g(t_1) = B < d(t_1)$ for the goal $t_1 = \hat{t}$, which is consistent and is not s_0. By (R) its parent u_2 has $g(u_2) = g(t_1) - c(u_2, t_1) < d(u_2)$, since $d(t_1) \leq d(u_2) + c(u_2, t_1)$, and $g(u_2) + h(u_2) \leq g(u_2) + c(u_2, t_1) + h(t_1) = B$ by the consistency of h. Inductively, let u_k ($k \geq 2$) have $g(u_k) < d(u_k)$, $g(u_k) \leq B - (k-1)c_{\min}$ and $g(u_k) + h(u_k) \leq B$. It is consistent, since otherwise $k(u_k) \leq [g(u_k) + h(u_k); g(u_k)] < K^*$, the second component being below B, against (Q); it is not s_0, since $d(s_0) = 0 \leq g(s_0)$; and its parent satisfies the same with $k + 1$, by the argument for u_2. The values $g(u_k)$

fall by at least $c_{\min}$ at each step and are nonnegative (each finite g being an rhs, i.e., a sum of costs), a contradiction. (b) $B \le d(t^*)$. Let $s_0 = w_0, \ldots, w_n = t$ be a shortest path to a goal with $d(t) = d(t^*)$, and suppose that $d(t) < B$; then $g(t) \ge m(t) \ge B > d(t)$, so some w_i has $g(w_i) > d(w_i)$, and we take the least such i. Either $i = 0$, and s_0 is inconsistent ($\mathrm{rhs}(s_0) = 0 < g(s_0)$) with $k(s_0) = [h(s_0); 0]$, where $h(s_0) \le d(t) < B$; or $\mathrm{rhs}(w_i) \le g(w_{i-1}) + c(w_{i-1}, w_i) \le d(w_{i-1}) + c(w_{i-1}, w_i) = d(w_i) < g(w_i)$, and hence w_i is inconsistent and the first component of its key is at most $d(w_i) + h(w_i) \le d(t) < B$ (a consistent heuristic that is 0 on the goals is admissible). In both cases an inconsistent state lies below K^*, against (Q). If B is infinite, (b) shows that $d(t^*)$ is infinite. (c) With $B = d(t^*)$ finite, the induction of (a), started from $\hat{t}$ with $g(\hat{t}) + h(\hat{t}) = B$, shows that the chain of parents from $\hat{t}$ goes through consistent states (all but $\hat{t}$ having $g < B$, and $\hat{t}$ by its choice) whose g decreases strictly by the costs of the edges; since $g \ge 0$ the chain is finite, and it ends at a consistent state without a parent, i.e., with rhs $= 0$, which by (R) is s_0, with $g(s_0) = 0$. The cost of the chain is $g(\hat{t}) - g(s_0) = B$. □

The theorem is static: it does not care how the values were reached, and the work of an incremental search is to establish (R) and (Q) at the end of each search and to show that it ends.

Proposition 6.21 (LPA*). Let the hypotheses of Theorem 6.20 hold for the costs of each search, and let the changes be told as above. Then each `solve()` of `LPAStar` that does not reach the node limit returns a shortest path with the current costs, or `Status::unsolvable` if no goal is reachable; a `solve()` after no change expands no state.

Proof. (R) holds after the constructor ($\mathrm{rhs}(s_0) = 0$, every other state with rhs and g infinite) and is kept by every step. An overconsistent expansion lowers $g(u)$, so that the minimum defining $\mathrm{rhs}(v)$ for a successor v becomes the least of the old one and $g(u) + c(u, v)$, which lines 10 and 11 compute; the successors not yet created are created with their rhs lowered, the others keep (R) since their other terms are unchanged. An underconsistent expansion raises $g(u)$ to ∞: a successor whose parent is not u keeps a minimum attained by its parent, whose term has not changed and was at most the old term of u, and those whose parent is u (and u itself, if it is its own predecessor) are recomputed by UPDATEVERTEX. A decrease of $c(u, v)$ gives the new minimum $\min(\mathrm{rhs}(v), g(u) + c(u, v))$ (line 17), creating v if $g(u)$ is finite, which is the only case in which a state never created acquires a predecessor of finite g; an increase leaves the minimum and its parent unchanged unless u was the parent, when UPDATEVERTEX computes it (line 18). A goal added or removed changes no rhs. At the exit of line 5 the least valid entry of U has a key not below K^* and every inconsistent goal has $\min(g, \mathrm{rhs}) > B$, which is (Q), since the keys stored in U are lower bounds of the current ones (line 6 pushes again an entry whose key has risen) and the entry returned by line 6 has its exact key. As remarked in the proof of Theorem 6.20, (Q) makes the least $m(t)$ of the goals equal to their least g, which is the B that the code keeps. Hence, Theorem 6.20 applies at the exit. The search ends, since with a consistent heuristic each state is expanded at most twice per search, once overconsistent and once underconsistent (Theorem 4 of [54], section 5.1), and the states of key below K^* are finitely many. After no change, (Q) still holds and line 5 expands nothing. □

A repair typically expands far fewer states than a search from scratch, although a change near the start may (somewhat obviously) undo most of the previous search. The test `lpa` checks LPA* against A* from scratch on grids whose cells change (told by the events of the cells, or by `update()`), on 300 random directed graphs whose arcs and goals change, and on the gridworlds of figures 8 and 9 of [54], 500 episodes each, where it expects LPA* to expand fewer than an eighth of the states of A* with the ties to the smaller g (the paper reports 23.71 expansions per episode against 307.93 on figure 8, and 15.56 against 241.77 on figure 9) and the cost of the path to change in 4 to 11 percent of the episodes of figure 8 (7.3 percent in figure 11 of the paper).

D* Lite

D* Lite of Koenig and Likhachev [143] is LPA* on the reversed graph, from the goal of a robot to the robot, which moves. `RobotDomain< D >` (`LPAStar.h`) is the domain D with its edges reversed: the successors of a state are its predecessors in D and vice versa, with the same costs, the initial state is the goal s_g of the robot, the only goal is the state s_r of the robot, and the heuristic of s is `pairwise_heuristic( s_r , s )` of D, an estimate of the cost from s_r to s in D, which has to be consistent and to satisfy the triangle inequality; the events of D are handed with their edges reversed, and the hashes, the features, the abstract states and the perfect hash are those of D. `DStarLite< D , Queue >` is `LPAStar` on it:

```
template< class D , template< class , class > class Queue = LPAHeap >
 requires PredecessorDomain< D > && PairwiseDomain< D >
class DStarLite {
 public:
```

```
DStarLite( D & d , const State & goal , const State & robot ,
           const SearchOptions & options = {} );
[[nodiscard]] SolutionT solve( void );      // path.front() is the robot
void edge_changed( const State & u , const State & v , Cost from , Cost to );
void take_events( void ) requires EventDomain< D >;
void move_to( const State & r );
[[nodiscard]] const State & robot( void ) const;
};
```

`move_to( r )` moves the robot from a to r: it removes the goal a, adds the goal r, and calls `shift_keys( h(a , r) )`, which adds the estimate of the move to k_m and makes every state compute its h again at its next use, by a version of the heuristic kept in the search and in each state.

Lemma 6.22 (The keys of D* Lite). If the pairwise heuristic satisfies the triangle inequality, the keys stored in U before a move of the robot are lower bounds of the keys of the same states after it.

Proof. A state s with $m = \min(g(s), \mathrm{rhs}(s))$ stored with the key $[m + h(a, s) + k_m; m]$ has, after the move to r, the key $[m + h(r, s) + k_m + h(a, r); m]$, and $h(a, s) \le h(a, r) + h(r, s)$. □

Hence, line 6 of Algorithm 6.10, which pushes again an entry whose key has risen instead of expanding it, keeps the order of the expansions by the current keys without reordering U after a move, which is the point of k_m (figure 5, line {30"}, of the T-RO version of [143]). The test `lpa` walks a robot along its path on six gridworlds of 31×31 cells whose costs change after each step, and checks each path against A* from the robot.

6.9 Hash-distributed incremental search

`HDLPAStar` (`HDLPAStar.h`) runs LPA* on the workers of HDA* (Chapter 9), so that a search is repaired in parallel after a change of the costs, and `HDDStarLite`, in the same header, is the same on the `RobotDomain` of a domain, i.e., a parallel D* Lite. The worker i owns the states v with $\mathrm{owner}(v) = i$ and is the only one that writes $g(v)$, $\mathrm{rhs}(v)$, the parent $\mathrm{par}(v)$ and the place of v in its queue U_i; the others read $g(v)$, by an atomic load, only when they recompute an rhs over all the predecessors (a *rescan*). The goals are joined to the virtual target t^* by edges of cost 0, and the owner of t^* keeps, for each goal t, the last value of $m(t) = \min(g(t), \mathrm{rhs}(t))$ that its owner has sent (the *mirror* of t), in an indexed heap, and publishes the least, B, in an atomic word, so that the test of the end costs a load. A state u is *eligible* when it is inconsistent and either $k(u) < [B + k_m; B]$, or u is a goal and $k(u) \le [B + k_m; B]$.

Three messages cross the workers, all with absolute values: $\mathrm{Relax}(v, x, u)$, sent after the overconsistent expansion of u to the owner of each successor v with $x = g(u) + c(u, v)$, sets $\mathrm{rhs}(v) := x$ and $\mathrm{par}(v) := u$ if $x < \mathrm{rhs}(v)$; $\mathrm{Invalidate}(v, u)$, sent after the underconsistent expansion of u, makes the owner of v rescan v if $\mathrm{par}(v) = u$, and is dropped otherwise; and Wake, sent by the owner of t^* to the others when B rises, since a worker of the mailbox that has found no work becomes active again only by receiving. The new $m(t)$ of a goal travels to the owner of t^* as a message of its own (a Relax to t^*, counted with the Relax messages). An *episode*, i.e., the work of one `solve()` between two barriers, runs as follows:

Algorithm 6.11 (HD-LPA*).

```
 1  procedure WORKER(i, φ)                          ▷ one phase of one episode
 2     while the mailbox is not quiescent do
 3        for each message m received do RECEIVE(m)
 4        if active then
 5           u ← the least eligible state of U_i, the stale entries
                dropped and, if φ = 1, the overconsistent ones expanded
                underconsistent in this episode put aside (frozen)
 6           if u exists then EXPAND(u, φ) else become passive
 7  procedure EXPAND(u, φ)
 8     if g(u) > rhs(u) then                          ▷ overconsistent
 9        g(u) ← rhs(u), by an atomic store, then the sends
10        SEND Relax(v, g(u) + c(u, v), u) to each successor v,
             and Relax(t*, g(u), u) if u is a goal
11     else                                           ▷ underconsistent
12        g(u) ← ∞;  raised(u) ← the episode
13        SEND Invalidate(v, u) to each successor v, and
```

```
            Invalidate(t*, u) if u is a goal;  RESCAN(u)
14  procedure RECEIVE(m)                              ▷ on the owner of m.v
15     if m.v = t* then update the mirror and B;  if B rose and φ = 1,
          SEND Wake to the other workers
16     else if m = Relax(v, x, u), v ≠ s0 and x < rhs(v) then
          rhs(v) ← x;  parent(v) ← u;  fix v in U
17     else if m = Invalidate(v, u) and parent(v) = u then RESCAN(v)
18  procedure RESCAN(v)
19     rhs(v) ← 0 if v = s0, otherwise the least atomic load of g(w) plus
          c(w, v) over the predecessors w, parent(v) ← that w;  fix v in U
20  procedure EPISODE(changes)                        ▷ between barriers
21     each worker applies the changes of the edges into its states, as
          LPAStar::edge_changed(), and the owner of t* those of the goals
22     restart, or repair: if an increase met a parent edge or the key of
          t* rose, WORKER(·, 1) to quiescence and a barrier; then the
          frozen states back in U and WORKER(·, 2) to quiescence
```

A SEND to the worker itself is applied at once, and one to another worker goes in its outbox; a Relax creates the state it names if it is new, and an Invalidate of a state unknown is dropped. The first episode, and every restart, is phase 2 alone from an empty table, i.e., HDA* on the pair (g, rhs) with no pruning at insertion. In phase 1 a state expanded underconsistent in the episode is *frozen*, i.e., not expanded overconsistent, until phase 2: with `Freeze::raised_parent` (the default) only when its parent has been expanded underconsistent in the episode too, with `Freeze::all` always. With one worker every message is applied at once and the least eligible state is expanded, which is the order of LPA*, and nothing is frozen (`HDLPAStar.h:837-849`).

The interface

```
template< PredecessorDomain D , class Dist = StateHashDistribution< D > ,
          ADS::WorkerExecutor Executor = ADS::ThreadPool ,
          template< class , class > class Queue = LPADefaultQueue >
class HDLPAStar {
 public:
  HDLPAStar( const D & domain , const Dist & dist ,
             const HDLPAOptions & options = {} );
  void set_executor( Executor & ex ) noexcept;
  [[nodiscard]] SolutionT solve( void );
  void edge_changed( const State & u , const State & v , Cost from , Cost to );
  void goal_changed( const State & t , bool added );
  template< class E > void take_events( E & d ) requires EventDomain< E >;
  void shift_keys( Cost delta );
  void restart( void ) noexcept;
  [[nodiscard]] std::uint32_t max_raises( void ) const;
  };
```

The use is that of `LPAStar`: the first `solve()` searches from scratch, the caller changes the domain and reports each change, and the next `solve()` repairs the search, or restarts it; `restart()` makes the next `solve()` start from scratch, and `max_raises()` returns the largest number of underconsistent expansions of one state in the last episode. The distribution (Chapter 9) must give each state always the same owner, which a `static_assert` checks against the distributions that change it; the queue `Queue` of each worker is `LPADefaultQueue`, i.e., `LPABuckets` for integer costs and `LPAHeap` otherwise. With an `EventDomain` that declares `CostChanges::decrease_only` (or `increase_only`), `edge_changed()` refuses a change of the other direction, and the repair of that direction is compiled out. The options are `HDLPAOptions`, which derive from the `ParallelOptions` of §8.1 (the number of workers, the size of the batches of the messages, the period of the flushes and the seed, which places t^*) and from

```
struct IncrementalOptions {

 /// the choice of an episode
 enum class Policy {
  adaptive ,                                  ///< by the prediction
  repair ,                                    ///< always repair
```

```
 restart                                          ///< always restart
 };

/// the states that phase 1 does not lower
enum class Freeze {
 all ,                                            ///< every one raised
 raised_parent                                    ///< those whose parent
 };                                               ///< was raised too

Policy policy = Policy::adaptive;                 ///< the choice
Freeze freeze = Freeze::raised_parent;            ///< the rule of phase 1
double rho = 0.5;                                 ///< the bound on changes
double gamma = HUGE_VAL;                          ///< on the predicted work
double beta = 2;                                  ///< on the repair
double mean_weight = 0.5;                         ///< of the last value
double window = 0;                                ///< its first width
bool audit = false;                               ///< check the invariants

};  // end( struct( IncrementalOptions ) )
```

whose fields are described below (`audit` checks the invariants (R) and (Q), the mirror and the queues at the end of every phase, and makes `solve()` throw `std::logic_error` if one fails; the tests set it). The counters of a `Solution` are those of its episode: `under` counts the underconsistent expansions, `reexpanded` the expansions of a state already expanded in the episode, `sent` the messages to another worker and `messages` their batches, `relaxes`, `invalidates` and `wakes` the messages of each kind, `rescans` and `remote_reads` the rescans and the loads of a g of another worker that they make, `phases` the phases run, `restarts` 1 if the episode restarted, `waits` the polls of a worker held back by the window (below), and `delta` the states whose g changed. The table of each worker is a concurrent hash table that the others read while its owner inserts, emptied by a restart in $O(1)$ by epochs, the nodes of the old table being kept in a list of free nodes for the new one.

Counting to infinity

The freeze is needed for an episode to end, as the following example shows.

Example 6.23 (Counting to infinity). Take the edges $s_0 \to u$, $u \to v$, $v \to u$ and $v \to t$ of cost 1, $h = 0$, the goal t, and u and v on two workers, and the episode that removes $s_0 \to u$, after which t is unreachable. At the barrier u is rescanned, $\mathrm{rhs}(u) = g(v) + 1 = 3$, and expanded underconsistent ($g(u) := \infty$), after which its own rescan reads the old $g(v) = 2$ and gives $\mathrm{rhs}(u) = 3$ again, and hence u is overconsistent and is lowered to 3. Meanwhile v, invalidated, reads $g(u)$; if it reads 3, it is raised and lowered to 4, which makes u raised and lowered to 5, and so on: without the freeze, the two workers count to infinity as the routers of an asynchronous distance-vector protocol do, with B rising along (the goal t follows v) and the keys of u and v staying below it. Sequential LPA* cannot do it, since its keys do not decrease within a search. With `Freeze::raised_parent` , v, raised, is not lowered in phase 1 through u, which has been raised in the episode too; the Invalidate of v then makes u rescan, find $g(v) = \infty$ and be raised for good, after which v and t are rescanned to ∞ as well, and phase 2 finds every state consistent. With `Freeze::all` the first lowering of u already waits for phase 2.

The test `hdlpa-thread` (and `hdlpa-openmp`, `hdlpa-fastflow`) runs this episode 50 times with two workers, both rules of the freeze and the policy `repair`, and checks that it ends with no path.

The correctness

Definition 6.24 (The hypotheses of HD-LPA*). (H1) The costs are at least $c_{\min} > 0$, the graph is locally finite and h is consistent, with $h(t) = 0$ on the goals, for the costs of each episode. (H2) The costs and the goals change only at the barriers. (H3) Only owner(v) writes $g(v)$, $\mathrm{rhs}(v)$ and $\mathrm{par}(v)$, so that the modification order of $g(v)$ is the program order of its owner. (H4) Every write of $g(u)$ is sequenced before the messages it implies, which go to every successor of u, and every change of $m(t)$ of a goal before its message to t^*. (H5) The messages of a worker to another arrive in the order in which it sent them, and a send happens before the actions of the receiver after the drain. (H6) The loads of g are atomic, so that a load returns neither a write older than one that happens before it, nor one older than what an earlier load of the same thread returned. A *quiescent point* is an instant at which no worker is active and no message exists, which the counter of the mailbox detects; the barriers are quiescent points.

(H2) to (H6) hold by construction: the changes are applied in the first stage of an episode, the owners alone write, the store of g precedes the sends (line 9), the channels of the mailbox are first in first out per pair of workers, and g is a `std::atomic` read with acquire and written with release. (H1) is a hypothesis on the domain, as for `LPAStar`. We want (R) and (Q) of Definition 6.19 at the end of each episode, with $m(t)$ the mirror; by Theorem 6.20, they give a shortest path.

Theorem 6.25 (No update lost). Under (H2) to (H6), if (R) holds at the start of a phase, then (R) holds at the next quiescent point, whatever the interleaving of the workers, the order in which they choose their states, the delays of the messages and the values that the loads return within (H6).

Proof. Fix $v \neq s_0$, let $g_f(w)$ be the value of $g(w)$ at the quiescent point Q, and call an *assignment* of v any action that writes $\mathrm{rhs}(v)$ and $\mathrm{par}(v)$ (the state at the start of the phase, a Relax that lowers rhs, a rescan). Upper bound: let w be a predecessor of v. If $g(w)$ was not written in the phase, (R) gave $\mathrm{rhs}(v) \leq g_f(w) + c(w, v)$ at the start, a Relax only lowers rhs, and any later rescan loads $g_f(w)$, the only value since the barrier. If the last write W of $g(w)$ is finite, it is an overconsistent expansion followed by $\mathrm{Relax}(v, g_f(w) + c(w, v), w)$ (H4), processed before Q, after which rhs rises only by rescans that happen after W (H5) and therefore load W or a later write (H6), i.e., $g_f(w)$. If it is ∞, the bound is trivial. Hence $\mathrm{rhs}(v) \leq \min_w g_f(w) + c(w, v)$ at Q. Lower bound and parent: let A be the last assignment of v before Q, which set $\mathrm{rhs}(v) = \gamma + c(u, v)$ and $\mathrm{par}(v) = u$, γ being a value of $g(u)$ produced by a write W_γ, and suppose that $\gamma \neq g_f(u)$, so that a first write W' of $g(u)$ follows W_γ, with its message M' to v. (i) M' is processed after A: if A is a Relax, it was sent before W' and precedes M' in the channel (H5); if A is a rescan and M' were processed before it, W' would happen before the load, which then could not return W_γ (H6); if A is the state at the barrier, everything of the phase follows it. (ii) M' causes an assignment: when it is processed $\mathrm{par}(v) = u$ (no assignment came after A), and either W' is underconsistent and M' is an Invalidate that names the parent, or W' is overconsistent and writes a value below γ (an overconsistent expansion lowers g), and hence M' is a Relax that lowers rhs. This contradicts the choice of A; hence $\gamma = g_f(u)$, and $\mathrm{rhs}(v) = g_f(u) + c(u, v)$ is attained by its parent. The mirror of a goal is the last $m(t)$ sent, by (H4) and (H5). □

The proof uses neither the order of the expansions, nor the monotonicity of g, nor that a rescan is atomic: its loads may be spread over time and see values of different ages. At the barrier nobody writes, and the changes keep (R) state by state, as in the proof of Proposition 6.21.

Theorem 6.26 (Termination). Under (H1), on a finite graph (or a locally finite one whose states of key below $\sup B$ are finitely many), every phase reaches a quiescent point in finitely many steps. With `Freeze::all` each state is expanded underconsistent at most once per episode; with `Freeze::raised_parent` at most once more than the number of its predecessors; in phase 2 no state is expanded underconsistent.

Proof. In phase 1, between two writes of ∞ the writes of $g(u)$ are a strictly decreasing run of finite values, each of them an rhs, i.e., a value of g at the barrier plus the cost of a walk, or the cost of a walk from s_0; by (H1) only finitely many of them lie below any bound, so each run is finite. With `Freeze::all`, after its ∞ a state is not lowered again in the phase, and it cannot be raised again, since $g = \infty \geq \mathrm{rhs}$. With `Freeze::raised_parent`, a raised state u is lowered only through a parent w not raised in the episode, as its owner sees it; the store of $g(w)$ releases after the stamp of the raise of w, and the load or the message by which u received that value acquires it, so that once u has its rhs from a value written after the raise of w, the raise is seen. Between two lowerings of u an Invalidate from its parent must have raised $\mathrm{rhs}(u)$, which means that the parent of the first lowering has been raised since, and can no longer serve; hence u is lowered at most once through each predecessor, and raised at most once more. The messages are one per write and successor, the rescans one per Invalidate or raise, and B, the least of the mirrors, rises finitely often, so that the Wakes, at most $p - 1$ per rise, are finitely many. In phase 2 there are only Relax messages, g, rhs and B only decrease, and the runs are finite as above; an underconsistent state has the key $[g + h + k_m; g]$, which does not change while it stays underconsistent, and at the end of phase 1 it is not eligible with the value B_1 of B (by Theorem 6.27), while B only decreases from B_1; it never becomes eligible, and if its rhs falls below g it is overconsistent. □

With one worker nothing is frozen, and the phase ends as LPA* does, each state being expanded at most twice (Proposition 6.21).

Theorem 6.27 (The stop at quiescence). At the end of each phase (Q) holds, except, at the end of phase 1, for the frozen states, which phase 2 expands.

Proof. A worker becomes passive only after finding no eligible state with the value B_r of B that it loaded, and stays passive until it drains a message, so that its queue is the same from that load to the quiescent point Q. If the final B is not above B_r, (Q) holds for its states, since eligibility only grows when B rises. If it is, some write of B after the one loaded rose, and its Wake was sent after it (H4); had the Wake been drained before the

load, the load would have returned that write or a later one (H5, H6), so it was drained after, which makes the worker active after its last passive moment, a contradiction. In phase 2, B only decreases and the first case always applies. □

The window

The workers may also be held near the order of LPA* by a window on the least key: each worker publishes, in an atomic word of its own, the least first key k_1 of its eligible states (∞ if none), and expands its least eligible state only if $k_1 \le k_{\min} + \Delta$, $k_{\min}$ being the least of the words it reads; a worker held back flushes its outboxes, polls and tries again, staying active. This is the bucket of Δ-stepping of Meyer and Sanders [145] on the keys of LPA*, with the difference that the workers do not wait for each other at the end of a bucket.

Lemma 6.28 (The window). With the window, Theorems 6.25 to 6.27 hold as stated, and every phase still reaches a quiescent point in finitely many steps, provided that every worker runs infinitely often.

Proof. The window restricts only the choice among the eligible states, which Theorem 6.25 and Theorem 6.26 leave free, and a worker held back has an eligible state, so that it does not become passive; the condition under which a worker becomes passive, and with it Theorem 6.27, is unchanged. Suppose that from some instant on no state is expanded. Then the messages in transit are drained in finitely many steps, after which the queues no longer change; each worker computes again its least eligible key and publishes it (a worker with none publishes ∞ before it becomes passive), and by the coherence of the atomic words each reader eventually loads these values. The worker W whose least eligible key k^* is the least of all then reads $k_{\min} = k^*$, so that its state is within the window, and W expands it, a contradiction. Hence expansions go on until the phase is quiescent, and by Theorem 6.26 there are finitely many of them. □

The window is used with more than one worker and `window` not negative (0 by default); its width Δ starts at `window` and is adapted by each worker: it doubles after 65 polls held back in a row (from the least of 1 and of the positive costs of the edges seen, when it is 0) and shrinks by a quarter, down to `window`, at each expansion of a state already expanded in the episode (`HDLPAStar.h:1086-1124`). We remark that the window trades the re-expansions of an asynchronous order for the polls of the workers held back, which the counts of the expansions do not show (they are in `waits`).

Corollary 6.29 (The episodes). Under (H1), every episode of `HDLPAStar` that is not stopped by its budget ends with $B = d(t^*)$ and returns a shortest path, or `Status::unsolvable` if no goal is reachable; an episode stopped by its budget restarts and ends in the same way.

Proof. (R) holds at the first barrier (an empty table) and after the changes (Theorem 6.25 and the remark after it); each phase ends (Theorem 6.26, Lemma 6.28) with (R) and, after phase 2, with (Q) (Theorem 6.27); Theorem 6.20 then gives the claim. A restart starts from an empty table, where (R) holds trivially, and runs phase 2 alone, i.e., an HDA* on (g, rhs), which ends by the same theorems. □

The rule `Freeze::raised_parent` and the bound on $\min(g, \mathrm{rhs})$ of the goals instead of g are ours. With the freeze of every raised state and the bound on g, one worker expanded about 4 times the states of LPA* on gridworlds of 21×21 cells, since a goal raised in phase 1 left B at ∞ until phase 2; with the bound on $\min(g, \mathrm{rhs})$ still about 1.7 times on those of figure 8 of [54], since a raised state waits for phase 2, and about 1.3 times with `Freeze::raised_parent`; this is why one worker freezes nothing and follows the order of LPA* (these counts were measured while designing the search, and are not checked by a test). The coalescing of the Relax messages to one state in an outbox by the least value (as long as no Invalidate to the state lies between them) would keep Theorem 6.25, the record dropped being dominated by the one kept, but the outboxes do not coalesce.

Repair or restart

Each episode chooses between the repair and a restart, by the policy of `IncrementalOptions`: `repair` and `restart` always do that, and `adaptive` (the default) chooses at the first barrier of the episode. Let $\hat{S}$ be the mean of the expansions of the restarts, each new value weighing `mean_weight` (0.5 by default). The episode restarts when the changes that matter, i.e., the decreases that lower an rhs and the increases of a parent edge, are more than $\rho\hat{S}$ (`rho`, 0.5 by default), or when the prediction of the repair exceeds $\gamma\hat{S}$ (`gamma`, ∞ by default, so that the prediction is not used); the prediction sums over the heads of the changes the expansions per head that the past repairs cost, learned by the place of the head between the start and the goal, in 8 buckets of $\min(g, \mathrm{rhs})/B$ (`HDLPAStar.h:1262-1300`). A repair that expands more than $\beta\hat{S}$ states (`beta`, 2 by default) is stopped, the workers checking the expansions of all at each flush of their outboxes, and restarted; before the first restart $\hat{S}$ is unknown

and no budget applies.

Proposition 6.30 (The cost of an episode). Let the policy be `adaptive`, $\hat{S} = S$, the expansions of a restart, and R those of the repair, and neglect the expansions made between two checks of the budget (at most `flush_period` per worker). An episode that repairs expands R states if $R \le \beta S$, and $(1+\beta)S$ otherwise; hence at most $(1+\beta)S$ always, and at most $(1+\beta)\max(1, 1/\beta)\min(R, S)$.

Proof. If the repair ends within its budget, it expands $R \le \beta S$ states. Otherwise it is stopped after βS expansions and the restart expands S more. For the ratio, if $R \le \beta S$ the episode expands R, which is at most $\max(1, \beta)\min(R, S)$ (since $R \le \beta S$ and $R \le R$); if $R > \beta S$ it expands $(1+\beta)S$, and $\min(R, S)$ is at least $\min(\beta, 1)S$. In both cases the ratio to $\min(R, S)$ is at most $(1+\beta)\max(1, 1/\beta)$. □

For $\beta = 1$ the ratio is 2, the deterministic bound of the ski rental problem [169]: an episode costs at most twice what the better of the two choices would have cost, and no β does better. We remark that the prediction is a guess, which the budget makes safe but not good: by default it is not used ($\gamma = \infty$), and $\beta = 2$, since the repairs that cost between one and two restarts are kept, and with $\beta = 1$ each of them would cost twice a restart.

The test `hdlpa-thread` runs `HDLPAStar` with 1, 2, 3 and 8 workers on six gridworlds whose costs change, 21 episodes each, against A* from scratch with the invariants audited at each barrier, with both rules of the freeze, the three policies, batches of 1 and 64, and with and without the window; with `Freeze::all` it checks that no state is expanded underconsistent twice in an episode, with the policy `restart` that every episode restarts, and with one worker and the policy `repair` that the expansions are within 1.1 times those of `LPAStar`. It also runs 150 random directed graphs whose arcs and goals change, the episode of Example 6.23, a grid with 8 moves told by the events of its cells, `HDDStarLite` with a robot that walks, a budget too small for any repair ($\beta = 0.01$), and a search with no change, which must expand nothing.

6.10 Benchmark

We compare the bounded-suboptimal searches of sections 6.3, 6.5 and 6.6 on the 100 instances of the 15-puzzle of Korf and on random grids of the kind of Chapter 2, with the weights 1.5, 2 and 5, the sequential A* ($w = 1$) being the reference. For each search and weight, Table 6.1 gives the expansions, the cost of the path over C^* (its suboptimality, which is usually much below w), the cost over the bound that the search proves (the suboptimality that the user can certify), and the time; for the anytime searches, the time to the first path and to the proof of optimality.

Table 6.1. The sequential bounded-suboptimal searches on the 15-puzzle and on grids, with the weights 1.5, 2 and 5. Each row gives the expansions and the seconds summed over the instances, and the means of cost / C^* and of cost / bound; an anytime search gives also the seconds to its first path. The machine and the compiler are given in the note of the table.

search	w	expanded	cost / C*	cost / bound	seconds	seconds to a path

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The incremental searches are compared on the gridworlds of figures 8 and 9 of Koenig, Likhachev and Furcy [54] (generated by `DynamicGraph::gridworld()` and changed by `perturb()`), with 50 episodes on each of 10 worlds: `LPAStar`, `HDLPAStar` with 1, 2, 4 and 8 workers and the three policies, and A* from scratch with the ties to the smaller g (version 1 of the paper, which breaks the ties as LPA* does).

Table 6.2. LPA* and HD-LPA* against A* from scratch on the gridworlds of figures 8 and 9 of [54]. Each row gives, per episode, the expansions, the underconsistent expansions, the messages, the restarts and the seconds. The machine and the compiler are given in the note of the table.

search	workers	policy	expanded	under	messages	restarts	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The clock is the timer of the `Solution`: it covers `solve()`, i.e., for an incremental search the repair (with the changes told before it applied at its first barrier), and not the changes made in the domain, the generation of the instances and the start of the process. We leave both tables as placeholders until the campaign is run on an idle machine. The counts that do not depend on the machine and that the tests check are given in the sections above (e.g., the factor 8 between the expansions of LPA* and of A* on the gridworlds of the paper, which reports 23.71 against 307.93 on figure 8).

6.11 Notes and references

Weighted A* is due to Pohl [46]; its bound wC^* is classical, and the observation that it holds without reopening when the heuristic is consistent is in Likhachev, Gordon and Thrun [48] (Theorem 1), where it is the basis of ARA*. The bound of weighted search in any order of expansion, with the pruning by $wf \geq z$, is in substance Theorem 4 and Corollary 2 of Burns, Lemons, Ruml and Zhou [3] (section 5.1), who also introduced the weak duplicate dropping (section 5.2, Theorem 5) and the anytime weighted and restarting forms of PBNF (sections 7 and 7.1, with the comparison of section 8.4). The rule of duplicate dropping of `Weight`, with a budget that depends on the heuristic of the parent, and its proof in any order of expansion with a consistent heuristic (Proposition 6.4) are ours, as is the observation of Example 6.5 that it does not extend to inconsistent heuristics. The anytime weighted A* is that of Hansen and Zhou [47], and ARA* that of Likhachev, Gordon and Thrun [48]; Richter, Thayer and Ruml [103] showed that restarting the search at each new weight, instead of repairing it, often finds better paths sooner, which `AnytimeAStar` does not offer.

The optimistic search is that of Thayer and Ruml [49], the improved one that of Chen, Sturtevant, Doyle and Ruml [50], whose Algorithm 1 goes on from the OPEN of the first search; we restart from the root (section 6.6) for the reason given there. The focal search is the A^*_ϵ of Pearl and Kim [51], Explicit Estimation Search is due to Thayer and Ruml [52], and the dynamic potential search to Gilon, Felner and Stern [53], whose infinite potential for the nodes of $h = 0$ we replace by a large one, which keeps the bound when a goal of large g is in OPEN. The batch weighted A* is that by which Agostinelli, McAleer, Shmakov and Baldi [126] solve the Rubik's cube with a learned heuristic.

Lifelong Planning A* is due to Koenig, Likhachev and Furcy [54], D* Lite to Koenig and Likhachev [143], and both build on the incremental shortest paths of Ramalingam and Reps [144]; Koenig and Sun [119] compare the incremental searches with the real-time ones for agents that act while they plan. The hash-distributed LPA* of section 6.9, with its two phases, its freeze and its proofs, is ours; the counting to infinity of Example 6.23 is the classical weakness of the asynchronous distributed Bellman-Ford algorithms of the routers, and the window is the bucket of Δ-stepping of Meyer and Sanders [145]. The policy that bounds the cost of an episode by a budget on the repair is the strategy of the ski rental problem [169].

The library offers no weighted or anytime form of IDA* beyond the weighted thresholds of section 6.3, no parallel focal or optimistic search, and no incremental search with a weight (the anytime D* of Likhachev et al. would combine ARA* and D* Lite); `LPAStar` and `HDLPAStar` refuse a weight other than 1.

6.12 Exercises

Exercise 6.1 (`ctest -R PBeFS.weighted`). Compute the fraction a/b of `Weight< int >` for $w = 1.5$ and $w = 1.1$, and check by hand that, with $w = 1.5$, `key( 2 , 3 )` is 13, `scale( 10 )` is 20, `lower_bound( 10 )` is 7 and `times( 10 )` is 15, and that, with $w = 1.1$, `key( 1 , 1 )` is 21 and `times( 10 )` is 11. Show by an example that the keys $\lfloor g + 1.5h \rfloor$ would order two nodes differently from $g + 1.5h$, and why the exact keys of Lemma 6.2 matter for the test of the end of a parallel search. The test checks these values.

Exercise 6.2 (`ctest -R PBeFS.weighted`). With $w = 2$ and duplicate dropping, decide whether a closed node of $g = 10$ and $h = 4$ is reopened by a path of cost 8 from a parent of $h = 5$, and by a path of cost 9 from a parent of $h = 6$; and whether the second one reopens the node when it is not closed. Then show that with $w = 1$ the rule of `Weight::improves()` reopens every node reached through a cheaper path, so that `drop_duplicates` changes nothing (the test checks this on an instance of Korf, comparing the expansions with those of A*).

Exercise 6.3 (`ctest -R PBeFS.weighted`). Run by hand weighted A* with duplicate dropping on the graph of Example 6.5 with $L = 1$, and check that it returns a path of cost 11 while $wC^* = 10$. Then run it on the graph of the test `reopen` of Chapter 2, with $h = (0, 3, 0, 1, 0)$, on which the test checks that the bound holds for

$w \in \{1.5, 2, 5\}$, and explain why the bound holds there although the heuristic is inconsistent (look at the sum D_j of the proof of Proposition 6.4).

Exercise 6.4 (`ctest -R PBeFS.weighted-wrapper`). A weighted A* with an integer weight is obtained by a wrapper of the domain whose heuristic is wh on every state, on which `AStar` runs unchanged. Show that `AStar` on the wrapper with $w = 2$ and `AStar` with `SearchOptions::weight = 2` on the domain compute the same keys, and hence select the same entries in the same order, and that the paths of both cost at most $2C^*$. What does the wrapper report as `bound`, and what does the weighted search report? The test checks the bound of the wrapper on 200 states of the 8-puzzle.

Exercise 6.5 (`ctest -R PBeFS.shift`). Let $\delta > 0$ and $w \geq 1$. Show that weighted A* with the heuristic $h + \delta$ on every state, goals included, selects the same entries in the same order as with h (the keys all grow by $w\delta$), and therefore returns a path of cost at most wC^* although $h + \delta$ is not admissible. Which step of the proof of Theorem 6.7 fails for $h + \delta$, and why does the conclusion hold anyway? The test checks the case $w = 1$ on 200 states of the 8-puzzle, comparing the costs and the expansions.

Exercise 6.6 (`ctest -R PBeFS.anytime`). Show that the first path of AWA* costs at most wC^* and that every bound it records is at most C^* (Theorem 6.10), and that the costs of its improvements decrease strictly. Then explain why AWA*, stopped at its node limit, returns the best path found with the status `node_limit` (the test checks this on an instance of Korf with $w = 5$ and half the nodes of the full search), and why its bound is then the last least f of FQ and not the cost.

Exercise 6.7 (`ctest -R PBeFS.anytime`). Prove Lemma 6.11 when the heuristic is inconsistent, following the node s_j of an optimal path through OPEN, INCONS and line 20, and show that the bound $w_i C^*$ of each iteration may fail with an inconsistent heuristic, while the final path is still optimal. The test runs ARA* with the schedule $(w, 1 + (w - 1)/2, 1 + (w - 1)/4)$ on the instances of the test, the inconsistent one included, and checks that a schedule that increases is refused.

Exercise 6.8 (`ctest -R PBeFS.focal`). Show that with $h = 0$ on a node n, the potential of DPS with $h(n) = 10^{-30}$ puts n before every node of $h > 0$ if $g(n) < wf_{\min}$ and after all of them if $g(n) > wf_{\min}$, and use it to prove Proposition 6.17. Give a small graph on which the infinite potential of the paper for the nodes of $h = 0$ returns a path of cost above wC^*. The test runs DPS with $w \in \{1.2, 2, 4\}$ on the 8-puzzle, an instance of Korf, three grids, Romania and the inconsistent instance of `reopen`.

Exercise 6.9 (`ctest -R PBeFS.focal`). Show that the estimate $\hat{f}$ of EES is never below g, so that a goal chosen at line 3 or 4 of Algorithm 6.7 costs at most w times the least f of CLEANUP, whatever the errors measured. Then give an inconsistent heuristic on which the first phase of IOS (weighted A* without reopening) selects a node of $g + w_o h > w_o C^*$, so that the L of line 2 of Algorithm 6.5 exceeds C^*; this is why the test runs IOS only on the instances with a consistent heuristic, while it runs the optimistic search, the four orders of `FocalSearch` and DPS on the inconsistent one too.

Exercise 6.10 (`ctest -R PBeFS.lpa`). Prove from (Q) that a `solve()` of `LPAStar` after no change expands no state (the test checks this on the 8-puzzle), and that after a change of one edge (u, v) that is not the edge of the parent of v and does not lower $\text{rhs}(v)$ the repair expands no state either. Then show that a decrease into a state never created creates it only when $g(u)$ is finite, and why this keeps (R) for the states that are not in the table.

Exercise 6.11 (`ctest -R PBeFS.lpa`). Prove Lemma 6.22 and show by an example that, if the pairwise heuristic violates the triangle inequality, an entry of U may have a stored key above its current one after a move of the robot, so that line 6 of Algorithm 6.10 does not detect it. The test walks a robot with D* Lite on six gridworlds whose costs change after each step, against A* from the robot.

Exercise 6.12 (`ctest -R PBeFS.bounded-thread`). Show that the anytime `HDAStar` records with each path only the bound $h(s_0)$, and why the least f of the OPEN of one worker is not a lower bound on C^* while the others are expanding; then show that it ends with an optimal path (Theorem 6.9 (iii)). The test runs the weighted, duplicate-dropping and anytime HDA*, farm, SPA*, MQA* and SafePBNF, and ARPBNF, with 1, 3 and 8 workers.

Exercise 6.13 (`ctest -R PBeFS.hdlpa-thread`). Run the episode of Example 6.23 by hand with `Freeze::all`, and count the underconsistent expansions of each state; show that it never exceeds 1 with `Freeze::all` (Theorem 6.26), which the test checks through `max_raises()` on six gridworlds with 1 to 8 workers. Why does the proof of Theorem 6.25 not need the freeze?

Exercise 6.14 (`ctest -R PBeFS.hdlpa-thread`). With $\beta = 0.01$ (and ρ, γ large), show that every repair whose expansions exceed $0.01\hat{S}$ is stopped and restarted, and that the paths are still optimal (Corollary 6.29); the test

checks that some of 20 episodes on a gridworld restart and all of them are optimal. Then prove Proposition 6.30 and show that $\beta = 1$ minimizes its ratio.

Exercise 6.15 (`ctest -R PBeFS.batch`). Show that the batch weighted A* with $K = 1$ and $w = 1$ expands the nodes of `AStar` in the same order, and that with $K > 1$ and $w = 1$ it still returns an optimal path (Proposition 6.18), although a goal may be selected before nodes of smaller f of the same batch. Bound the number of calls of `heuristic_batch()` in terms of the expansions and K; the test checks that with $K = 64$ there is at most one call per 32 expansions, and that the heuristic is computed once per state stored.

7. Bidirectional and external-memory search

[Source: include/PBeFS/astar/Bidirectional.h (BidirectionalOptions, BiNode, BiQueue, BiFronts, ReverseDomain, BlindDomain), BiBestFirst.h (BiBestFirst, bidijkstra, mm, mme, fmm, bae), NBS.h (NBS, nbs), DVCBS.h (DVCBS, dvcbs), GBFHS.h (GBFHS, gbfhs, bibfs), ParallelBidirectional.h (ParallelBidirectional, pnba), PEMSearch.h (PEMSearch, PEMOptions), IDDAStar.h (IDDAStar, IDDOptions, idd_astar), include/PBeFS/astar/Domain.h (BidirectionalDomain, BackwardOperatorDomain, MinCostDomain, for_each_parent); the tests bidir, bidir-thread, pem-thread and idd of test/bidirectional.cpp]

We are concerned in this chapter with two ways of reducing what a search must hold to find a path of least cost. The first is to search from both ends: a *front-to-end bidirectional search* grows a forward search from the initial state s and a backward search from the goal t, along the edges reversed, each guided by a heuristic towards the root of the other, and it stops when the two have met and a lower bound on the optimal cost C^*, drawn from their open lists, proves that no cheaper path joins them. Since the work of a search grows, roughly, exponentially with the depth it reaches, two searches of half the depth may cost far less than one of the whole depth (cf. the last row of Table 1.1 in §1.3); the difficulty, which took some decades to settle, is to stop as soon as the two searches meet without giving up optimality, and without letting either of them go beyond the middle. The second way is to keep the lists on the disk, as the delayed duplicate detection of §4.6 does for A*: here we treat the parallel external-memory bidirectional searches of Siag et al. [57], which combine the two ideas, and the A* with *immediate* duplicate detection on the disk of Lin and Fukunaga [95], whose closed list is a file read at random, as fits a solid state drive.

The structure of the chapter is as follows. Section 7.1 states the interface, i.e., the concepts that a domain has to satisfy to be searched backward, the options of the searches and the two adapters of a domain. Section 7.2 describes the table of the nodes that the two directions share, the meeting of the two fronts and the two lemmas on which every proof of the chapter rests, and section 7.3 the lower bounds on C^* by which the searches stop. Sections 7.4 to 7.7 give the searches that hold their lists in memory: those that expand one node at a time (the bidirectional Dijkstra's algorithm, MM, MMe, fMM and BAE*), NBS with the must-expand graph, DVCBS with its dynamic vertex cover, and GBFHS. Section 7.8 runs the two directions at once on two workers (PNBA* and the parallel versions of the others), section 7.9 gives the parallel external-memory searches of the framework PEM-BiHS, and section 7.10 the A* with immediate duplicate detection on the disk. Sections 7.11 to 7.13 give the benchmark, the notes and references, and the exercises. We use the notation of Chapter 1, the results on A* of Chapter 2, the lazy priority queues and the tables of Chapter 3, and the buckets and files of the external search of §4.6, which we do not repeat.

7.1 The problem and the interface

We consider the problem of Chapter 1 with a single goal state t. A bidirectional search needs, besides a `Domain`, the goal itself (from which the backward search starts), the predecessors of a state (along which that search moves) and a heuristic towards s (by which it is guided). These are the requirements of the concept `BidirectionalDomain` (Domain.h:476): a `Domain` that also has `goal_state()`, the one state for which `is_goal()` holds, `backward_heuristic( s )`, an estimate of the cost of a path from the initial state to `s`, and `for_each_predecessor( s , hb_s , f )`, which calls `f( t , c , hb_t )` for each state `t` with an edge (t, s) of cost `c`, `hb_t` being the backward heuristic of `t`, computed (incrementally, if the domain can) from that of `s`. The refinement `BackwardOperatorDomain` (Domain.h:496) has, moreover, `for_each_predecessor( s , hb , op , f )`, which skips the predecessor that the inverse of `op` leads to, `op` being the operator by which the backward search generated `s`, exactly as an `OperatorDomain` skips the move back to the parent of a successor

(Lemma 1.8); the function template `for_each_parent` (`Domain.h:773`) is the backward counterpart of `for_each_child` (Algorithm 1.1), and it calls the form with the operator when the domain has it, and the plain one otherwise. Finally, `MinCostDomain` (`Domain.h:512`) is a `Domain` with `min_edge_cost()`, a lower bound $\varepsilon \geq 0$ on the cost of an edge, which several of the bounds below use. The sliding-tile puzzles (whose predecessors are their successors, the moves being reversible at the same cost), the grids (whose predecessors are the moves reversed, at the cost of the cell left) and the explicit graphs satisfy these concepts, and the test `bidir` checks, on random grids with weighted cells, that the edges seen backward are those seen forward, with their costs.

Definition 7.1 (Notation of the bidirectional searches). We write $d(u, v)$ for the cost of a cheapest path from u to v, $C^* = d(s, t)$, F and B for the forward and the backward direction, and $\bar{D}$ for the direction other than D. For a state n, $g_F(n)$ is the cost of the best path found from s to n and $g_B(n)$ that of the best path found from n to t, each ∞ until its direction reaches n; we write $d_F(n) = d(s, n)$ and $d_B(n) = d(n, t)$ for their optimal values. $h_F(n)$ estimates $d(n, t)$ and $h_B(n)$ estimates $d(s, n)$, and $f_D = g_D + h_D$. The heuristic h_F is *admissible* if $h_F(n) \leq d(n, t)$ and *consistent* if $h_F(u) \leq c(u, v) + h_F(v)$ for every edge (u, v) (Definitions 2.1 and 2.2), and symmetrically h_B is admissible if $h_B(n) \leq d(s, n)$ and consistent if $h_B(v) \leq c(u, v) + h_B(u)$. Finally, $\varepsilon \geq 0$ is a lower bound on the cost of every edge, and U is the *incumbent*, the cost of the best path from s to t found so far (∞ if none), i.e., the z of Table 1.2, which we call U here as the headers and the literature on bidirectional search do.

Two adapters change a `BidirectionalDomain` without copying it. `ReverseDomain< D >` (`Bidirectional.h:640`) swaps the initial and the goal state: its successors are the predecessors of `D`, its heuristic is the backward heuristic of `D`, and the other way round, so that A* on it is the *reverse* A^* (rA*) of Siag et al. [57], and a path found on it, reversed, is a path of `D`; the operators are kept when `D` is both an `OperatorDomain` and a `BackwardOperatorDomain`, and `min_edge_cost()` when `D` is a `MinCostDomain`. `BlindDomain< D >` (`Bidirectional.h:709`) replaces both heuristics by 0, so that A* on it is Dijkstra's algorithm, and MM on it is the brute-force bidirectional search MM_0 of Holte et al. [61]. Both hold a reference to `D`, which must outlive them.

The in-memory searches take the options

```
struct BidirectionalOptions : public SearchOptions {

 /// the least cost \f$ \epsilon \f$ of an edge, by which the bound
 /// \f$ g_F + g_B + \epsilon \f$ is raised; negative to take
 /// min_edge_cost() of a MinCostDomain, and 0 for any other Domain
 double epsilon = -1;

 /// true if both heuristics are consistent: a search then does not open a
 /// node that the other has closed, and BAE* may use its bound; false
 /// asks only for admissible heuristics, and BAE* refuses it
 bool consistent = true;

 /// true to stop at the largest of all the lower bounds valid under the
 /// heuristics (those of A*, MM and, with consistent heuristics, BAE*)
 /// instead of the one of the algorithm alone
 bool all_bounds = false;

 /// the direction of the next expansion, for the algorithms that expand
 /// one node at a time
 SideChoice side = SideChoice::automatic;

 /// the fraction \f$ p \in ( 0 , 1 ) \f$ of the optimal path at which
 /// fMM( p ) meets, and the share of the forward search in the split of
 /// GBFHS
 double fraction = 0.5;

 /// for NBS and DVCBS, true to make ready at once the nodes whose
 /// \f$ f \f$ equals the lower bound, which is the version that covers the
 /// pairs needed to find all the optimal paths [Shperberg et al., AAAI
 /// 2019], false for the version of Chen et al. (IJCAI 2017), which makes
 /// them ready one at a time
 bool eager = false;

 };  // end( struct( BidirectionalOptions ) )
```

i.e., those of every search (`SearchOptions`, of which `node_limit` bounds the nodes stored by the two directions together, `expected_nodes` sizes the table of the nodes at the start, and a `weight` other than 1 is refused with `std::invalid_argument`) and the following ones. `epsilon` is the ε of Definition 7.1: a negative value (the default) takes `min_edge_cost()` of a `MinCostDomain`, and 0 for any other domain. `consistent` (true by default) asserts that both heuristics are consistent, which lets a search drop a node that the other direction has closed (Lemma 7.4) and is required by BAE* and by the parallel searches; it is an assumption on the domain that the in-memory searches do not check, and a domain whose heuristics are only admissible must set it to false, or the cost returned may be wrong. `all_bounds` asks a search to stop at the largest of the lower bounds of §7.3 that hold under the heuristics, instead of its own. `side` chooses the direction of the next expansion, among the values of

```
enum class SideChoice {
 automatic ,    ///< the choice of the algorithm as published
 priority ,     ///< the direction whose open list has the least priority
 alternate ,    ///< forward and backward in turn
 cardinality    ///< the direction with fewer open nodes [Pohl, 1971]
 };
```

where `automatic` is the choice of each algorithm as published; the directions themselves are indices into the arrays of a node,

```
enum Side : unsigned {
 forward_side = 0 ,   ///< the search from the start
 backward_side = 1    ///< the search from the goal
 };
```

Finally, `fraction` is the p of fMM(p) and the share of the forward search in the split of GBFHS, and `eager` selects the variant of NBS and DVCBS that covers all the optimal paths (§7.5). Each search of sections 7.4 to 7.8 is a class template with a constructor that takes the domain (which must outlive the search) and the options, and a method `solve()` that returns a `Solution` (the path, its cost, the status and the counters of §2.10); a function of one line, named after the search, builds it and calls `solve()`.

7.2 The two fronts

The searches of sections 7.4 to 7.7 share one table of the nodes, built by the class template `BiFronts` (`Bidirectional.h:331`), and differ only in the order in which they expand its nodes and in the bound at which they stop.

The node, the table and the queues

A node of the table is a `BiNode` (`Bidirectional.h:191`): the state and, for each direction D (the index 0 for F and 1 for B), g_D, h_D, the parent on the path of D and the operator by which D generated the node, with a byte of flags which holds, for each direction, whether the node is closed, whether it has ever been expanded (to count the reexpansions), whether it is open, and whether it is *ready* in the sense of NBS and DVCBS. A node is created by the first direction that reaches its state, with g of the other direction infinite, and h of the other direction is computed when that direction reaches it (or at once, for the searches that need both heuristics of every node, such as BAE*). The two directions therefore look up their states in one table (the open-addressing table of §3.6), and the lookup by which a direction detects its duplicates also tells whether the other direction has reached the state: this is the *immediate solution detection* of Sturtevant and Chen [94], which costs nothing beyond the duplicate detection.

The open lists are lazy, as that of A* (Lemma 3.1): an open list holds entries (n, key, g), not the nodes, and an entry is *stale* when its node has been closed in that direction or reached again with a smaller g since the entry was pushed, or, with `consistent`, when its node has been closed in the other direction; a stale entry is dropped when it reaches the top (`Bidirectional.h:476`), and the entries that are not stale are *fresh*. The class template `BiQueue` (`Bidirectional.h:239`) is such a list, ordered by the key and then by the larger g; it is the two-level bucket queue of §3.3 when the key and the cost are the same integer type (the keys being nonnegative), and the d-ary heap of §3.2 otherwise, e.g., for the real keys of fMM. Neither decreases a key: a node whose key changes is pushed again.

The expansion and the meeting

`BiFronts::start` creates the two roots and opens s forward and t backward; when they are the same state the incumbent is $U = 0$ at once. `BiFronts::expand( d , n , push )` (`Bidirectional.h:391`) closes n in the direction d, enumerates its successors (forward) or its predecessors (backward) by `for_each_child` or `for_each_parent`, prefetching the place of each in the table (§3.8), and then, for each state m so generated, with $g = g_d(n) + c$ and its heuristic h in the direction d, does the following.

1. If $g \geq U$ or $g + h \geq U$, it drops m, since no path through it in this direction is cheaper than the incumbent (with an admissible heuristic).
2. Otherwise it looks m up, creating its node if absent; if the node has $g_d(m) \leq g$ it counts a duplicate and drops it, and otherwise it sets $g_d(m) = g$ and the parent and the operator of d, clears the flag closed of d (a reopening, if it was set), and, if $g_{\bar{d}}(m)$ is finite and $g + g_{\bar{d}}(m) < U$, sets $U = g + g_{\bar{d}}(m)$ and records m as the meeting node.
3. If `consistent` holds and m is closed in the direction $\bar{d}$, it does not open m, since every path through it is known (Lemma 7.4); otherwise it opens m and calls `push`, by which the search inserts m in its own lists.

When the search ends, the path is the chain of the forward parents from the meeting node back to s, reversed, followed by the chain of the backward parents from it to t (`Bidirectional.h:497`). Figure 7.1 shows the two fronts of the bidirectional Dijkstra's algorithm on a grid, before and after they meet.

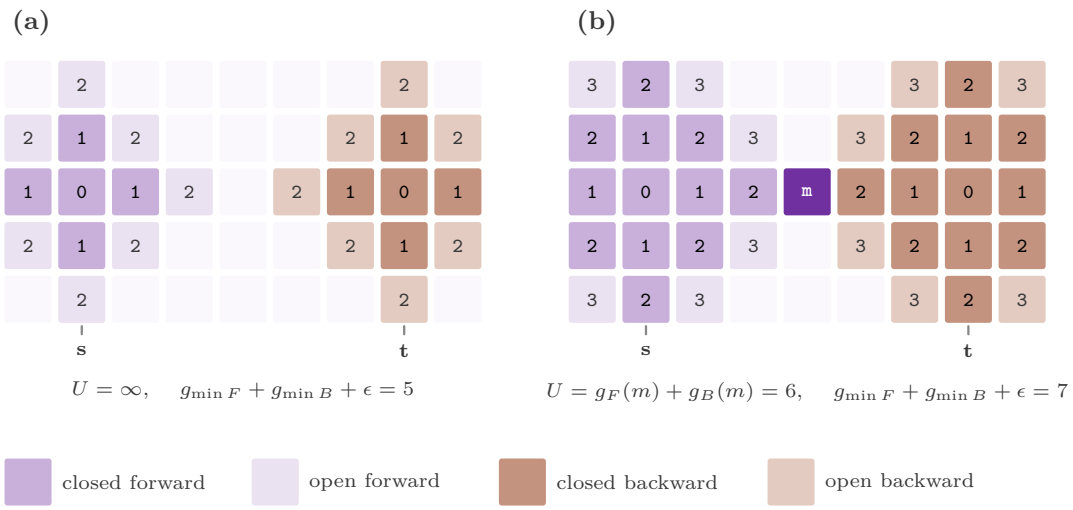


Figure 7.1. The two fronts of a bidirectional search on a grid with 4 moves of unit cost, from s and from t, $C^* = 6$; each cell carries its g in the direction that reached it. (a) After the expansion of the states within distance 1 of each root: no state is reached by both, $U = \infty$, and the bound $g_{\min F} + g_{\min B} + \varepsilon$ of the bidirectional Dijkstra's algorithm is $2 + 2 + 1 = 5$. (b) After those within distance 2: the state m is reached by both, at $g_F(m) = g_B(m) = 3$, which makes $U = 6$, while the bound is now $3 + 3 + 1 = 7 \geq U$, and the search stops.

The invariants

We now state the facts on the table that all the searches of the chapter use. The first is that U is always the cost of a path, and the second, the main one, is that an optimal path always lies between the two fronts while $U > C^*$; it is the argument of Holte et al. (section 4 of [81]), written for any search that detects the meetings at once and reopens a node reached by a cheaper path. In what follows, a search *closes a node unexpanded* when it pops a node whose f in the direction of the pop is not smaller than U, which all the searches do (e.g., `BiBestFirst.h:342`), and a *step* of a search is an expansion or such a closure.

Lemma 7.2 (Values of g and of U). At any time, each finite $g_F(n)$ is the cost of a path from s to n and each finite $g_B(n)$ that of a path from n to t, so that $g_D(n) \geq d_D(n)$; U is ∞ or the value $g_F(m) + g_B(m)$ at some time of a node m, hence $U \geq C^*$; and the parent chains of the meeting node give a path from s to t of cost at most U.

Proof. g_D is set only to 0 at the root of D and to $g_D(n) + c$ along an edge of D from a node n, which by induction is the cost of a path. U is lowered only to $g_F(m) + g_B(m)$ for the node m recorded with it, and a later decrease of $g_F(m)$ or of $g_B(m)$ lowers U again through the same test (step 2 above). The parent of m in D is

set together with $g_D(m) = g_D(n) + c$, and $g_D(n)$ can only decrease afterwards, so that the parent pointers of D form a tree in which the cost of the chain of each node is at most its g_D, as for A* (Lemma 2.10). □

Hence a search that returns U when it can prove $U \leq C^*$ returns an optimal path. Let now $P = (s = p_0, p_1, \ldots, p_k = t)$ be an optimal path, and say that p_l is *settled forward* if it is closed forward with $g_F(p_l) = d_F(p_l)$, and *settled backward* if it is closed backward with $g_B(p_l) = d_B(p_l)$. The flag `consistent` needs the following hypothesis, which Lemma 7.4 proves for the searches of the chapter.

(H) While $U > C^*$, every node of an optimal path from s to t that is closed in a direction D has its optimal g_D.

Lemma 7.3 (An optimal path between the fronts). Let a search expand by the procedure above, and, if it uses the flag `consistent`, let (H) hold. Then, between two steps, if $U > C^*$ there are indices $i < j$ such that p_i is open forward with $g_F(p_i) = d_F(p_i)$ and a fresh entry in the forward open list, and p_j is open backward with $g_B(p_j) = d_B(p_j)$ and a fresh entry in the backward open list. In particular, both open lists have a fresh entry.

Proof. We first remark that no node p_l of P ever has both $g_F(p_l) = d_F(p_l)$ and $g_B(p_l) = d_B(p_l)$ while $U > C^*$: the later of the two assignments finds the other g finite and lowers U to at most $d_F(p_l) + d_B(p_l) = C^*$ (step 2, or the start if $s = t$), and U never increases.

Let i be the least index such that p_i is not settled forward. It exists, since otherwise $t = p_k$ would have $g_F(t) = C^*$ and $g_B(t) = 0$. If $i = 0$, $p_0 = s$ has $g_F(s) = 0$. If $i > 0$, p_{i-1} is settled forward, and it was closed with its present g_F, since a later decrease would have reopened it; it was not closed unexpanded, since $f_F(p_{i-1}) \leq d_F(p_{i-1}) + d_B(p_{i-1}) = C^* < U$ by admissibility, U having been larger than C^* then as now. Hence it was expanded with $g_F(p_{i-1}) = d_F(p_{i-1})$, and it generated p_i with $g = d_F(p_i)$ and $g + h_F(p_i) \leq C^* < U$, so that step 1 did not drop it, and after that expansion $g_F(p_i) = d_F(p_i)$, which by Lemma 7.2 cannot decrease. In both cases p_i has its optimal g_F and is not closed forward, not being settled. When $g_F(p_i)$ took its present value, p_i was either opened and pushed with it or, under `consistent`, found closed backward, hence settled backward by (H), which by the first remark contradicts $U > C^*$. The entry pushed then is fresh: p_i has not been closed forward since (it would be settled forward), nor backward under `consistent` (it would be settled backward by (H)). The symmetric argument gives the largest index j such that p_j is not settled backward, open backward with $g_B(p_j) = d_B(p_j)$ and a fresh entry.

Finally, if $j \leq i$ the node p_j would have both optimal values of g: $g_B(p_j)$ by the choice of j, and $g_F(p_j)$ because p_j is settled forward if $j < i$, and by the argument above if $j = i$. By the first remark this contradicts $U > C^*$, and hence $i < j$. □

We remark that Lemma 7.3 asks nothing of the order of the expansions, and that without the flag `consistent` it holds with heuristics that are only admissible, since a node reached by a cheaper path is then always reopened. The flag is the "nipping" of the BS* of Kwa [90]: a direction that reaches a node closed by the other one does not open it, which is sound when the node has its optimal g in the other direction, and this is what the order of the expansions guarantees under consistent heuristics.

Lemma 7.4 (Closed with the optimal g). Let both heuristics be consistent, and let each direction D of a search pop its nodes by the least key pr_D, ties broken by the larger g_D, where pr_D is nondecreasing along an edge of D and strictly increasing in g_D at a fixed state. (a) Without the flag `consistent`, every node is expanded in each direction with its optimal g, and none is reexpanded. (b) With the flag, (H) holds. The keys f, $\max\{f, 2g\}$, $\max\{f, 2g + \varepsilon\}$, $\max\{f, g/p\}$, g and $b_D = f_D + g_D - h_{\bar{D}}$ have these properties, and (a) and (b) hold also for the orders of NBS, DVCBS and GBFHS.

Proof. The keys are monotone along an edge of cost c: f by consistency (Lemma 2.12), g and $2g$ trivially, and b_D because it changes by $2c + \Delta h_D - \Delta h_{\bar{D}} \geq 2c - c - c = 0$, both differences of the heuristics being at least $-c$ along the edge by consistency; each of them is strictly increasing in g at a fixed state.

(a) Let n be the first node expanded in D with $g_D(n) > d_D(n)$, and Q an optimal path of D from the root of D to n; let x be the first node of Q that is not closed in D with its optimal g_D. The nodes of Q before x were expanded, not closed unexpanded: if one, q, had been closed unexpanded, then $f_D(q) \geq U$ then, and $f_D(n) > d_D(n) + h_D(n) \geq d_D(q) + h_D(q) = f_D(q)$ by consistency along Q, so that n too would be closed unexpanded, U not having increased. Hence the predecessor of x on Q (if any) generated x with $g_D(x) = d_D(x)$ and $f_D(x) \leq d_D(n) + h_D(n) < f_D(n) < U$, so that x is open in D with a fresh entry (no node is dropped as closed by the other direction without the flag), and $x \neq n$. Along Q, $\mathrm{pr}_D(x) \leq \mathrm{pr}_D(n \text{ at } d_D(n)) < \mathrm{pr}_D(n \text{ at } g_D(n))$, so that x is popped before n, a contradiction. A node expanded with its optimal g is never reached again with a smaller one, and is not reexpanded.

(b) Let $p = p_l$ be the first node, over the run, of an optimal path P from s to t that is closed in a direction

while $U > C^*$ with a g larger than its optimal one; say B, the forward case being symmetric. As in the proof of Lemma 7.3, let $a > l$ be the largest index such that p_a is not settled backward; it exists, since otherwise p_{l+1} would have been expanded with its optimal g_B (it is not closed unexpanded, its f_B being at most $C^* < U$) and would have given p_l its optimal g_B, which cannot then increase. Thus p_a has its optimal g_B and is not closed backward; if it was not opened backward, or its entry is stale, because it is closed forward, then by the choice of p it is closed forward with its optimal g_F, and the first remark of the proof of Lemma 7.3 contradicts $U > C^*$. Hence p_a is open backward with a fresh entry and, as in (a), $\mathrm{pr}_B(p_a) < \mathrm{pr}_B(p_l)$, so that it is popped first, a contradiction.

For NBS and DVCBS, a node is expanded from the ready nodes, by least g_D, and every open node with f_D below the bound (C_{lb} or LB) is ready when a pair or a cluster is chosen (Lemma 7.10); since the node x (or p_a) has f_D smaller than that of the node popped, which is at most the bound, it is ready, and its g_D is smaller, so that it precedes the node popped; it cannot be generated by the cluster being expanded, whose children have a g_D at least that of the cluster. For GBFHS the nodes of a level are popped by f, and x (or p_a) has f_D and g_D smaller than those of the node popped, hence within the limits of its level, and it is popped first. □

Hence, under the flag `consistent`, a direction that generates a node of an optimal path closed by the other direction does not open it: its two values of g give the path through it, and the meeting is recorded at the generation (step 2). We remark that (b) does not extend to the nodes off the optimal paths: a node may be expanded with a g larger than its optimal one when the cheaper path to it goes through a node that the flag did not open, which costs at most some search, not the correctness. The test `bidir` runs every search with this flag on the 8-puzzle, the 15-puzzle, the grids and the graphs whose heuristics are consistent, and without it on a graph with an admissible heuristic that is not consistent (that of the test of the reopenings of §2.7), where all the searches but BAE* return the optimal cost.

7.3 The lower bounds

A front-to-end search stops when U is not larger than a lower bound LB on C^*, or when one of its open lists is empty. All the bounds below are computed from the least keys of the open lists, and all of them rest on Lemma 7.3: they need not be lower bounds on C^* at every time, but only while $U > C^*$, which is what the stopping rule needs. We write $f_{\min D}$, $g_{\min D}$, $b_{\min D}$ and $d_{\min D}$ for the least f_D, g_D, b_D and $\delta_D = g_D - h_{\bar{D}}$ over the fresh entries of the open list of D.

Theorem 7.5 (The lower bounds). Under the hypotheses of Lemma 7.3, whenever $U > C^*$ each of the following is at most C^*:

1. $f_{\min F}$ and $f_{\min B}$, the bounds of A*;
2. $g_{\min F} + g_{\min B} + \varepsilon$, the bound of the bidirectional Dijkstra's algorithm;
3. for $p \in (0,1)$, the least key of fMM(p), $C_p = \min\{\min_F \max\{f_F, g_F/p\}, \min_B \max\{f_B, g_B/(1-p)\}\}$, which for $p = 1/2$ is the least key C of MM, and the least key of MMe, $\min_D \min \max\{f_D, 2g_D + \varepsilon\}$;
4. if both heuristics are consistent, $(b_{\min F} + b_{\min B})/2$, the bound of BAE*;
5. if both heuristics are consistent, $f_{\min F} + d_{\min B}$ and $d_{\min F} + f_{\min B}$.

With integer costs each of them may be rounded up.

Proof. Let $n = p_i$ and $m = p_j$ be the nodes of Lemma 7.3, so that $C^* = g_F(n) + d(n,m) + g_B(m)$ and $d(n,m) \geq \varepsilon$, since $i < j$. (1) $f_F(n) = g_F(n) + h_F(n) \leq d(s,n) + d(n,t) = C^*$ by admissibility, and the same for $f_B(m)$. (2) $g_F(n) + g_B(m) + \varepsilon \leq C^*$. (3) Since $g_F(n) + g_B(m) \leq C^*$, either $g_F(n) \leq p\,C^*$ or $g_B(m) \leq (1-p)\,C^*$; in the first case $\max\{f_F(n), g_F(n)/p\} \leq C^*$, by (1), and in the second the same holds for m in B; for MMe, if $g_F(n) \leq g_B(m)$, then $2g_F(n) + \varepsilon \leq g_F(n) + g_B(m) + \varepsilon \leq C^*$, and symmetrically otherwise. (4) By consistency along the subpath from n to m (Lemma 2.3), $h_F(n) \leq d(n,m) + h_F(m)$ and $h_B(m) \leq d(n,m) + h_B(n)$, whence

$$b_F(n) + b_B(m) = 2g_F(n) + 2g_B(m) + (h_F(n) - h_F(m)) + (h_B(m) - h_B(n)) \leq 2(g_F(n) + d(n,m) + g_B(m)) = 2C^*.$$

(5) $f_F(n) + \delta_B(m) = g_F(n) + g_B(m) + h_F(n) - h_F(m) \leq g_F(n) + g_B(m) + d(n,m) = C^*$, and symmetrically. Each least key over the fresh entries is at most the key of n or of m, which have fresh entries. With integer costs C^* is an integer, and a lower bound on it may be rounded up. □

Theorem 7.6 (The stopping rule). Let a search satisfy the hypotheses of Lemma 7.3, and stop when an open list has no fresh entry, or when $U < \infty$ and $U \leq$ LB for a value LB that is at most C^* whenever $U > C^*$. Then, when it stops (and the node limit has not stopped it first), $U = C^*$ if t is reachable from s, and $U = \infty$ otherwise.

Proof. If t is not reachable, $U = \infty$ by Lemma 7.2. Otherwise, if $U > C^*$ when the search stops, both open lists have a fresh entry by Lemma 7.3, and $\mathrm{LB} \leq C^* < U$; neither condition holds, a contradiction. Hence $U \leq C^*$, and $U \geq C^*$ by Lemma 7.2. □

Each search uses its own bound by default, and with `all_bounds` the largest of those that hold under its assumptions: the bounds 1 and 2 always, the bound 3 for MMe, whose key is never smaller than that of MM, and, with `consistent`, the bounds 4 and 5. Of course, all of them are dominated by the bound of the pairs of NBS of §7.5, which, however, needs a search over the pairs of open nodes. We remark that the bounds of Theorem 7.5 are lower bounds on C^* only while $U > C^*$; once $U = C^*$ they may exceed C^*, which is harmless, since they then only stop the search.

7.4 One node at a time: MM, fMM, BAE* and the bidirectional Dijkstra's algorithm

The class template `BiBestFirst< D , P >` (`BiBestFirst.h`) expands, at each step, the node of least priority in the direction chosen, the priority being one of

```
enum class BiPriority {
 dijkstra ,   ///< g, the bidirectional Dijkstra's algorithm
 mm ,         ///< max( f , 2 g ), MM
 mme ,        ///< max( f , 2 g + epsilon ), MMe
 fmm ,        ///< max( f , g / p ) and max( f , g / ( 1 - p ) ), fMM( p )
 bae          ///< f + g - h of the other direction, BAE*
 };
```

i.e., for a node n open in D, respectively,

$$g_D(n), \quad \max\{f_D(n), 2g_D(n)\}, \quad \max\{f_D(n), 2g_D(n)+\varepsilon\}, \quad \max\{f_D(n), g_D(n)/p_D\}, \quad b_D(n) = f_D(n)+g_D(n)-h_{\bar{D}}(n),$$

for the bidirectional Dijkstra's algorithm, MM of Holte et al. [61], [81], its variant MMe of Sharon et al. [82], the fractional MM of Shaham et al. [83], with $p_F = p$ and $p_B = 1 - p$, and BAE* of Sadhukhan [88] as defined by Alcázar et al. [89], whose priority adds to f the error $g_D - h_{\bar{D}}$ of the heuristic of the other direction at the node. Its declarations, without the bodies, are

```
template< BidirectionalDomain D , BiPriority P >
class BiBestFirst {
 public:
 using Key = std::conditional_t< P == BiPriority::fmm , double , Cost >;
 explicit BiBestFirst( const D & domain ,
                       const BidirectionalOptions & options = {} );
 [[nodiscard]] SolutionT solve( void ) const;
 };

template< BidirectionalDomain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > > mm(
 const D & domain , const BidirectionalOptions & options = {} );
```

and the same for `mme`, `fmm` (whose p is `options.fraction`), `bae` and `bidijkstra`, the last being `BiBestFirs t< BlindDomain< D > , BiPriority::dijkstra >`, i.e., the MM_0 of [61], which ignores the heuristics. The constructor throws `std::invalid_argument` for BAE* when `consistent` is false, and for fMM when the fraction is not in $(0, 1)$. The key is real for fMM, whose g/p is not an integer, and of the cost type otherwise.

Algorithm 7.1 (The searches one node at a time, the pseudocode of the header `BiBestFirst.h:29-40`).

```
1  procedure BiBestFirst(s, t)
2     U ← ∞; OPEN_F ← {s}; OPEN_B ← {t}
3     while OPEN_F ≠ ∅ and OPEN_B ≠ ∅ do
4        if U ≤ LB then return U
5        D ← the direction chosen
6        n ← the node of least pr_D in OPEN_D, of largest g_D among those
7        if f_D(n) < U then expand n in the direction D, and
```

```
            U ← min(U, g_F(m) + g_B(m)) for each m generated
8     return U
```

Here OPEN_D is the open list of D by pr_D, its stale entries dropped at the top, and the expansion of line 7 is that of §7.2 (we write $g_F(m) + g_B(m)$ where the header writes $g_D(m) + g_{\bar{D}}(m)$); a node of $f_D(n) \geq U$ is closed unexpanded. The bound of line 4 is

$g_{\min F} + g_{\min B} + \varepsilon$ (Dijkstra), $\max\{C, f_{\min F}, f_{\min B}, g_{\min F} + g_{\min B} + \varepsilon\}$ (MM, MMe, fMM), $\lceil (b_{\min F} + b_{\min B})/2 \rceil$ (BAE

C being the least priority of the two open lists (section 4 of [81]), and the ceiling applying to integer costs only; with `all_bounds` it is the largest of the bounds of Theorem 7.5 that hold. The direction of line 5 is, for `SideChoice::automatic`, the one of least priority (the forward one on a tie, unless the top of the backward list has a smaller g) for MM, MMe, fMM and Dijkstra's algorithm, which is `SideChoice::priority`, and the alternation of the two for BAE*, as in its definition and in PEM-BAE* [57]; `alternate` and `cardinality` (the direction with fewer open nodes, the criterion of Pohl [91]) are available to every priority.

Theorem 7.7 (Correctness, and the meeting in the middle). Algorithm 7.1 returns C^* (or reports t unreachable) for each priority, with admissible heuristics and `consistent` false, or with consistent heuristics; for BAE* the heuristics must be consistent. With the choice of the direction by priority, moreover, MM expands no node with $g_D(n) > C^*/2$, MMe none with $2g_D(n) + \varepsilon > C^*$, and fMM($p$) none with $g_F(n) > p\,C^*$ or $g_B(n) > (1-p)\,C^*$.

Proof. The priorities of the five searches satisfy Lemma 7.4, so that (H) holds under the flag `consistent`, and with it Lemma 7.3; the bound of each is one of Theorem 7.5 (the bound C of MM, MMe and fMM is the bound 3, and that of BAE* the bound 4, which needs consistency), and Theorem 7.6 gives the first claim. For the second, let n be expanded in D. If $U > C^*$ at that time, with the direction of least priority $\mathrm{pr}_D(n) = C \leq C^*$ by Theorem 7.5. If $U \leq C^*$, then $U = C^*$, and the search did not stop at line 4, so that $C \leq \mathrm{LB} < U = C^*$. In both cases $\mathrm{pr}_D(n) \leq C^*$, i.e., $2g_D(n) \leq C^*$ for MM, $2g_D(n) + \varepsilon \leq C^*$ for MMe, and $g_D(n) \leq p_D\,C^*$ for fMM. □

The meeting in the middle depends on the choice of the direction: with `alternate` or `cardinality` the node popped in the direction chosen may have a priority larger than C, and then larger than C^*, so that the second claim of the theorem holds only for `SideChoice::priority` (the default of these searches). The test `bidir` checks it on 10 states of the 8-puzzle, for MM and for fMM with $p = 0.25$ and $p = 0.8$, by recording the states that each direction expands and comparing their distances from s and to t, computed by a breadth-first search, with $p\,C^*$ and $(1-p)\,C^*$; and it checks, with the same distances, that MM, NBS, DVCBS and GBFHS expand no state whose optimal g in the direction plus its heuristic exceeds C^*.

Proposition 7.8 (Termination). On a finite graph with nonnegative costs, Algorithm 7.1 terminates, and so do the searches of sections 7.5 to 7.7.

Proof. Each step pops an entry, and an entry is pushed only after a strict decrease of some $g_D(n)$, which is then the cost of a path (Lemma 7.2). A path of cost at most a given value K has at most K/δ edges of positive cost, δ being the least positive cost of an edge of the finite graph, and therefore the distinct costs of such paths are finitely many; hence each $g_D(n)$ decreases finitely many times, the entries are finitely many, and so are the steps, as in Theorem 2.11. In the procedure that chooses the pair of NBS (or the cluster of DVCBS) each raise of the bound is followed by a return or by the move of an entry from waiting to ready, since the new bound is the least of the quantities that the two tests compare with it, and an entry is moved at most once; GBFHS raises its bound by one unit at a time, and when the bound and the limits exceed the largest f and g of the finitely many paths without repeated states every open node is expandable. □

The implementation

The search keeps, for each direction, the open list by priority and, for the bounds, up to five further lazy queues by f, by g, by the priority of MMe, by b and by $\delta = g - h_{\bar{D}}$, each kept only when the bound of the algorithm (or, with `all_bounds`, one that holds) needs it; e.g., MM keeps the queues by f and by g, and the bidirectional Dijkstra's algorithm none. A node is pushed in each queue kept when its g changes, and the least key of a queue is read after dropping its stale entries, so that each bound costs as much as the open list, an amortized $O(1)$ per entry on the bucket queue (Proposition 3.5) and $O(\log n)$ on the heap. Both heuristics of a node are computed at its creation only for BAE* and for the bounds that need $h_{\bar{D}}$; the other searches compute $h_{\bar{D}}$ only when $\bar{D}$ reaches the node.

7.5 Expanding pairs: NBS and the must-expand graph

For a node u open forward and a node v open backward, $u \neq v$,

$$\mathrm{lb}(u,v) = \max\{f_F(u),\ f_B(v),\ g_F(u) + g_B(v) + \varepsilon\}$$

is a lower bound on the cost of a path from s through u and then v to t (the bounds 1 and 2 of Theorem 7.5 for one pair). The Near-Optimal Bidirectional Search (NBS) of Chen, Holte, Zilles and Sturtevant [85] expands *both* nodes of a pair of least lb, and it is the only search of the chapter with a guarantee on the number of its expansions relative to every other front-to-end search, which we now state.

Definition 7.9 (Must-expand graph). Let both heuristics be consistent. The *must-expand graph* G_{MX} is the bipartite graph with a forward copy and a backward copy of each state, in which the forward copy of u and the backward copy of v are joined if

$$\max\{d_F(u) + h_F(u),\ d_B(v) + h_B(v),\ d_F(u) + d_B(v) + \varepsilon\} < C^*,$$

i.e., if the pair has $\mathrm{lb} < C^*$ when both nodes have their optimal g. We write VC for the size of a minimum vertex cover of G_{MX}.

Theorem 1 of Eckerle et al. [84], as reported by the headers and by Chen et al. [85], states that every admissible front-to-end bidirectional search (one that knows only the values of g, h_F and h_B of its nodes and ε) must expand u forward or v backward for each edge (u,v) of G_{MX}, since otherwise a path of cost $\mathrm{lb} < C^*$ joining them could exist on an instance indistinguishable from the given one; hence VC is a lower bound on the expansions of every such search.

Algorithm 7.2 (The choice of the pair of NBS, the pseudocode of the header `NBS.h:35-45`).

```
1  procedure PrepareBest
2     move from waiting_D to ready_D every node with f_D < C_lb
3     loop
4        if ready_D ∪ waiting_D = ∅ for some D then return none
5        if ready_F.g + ready_B.g + ε ≤ C_lb then return the two tops
6        if waiting_D.f ≤ C_lb for some D then
            move the top of waiting_D to ready_D (for each such D)
7        else C_lb ← min(waiting_F.f, waiting_B.f,
                         ready_F.g + ready_B.g + ε)
```

The pair is found as in section 7 of [85], with a lower bound C_{lb} on the least lb raised until a pair reaches it: in each direction D, a queue waiting$_D$ of the open nodes by f_D, the larger g_D first on a tie, and a queue ready$_D$ of those with $f_D \le C_{lb}$, by g_D and then by the smaller f_D; `ready_D.g` is the least g of ready$_D$ and `waiting_D.f` the least f of waiting$_D$, both over the fresh entries. The search calls PrepareBest, stops if it returns none or if $U \le C_{lb}$, and otherwise expands the top of ready$_F$ and then that of ready$_B$, each closed unexpanded if its f is not below U (`NBS.h:234-246`). With `eager`, line 2 moves the nodes with $f_D \le C_{lb}$, which is the NBS$_A$ of Shperberg et al. [86], whose pairs cover those with $\mathrm{lb} \le C^*$, needed to find all the optimal paths, and line 6 is then never taken; with $\varepsilon = 0$ the bound is that of Chen et al. without it (the base case of Shperberg et al.). The declarations are those of `BiBestFirst` without the parameter `P`, i.e., `NBS< D >( domain , options )` with `solve()`, and `nbs( domain , options )`.

Lemma 7.10 (The bound of the pairs). (a) Within a call of PrepareBest, after line 2, every node of waiting$_D$ has $f_D \ge C_{lb}$ (with `eager`, $f_D > C_{lb}$), so that every open node with $f_D < C_{lb}$ is ready, and C_{lb} strictly increases at each execution of line 7. (b) Each time C_{lb} is set at line 7, it is at most the least lb of the pairs of open nodes, and hence at most C^* if $U > C^*$. (c) A pair returned at line 5 has $\mathrm{lb}(u,v) \le C_{lb}$.

Proof. (a) Line 2 leaves in waiting$_D$ the nodes with $f_D \ge C_{lb}$, and line 7 is reached only when the tops of waiting have $f > C_{lb}$ (line 6) and the ready tops have $g_F + g_B + \varepsilon > C_{lb}$ (line 5), so that the new value is larger and is again at most the least f of waiting; no node enters waiting within the call. (b) Let u be open forward and v open backward, each with a fresh entry in ready or in waiting. If u or v is waiting, $\mathrm{lb}(u,v)$ is at least its f, which is at least the least f of its waiting queue; if both are ready, $\mathrm{lb}(u,v) \ge g_F(u) + g_B(v) + \varepsilon$, which is at least the sum of the least values of g of the two ready queues plus ε. The new C_{lb} is the minimum of these quantities. If $U > C^*$, the pair (p_i, p_j) of Lemma 7.3 has $\mathrm{lb} \le C^*$ (Theorem 7.5, bounds 1 and 2), whence $C_{lb} \le C^*$. (c) A node enters ready$_D$ with $f_D \le C_{lb}$, C_{lb} only grows, and the f of a ready node does not change while its entry is fresh; the condition of line 5 bounds the third term of lb. □

Theorem 7.11 (Correctness and near-optimality of NBS). NBS returns C^* (or reports t unreachable) under the

hypotheses of Theorem 7.7. If both heuristics are consistent and no node is reexpanded (which holds with `consistent` false, by Lemma 7.4(a)), the pairs that it expands with $\mathrm{lb} < C^*$ form a matching of G_{MX}, so that it expands at most 2 VC nodes in them; the other pairs expanded have $\mathrm{lb} = C^*$.

Proof. NBS satisfies Lemma 7.4, so (H) and Lemma 7.3 hold, and by Lemma 7.10(b) the value of C_{lb} at the test $U \le C_{lb}$ is at most C^* whenever $U > C^*$, since U only decreases after C_{lb} is set; Theorem 7.6 gives the correctness. Let the heuristics be consistent, and (u, v) be a pair expanded. Since $g_F(u) \ge d_F(u)$ and $g_B(v) \ge d_B(v)$, the quantity of Definition 7.9 is at most $\mathrm{lb}(u, v)$, and (u, v) is an edge of G_{MX} if $\mathrm{lb}(u, v) < C^*$. If no node is reexpanded, each state is expanded at most once in each direction, and two pairs expanded share no forward copy and no backward copy: they form a matching M. Every vertex cover of G_{MX} contains an end of each edge of M, which are disjoint, hence $|M| \le \mathrm{VC}$, and the pairs of M make at most $2|M| \le 2\,\mathrm{VC}$ expansions. Finally, a pair is expanded only if $U > C_{lb} \ge \mathrm{lb}(u, v)$ by Lemma 7.10(c); if $U = C^*$ this gives $\mathrm{lb} < C^*$, and if $U > C^*$ it gives $\mathrm{lb} \le C_{lb} \le C^*$. □

This is theorem 3 of [85], and their theorem 5 shows that no admissible front-to-end search can guarantee less than 2 VC in the worst case. With the flag `consistent`, a node off the optimal paths may be expanded with a g larger than its optimal one (see the remark after Lemma 7.4) and later reexpanded, which the counter `reexpanded` of the `Solution` reports; the bound of the theorem holds for the runs in which it is 0. We also remark that the two expansions of a step are those of the two tops returned by PrepareBest: the forward expansion cannot make the backward top stale, since $u \ne v$ (if $u = v$, the node would have both values of g, and $U \le g_F(u) + g_B(u) \le C_{lb}$ would have stopped the search), and it writes only the forward fields.

7.6 Expanding clusters: DVCBS

NBS expands both ends of one edge of G_{MX} at a time, which is optimal only up to the factor 2; the Dynamic Vertex Cover Bidirectional Search (DVCBS) of Shperberg, Felner, Sturtevant, Shimony and Hayoun [86] instead expands, at each step, one vertex of a minimum vertex cover of the part of the must-expand graph that it knows. The class template `DVCBS< D >` (`DVCBS.h`), with the function `dvcbs( domain , options )`, keeps the high level of NBS: a lower bound LB on C^*, raised until $U \le \mathrm{LB}$, the queues waiting$_D$ by f_D, and the ready nodes, those with $f_D < \mathrm{LB}$ ($\le$ with `eager`). The nodes move from waiting$_D$ to ready$_D$ one value of f_D at a time, and the ready nodes are grouped in *clusters*, one per value of g_D. The clusters are the vertices of the *dynamic must-expand graph* DGMX, a forward cluster of $g_F = a$ and a backward one of $g_B = b$ being joined if $a + b + \varepsilon \le \mathrm{LB}$, which makes every pair of their nodes have $\mathrm{lb} \le \mathrm{LB}$; and DVCBS expands one cluster of a minimum vertex cover of DGMX, weighted by the sizes of the clusters (algorithm 3 of [86]). The header has no pseudocode of its own: the choice of the clusters is that of Algorithm 7.2 with the moves of whole values of f and the test of line 5 on the least g of the clusters (`DVCBS.h:235-272`), and the cover is computed as follows.

Let $a_0 < a_1 < \cdots < a_{m-1}$ be the values of g of the forward clusters with an edge and $b_0 < \cdots < b_{n-1}$ those of the backward ones, with weights $w(a_i)$ and $w(b_j)$, and let $N(a_i) = \{b_j : a_i + b_j + \varepsilon \le \mathrm{LB}\}$, a prefix of the backward clusters.

Lemma 7.12 (The covers of DGMX). For $0 \le i \le m$, the set $K_i = \{a_0, \ldots, a_{i-1}\} \cup N(a_i)$ (with $N(a_m) = \emptyset$) is a vertex cover of DGMX, and every vertex cover K contains some K_i; hence a minimum cover is a K_i of least weight $\sum_{l<i} w(a_l) + \sum_{b \in N(a_i)} w(b)$.

Proof. The neighbourhoods are nested: $N(a_{i'}) \subseteq N(a_i)$ for $i' > i$, since $b \le \mathrm{LB} - \varepsilon - a_{i'} \le \mathrm{LB} - \varepsilon - a_i$. An edge (a_l, b) is covered by K_i through a_l if $l < i$, and through $b \in N(a_l) \subseteq N(a_i)$ if $l \ge i$; hence K_i is a cover. Given a cover K, let i be the least index with $a_i \notin K$ ($i = m$ if there is none); then $a_0, \ldots, a_{i-1} \in K$, and every edge of a_i is covered by its backward end, i.e., $N(a_i) \subseteq K$, so that $K_i \subseteq K$, and the weight of K_i is at most that of K. □

The structure of the covers was shown by Shaham et al. [83] for the must-expand graph, and it makes the minimum cover a pass over $i = 0, \ldots, m$, with a pointer to the end of $N(a_i)$ that only moves down and the prefix sums of the backward weights, in time linear in the number of clusters (`DVCBS.h:199-232`). Among the minimum covers, the cluster expanded is the forward a_0 or the backward b_0 that lies in one of them, the one with fewer nodes and the backward one on a tie, which is the variant reported by Shperberg et al. (section 5.1 of [86]); its nodes are expanded in the order in which they became ready, each closed unexpanded if its f is not below U, and the expansion of the cluster stops as soon as $U \le \mathrm{LB}$. Figure 7.2 shows the clusters, the edges of DGMX and a minimum cover.

Proposition 7.13 (Correctness of DVCBS). DVCBS returns C^* (or reports t unreachable) under the hypotheses of Theorem 7.7.

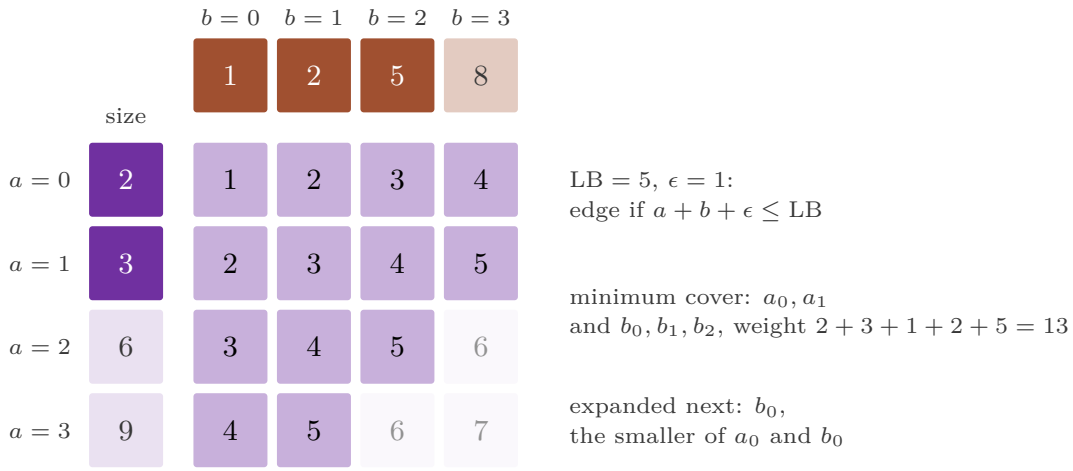


Figure 7.2. The pairs of NBS and the clusters of DVCBS. The forward clusters of $g_F = a$ (rows) and the backward ones of $g_B = b$ (columns) carry their sizes, and each cell the bound $a + b + \varepsilon$ of the pairs of its two clusters, with $\varepsilon = 1$; with LB = 5 the cells within it (darker) are the edges of DGMX, a staircase since $N(a_i)$ is a prefix that shrinks as i grows. Of the covers $K_0, \ldots, K_4$ of Lemma 7.12, of weights 16, 18, 13, 14 and 20, the least is $K_2 = \{a_0, a_1, b_0, b_1, b_2\}$; both a_0 and b_0 lie in it, and DVCBS expands the smaller, b_0.

Proof. Lemma 7.10 holds for LB with the clusters in place of the ready queues, with the same proof, since the moves of a whole value of f keep the nodes of waiting at $f \geq$ LB; the cluster expanded is a cluster of least g of its direction, so that Lemma 7.4 holds, and with it Lemma 7.3; the test $U \leq$ LB is made after each node of a cluster, and Theorem 7.6 applies. □

DVCBS has no bound of the kind of Theorem 7.11 on its expansions (section 5.2 of [86]), but it expanded fewer nodes than NBS on every domain of their table 1. The test `bidir` runs it, as NBS, with each variant of the options (all the bounds, eager, $\varepsilon = 0$, and the choices of the direction, which these two searches ignore), and checks that it never expands a node of f above C^*.

The implementation

A cluster is a vector of nodes in the order in which they became ready, with a count of its live nodes, and the clusters of a direction are a map by g (`DVCBS.h:123-134`). A ready node whose g decreases leaves its cluster (the count is decremented, and the cluster erased when it reaches 0) and goes back to waiting; under `consisten t` a node expanded in D leaves its cluster of $\bar{D}$ as well. Since a node may also die in its cluster by a closure in the other direction, whose leaving is not counted, the cluster expanded is erased as a whole at the end. The weights used by the cover are the live counts, i.e., the sizes of the clusters up to these dead nodes.

7.7 GBFHS and the bidirectional breadth-first search

The Generalized Breadth-First Heuristic Search (GBFHS) of Barley et al. [87] works with integer costs and raises a bound f_{Lim} on C^* one unit at a time; at each value it splits $f_{Lim} - \varepsilon + 1$ into two limits $g_{LimF} + g_{LimB}$ on the depths of the two searches, and it expands, in each direction D, every open node with $f_D \leq f_{Lim}$ and $g_D < g_{LimD}$.

Algorithm 7.3 (GBFHS, the pseudocode of the header `GBFHS.h:16-27`).

```
1  procedure GBFHS(s, t)
2     U ← ∞; f_Lim ← max(h_F(s), h_B(t), ε); g_LimF ← g_LimB ← 0
3     while OPEN_F ≠ ∅ and OPEN_B ≠ ∅ do
4        if U ≤ f_Lim then return U
5        raise g_LimF and g_LimB so that g_LimF + g_LimB = f_Lim − ε + 1
6        while some n in OPEN_D has f_D(n) ≤ f_Lim and g_D(n) < g_LimD do
7           expand the one of least f_D, of largest g_D among those,
            of the direction of the least such f
8           if U ≤ f_Lim then return U
9        f_Lim ← f_Lim + 1
```

The class template `GBFHS< D >` requires `std::is_integral_v< cost_t< D > >`, as do `gbfhs( domain , options )` and `bibfs( domain , options )`, the latter being GBFHS on the `BlindDomain` with each unit of depth given to the direction with fewer open nodes, i.e., the bidirectional breadth-first search with the criterion of Pohl [91]. The split of line 5 is chosen by `side`: `alternate`, and also `automatic`, the default, give each new unit to the smaller limit (the backward one on a tie) after a first split in halves; `cardinality` gives it to the direction with fewer open nodes; and `priority` gives the forward limit the share `fraction` of the sum, $\lfloor p\,(f_{Lim} - \varepsilon + 1)\rfloor$, clamped so that no limit decreases, as fMM(p) does (`GBFHS.h:177-206`).

Lemma 7.14 (A level finds its paths). Let the costs be integers, ε a lower bound on them, and the hypotheses of Lemma 7.3 hold. When the loop of lines 6 to 8 ends at the level f_{Lim}, every path P from s to t of cost $C \le f_{Lim}$ has $U \le C$.

Proof. Let $P = (p_0, \ldots, p_k)$, and suppose $U > C$. Let a be the least index such that p_a is not closed forward with $g_F(p_a)$ at most the cost of the prefix of P up to p_a, and b the largest index such that p_b is not closed backward with $g_B(p_b)$ at most the cost of the suffix of P from p_b. As in the proof of Lemma 7.3, with the costs of the prefixes in place of the optimal values of g and $C < U$ in place of $C^* < U$, p_a has $g_F(p_a)$ at most the cost of its prefix, is not closed forward and is open forward with a fresh entry, and symmetrically for p_b; and if $a \ge b$ the node p_b has both values of g within the costs of its prefix and of its suffix, whose sum, at most C, was offered to U. Hence $a < b$. Then $f_F(p_a) \le g_F(p_a) + d(p_a, t) \le C \le f_{Lim}$ by admissibility, and since the level ended, $g_F(p_a) \ge g_{LimF}$; symmetrically $g_B(p_b) \ge g_{LimB}$; and since P has at least one edge between p_a and p_b,

$$C \ge g_F(p_a) + \varepsilon + g_B(p_b) \ge g_{LimF} + g_{LimB} + \varepsilon = f_{Lim} + 1,$$

a contradiction. □

Theorem 7.15 (Correctness of GBFHS). Under the hypotheses of Lemma 7.14, GBFHS returns C^* (or reports t unreachable), $f_{Lim} \le C^*$ at every expansion, and hence it expands no node with $f_D > C^*$.

Proof. If $s = t$, $U = 0$ stops it at once. Otherwise $f_{Lim} \le C^*$ at the start, since $h_F(s) \le C^*$, $h_B(t) \le C^*$ and $\varepsilon \le C^*$. While $f_{Lim} < C^*$, $U \ge C^* > f_{Lim}$ (Lemma 7.2) and the search does not stop at lines 4 or 8; at the end of the level $f_{Lim} = C^*$, if it is reached, Lemma 7.14 gives $U \le C^*$, i.e., $U = C^*$, and the search returns it at line 4 of the next iteration, or earlier, within the level C^*, with $U \le f_{Lim} = C^*$. Hence every expansion happens at a level $f_{Lim} \le C^*$, of a node with $f_D \le f_{Lim}$. An empty open list stops it with $U = C^*$ or $U = \infty$ by Lemma 7.3 and Theorem 7.6. □

The order of line 7 is ours: Barley et al. leave free the order of the expandable nodes of a level, and the order of A* (least f, then largest g) reaches the meeting point within the last level much earlier than the order of g; the header reports 2.5 times fewer expansions on the 15-puzzle, a measure that we have not repeated here. Under consistent heuristics this order satisfies Lemma 7.4, which makes the flag `consistent` sound.

The implementation

Each direction has three lazy queues: waiting, the nodes with $f > f_{Lim}$ by f; ready, those with $f \le f_{Lim}$ by f and then by the larger g; and deep, those of ready whose g reached the limit, by g. A node at the top of ready with $g_D \ge g_{LimD}$ goes to deep, and comes back to ready when the limit passes it (`GBFHS.h:163-174`); thus a level is a sequence of moves between the queues and of expansions, each move being paid once per entry and per change of the limits. We remark that f_{Lim} grows by one unit even when no open node has its f near it, so that on a domain with large costs a run may pass through many empty levels, each of a constant cost.

7.8 The two directions in parallel

The class template `ParallelBidirectional< D , K , Executor >` (`ParallelBidirectional.h`) runs the two directions at once, each on a worker of an executor (the executors of §1.6), sharing the table of the nodes and the incumbent. The search of a worker is one of

```
enum class ParallelBiKind {
 nba ,        ///< NBA*: by f, with the two rejections (PNBA*)
 dijkstra ,   ///< by g, the bound g_F + g_B + epsilon
 mm ,         ///< by max( f , 2 g ), the bound of MM
 bae          ///< by b, the bound of BAE*
```

```
};
```

where `nba` is the NBA* of Pijls and Post [92], which expands by f and rejects some nodes as described below, so that the search is PNBA* of Rios and Chaimowicz [93], and the other three are the parallel versions of the bidirectional Dijkstra's algorithm, of MM and of BAE*. Its declarations are

```
template< BidirectionalDomain D , ParallelBiKind K ,
          ADS::WorkerExecutor Executor = ADS::ThreadPool >
class ParallelBidirectional {
 public:
 explicit ParallelBidirectional( const D & domain ,
                                 const BidirectionalOptions & options = {} );
 void set_executor( Executor & ex ) noexcept;
 [[nodiscard]] SolutionT solve( void ) const;
 };

template< BidirectionalDomain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > > pnba(
 const D & domain , const BidirectionalOptions & options = {} );
```

The constructor refuses `consistent` false and a weight other than 1 with `std::invalid_argument`, since the scheme needs consistent heuristics; without `set_executor` each search builds an executor of two threads. The counters `expanded_by` and `stored_by` have one entry per direction, and the node limit is checked every 64 expansions, a worker stopping both when twice the nodes it has created reach it.

Each worker runs the best-first search of its direction on its own open list, writes only the fields of its direction in the nodes (g_D, the parent, the operator and the flag closed$_D$), and reads those of the other; the values of g and the flags closed are accessed through atomic references, with sequentially consistent operations, and both workers lower U, with its meeting node, under a lock (`ParallelBidirectional.h:264`). After each expansion a worker publishes the least key of its open list and, for MM, its least f and its least g, each as the maximum of the value published before and the present one (`ParallelBidirectional.h:392`); it stops both workers when its open list is empty, or when U is not larger than the bound of the algorithm computed from the two published records (`ParallelBidirectional.h:275`), which is the bound of `BiBestFirst` for Dijkstra's algorithm, MM and BAE*, and $\max\{f_{\min F}, f_{\min B}\}$ for NBA*. Three facts make the scheme exact.

Lemma 7.16 (No meeting is missed). If both directions set $g_F(m)$ and $g_B(m)$ of a node m to finite values, then at least one of the two workers reads the value set by the other, and offers their sum to U.

Proof. When the worker of D sets $g_D(m)$ it then reads $g_{\bar{D}}(m)$, and the worker of $\bar{D}$ does the converse (`ParallelBidirectional.h:445-449`). With sequentially consistent accesses there is a total order of the four operations compatible with the order of each worker; the worker whose write comes second in that order reads after both writes, and hence reads the value of the other. A node created by one worker with its g is published by its insertion in the concurrent table, so that the other worker that finds it sees its fields. This is the argument of the mutual exclusion of Dekker. □

Lemma 7.17 (The published keys). With consistent heuristics, the least key of the nodes open or being expanded in a direction never decreases, and so each value published by a worker, however old, is at most the least key of the nodes open in its direction at any later time.

Proof. Each worker is a sequential best-first search of its direction whose key satisfies Lemma 7.4. A node popped has the least key of the open list, and its children have keys not smaller than its own, by the monotonicity along an edge; the nodes that the other worker closes, or that U prunes, only leave the list. Hence the least key over the open nodes and the node being expanded never decreases, and a published value, which is a least key at the time of its publication (or an earlier one, by the maximum), stays below it. □

Lemma 7.18 (The rejections of NBA*). Let the heuristics be consistent, $F_{\bar{D}}$ be a value published by the other direction, and x be popped in D with $g_D(x) + F_{\bar{D}} - h_{\bar{D}}(x) \geq U$. Then no path from the root of D to x of cost $g_D(x)$, continued through a node open in $\bar{D}$, costs less than U.

Proof. Let y be open in $\bar{D}$, and $d(x, y)$ the cost of the subpath between them; by consistency $h_{\bar{D}}(y) \leq d(x, y) + h_{\bar{D}}(x)$, so that a path through x and y costs at least

$$g_D(x) + d(x, y) + g_{\bar{D}}(y) \geq g_D(x) + f_{\bar{D}}(y) - h_{\bar{D}}(x) \geq g_D(x) + F_{\bar{D}} - h_{\bar{D}}(x) \geq U,$$

since $f_{\bar{D}}(y) \geq F_{\bar{D}}$ by Lemma 7.17. □

NBA* closes unexpanded the node x popped in D when $f_D(x) \geq U$ or when $g_D(x) + F_{\bar{D}} - h_{\bar{D}}(x) \geq U$ (section 3 of [92]), the second being the *rejection* of Lemma 7.18 (`ParallelBidirectional.h:472-476`).

Theorem 7.19 (Correctness of the parallel searches). With consistent heuristics, `ParallelBidirectional` returns C^* for each kind, or reports t unreachable, unless the node limit stops it.

Proof. By Lemma 7.16, every node that gets both values of g lowers U to at most their sum, which is the first remark of the proof of Lemma 7.3; each direction pops by a key of Lemma 7.4, so that (H) holds, its proof being the same with the two workers interleaved; and a node of an optimal path is neither pruned nor rejected while $U > C^*$, since $f_D \leq C^* < U$ and, by Lemma 7.18 with the node of the other direction given by Lemma 7.3, $g_D(x) + F_{\bar{D}} - h_{\bar{D}}(x) \leq C^* < U$. Hence Lemma 7.3 holds at any time at which a worker reads its open list: if $U > C^*$, the open list of each direction has a node of P with its optimal g. A worker that finds its list empty thus has $U \leq C^*$. A worker that finds $U \leq \mathrm{LB}$, LB being computed from the published values, has by Lemma 7.17 an LB that is at most the bound of Theorem 7.5 computed from the present open lists (each bound being a nondecreasing function of the least keys), and hence at most C^* if $U > C^*$; Theorem 7.6 concludes. □

The test `bidir-thread` (and `bidir-openmp`, `bidir-fastflow` on the other executors) runs the four kinds on 50 states of the 8-puzzle, on two instances of the 15-puzzle of Korf, on random grids with 8 moves and on random directed graphs, half of the runs on the 8-puzzle with an executor of their own and the others with one given. A parallel bidirectional search has two workers, so that its speedup is at most 2, and it pays the overheads of §8.2: the search overhead of a bound read late (a worker may expand nodes that a sequential search would have pruned by a bound already reached) and the coordination overhead of the atomic accesses to the shared nodes.

7.9 Parallel external-memory bidirectional search

The class template `PEMSearch< D , K , Executor >` (`PEMSearch.h`) is the framework PEM-BiHS of Siag, Shperberg, Felner and Sturtevant [57], whose open and closed lists are files on the disk, the nodes of one bucket being loaded, checked for duplicates and expanded by the workers of an executor. The search is one of

```
enum class PEMKind {
 astar ,      ///< PEM-A*, forward only
 mm ,         ///< PEMM
 bae          ///< PEM-BAE*
 };
```

i.e., PEM-A* (forward only), PEMM, the external MM of Sturtevant and Chen [94], and PEM-BAE*; the reverse PEM-rA* is PEM-A* on a `ReverseDomain`. The domain must be a `DiskDomain` of §4.6 (integer costs and trivially copyable states), and a `BidirectionalDomain` for PEMM and PEM-BAE*. The declarations are

```
template< DiskDomain D , PEMKind K ,
          ADS::WorkerExecutor Executor = ADS::ThreadPool >
 requires( ( K == PEMKind::astar ) || BidirectionalDomain< D > )
class PEMSearch {
 public:
 PEMSearch( const D & domain , unsigned threads = 1 ,
            const SearchOptions & options = {} , PEMOptions pem = {} );
 void set_executor( Executor & ex ) noexcept;
 [[nodiscard]] SolutionT solve( void ) const;
 };
```

with the options of the disk

```
struct PEMOptions {

 /// the partitions of a bucket by the hash of the state
 std::size_t partitions = 16;

 /// where the files go, in a directory of their own removed at the end;
 /// the temporary directory of the system if empty
```

```
std::filesystem::path directory;

/// the bytes of the buffer of each file being written
std::size_t buffer_bytes = std::size_t( 1 ) << 16;

/// the most bytes the files may hold on the disk (0 = no cap)
std::uint64_t disk_limit = 0;

};  // end( struct( PEMOptions ) )
```

i.e., the number P of partitions of a bucket (16 by default, 0 being refused), the directory of the files (a directory of their own is made in it, or in the temporary directory of the system, and removed at the end, also when the search stops at a limit or throws), the bytes of the buffer of each file being written, and a cap on the bytes held on the disk, at which the search stops with `Status::disk_limit`. The number of workers is `threads`; of the `SearchOptions`, `node_limit` bounds the states written to the closed files, and a weight other than 1 is refused. The counters are `expanded` (the states expanded, and `expanded_by` by direction), `generated`, `duplicates` (the copies dropped), `stored` (the states written to the closed files), `sent` (the bytes written to the disk) and `messages` (here the most bytes held on the disk at once).

The buckets and the meetings

A node of the direction D is kept in the bucket (D, g_D, h_F, h_B) (with $h_B = 0$ for PEM-A*), which carries all that the priorities and the bounds need, and the bucket is split into P partitions by the hash of the state, a partition being a file per worker that writes to it. Two copies of a state, in either direction, have the same h_F and h_B and the same partition, so that the duplicates of a partition are in the same partition of the buckets with the same heuristics, and so are the copies of the state in the other direction.

Algorithm 7.4 (PEM-BiHS, the pseudocode of the header `PEMSearch.h:23-39`).

```
 1  procedure PEM-BiHS(s, t)
 2     U ← ∞; OPEN_F ← {s}; OPEN_B ← {t}
 3     while OPEN_F ≠ ∅ ∧ OPEN_B ≠ ∅ ∧ U > LB do
 4        D ← the direction chosen; b ← the best bucket of OPEN_D
 5        for each partition p of b, in parallel, do
 6           T ← the states of the files of p, without copies
 7           T ← T \ the partition p of the closed buckets of D with the
                same h_F and h_B and a smaller g
 8           write T to the partition p of CLOSED_D(b)
 9           U ← min(U, g + g') for each state of T found, with g', in
                the partition p of the buckets of the other direction
                with the same h_F and h_B
10           for each u in T and each child v of u in D do
                append v to the partition of v of its bucket of OPEN_D
11     return U
```

Lines 6 to 8 are the processing of a file of DDD-A* (Algorithm 4.6), here for a partition made of the files of all the workers; line 9 is the *delayed solution detection* of Sturtevant and Chen [94]: a meeting is found when one of its two copies is expanded, by a scan of the open and the closed buckets of the other direction with the same heuristics, in the same partition (Figure 7.3). PEM-A* instead tests the goal when a node is generated, as Siag et al. do. The best bucket of a direction is the one of least priority (f for PEM-A*, $\max\{f, 2g\}$ for PEMM, b for PEM-BAE*), of least g among those, and then of least h of the direction, the lower-g-first order of Siag et al. (section 3.3 of [57]); PEMM takes the direction of least priority, the forward one on a tie, and PEM-BAE* alternates them. The partitions of a bucket are taken by the workers from a shared counter, and each worker writes the children through a buffer per file of the buckets it fills, the new buckets being added to the open ones when the bucket is done (`PEMSearch.h:519-684`). The path is traced back at the end through the hashes of the parents, as by Algorithm 4.7, from each of the two records of the meeting.

Since a meeting is found only when one of its copies is expanded, two open copies of a state on an optimal path may be unknown to each other, and the bound $g_{\min F} + g_{\min B} + \varepsilon$ is not valid (section 3.3 of [94]). The bound of line 3 is therefore $f_{\min F}$ for PEM-A*,

$$\max\{C, f_{\min F}, f_{\min B}, g_{\min F} + g_{\min B}\}$$

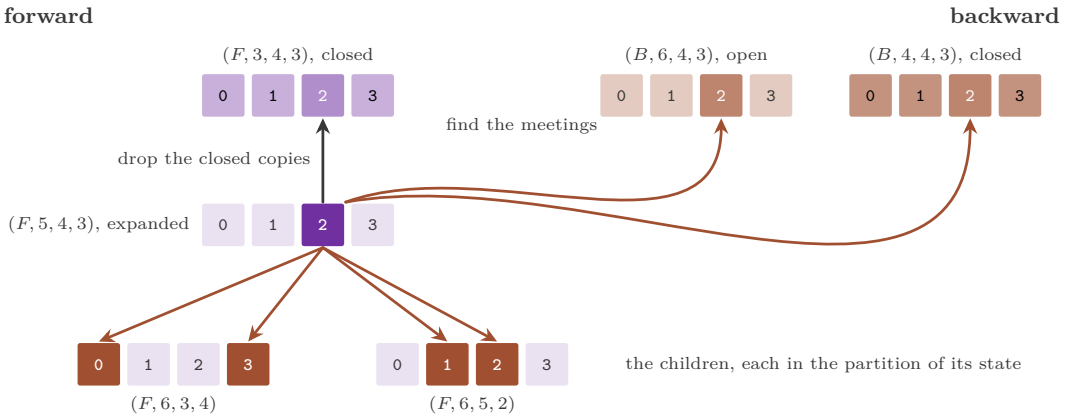


Figure 7.3. The buckets (D, g, h_F, h_B) of PEM-BiHS on the disk, each split into $P = 4$ partitions by the hash of the state. The partition 2 of the forward bucket $(F, 5, 4, 3)$ is loaded and cleared of its copies and of the states of the partition 2 of the closed forward buckets of the same heuristics and a smaller g (here $(F, 3, 4, 3)$); its states are looked up in the partition 2 of the backward buckets of the same heuristics, open and closed, to find the meetings; and each child is appended to the partition of its state in the bucket of its g and heuristics, by the worker that generated it.

for PEMM, and, for PEM-BAE*, the largest of $\lceil (b_{\min F} + b_{\min B})/2 \rceil$, $g_{\min F} + g_{\min B}$, $f_{\min F} + d_{\min B}$ and $d_{\min F} + f_{\min B}$ (the bounds of Alcázar et al. [89], which the PEM-BAE* of Siag et al. computes), all from the keys of the open buckets (`PEMSearch.h:380-406`).

Lemma 7.20 (The order of the buckets). Let the costs be positive and the key of a bucket be nondecreasing along every edge generated, which the search checks. Then, in each direction, no bucket gets a record after its expansion, and the copy of a state with its optimal g in that direction, if generated, is expanded before every other copy of it, all the later ones being dropped at line 7.

Proof. Let the bucket b of D be expanded. The children of b and of every later bucket of D have keys not smaller than their parents and, for equal keys, a larger g (the costs being positive); since the best bucket is the one of least key and then of least g, the pairs (key, g) of the buckets expanded in D never decrease, and a child of a later bucket b' has either a key larger than that of b or the same key and $g > g(b') \geq g(b)$, so that it is not in b. For the second claim, the copies of a state v have the same heuristics, so that in its buckets the key is a strictly increasing function of g (as in Lemma 7.4); the copy with the optimal g is in the bucket of least key among them, and, by the induction along an optimal path of Lemma 4.14, whose argument applies to any key monotone along the edges, it is there when that bucket is expanded. A later copy, of larger g, is in a later bucket of the same heuristics and the same partition, and line 7 finds v in the closed partition. □

The search throws `std::logic_error` on an edge of a cost that is not positive and on a child whose bucket has a smaller key than its parent (`PEMSearch.h:638`): for PEM-A* the second is a heuristic that is not consistent, and the test `pem-thread` checks the refusal on the graph of §2.7. For PEMM and PEM-BAE* the check is on their key, $\max\{f, 2g\}$ or b, and a heuristic that is not consistent but keeps the key monotone on the edges generated is not refused; for PEM-BAE* its bounds then need not hold, and the consistency of both heuristics is an assumption of the search, as the header states.

Lemma 7.21 (Delayed meetings). Let the hypotheses of Lemma 7.20 hold, $P = (p_0, \ldots, p_k)$ be an optimal path, and $U > C^*$ between two buckets. Then there are $i \leq j$ such that p_i has a copy in an open forward bucket with $g_F = d_F(p_i)$, and p_j one in an open backward bucket with $g_B = d_B(p_j)$.

Proof. Say that p_l is settled in D if its copy with the optimal g has been expanded in D. If p_l is settled in both directions, the later of the two expansions scanned the closed partition that holds the earlier one, and set $U \leq C^*$. As in Lemma 7.3, the first node p_i not settled forward has its optimal copy in an open forward bucket (generated by p_{i-1} with $g + h_F \leq C^* < U$, and not dropped at line 7 by Lemma 7.20), and the last node p_j not settled backward has its optimal copy in an open backward bucket. Suppose $j < i$: then p_j is settled forward and, if $j + 1 < i$, p_{j+1} is settled in both directions, which is excluded; so $j = i - 1$, p_{i-1} is settled forward and open backward, and p_i is settled backward and open forward. The forward expansion of p_{i-1} generated the forward copy of p_i, and the backward expansion of p_i generated the backward copy of p_{i-1}; whichever of the two expansions came later found, in an open bucket of the other direction with the same heuristics and partition, the copy generated by the earlier one (the forward copy of p_i, or the backward copy of p_{i-1}, both still open), and set

$U \leq C^*$. Hence $i \leq j$. □

Theorem 7.22 (Correctness of PEM-BiHS). Under the hypotheses of Lemma 7.20, and with consistent heuristics for PEM-BAE*, the searches of `PEMSearch` return C^*, or report t unreachable, unless a limit stops them first; each state is expanded at most once in each direction.

Proof. With $n = p_i$ and $m = p_j$ of Lemma 7.21, $g_F(n) + g_B(m) \leq C^*$ (now possibly with $n = m$), which proves the bound $g_{\min F} + g_{\min B}$; the bounds of f and C follow as in Theorem 7.5 (1 and 3), whose proofs use only $g_F(n) + g_B(m) \leq C^*$, and those of BAE* as in Theorem 7.5 (4 and 5), whose proofs use $d(n, m) \geq 0$. Every key of an open bucket is the key of the copies in it, so the least keys of the buckets are at most those of the copies of n and m. Theorem 7.6, with Lemma 7.21 in place of Lemma 7.3, gives the result for PEMM and PEM-BAE*; PEM-A* is DDD-A* with the goal tested at the generation and the stop at $U \leq f_{\min F}$, which is correct by the same argument with $j = k$. A state is written to a closed file of D only once by Lemma 7.20, and it is expanded only from there. □

The implementation

The closed buckets are indexed by (D, h_F, h_B), each with the list of the values of g that have a closed file, so that the scan of line 7 reads only the closed partitions of the same heuristics and of a smaller g, and that of line 9 those of the other direction with the same heuristics; the open buckets are a set, scanned linearly to choose the best and to compute the bound, at a cost proportional to their number, which is small relative to the records. The memory holds, for each worker, the hash table of the partition it is processing and a buffer of `buffer_bytes` for each file it writes, i.e., at most P for each distinct bucket of the children of the current one; the disk holds every closed file to the end, as for DDD-A* (Proposition 4.16), since a duplicate may be in any closed bucket of the same heuristics and the path is traced through them. The test `pem-thread` (and `pem-openmp`, `pem-fastflow`) runs PEM-A*, PEMM, PEM-BAE* and PEM-rA* with 1, 2 and 4 workers and 5 partitions on 3 states of the 8-puzzle, on random grids with 4 moves and on the map of Romania, and with 2 workers on random walks of 40 moves of the 15-puzzle, against the cost of A*; it checks the stop at a cap of 4096 bytes on the disk and at a node limit of 1000, the refusal of an inconsistent heuristic, and that no file is left when the searches end, stop or throw.

7.10 A* with immediate duplicate detection on the disk

The external searches above, and DDD-A* of §4.6, detect the duplicates in batches by scanning files, which suits a disk read sequentially. On a solid state drive a random read costs much less, and Lin and Fukunaga [95] revisited the *immediate* duplicate detection on the disk: their A*-IDD keeps the closed list in a file read at random, and in memory only a compact table of the positions of its records. The class template `IDDAStar< D >` (`IDDAStar.h`) implements it for a `DiskDomain`, with the declarations

```
template< DiskDomain D >
class IDDAStar {
 public:
 explicit IDDAStar( const D & domain , const SearchOptions & options = {} ,
                    IDDOptions idd = {} );
 [[nodiscard]] SolutionT solve( void ) const;
 };

template< DiskDomain D >
[[nodiscard]] Solution< state_t< D > , cost_t< D > > idd_astar(
 const D & domain , const SearchOptions & options = {} ,
 const IDDOptions & idd = {} );
```

and the options

```
struct IDDOptions {

 /// where the files go, in a directory of their own removed at the end;
 /// the temporary directory of the system if empty
 std::filesystem::path directory;

 /// the places S of the table of the positions in memory, which bound
 /// the closed states to 7 S / 8; at least 8 blocks
```

```
std::size_t slots = std::size_t( 1 ) << 20;

/// the partitions P of the states, one buffer each
std::size_t partitions = 100;

/// the states B of a full buffer, i.e., of a block of the file
std::size_t block = std::size_t( 1 ) << 10;

/// the records of the top of an open bucket kept in memory
std::size_t open_block = std::size_t( 1 ) << 10;

/// the most bytes the files may hold on the disk (0 = no cap)
std::uint64_t disk_limit = 0;

};  // end( struct( IDDOptions ) )
```

i.e., the directory of the files (as for `PEMSearch`), the number S of places of the table of the positions, which bounds the closed states to $7S/8$, the number P of partitions, the number B of states of a block of the closed file, the number of records of the top of an open bucket kept in memory, and the cap on the bytes on the disk. A zero option, or $S < 8B$, is refused with `std::invalid_argument`, as is a weight other than 1.

The lists

OPEN is a set of buckets (f, g), ordered by least f and then by largest g, each a stack of records on the disk with its top in memory: a push adds to the top, and when the top holds twice the records of `open_block`, its lower half goes to the file of the bucket; a pop takes from the top, and when the top is empty the last block of the file is read back (`IDDAStar.h:350-398`). A record holds the state, its g, the state of its parent and, in an `OperatorDomain`, its operator. CLOSED has three parts: the records, in a file written only at its end (or rewritten in place); a table in memory of S places of $\lceil \log_2(S+1) \rceil$ bits each, packed in words, each holding the position plus one of a record in the file (0 for an empty place), probed linearly from the hash of the state; and P buffers in memory, one per partition of the states by a second hash, which collect the states closed until a buffer holds B of them and is written to the file as one block, its positions then being put in the table. Since each block holds the states of one partition, a vector of one partition number per block tells whether the record that a place points to may be the state looked for, and the file is read only when it may (section 4 of [95]).

Algorithm 7.5 (A*-IDD, the pseudocode of the header `IDDAStar.h:25-39`).

```
 1  procedure A*-IDD(s₀)
 2     push (s₀, g = 0) in OPEN
 3     while OPEN ≠ ∅ do
 4        n ← pop the top of the bucket of least f and largest g
 5        if n is in the buffer of its partition or, probing the table
             and reading only the records of blocks of its partition,
             in the file, with a g not larger, then continue
 6        put n in the buffer of its partition (or update its record)
 7        if the buffer has B states, append them to the file as a
             block, and put their positions in the table
 8        if n is a goal then return the path, traced by the parents
 9        push each successor of n, with its g and parent, in OPEN
10     return unsolvable
```

The duplicates are thus detected when a node is popped, before its expansion, and not when it is generated: OPEN holds every copy generated, as the lazy list of §3.1, and a state popped with a smaller g than its closed record is reopened, its record rewritten in the buffer or in place in the file. The path is traced from the goal by looking up the parent of each record among the closed states, down to the root, whose parent is itself.

Lemma 7.23 (The lookup is exact). At each pop, (a) every closed state is either in the buffer of its partition or in the file, at a position a such that the place holding $a+1$ is reached by the linear probing from its hash before an empty place, and not both; (b) the block $\lfloor a/B \rfloor$ of the file holds only states of one partition, the one recorded for it; (c) the table has an empty place. Hence the probe of line 5 terminates, and it finds the closed record of n if there is one, reading from the disk only records of the partition of n.

Proof. (b) A buffer is written only when it holds exactly B states, all of its partition, so that each block is full and of one partition, and its partition is appended to the vector at the same time (`IDDAStar.h:327-348`). (a) A state is put in the buffer only when it is found neither there nor in the file, and it leaves the buffer only when the buffer is written, each record of the block getting a place found by the linear probing from its hash; the places are never cleared, so that the probing from the hash of a state reaches its place before any empty one (Lemma 3.9), and a reopening rewrites the record at the same position. (c) The search stops when $8\,used \geq 7S$ after a step, so that before a flush fewer than $7S/8$ places are taken, and a flush adds $B \leq S/8$ of them (the constructor requires $S \geq 8B$); hence fewer than S places are ever taken, the stop coming right after the flush that reaches $7S/8$. Thus the probe meets an empty place, and a place whose block has another partition is skipped without reading the record, which cannot be that of n by (b). □

Theorem 7.24 (Correctness of A*-IDD). With an admissible heuristic and positive costs, A*-IDD returns a path of cost C^*, or reports the instance unsolvable on a finite graph, unless a limit stops it; with a consistent heuristic each state is expanded once, and *expanded* = *stored*.

Proof. By Lemma 7.23 the test of line 5 is exact, and the search is A* with the duplicates of OPEN dropped at the selection rather than at the generation, the reopening of a closed state popped with a smaller g, and the goal tested at the selection. Every state popped and not dropped has a g smaller than that of every earlier closure of it, so that the invariant of OPEN of Lemma 2.7 holds (an optimal path has a copy of one of its nodes in OPEN with its optimal g, all its predecessors on the path being closed with the optimal one), and the argument of Theorem 2.9 gives that the first goal popped has $g = C^*$; the path traced through the records of the parents has a cost at most g, each parent having been closed with a g that only decreased since, and it ends at the root, since g strictly decreases along it. With a consistent heuristic the order by f closes each state with its optimal g (Theorem 2.13), so that no reopening occurs, and each closed state is stored and expanded once. On a finite graph the closures are finitely many, as in Theorem 2.11. □

Proposition 7.25 (Memory and accesses of A*-IDD). The memory holds the table, $S\lceil \log_2(S+1) \rceil/8$ bytes, the P buffers, fewer than PB records, the vector of the partitions of the blocks, and the tops of the open buckets, fewer than $2\,open_block$ records each; the disk holds the closed records and the bottoms of the open buckets. A lookup in the file reads at most one record for each place of its partition that the probe visits.

Proof. The bounds follow from the construction above and from Lemma 7.23. □

Hence, for $S = 2^{20}$ (the default), a place takes 21 bits and the table about 2.75 MB, for up to 917 504 closed states. The counter `messages` counts the records read at random from the closed file, i.e., the probes that reach the disk, and `sent` the bytes written. The search stops with `Status::node_limit` when the table is 7/8 full or when the closed states reach `node_limit`, and with `Status::disk_limit` when the files exceed the cap. The test `idd` runs A*-IDD with $S = 2^{19}$, $B = 16$, 7 partitions and open blocks of 8 records (so that most of the closed list is on the disk) on 50 states of the 8-puzzle against their exact distances, on the instance 55 of the 15-puzzle of Korf, where it checks that *expanded* = *stored* and that the closed file is read, on random grids and on the map of Romania; it checks the reopening on the graph with an inconsistent heuristic of §2.7, the stop when the table of 1024 places is full and at a cap of 4096 bytes on the disk, the removal of the files, and the refusal of a table smaller than 8 blocks.

We remark that a duplicate found in the file is read twice, once by the probe and once to compare its g (`IDDAStar.h:429-435`), so that `messages` counts two reads for it; and that the table is never cleared, so that a state that is closed once keeps its place for the rest of the search, which is why the closed states are bounded by $7S/8$ rather than by the disk.

7.11 Benchmark

We compare the bidirectional searches with A* and with the reverse A* (A* on the `ReverseDomain`) on the instances of the 15-puzzle of Korf with the Manhattan distance in both directions, and on random grids with 8 moves and the octile heuristic, both domains having consistent heuristics and reversible moves; and the external searches with DDD-A* of §4.6 on the same instances of the 15-puzzle. The runs are made by `AStarBench`, whose option `--algo` names the searches of this chapter (`ra`, `bidij`, `mm`, `mme`, `fmm`, `bae`, `nbs`, `dvcbs`, `gbfhs`, `bibfs`, `pnba`, `pdij`, `pmm`, `pbae`, `pema`, `pemra`, `pemm`, `pembae` and `idd`), each run twice in separate processes on an idle machine, the gap between the two runs being the A/A noise given in the note of the tables.

Table 7.1. The in-memory bidirectional searches. For each set of instances and each search, the expansions in the forward and in the backward direction, the nodes stored, and the seconds summed over the instances (each the median of its two runs). The machine and the compiler are given in the note of the table.

instances	search	expanded forward	expanded backward	stored	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`*; no number is given until then.*

Table 7.2. The external searches on the 15-puzzle. For each search and number of workers, the states expanded, the states stored in the closed files, the bytes written to the disk, the most bytes held on the disk at once, and the seconds summed over the instances; the directory of the files is on the local drive of the machine named in the note.

search	workers	expanded	stored	bytes written	bytes on disk at most	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`*; no number is given until then.*

The clock is the timer of the `Solution`: it includes the creation of the table, the start of the threads of the executor when none is lent, the search, the writing and the reading of the files and the rebuilding of the path, and it does not include the construction of the domain nor the removal of the directory of the files. We leave the tables as placeholders until the campaign is run on an idle machine.

7.12 Notes and references

Bidirectional search is due to Pohl [91], who also proposed to expand in the direction with fewer open nodes; the "nipping" of the nodes closed by the other direction is from the BS* of Kwa [90]. The front-to-end searches of this chapter come from a line of work started by MM of Holte, Felner, Sharon and Sturtevant [61], whose journal version [81] gives the bounds and the proof of the meeting in the middle that we followed in sections 7.2 to 7.4; the improved priority of MMe is from Sharon et al. [82], and fMM and the structure of the minimum covers of the must-expand graph are from Shaham et al. [83]. The sufficient conditions for node expansion and the must-expand graph are due to Eckerle et al. [84]; NBS to Chen et al. [85], whose section 7 gives the queues waiting and ready that we use; DVCBS, NBS$_A$ and the eager variants to Shperberg et al. [86]; and GBFHS to Barley et al. [87]. BAE* was introduced by Sadhukhan [88], whose paper we could not consult, and Alcázar, Riddle and Barley [89] derived its bound, and the individual bounds used with `all_bounds`, from a unified view of the errors of the heuristics; we followed their definition. The immediate and the delayed solution detection, and the external MM, are due to Sturtevant and Chen [94], and the framework PEM-BiHS, with PEM-BAE* and the reverse A*, to Siag et al. [57]. NBA* is due to Pijls and Post [92], and its parallel version PNBA* to Rios and Chaimowicz [93], whose paper we know only from its abstract; the argument of Lemma 7.16, by which no meeting is missed, is that of the mutual exclusion of Dekker, and we give it for the atomic accesses of the library rather than for the scheme of their paper. A*-IDD is due to Lin and Fukunaga [95], and the hash-based delayed duplicate detection on which the external searches rest to Korf (see §4.6).

The library implements only front-to-end searches, i.e., searches whose heuristics estimate the distance to the root of the other direction; the front-to-front searches, which estimate the distance to the nodes of the other front, are not treated here. Also, the searches of sections 7.4 to 7.7 are sequential, and the only parallelism over the two directions is that of §7.8, with two workers, while the parallelism within a direction is that of the external searches of §7.9, over the partitions of a bucket. A comparison of the expansions of MM, NBS, DVCBS, GBFHS and BAE* across domains is in [86] (table 1) and in [89]; Table 7.1 is to give our own counts, on our instances.

7.13 Exercises

Exercise 7.1 (`ctest -R PBeFS.bidir`). Show that the predecessors of a cell of a grid with weighted cells, as defined by the domain, are the moves reversed at the cost of the cell left, and hence that the forward and the backward graph are the same graph with the edges reversed. The test checks it on 4 random grids of 11×7 cells with 8 moves.

Exercise 7.2 (`ctest -R PBeFS.bidir`). Prove the bound of MMe, $\min_D \max\{f_D, 2g_D+\varepsilon\} \leq C^*$ while $U > C^*$, directly from Lemma 7.3, and show that it is never smaller than the bound C of MM. Then build a graph of three nodes on which MM expands a node that MMe does not. The test runs both on 20 states of the 8-puzzle with 6 variants of the options.

Exercise 7.3 (`ctest -R PBeFS.bidir`). Theorem 7.7 assumes the choice of the direction by priority. Build a small instance on which MM with `SideChoice::alternate` expands a node with $g_D > C^*/2$, and explain why the cost returned is still C^*. The test checks the meeting in the middle only with the default choice.

Exercise 7.4 (`ctest -R PBeFS.bidir`). On the graph of five nodes of the test, with the admissible but inconsistent heuristic $h = (0, 3, 0, 1, 0)$, run MM by hand with `consistent` false, and show where a node is reopened. Then find the step of the proof of Lemma 7.4 that fails with this heuristic, and explain why `bae` refuses `consistent` false.

Exercise 7.5 (`ctest -R PBeFS.bidir`). Prove that, with `eager`, every pair with $\mathrm{lb} \leq C^*$ has an end expanded by NBS before the search stops. Which line of Algorithm 7.2 is then never executed, and why?

Exercise 7.6 (`ctest -R PBeFS.bidir`). In Figure 7.2, change the size of the cluster a_0 to 1 and recompute the weights of the covers $K_0, \ldots, K_4$ of Lemma 7.12; which cluster does DVCBS expand? Show that NBS, on the same clusters, may expand nodes of the four clusters a_0, a_1, b_0 and b_1 before it expands any of b_2.

Exercise 7.7 (`ctest -R PBeFS.bidir`). Show that GBFHS with $\varepsilon = 0$ is still correct, and that each level then raises the sum of the two limits by one. The test runs GBFHS and the bidirectional breadth-first search on random directed graphs with arcs of cost 0 to 9, and with the option $\varepsilon = 0$ on the 8-puzzle and the grids; which bound of the other searches becomes weaker with $\varepsilon = 0$?

Exercise 7.8 (`ctest -R PBeFS.bidir-thread`). In Lemma 7.16, replace the sequentially consistent accesses by a store with release order and a load with acquire order, and give an execution of the two workers on a path of three nodes in which neither reads the g written by the other. Then show that the published keys of Lemma 7.17 may be read late without harm, but that a published key that could decrease would break Theorem 7.19.

Exercise 7.9 (`ctest -R PBeFS.pem-thread`). Lemma 7.21 allows $i = j$, which makes the bound $g_{\min F} + g_{\min B} + \varepsilon$ invalid for the delayed detection. Build an instance with two open copies of a state on an optimal path, one per direction, on which that bound exceeds C^* while $U > C^*$, and check that PEMM uses the bound without ε. The test runs PEMM, PEM-BAE*, PEM-A* and PEM-rA* with 1, 2 and 4 workers.

Exercise 7.10 (`ctest -R PBeFS.pem-thread`). Show that the bytes on the disk of `PEMSearch` never fall below the bytes of the closed files, and estimate them, for the 15-puzzle, from the number of states stored and the size of a record (the state, the hash of the parent and the operator). The test checks the stop at a cap of 4096 bytes.

Exercise 7.11 (`ctest -R PBeFS.idd`). With $S = 2^{19}$ places, how many bits does a place of the table of A*-IDD take, and how many closed states can the search hold? Show that, with 1024 places and blocks of 64 states as in the test, the search stops before the probing of Lemma 7.23 could find the table full, and count the places taken at the stop.

Exercise 7.12 (`ctest -R PBeFS.idd`). On the graph of five nodes with the inconsistent heuristic, follow Algorithm 7.5, show which state is reopened, and whether its record is rewritten in the buffer or in the file with the options of the test ($B = 16$, 7 partitions). Then explain why the test checks *expanded = stored* on the 15-puzzle, whose heuristic is consistent, and not on this graph.

8. Parallel A* in shared memory

[Source: include/PBeFS/astar/SPAStar.h (SPAStar), include/PBeFS/astar/MQAStar.h (MQAStar), include/PBeFS/astar/FarmAStar.h (FarmAStar), include/PBeFS/astar/PASE.h (PASE, PASEOptions), include/PBeFS/astar/EPASE.h (EPASE, EPASEOptions), include/PBeFS/astar/PWAStar.h (PWAStar, PWAStarOptions), include/PBeFS/astar/MPLP.h (MPLP, MPLPOptions), the concepts EdgeDomain, HeterogeneousEdgeDomain and LazyEdgeDomain of include/PBeFS/astar/Domain.h, include/PBeFS/astar/ConcurrentClosed.h and its three implementations in include/PBeFS/astar/structures/, include/PBeFS/core/SearchUtils.h (ParallelParams, run_workers), include/PBeFS/astar/Solution.h (SearchStats), include/PBeFS/astar/SlowDomain.h (SlowDomain, SlowEdgeDomain, EdgeWork)]

We are concerned in this chapter with the parallel searches of the library in which the p workers share one OPEN list, or one owner of it, i.e., Simple Parallel A* (SPA*), MultiQueue A* (MQA*), the master-worker farm and Parallel A* for Slow Expansions (PA*SE), and the searches that parallelize the evaluations of the edges rather than the expansions of the states (ePA*SE, GePA*SE, their anytime variants, PwA* and MPLP). Since this is the first chapter on parallel search, we also state in it by which overheads the parallel searches of the library are measured (§8.2) and what they must guarantee, i.e., an optimal path with an admissible heuristic and a sound decision of when to stop (§8.3). For each search we then give the algorithm as the code runs it, the invariant from which its optimality follows, the proof that its test of termination is exact, and the cost of its operations. The decentralized searches, Hash Distributed A* and Safe PBNF, are the subject of Chapter 9 and Chapter 11, the further parallel schemes of the library (PSDD, KBFS, TDS, the parallel IDA* and the parallel greedy searches, among others) that of Chapter 12, and the last sections of this chapter compare the searches of the three chapters. We use the notation of Chapter 1, i.e., $g^*(u)$, $h^*(u)$ and the optimal cost $C^* = h^*(s_0)$ of §1.1 and the conventions of the pseudocode of §1.4 (the workers, the executors and the options of the parallel searches are introduced in §1.6), and we refer to Chapter 2 for the sequential A* and the lemmas on which the proofs below rest.

8.1 The interface of the parallel searches

Each search of this chapter is a class template over the `Domain` D of the problem (see Chapter 1) and over the executor that runs its workers, i.e., the pool of C++ threads of the library (the default), OpenMP or FastFlow [12] (§1.6); SPA* and the farm also take the OPEN list as a template parameter, which by default is the bucket queue when the costs are integers and a heap otherwise (a 4-ary one with the default data structures, see Chapter 3). The same code of a search serves each executor; the tests run SPA* and MQA* on the C++ threads and on OpenMP, and the farm, PA*SE and the searches by edges on all three. A search is built from the `Domain` and from a `ParallelOptions`, and `solve()` returns a `Solution`, as the sequential A* does. The following example, from the case `manual` of the tester, which the build compiles and runs (`ctest -R PBeFS.manual`), solves an instance of the 15-puzzle with SPA* on the default executor and with MQA* on an executor built once by the caller and given to the search by `set_executor()`:

```
const FifteenPuzzle d( ki.board );
ParallelOptions o;
o.threads = 4;          // the only option of SPA*, MQA* also reads o.seed
SPAStar< FifteenPuzzle > spa( d , o );
const auto s = spa.solve();     // on a pool of o.threads threads of its own
Executor ex( 4 );               // one executor for several searches,
                                // which must outlive them
MQAStar< FifteenPuzzle > mq( d , o );
mq.set_executor( ex );
```

```
const auto m = mq.solve();
const auto expanded = s.stats.expanded;        // by all the workers
const auto reexpanded = s.stats.reexpanded;
const double lb = s.stats.load_balance();      // 1 if perfectly balanced
```

The four searches have the same interface, i.e., a constructor that takes the `Domain` (which must outlive the search) and the options, `set_executor()` and `solve()`; for SPA* it reads as follows (the declarations are quoted from `SPAStar.h`, where they come with their bodies), and MQA*, the farm and PA*SE differ only in their template parameters (`MQAStar< D , Executor >`, `FarmAStar< D , Executor , Open >` and `PASE< D , Executor >`, the last with a `PairwiseDomain D`) and in the options of PA*SE, `PASEOptions`; the searches by edges of §8.9 (`EPASE`, `PWAStar` and `MPLP`) have the same interface as well.

```
template< Domain D , WorkerExecutor Executor = ThreadPool ,
          class Open = /* buckets for integer costs, a heap otherwise */ >
class SPAStar {
 public:
 explicit SPAStar( const D & domain , const ParallelOptions & options = {} );
 void set_executor( Executor & ex ) noexcept;  // must outlive the searches
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };
```

```
struct PASEOptions : public ParallelOptions {

 /// the inflation \f$ \varepsilon_p \geq 1 \f$ of the independence rule:
 /// with \f$ \varepsilon_p = 1 \f$ the search is optimal, otherwise the
 /// cost is within a factor \f$ \varepsilon_p \f$ of the optimum; it is
 /// raised to the weight \f$ w \f$ of SearchOptions if smaller, which
 /// orders OPEN by \f$ f' = g + w h \f$ (wPA*SE if above 1)
 double eps = 1;

 /// the states of OPEN examined for a safe one, from the first (0: all),
 /// those that the prune skips not counted
 std::size_t candidates = 64;

 /// once a goal of cost \f$ z \f$ is in OPEN, no state with
 /// \f$ g + \varepsilon_p h \geq z \f$ is expanded, which keeps the bound
 bool prune = true;

 };  // end( struct( PASEOptions ) )
```

`ParallelOptions` joins the options of the sequential search (the node limit, among others) with the `ParallelParams` of `include/PBeFS/core/SearchUtils.h:98`: the number of workers `threads` (p, 1 by default), the nodes per message `batch` (64), the expansions between two flushes of an outbox `flush_period` (64) and the `seed` of the random choices (1); `normalize()` sets to 1 any of the first three that is 0 (`SearchUtils.h:106`). Of these, the searches of this chapter read `threads`, the farm also `batch` (the nodes per task) and MQA* the `seed` of its draws; `PASEOptions` adds the inflation $\varepsilon_p \geq 1$ of PA*SE, `eps` (1 by default), which the constructor raises to the weight w of the options when smaller (`PASE.h:238`), the number `candidates` of the states of OPEN that a selection examines (64, and 0 for all of them), and `prune` (true), which keeps the workers from expanding the states through which no goal can be cheaper than the bound allows; all three are described in §8.8. The field `expected_nodes` of the options, which presizes the closed list of the sequential A*, is not read by the parallel searches. PA*SE needs a `PairwiseDomain` (`Domain.h:295`), i.e., a `Domain` with a function `pairwise_heuristic( s , t )` that estimates the cost of a path from s to t, whose requirements are stated in §8.8, and the searches by edges of §8.9, whose options are given there, need a `Domain` that evaluates its edges one at a time.

The weight w of `SearchOptions` (1 by default), the dropping of the duplicates and the anytime search change SPA*, MQA* and the farm as they change HDA* and Safe PBNF: OPEN is ordered by $f' = g + wh$ and the incumbent prunes accordingly, so that the path costs at most wC^*, or the search goes on to an optimal incumbent recording the paths found on the way. These are the bounded-suboptimal and anytime searches of Chapter 6, which proves their bounds; in this chapter the weight is 1, except in the wPA*SE of §8.8 and in the searches by edges, whose weight orders their keys. The synchronous rounds of the farm (`FarmOptions::synchr`

onous) are the K-best-first search of §12.3.

An executor given by `set_executor()` must outlive the searches that use it and, for the farm, have at least `threads + 1` threads. Without `set_executor()`, `solve()` builds an executor with as many threads as workers (one more for the farm, whose master runs on a separate thread), and runs the body of each worker on it (by `run_workers()` in SPA*, MQA* and PA*SE, by the farm of the executor in the farm), pinned to a core when `PBEFS_PIN` is set, unless the body runs on the thread of the caller (§1.6). As in the sequential search, a `Solution` holds the status (`solved`, `unsolvable` or `node_limit`), the cost and the path, which is rebuilt from the goal through the parent pointers after all the workers have returned, and the counters of `SearchStats` (`Solution.h:73`): the expansions (re-expansions included), the re-expansions, the successors generated, the duplicates, the nodes sent to another worker, the batches sent, the nodes stored, the seconds of the wall clock, and the expansions and the stored nodes of each worker (a single entry, that of the master, in the farm). In a parallel search the node limit bounds the nodes stored by each worker (by the master in the farm), and the search that reaches it returns `Status::node_limit` with no path.

8.2 The overheads of a parallel search

A parallel search with p workers is rarely p times faster than the sequential one, and the gap is usually split, following the survey of Fukunaga et al. ([1], section 3.1) and the analysis of Jinnai and Fukunaga ([5], section 2.1), into four overheads. We give them in the form in which the counters of the library compute them; E_i denotes the expansions of the worker i (the entry i of `expanded_by`, re-expansions included) and E^{seq} those of the sequential A* on the same instance, with the same heuristic and the same tie-breaking.

Definition 8.1 (Search overhead). The search overhead of a parallel run is

$$\text{SO} = \frac{\sum_{i=0}^{p-1} E_i}{E^{\text{seq}}} - 1. \tag{8.1}$$

It counts the expansions that the sequential search does not make, i.e., the nodes with $f > C^*$, the extra nodes of the last layer $f = C^*$ and the re-expansions, which a parallel search pays even with a consistent heuristic, since a worker may expand a node before another worker has found the cheaper path to it (Figure 8.3). A negative SO is possible (and may give a superlinear speedup), since a parallel search may reach a goal of the last layer after fewer of its nodes than the sequential one. Since it needs a second, sequential run, the library does not compute SO; one obtains it from the field `expanded` of the two `SearchStats`, typically from a run of `AStar` on the same instance.

Definition 8.2 (Communication overhead). The communication overhead is the fraction of the generated nodes that change worker,

$$\text{CO} = \frac{\text{sent}}{\text{generated}}, \tag{8.2}$$

and 0 when no node is generated, as `communication_overhead()` computes it (`src/Solution.cpp:69`). Each such node costs a copy, a message or a lock. Note that the numerator counts nodes, as opposed to messages, as in section 2.1 of [5]; the number of the messages, i.e., of the batches, is the separate counter `messages`.

Definition 8.3 (Load balance). The load balance is the ratio of the busiest worker to the mean,

$$\text{LB} = \frac{\max_i E_i}{\frac{1}{p}\sum_{i=0}^{p-1} E_i} \geq 1, \tag{8.3}$$

1 being a perfect balance (the inequality holds since a maximum is not smaller than a mean); `load_balance()` (`src/Solution.cpp:55`) returns 1 when no worker has expanded a node. Note that section 2.1 of [5] counts in (8.3) the nodes assigned to a worker, whereas the library counts its expansions, as Kishimoto, Fukunaga and Botea [2] do (their section 4.2.1).

Definition 8.4 (Coordination overhead). The coordination (or synchronization) overhead is the fraction of the time of the workers spent waiting, for a lock, for a message or for the other workers, instead of expanding.

No counter of the library measures the coordination overhead, which depends on the machine (and, in shared memory, also on the contention for the memory bus, cf. section 2.1 of [5]). It shows in the wall clock and in a

profiler. Note also that the shared-memory searches of this chapter report sent = 0, and hence CO = 0 by (8.2), although each node of SPA* and of MQA* goes through a shared structure and each node of the farm travels to a worker and back. In fact, (8.2) measures the nodes that change owner in a distributed search (Chapter 9), and the searches of this chapter have no owners. For the same reason the farm reports the expansions of the master only, and its LB is 1 by construction.

These overheads are not independent ([5], section 2.1). A distribution of the states that keeps the successors of a node on its worker lowers CO, but it tends to gather the nodes of least f on a few workers. This raises LB and, through it, SO, since the other workers then expand nodes that the sequential search would not reach. This is the band effect of section 3.1.1 of [5], which is still visible when the expansions are made artificially slow, and which the authors therefore attribute to the load imbalance (their Observation 1). Conversely, a uniform random distribution balances the load at the price of sending a fraction $1 - 1/p$ of the nodes. The centralized searches of this chapter sit at one end of this trade-off, since they have no owners, hence no CO in the sense of (8.2), and a balanced load, and they pay with coordination, since each expansion goes through a shared structure or through a master.

8.3 Correctness and termination of a parallel search

The sequential A* stops at the first goal it selects, since by Lemma 2.8 every node it selects has $f \le C^*$. A parallel search cannot, since, when a worker selects a goal of cost g, another worker may hold a node of smaller f, or be expanding one, or such a node may be travelling towards the worker that will expand it ([1], sections 3.1 and 3.3.1). Hence, the parallel searches of the library share the cost z of the best goal found so far (the *incumbent*, $z = \infty$ before the first goal), prune each node with $f \ge z$, test the goals when a node is selected for expansion, and stop only when no node that could improve z is left in any worker or in transit, a condition that we now make precise.

An *entry* is a pair (u, γ) of a node and a value of its g, put in OPEN when $g(u)$ takes the value γ. The entry is *stale* when u has been closed or $g(u) < \gamma$ since then (the lazy deletion of §3.1), and its key is $f = \gamma + h(u)$.

Definition 8.5 (Entries in flight). At an instant τ of a parallel search, an entry is *in flight* if it is (i) in an OPEN list, (ii) held by a worker that has selected it and has not yet finished processing it, the successors it generates being part of what it holds until they are pushed, or (iii) in a message (or task) that has been sent and is still to be received.

We say that a parallel search is *well-formed* if the following three conditions hold. (a) Each entry is in flight from its creation until it is selected. (b) A worker that selects a non-stale entry (u, γ) with $\gamma + h(u) < z$, u not a goal, expands u, i.e., for each successor v it either prunes v, because $\gamma + c(u, v) + h(v) \ge z'$ for a value z' that z had at some earlier instant, or makes $g(v) \le \gamma + c(u, v)$ hold, lowering $g(v)$ (or setting it, if v is new) to the cost of a path to v when it is larger, and creating the entry of each new value, while still holding u. (c) A goal u selected in a non-stale entry with $g(u) < z$ sets $z \leftarrow g(u)$ and records u. All the searches of this chapter are well-formed, as the proofs of their sections show, and so are those of Chapter 9 and Chapter 11. We remark that pruning against a value z' of the past is sound because z never increases, and hence $z' \ge z$. Condition (b) allows a worker to skip a successor v for which $g(v) \le \gamma + c(u, v)$ already holds (the pruning of the parent at the end of this section) and to give v a value smaller than $\gamma + c(u, v)$ when $g(u)$ has decreased since the selection (the farm of §8.7).

Theorem 8.6 (The stopping criterion). Let h be admissible, the costs nonnegative and the search well-formed, and let τ be an instant at which no entry in flight that is not stale has $f < z$. Then $z = C^*$ at τ (with $C^* = \infty$ if no goal is reachable).

Proof. The argument is that of Lemma 2.7, with the entries in flight in the place of OPEN. If no goal is reachable, $z = \infty = C^*$, since z is only set to the g of a goal. Otherwise, suppose $z > C^*$ at τ, and let $P = (s_0, \dots, s_k)$ be an optimal path to a goal s_k, with $g^*(s_k) = C^*$, chosen so that no s_j with $j < k$ is a goal (a goal s_j on P has $g^*(s_j) \le C^*$, hence $g^*(s_j) = C^*$, and P may be cut there). We first remark that $g(u) \ge g^*(u)$ at every instant, since $g(u)$ is the cost of a path (cf. Lemma 2.6, whose induction holds in the parallel searches because each assignment of g is made under the lock of its node, or by its single owner), and that the successive values of $g(u)$ strictly decrease. Hence, for each j the value $g^*(s_j)$ is taken by $g(s_j)$ at most once, the entry $(s_j, g^*(s_j))$ is created at most once, and once created it can become stale only by the closing of s_j, which (since $g(s_j) = g^*(s_j)$ from then on) only the selection of this very entry can cause. Call the expansion of s_j *done* if a worker has selected the entry $(s_j, g^*(s_j))$ and finished processing it before τ; by the remark, the entry was not stale when selected, and its key $g^*(s_j) + h(s_j) \le g^*(s_j) + h^*(s_j) = C^*$ was smaller than the value of z at that time, which is at least the

value $z > C^*$ at τ. Clearly, the expansion of s_k is not done, since by (c) it would have set $z \leq C^*$; let j be the least index whose expansion is not done. If $j = 0$, the entry $(s_0, 0)$ created at the start is in flight by (a), or held by a worker that has selected it. If $j > 0$, the expansion of s_{j-1} is done and, s_{j-1} not being a goal, (b) applies to it with $\gamma = g^*(s_{j-1})$. It did not prune s_j, since $g^*(s_{j-1}) + c(s_{j-1}, s_j) + h(s_j) = g^*(s_j) + h(s_j) \leq C^* < z \leq z'$ (h being admissible); hence, it left $g(s_j) \leq g^*(s_j)$, i.e., $g(s_j) = g^*(s_j)$, creating the entry $(s_j, g^*(s_j))$ if it was not already there. Thus, for $j = 0$ as for $j > 0$, this entry exists; it is in flight, or it has been selected, and in the second case its processing is not finished (by the choice of j), and it is held. By the remark it is not stale, and its key is at most $C^* < z$, which contradicts the hypothesis. Hence $z \leq C^*$; since z is the g of a goal, i.e., the cost of a path to it, $z = C^*$. □

We draw the theorem in Figure 8.1: a search may stop when the least f in flight is at least the incumbent, and only then. In fact, the searches differ only in how they decide that this condition holds, which is easy when one lock or one master sees all the entries (SPA* and the farm), needs a counter when the OPEN list is spread over many queues (MQA*), and is the delicate part of a decentralized search, in which no worker sees the others at the same instant ([1], section 3.3.1). Once the search has stopped, the path is rebuilt from the recorded goal u through the parent pointers, after all the workers have returned. Its cost is at most the final $g(u) \leq z$ by the inequality of Lemma 2.10, which holds in the parallel searches at the end for the same reason as Lemma 2.6 in the proof above, and it is at least C^*; hence it is exactly C^*.

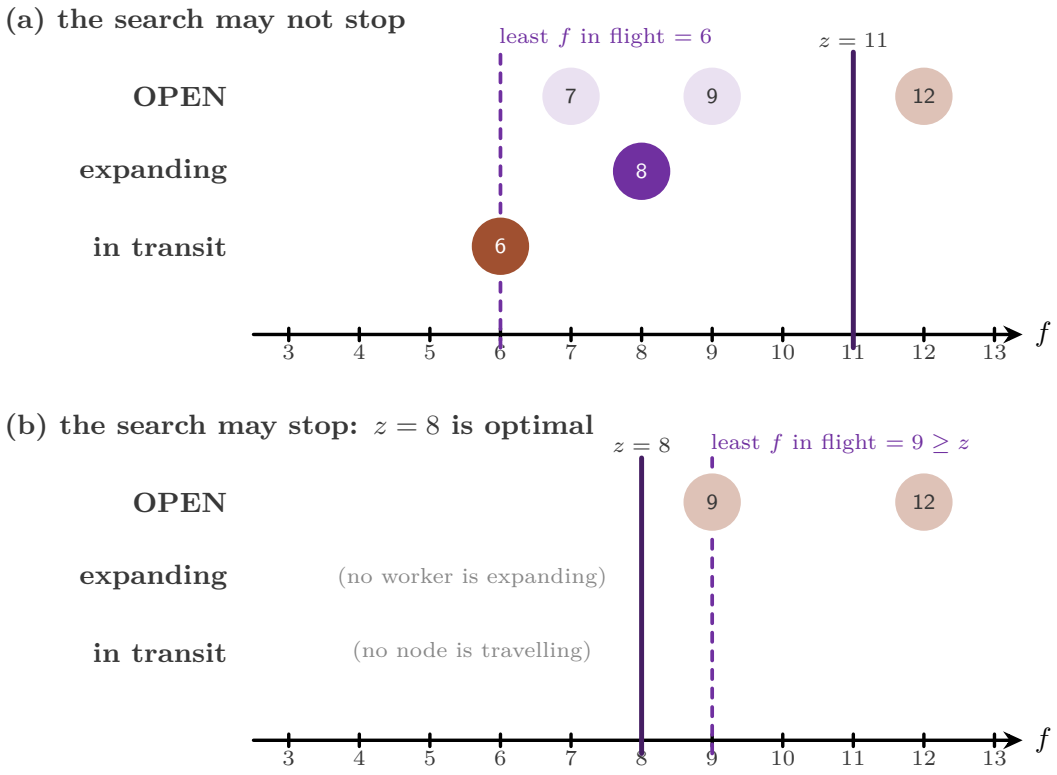


Figure 8.1. The stopping criterion of Theorem 8.6. In (a) an entry with $f = 6$ is in a message and one with $f = 8$ is being expanded, both below the incumbent $z = 11$, and the search may not stop, even if all the OPEN lists held only the entries of $f \geq 11$. In (b) a goal of cost 8 has been found; all the entries in flight have $f \geq z$ (and are pruned), and $z = 8$ is optimal.

Proposition 8.7 (Finiteness). On a finite graph with nonnegative costs, a well-formed search creates finitely many entries. Hence, if each of its workers selects an entry, or waits, in a finite time, and a worker without an entry to select waits only for a step of another worker, the search stops.

Proof. An entry is created only when $g(v)$ takes a new value, which is strictly smaller than the previous one and is the cost of a path from s_0 to v. The argument (i) of Theorem 2.11 shows that these values are finitely many for each of the finitely many states. Each selection removes an entry from the flight set; hence, after finitely many selections no entry is left and the stopping condition of the search (whose form is proved for each search below) is met. □

On an infinite graph a parallel search needs at least what A* needs, i.e., costs at least some $\varepsilon > 0$, a finite

branching and a reachable goal (argument (ii) of Theorem 2.11); with costs that only avoid nonpositive cycles it may run forever (the example after Theorem 2.11). We do not prove that these hypotheses suffice for every search of this chapter: the relaxed order of MQA* (§8.6), for instance, gives no deterministic bound on how long an entry of small f may wait in its queue.

Remark 8.8 (Idle flags are not enough). A test of termination that only polls each worker, possibly twice in a row, whether it is idle is not sound when the entries travel in messages: worker 0 may send an entry of small f to worker 1 and become idle, while worker 1, idle, has not received it yet. Both answer "idle" in both rounds if the message is slow, the search stops, and the entry, which may be on the only optimal path, is lost. A sound test must also count the entries in transit, as condition (iii) of Definition 8.5 and Theorem 8.6 require. In this chapter, the centralized searches count them by construction (a counter under the lock of OPEN, an atomic counter, the tasks out at the master), as the proofs below show. The decentralized ones use the counting of Dijkstra and Scholten [35] in shared memory and the four counters of Mattern [11] over MPI, which Chapter 9 treats.

Finally, all the parallel searches of the library skip the move back to the parent in an `OperatorDomain` (`Domain.h:356`), as the sequential A* does (§2.9 and [4]): the inverse of a move costs as much as the move, and therefore the parent is never reached more cheaply through its child, and the lookup of about one successor in three of the sliding-tile puzzles is saved. This keeps the searches well-formed, as in the proof of Proposition 2.16. Indeed, a worker reads the operator o of u together with $g(u) = \gamma$ (under the lock of u, or on the master), and o is the move from the node p that gave u that value; hence $\gamma = g_0 + c(p, u)$, where g_0 is the value of $g(p)$ at that time, and since $g(p)$ never increases, the skipped successor p has $g(p) \le \gamma - c(p, u) \le \gamma + c(u, p)$, as condition (b) requires.

8.4 Simple Parallel A*

The simplest parallel A* lets the p workers share the two lists of the sequential search, i.e., each worker takes the best entry of the shared OPEN, expands its node and puts the successors back ([1], Algorithm 2, and section 2.2 of [3]). Thus, the order of the expansions is usually close to that of A*, but p nodes are expanded at once, and a node may be expanded before the node that gives it a cheaper path. There are no owners, hence no communication in the sense of (8.2), and the load is balanced by construction. Its price is the coordination, since each selection and each insertion goes through the lock of OPEN.

The algorithm

Algorithm 8.1 (SPA*). Here, the lock L protects OPEN and the counter busy of the workers that hold an entry; the fields of the nodes that change are protected by the locks of CLOSED (§8.5), and z is atomic, i.e., it is read and written as one word, without a lock.

```
 1  procedure SPA*(s₀, p)
 2     CLOSED ← {s₀}, with g(s₀) ← 0;  OPEN ← {(f(s₀), g(s₀), s₀)}
 3     z ← ∞;  busy ← 0;  done ← false
 4     in parallel, on each worker i = 0, ..., p − 1 do
 5        while not done do
 6           lock L
 7           if OPEN has an entry of f < z then
 8              (f, g, u) ← the first entry of OPEN, removed
 9              busy ← busy + 1
10           else if busy = 0 then done ← true
11           unlock L
12           if no entry was removed then wait a little;  continue
13           F ← ∅;  lock u
14           if u is closed or g ≠ g(u) then u is not expanded  ▷ stale
15           else close u
16           unlock u
17           if u is expanded and u is a goal then
18              z ← min(z, g(u)), recording u if g(u) < z
19           else if u is expanded then
20              for each successor v of u with g(u) + c(u,v) + h(v) < z do
21                 g' ← g(u) + c(u, v);  lock v
22                 if v ∉ CLOSED or g' < g(v) then
```

```
23                    g(v) ← g';  parent(v) ← u;  v is not closed
24                    F ← F ∪ {(g' + h(v), g', v)}
25                unlock v
26          lock L;  push the entries of F in OPEN;  busy ← busy − 1
27          unlock L
28    return the path to the goal of cost z, if z < ∞
```

This is the pseudocode of SPAStar.h:16. A worker takes the lock of OPEN only twice per expansion, to select an entry (lines 6 to 11) and to push all the successors of its node at once (lines 26 and 27). The expansion itself, at lines 13 to 25, runs without it, and this is where the parallelism is. Figure 8.2 draws one selection and one insertion. At line 14, the test of staleness is the lazy deletion of the sequential search, done under the lock of the node since another worker may be lowering its g. The goal test is at the selection (line 17), and a goal does not end the search, which goes on until line 10 finds no entry of $f < z$ and no busy worker. The wait of line 12 is an exponential backoff: the k-th consecutive wait spins for 2^k pauses of the processor, for $k = 0, \ldots, 6$, and each further wait yields the core to the system, so that a worker that finds OPEN empty only for a moment does not sleep. When the node limit is set, a worker that has stored that many nodes sets done and records that the limit was hit, and the other workers leave the loop at line 5 after their current step.

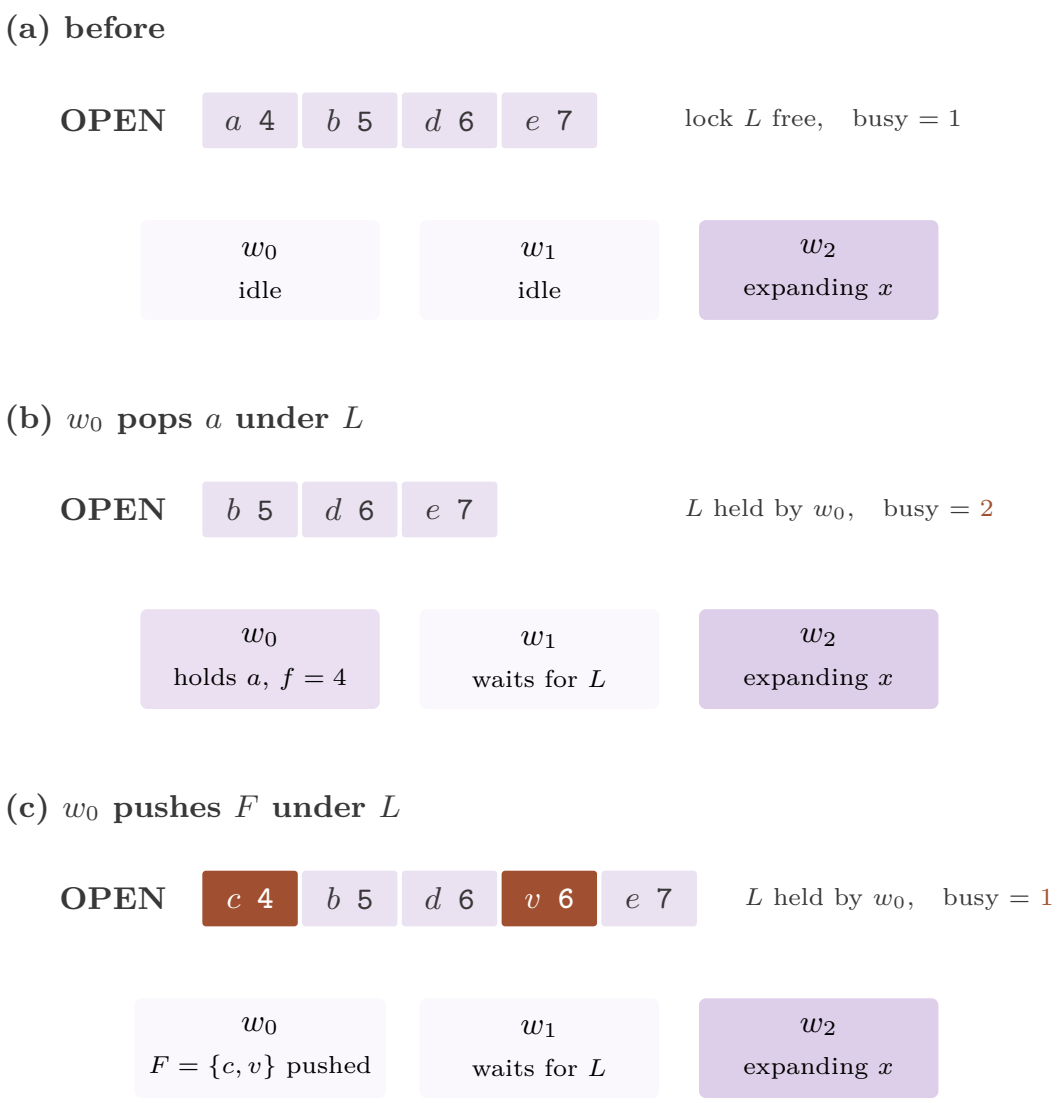


Figure 8.2. A step of SPA* with three workers. (a) Worker w_2 is expanding a node and busy = 1. (b) Worker w_0 takes L, removes the best entry a of OPEN and raises busy, while w_1 waits for L. (c) After the expansion of a, outside L, w_0 takes L again, pushes its successors c and v and lowers busy in the same critical section.

The code makes the expansion of lines 13 to 25 in two phases, which SPA* and MQA* share. In the first phase

the worker generates the successors, prunes them by z, computes their hashes and *prefetches* their places in CLOSED, i.e., it asks the processor to bring the lines of memory that their lookups will read into its cache without waiting for them (§3.8); in the second it updates them one by one. The cache misses of the lookups of all the successors therefore overlap, instead of following one another.

Algorithm 8.2 (The expansion of SPA* and MQA*). The entry $(\gamma + h(u), \gamma, u)$ has been selected; UPDATE is the operation of the shared CLOSED (Algorithm 8.3), and the procedure returns the set F of the entries to push.

```
 1  procedure EXPAND(u, γ)
 2     lock u
 3     if u is closed or g(u) ≠ γ then unlock u;  return ∅   ▷ stale
 4     close u;  read g(u) and the operator o of u;  unlock u
 5     if u is a goal then
 6        lock the goal record
 7        if g(u) < z then z ← g(u);  record u
 8        unlock the goal record;  return ∅
 9     S ← ∅                                    ▷ phase 1: no shared write
10     for each successor v of u not reached by the inverse of o do
11        if g(u) + c(u, v) + h(v) < z then
12           compute the hash of v;  prefetch its place in CLOSED
13           S ← S ∪ {(v, g(u) + c(u, v))}
14     F ← ∅                                    ▷ phase 2: the updates
15     for each (v, g') ∈ S do
16        if UPDATE(v, g', u) returns the node of v then
17           F ← F ∪ {(g' + h(v), g', v)}
18     return F
```

The incumbent is read at line 11 without any ordering of the memory (a *relaxed* read): the value read is one that z had at some instant of the expansion, and since z only decreases it is not smaller than the current one, so that the pruning is the one that condition (b) of §8.3 allows. The goal is recorded under a separate mutex, taken only when a goal is selected. Furthermore, the lock of OPEN starts a cache line, followed by OPEN and busy, which it protects (`SPAStar.h:216`), the incumbent starts another one (`SPAStar.h:220`), so that the reads of z at line 11 do not share a line with the lock, and each worker with its counters starts a line of its own (`SPAStar.h:235`); thus, the counters of two workers never share a line.

Invariants and correctness

We now prove that SPA* is correct. The proof rests on one invariant, which ties the counter busy to the entries that are not in OPEN.

Lemma 8.9 (The invariant of busy). Whenever L is free, and also while a worker holds L at line 7, every entry in flight is in OPEN or is held by one of the busy workers, i.e., by a worker between lines 9 and 26, and busy is the number of these workers.

Proof. Since SPA* has no messages, an entry in flight is in OPEN or held (Definition 8.5). An entry leaves OPEN only at line 8, where the worker raises busy in the same critical section, and it is held (with the entries of F that it creates) until line 26, where the worker pushes F and lowers busy in the same critical section. Since both changes of busy are made under L together with the change of OPEN, the claim holds whenever L is released, and therefore also when it is taken again, until the worker that holds it changes OPEN or busy (which it does only after line 7). □

Theorem 8.10 (Correctness and termination of SPA*). If h is admissible and SPA* ends without reaching the node limit, it returns an optimal path (or none, when no goal is reachable); on a finite graph with nonnegative costs it ends.

Proof. SPA* is well-formed: (a) and (c) are lines 8, 24, 26 and 18, and (b) is lines 20 to 25, where the update of v is made under its lock, and where z is read without a lock, so that the pruning at line 20 uses a value that z had at some instant of the expansion (the successor skipped by the pruning of the parent of §8.3 is also allowed by (b)). When a worker sets done at line 10, it holds L and finds no entry of $f < z$ in OPEN and busy $= 0$. By Lemma 8.9 no entry in flight is held, and hence none has $f < z$, and Theorem 8.6 gives $z = C^*$. A worker that finds no entry to select at line 12 waits only for the busy workers, which end their expansion without waiting for anything but locks held for a bounded number of steps; hence Proposition 8.7 gives the termination. □

Re-expansions

Unlike A* (cf. Theorem 2.13), SPA* may expand a node more than once even with a consistent heuristic ([1], section 3.2): in Figure 8.3, two workers expand a and b at once, the second best entry b generates v through an expensive edge, and v is selected and closed by w_1 before w_0, through c, finds the cheaper path to it. As one can check on the figure, lines 22 and 23 then reopen v (`v is not closed`), and it is expanded again; the counter `re expanded` counts these expansions, which belong to the search overhead (8.1). With $p = 1$, SPA* is A*, and the re-expansions are those of an inconsistent heuristic (§2.7). The number of the re-expansions is bounded as follows, for SPA* and, with the same proof, for MQA* and the farm.

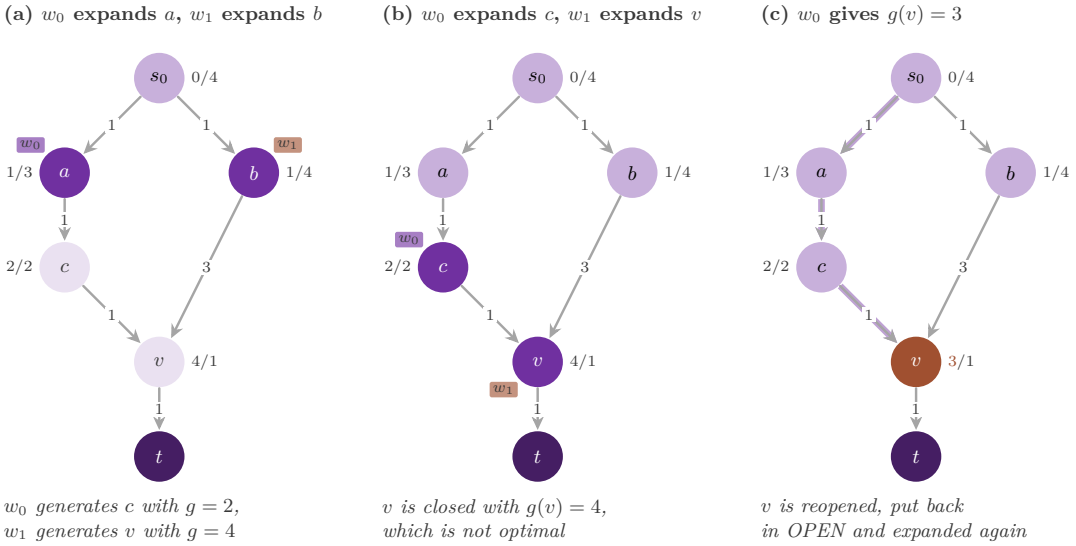


Figure 8.3. A re-expansion in SPA* with a consistent heuristic (values g/h next to the states). (a) w_0 and w_1 expand a and b at once. (b) w_1 selects v, of $f = 5$, with $g(v) = 4$, while w_0 expands c. (c) The expansion of c gives $g(v) = 3$: v is reopened and will be expanded again, now with its optimal g.

Lemma 8.11 (One expansion per value of g). In SPA*, MQA* and the farm, between two expansions of a node u the value of $g(u)$ strictly decreases. Hence, u is expanded at most as many times as the distinct values that $g(u)$ takes, and the re-expansions are at most the decreases of g after a first expansion.

Proof. An expansion of u selects an entry (u, γ) that is not stale, i.e., with u not closed and $g(u) = \gamma$, and closes u, in one critical section on the lock of u (lines 2 to 4 of Algorithm 8.2; in the farm, only the master does both). A later expansion needs an entry that is not stale, hence u not closed; but u is made not closed only by an update that sets $g(u)$ to a strictly smaller value (line 23 of Algorithm 8.1, line 9 of Algorithm 8.3, the INSERT of the farm). □

The lemma does not bound the decreases of g in terms of p; they are bounded only by the number of the distinct costs of the paths to the node that are smaller than its first value of g (cf. Proposition 8.7), and we know no bound that depends on p alone. With $p = 1$ and a consistent heuristic no decrease follows an expansion, and the lemma gives the single expansion of Theorem 2.13.

The cost of an expansion

We now count the synchronization of a step of SPA*, since it limits the search when the expansions are fast.

Proposition 8.12 (The cost of a step of SPA*). A worker that selects an entry takes L twice (lines 6 and 26), the lock of u once, and, for each successor v that is not pruned, at most one lock of CLOSED (with the default data structures, only when v was already stored and g' improves on the $g(v)$ read without the lock, cf. Lemma 8.13); under L it makes one removal and $|F|$ insertions of the OPEN list. The nodes stored are one per state reached, and the entries pushed in OPEN, stale ones included, are one for each value taken by the g of a node, i.e., one for each state reached plus one for each later improvement of its g.

Proof. The counts are read on Algorithms 8.1, 8.2 and 8.3; a worker that finds no entry to select takes L once and waits. Each state has one node by Lemma 8.13(i), and an entry is pushed only for the root and for each update that returns a node, i.e., for each new state or improvement of g (Lemma 8.13(iii)). □

Hence, the lock of OPEN is contended by all the workers twice per expansion, and the survey reports that, when the expansions are as fast as on the sliding-tile puzzle, SPA* is usually slower than the sequential A*, even with lock-free data structures ([1], section 3.2, after [3]). It may pay off when the expansions are slow, since the time under L is then a small part of an expansion. MQA* (§8.6) removes this lock.

8.5 The shared closed list

SPA* and MQA* share one CLOSED list, the class `ConcurrentClosed< D , Lock >` of `ConcurrentClosed.h`, which maps each state to its canonical node and protects the fields of a node that change during the search (g, the parent, the operator and the flags closed and expanded); h never changes and is read freely. Its operation `update( t , hash , g , h , parent , op , ... )` records the path to t of cost g: it returns the node of t when t is new or g is smaller than its g, which is then set with the parent and the operator and makes the node not closed, and returns a null pointer otherwise, counting a duplicate. Its implementation is chosen at build time with the set of data structures (`Structures.h`, §3.12).

With the default data structures, CLOSED is a lock-free hash table of pointers to the nodes, compared by their states, one word of 8 bytes per node, which a lookup reads and an insertion fills by a *compare-and-swap*, i.e., an atomic instruction that writes a word only if it still holds the value read before. The fields that change are protected by 256 locks, each on a cache line of its own, the lock of a node being chosen by a multiplicative hash of its address. Each worker keeps a *spare* node, allocated from its private pool, which it fills with the state before the insertion and keeps for the next call when the state was already there. Figure 8.4 draws the three cases of the update, which the following procedure makes.

Algorithm 8.3 (The update of the shared CLOSED). The path to v of cost g' through u is recorded; the procedure returns the node of v when an entry must be pushed, and nil otherwise.

```
 1  procedure UPDATE(v, g', u)
 2     n ← the spare node of the worker, filled with v, g', h(v), u
 3     x ← the node of v in CLOSED, or n inserted by a compare-and-swap
 4     if x = n then the worker takes a new spare node;  return x
 5     if g' < g(x), read without the lock of x, then
 6        lock x
 7        if g' < g(x) then
 8           g(x) ← g';  parent(x) ← u;  the operator of x ← that of v
 9           x is not closed;  unlock x;  return x
10        unlock x
11     count a duplicate;  return nil
```

With the data structures of the standard library or of Boost, CLOSED is split into 256 shards by the top byte of the hash of the state, each a sequential table with a lock, which also protects the nodes of its states (`structures/std/ConcurrentClosed.h:134`); UPDATE then takes the lock of the shard of v and runs lines 3 to 11 under it, with the test of line 5 dropped and the node allocated only when v is new; the prefetch of Algorithm 8.2 then does nothing, since the table of a shard may be reallocated under its lock.

We now show that the shared CLOSED loses no improvement of g, although most of its operations take no lock. We assume of the lock-free table what its implementation guarantees and what we do not prove here, i.e., that the find-or-insert of line 3 is linearizable: of several calls on the same state, exactly one inserts its node, and the others return that node.

Lemma 8.13 (The lock-free update). With the lock-free table, (i) each state has at most one node; (ii) the g of a node never increases; (iii) if UPDATE returns nil, the g of the node was at most g' at some instant of the call, and if it returns the node, it has set g to g', which was smaller than g at that instant; and (iv) a call that finds $g(x) \leq g'$ at line 5 takes no lock and writes no field of a node. Hence no improvement of g is lost, and each improvement creates exactly one entry.

Proof. (i) Line 3 inserts a node only when the state is not in the table, by the assumption on the table. (ii) The only writes of g are the initialization of a new node at line 2, before it is published by line 3, and line 8, under the lock of the node, after the test of line 7 that the new value is smaller. (iii) The read of line 5 is an atomic read of the word of g, and it returns a value that g had at some instant; since g never increases, the current value is not larger. If the value read is at most g', the current one is too, and the call is a duplicate. Otherwise the call takes the lock and tests again, and the test of line 7 and the write of line 8 are exact, since all the writes are made under that lock. (iv) Such a call executes lines 2, 3, 5 and 11, which write the spare node, private to

the worker, its counter, and whatever the find-or-insert of line 3 writes in the table itself (with the default table, a counter of the calling thread on a cache line of its own, which a growth of the table reads), but no node. □

The duplicates, which are often the most frequent case of a search with many transpositions, thus take no lock and write no node, except those that lose a race between lines 5 and 7, which take the lock of the node and then find $g(x) \leq g'$. A node is allocated once per state inserted; the nodes are never moved, and a pointer to a node stays valid until the end of the search (as the parent pointers and the entries of OPEN require).

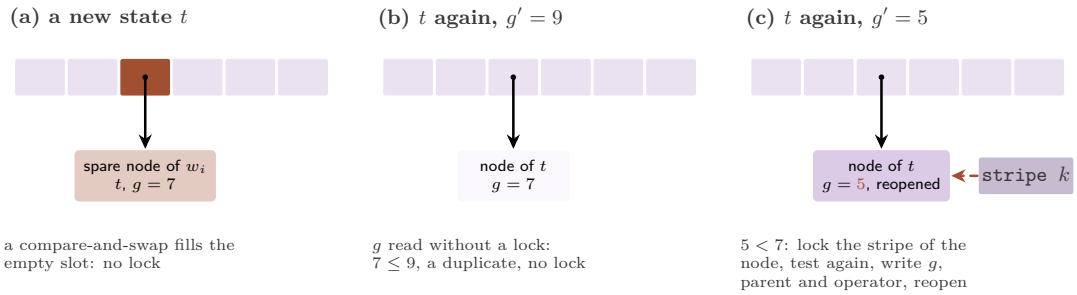


Figure 8.4. The three cases of `update` on the lock-free CLOSED. (a) A new state: a compare-and-swap fills the empty slot with the spare node of the worker. (b) A duplicate: the g read without a lock is not larger than g', and no lock is taken. (c) A better path: the worker takes the stripe of locks of the node, tests again, writes g, the parent and the operator and reopens the node.

8.6 MultiQueue A*

MQA* keeps the shared CLOSED of SPA* and replaces its OPEN, a priority queue under one lock, by a relaxed priority queue, the MultiQueue of Rihani, Sanders and Dementiev [37]: $n = 2p$ sequential priority queues, each with a separate lock, a push into a queue drawn at random, and a pop from the better of two queues drawn at random (Figure 8.5). Two workers then contend only when they happen to draw the same queue (which is rather rare with $2p$ queues), and even then the one that finds the lock taken draws again instead of waiting, while the order of the pops typically stays close to that of the keys.

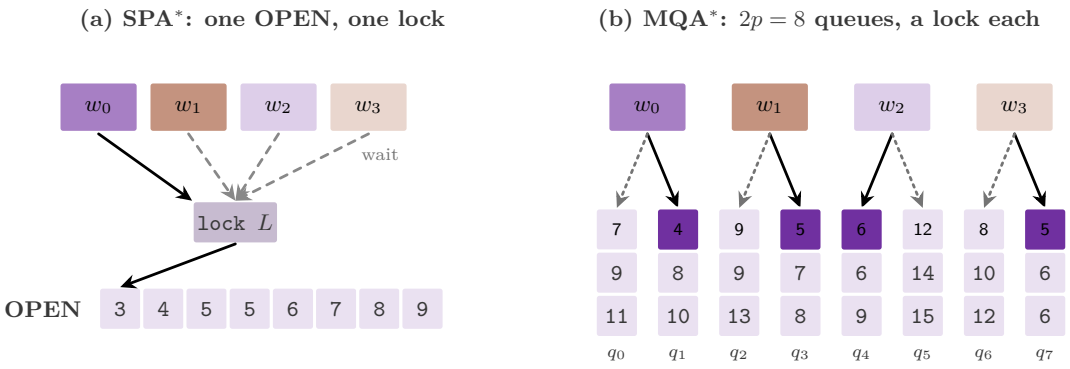


Figure 8.5. The OPEN list of SPA* and that of MQA* with $p = 4$ workers. (a) In SPA* the four workers contend for one lock, and three of them wait. (b) In MQA* each worker reads the least keys of two queues drawn at random (the pairs are adjacent here only for the sake of the drawing) and pops from the better one, under the lock of that queue only.

The MultiQueue

Each queue j keeps the key of its minimum in an atomic word top_j (∞ when the queue is empty), which the other workers read without its lock. A *try-lock* takes a lock only if it is free, and returns at once otherwise. MQA* uses the MultiQueue with a *stickiness* of 1 (`MQAStar.h:121`), i.e., each operation draws its queues anew, and with its successors pushed together, as follows.

Algorithm 8.4 (The MultiQueue of MQA*, with n queues).

```
 1  procedure PUSHALL(E)                    ▷ the entries E, into one queue
 2     loop
 3        j ← a queue drawn at random
 4        if a try-lock of j succeeds then break
 5     put the entries of E in the queue j;  top_j ← its least key
 6     unlock j
 7  function POP()
 8     loop
 9        j, j' ← two distinct queues drawn at random
10        if top_j' < top_j then j ← j'      ▷ read without the locks
11        if top_j = ∞ then return SCAN()
12        if a try-lock of j succeeds then break
13     if the queue j is empty then unlock j;  return SCAN()
14     e ← the least entry of j, removed;  top_j ← its least key, or ∞
15     unlock j;  return e
16  function SCAN()
17     for each queue j, from one drawn at random, do
18        lock j
19        if j is not empty then
20           e ← its least entry, removed;  update top_j;  unlock j
21           return e
22        unlock j
23     return nothing
```

Each queue is exact, i.e., it removes its least entry: it is an 8-ary heap behind two buffers of 16 entries, a sorted one that holds its least entries and from which a pop takes, and an unsorted one that a push fills and that goes into the heap only when it is full or when the sorted one runs out. Hence, the pop returns an entry that is not always the least of all (Figure 8.6): its *rank error* is the number of the entries present with a smaller key. In the sequential model of the analysis (the entries spread at random over the n queues and removed by the better of two random queues), the expected rank error is linear in n, and its maximum over a polynomial number of pops is $O(n \log n)$ with high probability, while a pop from a single random queue would make it grow without bound; these are results of the literature recalled in [38], which we do not prove here, and they hold in that model, not for the interleavings of a real run. Since SPA* expands p nodes at once anyway, a rank error of order $n = 2p$ departs from the order of A* by somewhat more than SPA* already does, and arguably not by much. Also, POP returns nothing only after SCAN has found all the queues empty, each locked in turn; with pushes running at the same time this does not prove that the structure is empty (a push may fill a queue already scanned), and MQA* therefore does not use it for its termination.

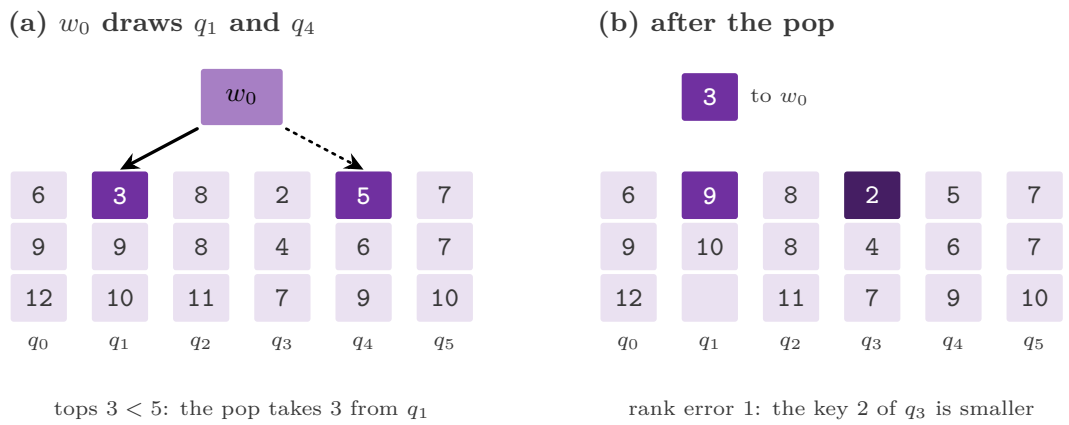


Figure 8.6. A pop from a MultiQueue of six queues (their least keys on top). (a) w_0 draws q_1 and q_4 and compares their least keys, 3 and 5. (b) It removes 3 from q_1, whose next key becomes its top; the rank error of this pop is 1, since the key 2 of q_3 is smaller.

Two refinements of the MultiQueue save cache misses. A worker may keep the two queues it drew for s operations, the stickiness, which keeps their lines in its cache, and draw again after s operations or when a lock is held (Figure 8.7). The rank errors then grow (about linearly in s, according to the measures available, since there is no theory yet), and MQA* uses $s = 1$, which keeps the model of the bounds above. Also, a worker that has

several entries to push at once, as the successors of a node, may push them all into one queue under one lock (PUSHALL, `MQAStar.h:299`). This takes one lock per expansion instead of one per successor, and relaxes the order as a stickiness of $|F|$ would, since the successors of a node then sit in the same queue; MQA* does so, and the bounds of the model then no longer apply as stated.

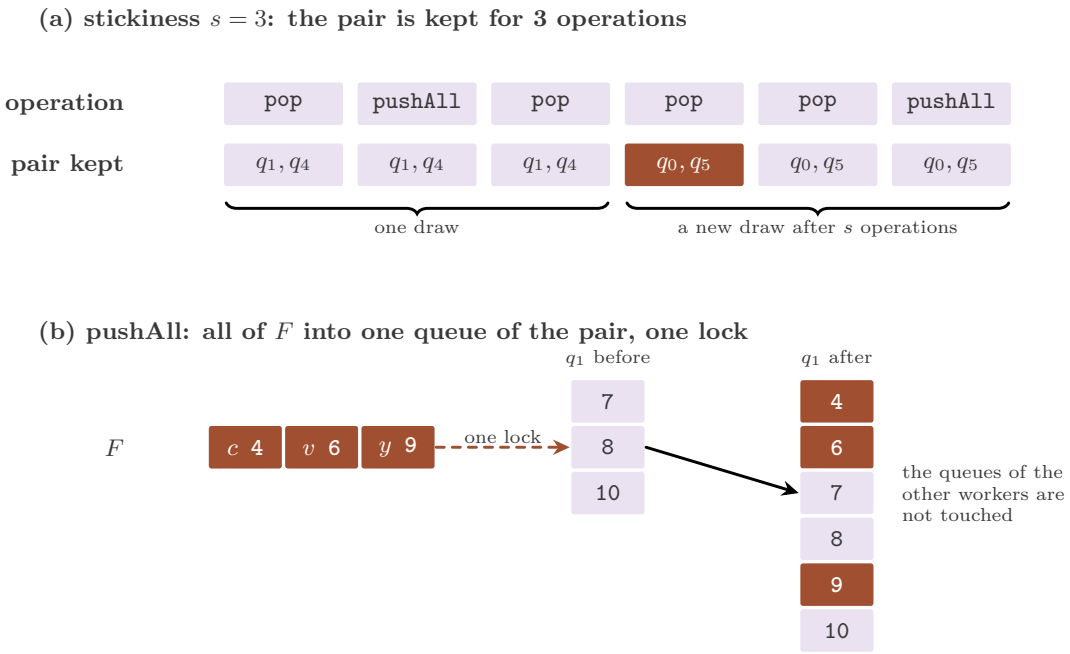


Figure 8.7. The two devices of the MultiQueue that save cache misses. (a) With stickiness $s = 3$ a worker keeps the pair of queues it drew for three operations and then draws a new pair. (b) A pushAll puts all the successors F of a node into one queue of the pair, under one lock, where they take their places among the keys of that queue.

The algorithm and its termination

Algorithm 8.5 (MQA*). Here, OPEN is the MultiQueue of Algorithm 8.4 and the atomic counter m of the entries that are in OPEN or held by a worker replaces busy and the lock L of SPA*.

```
 1  procedure MQA*(s₀, p)
 2     CLOSED ← {s₀}, with g(s₀) ← 0;  OPEN ← {(f(s₀), g(s₀), s₀)}
 3     z ← ∞;  m ← 1;  done ← false
 4     in parallel, on each worker i = 0, ..., p − 1 do
 5        while not done do
 6           e ← a pop of OPEN, possibly nothing
 7           if no entry was popped then
 8              if m = 0 then done ← true
 9              else wait a little
10              continue
11           (f, g, u) ← e;  F ← ∅
12           if f < z then F ← EXPAND(u, g)          ▷ Algorithm 8.2
13           m ← m + |F| − 1;  push the entries of F in OPEN
14     return the path to the goal of cost z, if z < ∞
```

This is the pseudocode of `MQAStar.h:19`. Since the entries of OPEN are not ordered across the queues, an entry popped with $f \geq z$ does not mean that OPEN has no better entry, and it is dropped (line 12) instead of ending the search. The end is decided by m only (Figure 8.8). The wait of line 9 is the backoff of SPA*, and the node limit is handled as there.

We now show that m counts all the entries in flight, whatever the interleaving of the workers.

Lemma 8.14 (The invariant of m). At every instant, $m = |\text{OPEN}| + H + Q$, where $|\text{OPEN}|$ is the number of entries in the queues, H the number of the workers that hold a popped entry and have not yet executed the addition of line 13, and Q the number of entries that have been counted at line 13 and are still to be pushed.

Hence $m = 0$ if and only if OPEN is empty and no worker holds an entry.

Proof. At line 3, $m = 1$ and OPEN holds the root. A pop (line 6) moves one entry from OPEN to a holding worker and leaves m unchanged. In turn, the atomic addition of line 13 ends the holding of one worker (H falls by 1) and counts the $|F|$ entries to be pushed (Q rises by $|F|$), and it adds $|F| - 1$ to m (when $|F| = 1$ the code skips the addition, which would add 0, and the holding ends at the same point of line 13). Each push of line 13 moves an entry from Q to OPEN. The three terms are nonnegative, which gives the last claim. □

The order of line 13 matters. Indeed, the entries of F are counted *before* they are pushed, and therefore m never misses an entry. Had they been counted after the pushes, another worker could pop one of them, expand it and subtract its 1 before it was added, and read $m = 0$ while entries were still in flight. Also, the code makes one atomic addition per expansion (`MQAStar.h:293`, an addition of $|F| - 1$ modulo 2^{64}, which is a subtraction of 1 when $F = \emptyset$), and none when $|F| = 1$, instead of one before and one after the pushes, which halves the traffic on the cache line of m. All the workers share that line, on which m comes first (`MQAStar.h:249`); we remark that the line is not padded after m, so that the first fields of CLOSED, which follow m in memory, may share it.

Theorem 8.15 (Correctness and termination of MQA*). If h is admissible and MQA* ends without reaching the node limit, it returns an optimal path (or none, when no goal is reachable); on a finite graph with nonnegative costs it ends.

Proof. MQA* is well-formed for the reasons given for SPA* in Theorem 8.10, the expansion being the same (Algorithm 8.2), and an entry dropped at line 12 has $f \geq z$, to which condition (b) does not apply. A worker sets done only when it reads $m = 0$, and then, by Lemma 8.14, no entry is in OPEN and none is held, i.e., none is in flight, and Theorem 8.6 gives $z = C^*$. A worker whose pop fails while $m > 0$ waits for the workers that hold entries or are about to push them, and these end their step without waiting for anything but locks of single queues; Proposition 8.7 gives the termination. □

The order of the pops changes only how many nodes are expanded before z is optimal and how many are reopened, never the cost of the path.

step	OPEN (all queues)	w_0 holds	w_1 holds	m	z
start	s_0 4			1	∞
w_0 pops		s_0 4		1	∞
w_0: $\|F\| = 3$	a 4, t 8, d 9			3	∞
both pop	d 9	a 4	t 8	3	∞
w_1: goal t	d 9	a 4		2	8
w_0: $F = \emptyset$	d 9			1	8
w_1 drops d			d 9 $\geq z$	0	8
w_0 finds nothing				0: end	8

Figure 8.8. The counter m of MQA* in a run with two workers (the entries with their f). The expansion of s_0 adds $|F| - 1 = 2$ before its three successors are pushed; the goal t sets $z = 8$ and adds -1; the expansion of a prunes its successors ($F = \emptyset$); d, popped with $f = 9 \geq z$, is dropped; w_0, whose pop then fails, reads $m = 0$, from which it concludes that no entry is left.

Proposition 8.16 (The cost of a step of MQA*). A step of a worker that pops an entry takes one lock of a queue for the pop and one for PUSHALL (none when $F = \emptyset$), the locks of Algorithm 8.2, and at most one atomic addition on m; unlike the lock L of SPA*, none of these locks is taken by every step of every worker. Each operation on a queue j costs that of the 8-ary heap of j, i.e., $O(\log |q_j|)$ comparisons in the worst case, $|q_j|$ being the number of entries of j.

Proof. The counts are read on Algorithms 8.4 and 8.5; PUSHALL takes one lock for all the entries of F, and the addition of line 13 is skipped when $|F| = 1$. The locks of a failed try-lock are not taken. The cost of a heap is

that of §3.2, the buffers only lowering it on average. □

The proposition does not bound the number of the draws of a pop, which grows with the contention for the queues; we have no bound on it in a real run, beyond the remark that with $2p$ queues and at most $p-1$ other locks held at any instant a draw finds a free queue rather often.

The key of OPEN

With integer costs the key of an entry of MQA* (and of PA*SE) is the packed word of §3.4,

$$\mathrm{key}(g,h) = 2^{32}(g+h) + (2^{32}-1-g), \qquad f(k) = \lfloor k/2^{32} \rfloor,$$

i.e., f in the high half and the complement of g in the low half (Figure 8.9), so that the comparison of two keys is one comparison of integers and gives the order of Lemma 3.6: by increasing f and, for equal f, by decreasing g, provided that $g, h \geq 0$ and $g+h < 2^{32}$, which the integer costs of the domains of the library satisfy by a wide margin. With real costs the key is $f = g+h$ alone, and the ties are broken by the queues. Line 12 of Algorithm 8.5 reads f from the key by the shift above.

The preference of the deeper node within a layer of equal f matters for a relaxed queue rather more than for an exact one, as we now explain. The last layer $f = C^*$ often holds a large part of the nodes (on the sliding-tile puzzles, for instance), and a search that prefers the deeper nodes typically reaches a goal after a part of that layer (§2.8). With f only, instead, the pops of the layer come in an arbitrary order, and typically much more of it is expanded before a goal is found. Since a MultiQueue scatters the entries of one layer over $2p$ queues, only the key can carry the preference for depth into each of them.

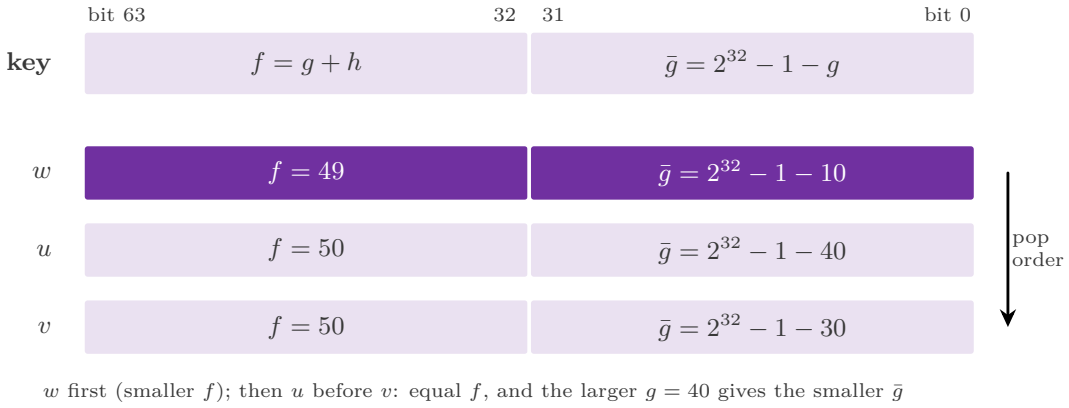


Figure 8.9. The packed key of MQA* and PA*SE: f in bits 63 to 32, the complement $\bar{g}$ of g in bits 31 to 0. The node w comes first by its smaller f; u comes before v, of the same f, since its larger g gives a smaller $\bar{g}$.

As for the expansion and CLOSED, the engineering is that of SPA* (prefetches, cache lines, lock-free duplicates). Finally, the test `mqa-thread` runs MQA* with 1, 2, 3 and 8 workers on two instances of Korf of the 15-puzzle and on the map of Romania, and with 8 and with 0 workers (which `normalize()` turns into 1) on the corner cases of `check_astar_edges` (e.g., an initial state that is a goal, one move and a node limit); `mqa-openmp` does the same on OpenMP.

8.7 The master-worker farm

The farm with feedback is a master-worker form of the centralized approach of section 3.2 of [1], written with the farm of the executor of the library (a farm of FastFlow [12] on that runtime). A master owns OPEN, CLOSED, all the nodes and the incumbent, and therefore needs no lock. It sends the best nodes of OPEN, in tasks of up to `batch` nodes, to the workers that are free (the farm schedules them on demand). A worker generates the successors of the nodes of its task, with their costs and their heuristic values, and sends them back on the feedback channel (Figure 8.10). The master then checks them against CLOSED and puts the new and the improved ones in OPEN. In other words, the scheme is that of SPA* with the master in the place of the locks.

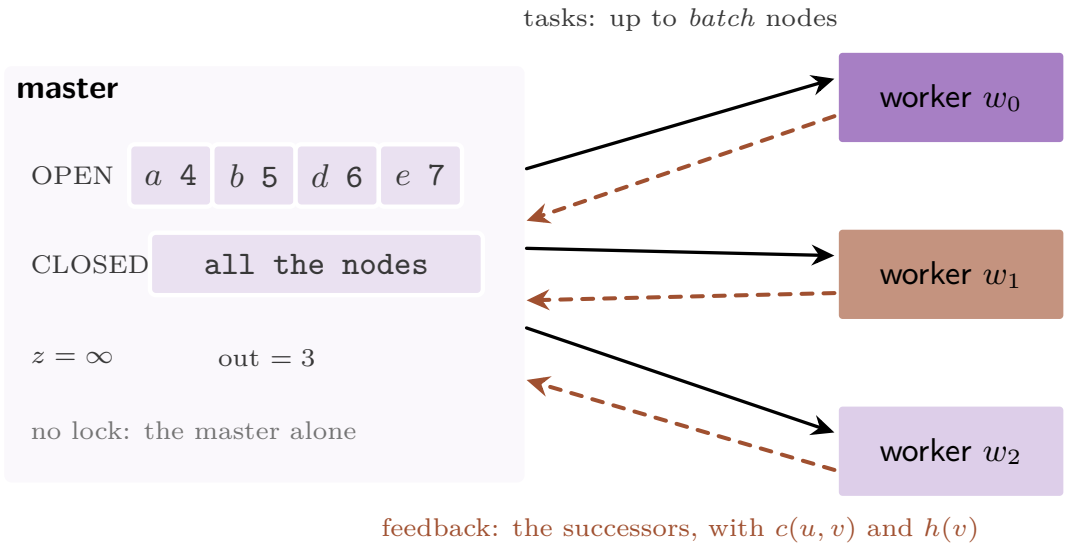


Figure 8.10. The farm with feedback: only the master holds OPEN, CLOSED, the incumbent z and the number out of the tasks at the workers; the tasks (solid) carry up to `batch` nodes to a free worker, and the feedback channel (dashed) brings back their successors with their costs and heuristic values.

Algorithm 8.6 (The farm). INSERT is the insertion of the sequential A* on the lists of the master (a new state goes in OPEN; a state reached with a smaller g is updated, reopened and put in OPEN again; otherwise it is a duplicate), and the stale entries of OPEN are dropped when they reach its top, as in A*.

```
 1  procedure MASTER(s₀, p)
 2     z ← ∞;  out ← 0;  INSERT(s₀, 0, nil)
 3     loop
 4        DISPATCH()
 5        if out = 0 and OPEN has no node of f < z then return
 6        receive a task T back from a worker;  out ← out − 1
 7        for each successor v of a node u of T, with c(u, v) and h(v) do
 8           if g(u) + c(u, v) + h(v) < z then INSERT(v, g(u) + c(u, v), u)
 9  procedure DISPATCH()
10     while out < p and OPEN has a node of f < z do
11        T ← ∅
12        while |T| < batch and OPEN has a node of f < z do
13           u ← the first node of OPEN, removed;  close u
14           if u is a goal then z ← min(z, g(u)), recording u if g(u) < z
15           else T ← T ∪ {u}
16        if T = ∅ then return
17        send T to a free worker;  out ← out + 1
18  procedure WORKER(T)                          ▷ on each of the p workers
19     for each u ∈ T, add to T the successors v of u, with c(u, v), h(v)
20     send T back to the master
```

This is the pseudocode of `FarmAStar.h:17`; the goal test is at the selection (line 14), which is the expansion point of the master, and Figure 8.11 draws one return of a task. A task carries, for each of its nodes, the pointer to the node and a copy of its operator, and the worker reads of a node only its state and its h, which never change; hence the master may update the other fields of a node that is out (its g, its parent, its operator) without a race. Lines 6 to 8 are made in two phases, as the expansion of SPA* is.

Algorithm 8.7 (The return of a task, on the master).

```
1  function BACK(T)
2     out ← out − 1
3     for each successor v in T do                ▷ phase 1
4        compute the hash of v;  prefetch its place in CLOSED
5     for each successor v of a node u of T, with c(u, v) and h(v) do
```

```
 6        if g(u) + c(u, v) + h(v) < z then INSERT(v, g(u) + c(u, v), u)
 7     if the master stores at least node_limit nodes then
 8        the limit is hit, and DISPATCH sends nothing more
 9     put T in the list of the free tasks;  DISPATCH()
10     return out > 0 or (the limit is not hit and OPEN has a node of f < z)
```

The farm of the executor calls BACK on each task that comes back, and it ends when BACK returns false, i.e., when the test of line 5 of Algorithm 8.6 holds (or the limit is hit and no task is out). The tasks are reused through the list of free tasks, and the farm allocates no memory for them in the steady state.

As for SPA*, the correctness of the farm rests on an invariant, here on the counter out, from which Theorem 8.6 applies.

Lemma 8.17 (The invariant of out). Whenever the master is between two of its steps (at line 5 of Algorithm 8.6, or at the end of BACK), every entry in flight is in OPEN or belongs to one of the tasks that are out, as one of their nodes or as a successor of one, and out is the number of these tasks.

Proof. Only the master changes OPEN, z and out, and it handles one returned task at a time. An entry leaves OPEN only at line 13, and it is either a goal, processed at once (line 14), or put in a task T, which leaves at line 17 with out raised by 1. The task holds its nodes and then their successors until it comes back, when line 6 lowers out by 1 and lines 7 and 8 insert the successors that are not pruned, before the master's next step. □

Theorem 8.18 (Correctness and termination of the farm). If h is admissible and the farm ends without reaching the node limit, it returns an optimal path (or none, when no goal is reachable); on a finite graph with nonnegative costs it ends.

Proof. The search is well-formed: a node selected at line 13 is either a goal (line 14) or carried in a task, whose successors are inserted at line 8, before the next test of line 5. Note that $g(u)$ at line 8 is the current g of the parent, which may be smaller than when the task left, if u was improved meanwhile. It is still the cost of a path to u, and $g(u) + c(u, v)$ is therefore the cost of a path to v, and u, reopened, is selected again. By Lemma 8.17, out $= 0$ and no node of $f < z$ in OPEN mean that no entry with $f < z$ is in flight, and Theorem 8.6 gives $z = C^*$. A worker never waits for anything but a task, and the master waits at line 6 only when out > 0; Proposition 8.7 gives the termination. □

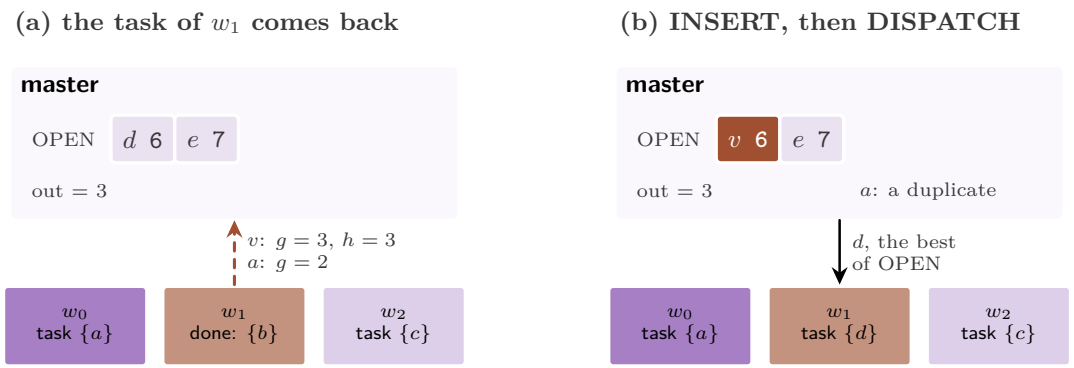


Figure 8.11. A return of a task. (a) The task of w_1, the node b, comes back with its successors v (with $g = 3$ and $h = 3$) and a (with $g = 2$). (b) The master inserts v in OPEN, counts a, already closed with $g = 1$, as a duplicate, and dispatches the best node d of OPEN to the free worker w_1; out is 3 again.

Proposition 8.19 (The work of the master). At any instant at most p tasks, and hence at most $p \cdot$ batch nodes, are out at the workers. For each task of b nodes whose successors are S, the master makes b removals of non-stale entries from OPEN (besides the stale entries it drops on the way), $|S|$ computations of a hash, at most $|S|$ lookups in CLOSED and at most $|S|$ insertions in OPEN, while the workers generate the $|S|$ successors in parallel.

Proof. Line 10 dispatches only while out $< p$, and a task has at most batch nodes (line 12). The counts follow from Algorithms 8.6 and 8.7: each successor is hashed in the first phase of BACK, and it is passed to INSERT, which makes one lookup and at most one insertion, unless it is pruned. □

Since the master does all the bookkeeping (the lookups in CLOSED, the updates of OPEN and the selections), it is typically a bottleneck unless the generation of the successors, the only part done by the workers in parallel,

costs much more than the bookkeeping of the same successors. Also, the batch trades the number of the tasks against the order of the expansions, since the nodes of a task are the batch best of OPEN at the time of the dispatch, not at the time of their expansion. The farm runs `threads` + 1 threads, and with `PBEFS_PIN` the master takes the first core of the affinity mask and the worker i the core $i+1$ (`FarmAStar.h:186`). Finally, the test `farm-thread` (and `farm-openmp`, `farm-fastflow`) runs the farm with 1, 2, 3 and 8 workers and tasks of 1 and of 4 nodes, on the 15-puzzle, on a random grid and on the map of Romania, proves a grid split by a wall unsolvable, checks the node limit and, with 8 workers and with the number of workers and the batch at 0, the corner cases of `check_astar_edges`.

8.8 Parallel A* for slow expansions

The searches above expand in parallel nodes whose g may not be optimal yet, and pay for it with re-expansions. PA*SE of Phillips, Likhachev and Koenig [6] expands in parallel only states whose g is already optimal (or within a factor of it), proved so by a pairwise heuristic, at the price of a test before each expansion; hence, no state is expanded twice. With a weight $w > 1$ on the heuristic it is the wPA*SE of section 4 of [6].

Independence and safety

Let $h(t, s)$ be the pairwise heuristic of the domain, an estimate of the cost $d(t, s)$ of a path from t to s, and let $h(s)$ be the heuristic of the search, an estimate of the cost $h^*(s)$ from s to the nearest goal. PA*SE needs the following properties, which we call (H): (H1) $h(t, s) \le d(t, s)$ for all t and s (the pairwise heuristic is admissible), (H2) h is admissible, and (H3) $h(t) \le h(t, s) + h(s)$ for all t and s (the forward-backward consistency of [6], a triangle inequality that the paper states for $h(\cdot, \cdot)$, in the form that the proofs below use; it holds when $h(t)$ is the pairwise heuristic to the nearest goal). The library leaves them to the `Domain` (`Domain.h:295`); the pairwise heuristics of the library are the distance between two cells of a grid, the sum of the displacements of the tiles of a sliding-tile puzzle and the number of the cells that are dirty in t and clean in s of the vacuum world, each with the matching heuristic h (the distance to the goal cell, the Manhattan distance and the number of the dirty cells), and each satisfies (H) since it is a distance, or a count of the moves that remain, for which the triangle inequality holds. OPEN is ordered by $f' = g + wh$, with $w \ge 1$ the weight of the options (1 for PA*SE), and the inflation ε_p of the test below is at least w (the constructor raises it to w when it is smaller, `PASE.h:222-239`).

Definition 8.20 (Independence and safety). For $\varepsilon_p \ge 1$, a state s is *independent* of a state t if

$$g(s) - g(t) \le \varepsilon_p \, h(t, s). \tag{8.4}$$

A state s of OPEN is *safe* if it is independent of each state t being expanded (the set BE, with its g at the expansion) and of OPEN with $f'(t) < f'(s)$.

Intuitively, a path through t reaches s with a cost at least $g(t) + d(t, s) \ge g(t) + h(t, s)$; hence, if s is independent of t, such a path cannot lower $g(s)$ below $g(s)/\varepsilon_p$ (for $g(t)$ optimal). The states t with $f'(t) \ge f'(s)$ need no check, since s is independent of them: by (H3), $f'(s) \le f'(t)$ and $w \le \varepsilon_p$ give

$$g(s) - g(t) \le w\big(h(t) - h(s)\big) \le w\, h(t, s) \le \varepsilon_p \, h(t, s), \tag{8.5}$$

which is Theorem 2 of [6] for OPEN, and the same argument for BE, whose states keep the g of their expansion (it is the observation of the section "Algorithm" of GePA*SE [122]). Figure 8.12 draws (8.4) on a line of states: the states that depend on t are those above a cone with its apex in t and slopes $\pm\varepsilon_p$, and a larger ε_p makes the cone steeper and more states independent.

The algorithm

Algorithm 8.8 (PA*SE). One lock L protects OPEN, BE and CLOSED; OPEN is an ordered set by increasing f' and, for equal f', by decreasing g, CLOSED holds all the states generated, each with the flag closed, and K is the number of the states of OPEN examined (all, if 0).

```
1  procedure PA*SE(s₀, p)
2     CLOSED ← {s₀}, with g(s₀) ← 0;  OPEN ← {s₀};  BE ← ∅
3     in parallel, on each worker do
4        lock L
```

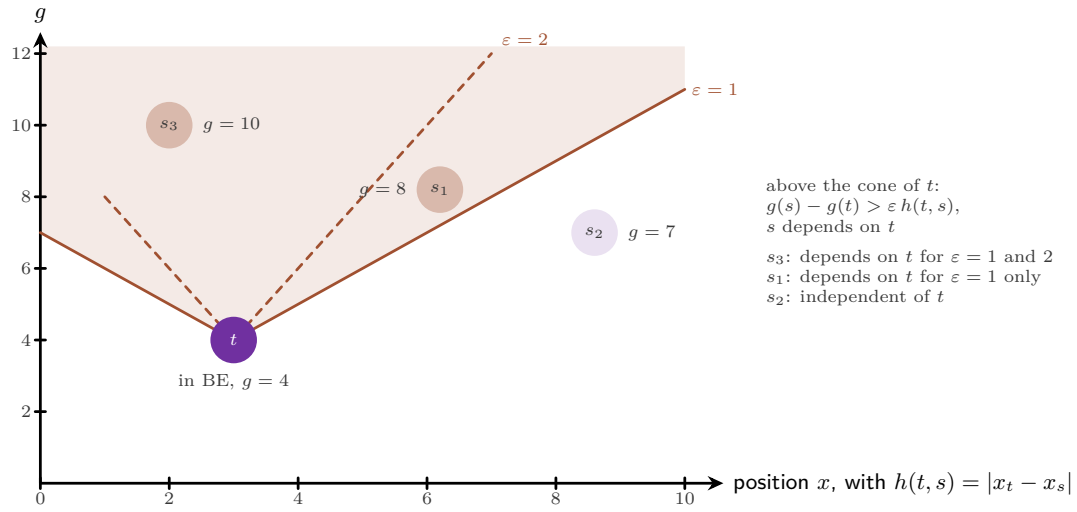


Figure 8.12. The independence (8.4) on a line of states, with $h(t,s) = |x_t - x_s|$ and g on the vertical axis. The state t, being expanded, casts a cone of slope ± 1 for $\varepsilon_p = 1$ (solid, shaded above) and ± 2 for $\varepsilon_p = 2$ (dashed); a state above the cone of t may still get a smaller g through t and is not safe. s_3 depends on t for both values, s_1 only for $\varepsilon_p = 1$, and s_2 is independent of t.

```
 5        while not done do
 6           s ← the first state of OPEN, among its first K, that is SAFE
                and, if a goal, ENDS
 7           if there is none then
 8              if OPEN = ∅ and BE = ∅ then done ← true;  break
 9              wait on L until OPEN or BE change;  continue
10           remove s from OPEN
11           if s is a goal then record s;  done ← true;  break
12           add s to BE;  close s;  unlock L
13           S ← the successors of s, with their costs   ▷ the slow part
14           lock L
15           for each (v, c(s, v)) ∈ S do
16              g' ← g(s) + c(s, v)
17              if v ∉ CLOSED then
18                 add v to CLOSED and OPEN, with g(v) ← g', parent(v) ← s
19              else if v is not closed and g' < g(v) then
20                 g(v) ← g';  parent(v) ← s;  reorder v in OPEN
21           remove s from BE;  wake the waiting workers
22        unlock L
23  procedure SAFE(s)
24     for each t ∈ BE with f'(t) < f'(s) do
25        if g(s) − g(t) > ε_p h(t, s) then return false
26     for each t ∈ OPEN with f'(t) < f'(s) do
27        if g(s) − g(t) > ε_p h(t, s) then return false
28     return true
29  procedure ENDS(s)
30     return no state t of OPEN ∪ BE has g(t) + ε_p h(t) < g(s)
```

This is the pseudocode of `PASE.h:18`, i.e., Algorithm 1 of [6] with the scan of line 6 bounded, the test of BE restricted to the states of smaller key by (8.5), and the test ENDS for the goals. The wait of line 9 is on a *condition variable*, which releases L and suspends the worker in one atomic step until another worker wakes it (line 21), and then takes L again. Figure 8.13 draws one selection, in which the first state of OPEN depends on the state being expanded and is skipped, and the second is safe and moves to BE. Note that, by line 19, a closed state is never improved and never reopened, which is sound only because of Lemma 8.21 below. The code makes the selection as follows, with the options `candidates` (K, 64 by default) and `prune` (true by default) and with z the least g of a goal in OPEN (∞ if none), which the insertions of line 18 and 20 keep (`PASE.h:377-441` and `453-476`).

Algorithm 8.9 (The selection of PA*SE, under L).

```
 1  function SELECT()
 2     for each x of OPEN, in order, until K pass line 3 (all if K = 0), do
 3        if PRUNED(x) then continue
 4        if SAFE(x) and (x is not a goal or ENDS(x)) then return x
 5     return nil
 6  function SAFE(s)
 7     for each t of BE, and then of OPEN, in the order of the set, do
 8        if f'(t) ≥ f'(s) then go on with the next set
 9        if not (t is in OPEN and PRUNED(t)) and not INDEPENDENT(s, t)
             then return false
10     return true
11  function ENDS(s)
12     for each t of OPEN, and then of BE, in order, with f'(t) < g(s) do
13        if g(t) + ε_p h(t) < g(s) then return false
14     return true
15  function INDEPENDENT(s, t)
16     δ ← 0 with integer costs, 1e−9 · max(1, |g(s)|) with real costs
17     return g(s) − g(t) ≤ ε_p h(t, s) + δ
```

Here PRUNED(x) holds when `prune` is set, x is not a goal and $g(x) + \varepsilon_p h(x) \geq z$; SAFE does not check the pruned states of OPEN, of which a selected state is independent anyway (the proof of Lemma 8.21). The scans of lines 7 and 12 stop at the first state whose key is not smaller, since OPEN and BE are sorted by f' first; with integer costs the key is the packed word of §8.6 of f' and g, followed by the address of the node, and with real costs the pair $(f', -g)$ followed by the address, which makes the keys distinct. Thus, the states of equal f' are not checked, as (8.5) allows, and a layer of equal f' costs a scan of BE per state instead of a scan of the layer. ENDS needs only the states of key below $g(s)$, since a state t with $f'(t) \geq g(s)$ has $g(t) + \varepsilon_p h(t) \geq f'(t) \geq g(s)$ ($w \leq \varepsilon_p$). The tolerance δ of line 16, the `cost_slack()` of `Node.h` with the relative tolerance `real_cost_tolerance`, keeps the rounding of real costs from making the goal depend on the states of equal f'; with real costs, Lemma 8.21 and Theorem 8.22 are therefore proved only for the exact test, and we do not quantify how the tolerance, and the rounding of g along a path, weaken their bounds. With integer costs the weight is the fraction $w = a/b$, with $b = 1$ if w is an integer and $b = 256$ otherwise (a rounded up, so that the weight is not smaller than the one given), and the key is $bg + ah$, exact, packed with g in one word, which needs $bg + ah < 2^{32}$; the `Solution` of a goal of cost c carries the lower bound $\lceil c/\varepsilon_p \rceil$ (c/ε_p with real costs) on C^* (`PASE.h:222-239` and `287-295`).

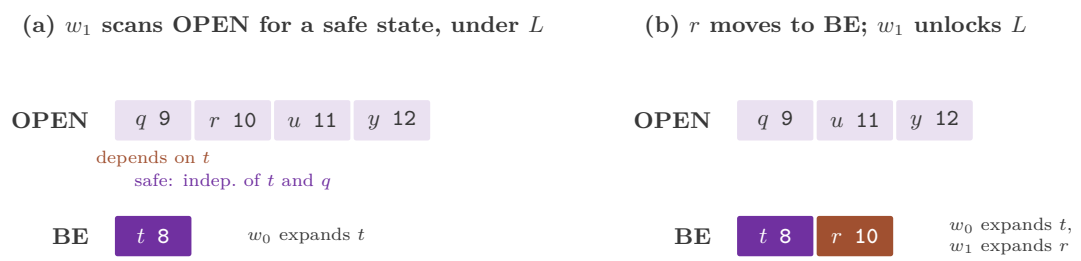


Figure 8.13. A selection of PA*SE. (a) w_0 expands t; under L, w_1 scans OPEN: q depends on t, while r is independent of t and of q, the only state of OPEN with a smaller f. (b) r leaves OPEN and joins BE, and w_1 releases L and expands it.

Correctness

We first show that a state found safe has a nearly optimal g; this is the invariant that allows line 19 to leave the closed states alone. Call a state *done* if it is closed and outside BE, i.e., its expansion is complete.

Lemma 8.21 (Safe states are nearly optimal). Under (H), with $w \leq \varepsilon_p$ and integer costs or the exact test, each state s that PA*SE selects has $g(s) \leq \varepsilon_p\, g^*(s)$; with $\varepsilon_p = 1$, each state is expanded with its optimal g, and hence at most once.

Proof. By induction on the selections (cf. the proof of Theorem 8.6, of which this is the analogue for BE). Suppose that the states selected before have been selected with $g \leq \varepsilon_p g^*$, and that s is found safe with $g(s) > \varepsilon_p g^*(s)$. Let P be an optimal path from s_0 to s, and let t be the first state of P that is not done; it exists, since s is in OPEN. We first show that t is in OPEN or in BE, with $g(t) \leq \varepsilon_p g^*(t)$. If t is closed, it is in BE, since it is

not done, and it was selected before s, so that the induction hypothesis gives the bound. Otherwise, if $t = s_0$, then t is in OPEN with $g(t) = 0$. Otherwise the predecessor t' of t on P is done, and it was selected with $g(t') \le \varepsilon_p g^*(t')$; its expansion generated t (the successor that the pruning of the parent of §8.3 skips is the node whose expansion gave t' its g, which is closed, while t is not), found t not closed (a state closed then would be closed now), and lines 17 to 20 left t in OPEN with $g(t) \le g(t') + c(t', t) \le \varepsilon_p(g^*(t') + c(t', t)) = \varepsilon_p g^*(t)$, using $\varepsilon_p \ge 1$ and $c \ge 0$; $g(t)$ has not increased since. Furthermore, $t \ne s$, since, if $t = s$, the bound just proved contradicts the assumption on s. Since t precedes s on an optimal path, $g^*(s) = g^*(t) + d(t, s)$, and

$$g(s) - g(t) > \varepsilon_p g^*(s) - \varepsilon_p g^*(t) = \varepsilon_p\, d(t, s) \ge \varepsilon_p\, h(t, s)$$

by (H1), so that s depends on t. If t is a pruned state of OPEN, then $g(t) + \varepsilon_p h(t) \ge z$, while $g(s) + \varepsilon_p h(s) < z$ if s is not a goal (line 3 did not skip it), and $g(s) \le g(t) + \varepsilon_p h(t)$ if s is a goal (ENDS(s) holds); in both cases $g(s) - g(t) \le \varepsilon_p(h(t) - h(s)) \le \varepsilon_p h(t, s)$ by (H3), with $h(s) = 0$ for a goal, and s is independent of t. Otherwise, if $f'(t) < f'(s)$, the state t, in BE or in OPEN, is checked by SAFE (lines 24 to 27), and s is not safe; if $f'(t) \ge f'(s)$, then s is independent of t by (8.5). All the cases contradict the dependence just shown. With $\varepsilon_p = 1$ the bound is $g(s) = g^*(s)$, and a state, once closed, is never put back in OPEN (line 19). □

For the goal returned at line 11, safety is not enough when there are several goal states: Lemma 8.21 bounds $g(s)$ by $\varepsilon_p g^*(s)$, the cost of a cheapest path to *that* goal, which may exceed $\varepsilon_p C^*$, and a safe goal may be returned while a goal of smaller f is still waiting. Algorithm 1 of [6] and its theorems are stated for one goal state; the test ENDS of line 6 extends the bound to a goal region.

Theorem 8.22 (Bounded suboptimality of PA*SE). Under (H), with $w \le \varepsilon_p$ and integer costs or the exact test, if PA*SE returns a goal s, then $g(s) \le \varepsilon_p C^*$; with $\varepsilon_p = 1$ the path returned is optimal. The pruning of the option `prune` does not change the bound.

Proof. Let P be an optimal path to a goal s', with $g^*(s') = C^*$, and let t be the first state of P that is not done when s is selected. As in the proof of Lemma 8.21, t is in OPEN or in BE with $g(t) \le \varepsilon_p g^*(t)$ (with the g of its expansion if in BE), and by (H2) and $h^*(t) = C^* - g^*(t)$ (since t is on an optimal path)

$$g(t) + \varepsilon_p h(t) \le \varepsilon_p g^*(t) + \varepsilon_p h^*(t) = \varepsilon_p C^*.$$

ENDS(s) holds when s is selected, i.e., no state of OPEN or BE has $g + \varepsilon_p h < g(s)$; hence $g(s) \le g(t) + \varepsilon_p h(t) \le \varepsilon_p C^*$ (if $t = s$, the same inequality holds with $h(s) = 0$). The pruning of line 3 of Algorithm 8.9 skips only states x with $g(x) + \varepsilon_p h(x) \ge z$, the g of a goal in OPEN: it never skips a goal, and if it skips the state t above, then $z \le \varepsilon_p C^*$, and the goal of cost z is already within the bound (the selections do not depend on the pruning in the argument above, which uses only ENDS). Finally, the path rebuilt through the parents has cost at most $g(s)$, as in Lemma 2.10, since all the writes of g and of the parents are made under L. □

These are Theorems 1 and 3 of [6], stated there for one goal state; the proofs above make explicit where the forward-backward consistency (H3) enters, i.e., only in (8.5), which restricts the checks to the states of smaller key. Exercise 8.6 asks for a graph with two goal states on which the test of safety alone returns a goal beyond the bound; the documentation of the library (`docs/theory/parallel-astar.md`) reports that, before ENDS, the test of safety alone returned paths up to 4% longer than the optimum on 8 of the 50 navigation instances of the map den520d with 5 workers (a count that we have not rerun).

Finally, we show when PA*SE neither waits forever nor runs forever.

Proposition 8.23 (Progress and termination of PA*SE). Let OPEN be not empty and BE empty, and let x be the first state of OPEN that line 3 of Algorithm 8.9 does not skip. Then SELECT returns x. Hence no worker waits forever while another could proceed, and on a finite graph PA*SE ends, with a goal or with OPEN and BE empty, in which case no goal is reachable.

Proof. The state x exists, since a goal in OPEN is never skipped and, without a goal in OPEN, $z = \infty$ skips nothing. Each state t of OPEN before x is skipped, i.e., $g(t) + \varepsilon_p h(t) \ge z > g(x) + \varepsilon_p h(x)$ (or x is a goal, with $g(x) = z$, see below), and SAFE does not check it (line 9), while BE is empty; hence x is safe, and it is among the first K states examined, since the skipped ones are not counted (line 2). If x is a goal, then $g(x) = z$, since a goal of $g = z$ has $f' = g$ not larger than that of x and comes first; the states before it are skipped, so that $g(t) + \varepsilon_p h(t) \ge z = g(x)$, and the states after it have $f' \ge g(x)$; hence ENDS(x) holds. A worker waits at line 9 only when no state is selected, i.e., when BE is not empty, while another worker expands outside L. That worker then takes L, changes BE at line 21 and wakes all the waiting workers, and no wake-up is lost, since the wait releases L atomically and all the changes are made under L. By line 19 a closed state never re-enters

OPEN, so that each state is closed at most once, and on a finite graph the selections are therefore finitely many. When OPEN and BE are both empty, all the states reachable from s_0 have been closed, since the successors of a closed state were all added at line 18, and no goal was selected, so that none is reachable. □

We remark that the proposition relies on two choices of the code (`PASE.h:399-416` and `456-461`): the states skipped at line 3 do not count among the K states examined, and SAFE does not check the pruned states of OPEN. Were they counted, with `prune` and $K > 0$ the first K states of OPEN could all be skipped while BE is empty, which is possible when $\varepsilon_p > w$ or with many states of $f' = z$ and $h = 0$, and no state would be selected; were they checked, the first state not skipped would be independent of them under (H3), as in the proof of Lemma 8.21, but with a pairwise heuristic that violates (H3) it could depend on one of them, and again no state would be selected. In either case each worker would wait at line 9 for a change that no worker makes. Since the goal of cost z is in OPEN until it is selected, and it is never skipped, the scan always reaches a state that is not skipped.

The next proposition states in one place what the two choices preserve; the comment of SAFE in the code refers to it.

Proposition 8.24 (The pruning in the selection). Under (H), with $w \le \varepsilon_p$, $\varepsilon_p \ge 1$, integer costs or the exact test, and `prune` set, let SAFE not check the pruned states of OPEN and let the K candidates of SELECT be counted among the states not pruned, as in Algorithm 8.9. Then (i) the workers never all wait with OPEN not empty and BE empty, and (ii) a goal s returned by PA*SE has $g(s) \le \varepsilon_p C^*$.

Proof. (i) With BE empty, the first state x of OPEN not pruned is among the candidates and is safe, and if it is a goal it has $g(x) = z$ and ENDS(x) holds (Proposition 8.23); hence SELECT returns a state. (ii) Since z is the least g of a goal in OPEN, it never increases, and the goal of cost z stays in OPEN until a goal is selected, since line 3 never skips a goal. Let P be an optimal path from s_0 to a goal, so that $g^*(v) + h^*(v) = C^*$ for each state v of P. We show, by induction on the selections, that each state v of P is selected with $g(v) \le \varepsilon_p g^*(v)$ unless $z \le \varepsilon_p C^*$ already holds. Let v be selected with $g(v) > \varepsilon_p g^*(v)$, and let q be the first state not done of an optimal path from s_0 to v, followed by the part of P after v, which is an optimal path to the goal; as in the proof of Lemma 8.21, $q \ne v$, q is in OPEN or BE with $g(q) \le \varepsilon_p g^*(q)$ (the states before it are done, and selected with that bound by the induction, unless $z \le \varepsilon_p C^*$ already, which is the claim), and v depends on q. If q is checked by SAFE, or if $f'(q) \ge f'(v)$, the dependence is contradicted as in that proof; otherwise q is a pruned state of OPEN, i.e., $g(q) + \varepsilon_p h(q) \ge z$, and by (H2)

$$z \le g(q) + \varepsilon_p h(q) \le \varepsilon_p g^*(q) + \varepsilon_p h^*(q) = \varepsilon_p C^*.$$

When the goal s is selected, ENDS(s) holds; if $z \le \varepsilon_p C^*$, then $g(s) \le z$, since the goal of cost z is in OPEN or is s, and $g(s) \le \varepsilon_p C^*$; otherwise the first state not done of P has $g \le \varepsilon_p g^*$, and the argument of Theorem 8.22 gives the bound. □

We remark that the hypotheses of the proposition are those of Theorem 8.22, and that (ii) uses (H3) only through (8.5) and the selection of each state at most once (line 19); the same argument holds for ePA*SE, whose SAFE skips the pruned states of OPEN and whose scan does not count them (`EPASE.h:584-587`), with $\bar{g}$ for the states of BE.

Proposition 8.25 (The cost of a selection). Under L, a call of SELECT evaluates the pairwise heuristic at most K ($|\text{BE}_<| + |\text{OPEN}_<|$) times ($|\text{OPEN}|$ in place of K if $K = 0$), where $\text{BE}_<$ and $\text{OPEN}_<$ are the states of BE and of OPEN with f' smaller than that of the last candidate scanned, and $|\text{BE}| < p$; ENDS reads at most $|\text{OPEN}| + |\text{BE}|$ states per goal examined. Each insertion in OPEN, removal and reordering costs $O(\log |\text{OPEN}|)$ in the ordered set, and the successors of an expansion cost one lookup in CLOSED each, under L.

Proof. SELECT tries at most K candidates, and SAFE makes one test per state of BE and per state of OPEN before the candidate with a smaller f'; BE holds the states being expanded, at most one per other worker. OPEN and BE are B-trees, whose operations are logarithmic, and lines 15 to 20 make one lookup per successor. □

The engineering

We now turn to the implementation, which we have kept close to the pseudocode. OPEN and BE are ordered sets (B-trees of the default data structures, `PASE.h:357-358`). The scan of line 6 is bounded by K states, which bounds the time under L (a state further down would have to be checked against a longer and longer prefix of OPEN), and by Proposition 8.23 the bound costs no progress under its condition. Also, the hashes of the

successors are computed outside L, and under L their places in CLOSED are prefetched before any lookup, and the lock is therefore held for about one cache miss instead of one per successor.

As Proposition 8.25 shows, PA*SE has a price, since a safe state costs a scan of BE and of the states of OPEN before it, all under one lock. PA*SE may therefore pay off only when an expansion is much slower than these scans (perhaps by orders of magnitude), as in motion planning with collision checks, which is the setting of [6]. To study this regime on any domain, `SlowDomain< D >` (`SlowDomain.h`) wraps a domain and adds to each expansion a given number of iterations of $j \leftarrow 11i \bmod 9999943$, $i \leftarrow j$, which keep a core busy without touching the memory, as in the experiment of section 3.1.1 of [5]; it forwards the other members of the domain (its features, its abstraction, its nblocks, its pairwise heuristic) when the domain has them. The test `pase-thread` checks, with 1, 2, 3 and 8 workers, that PA*SE finds the distances of a breadth-first search on 21 states spread over the state space of the 8-puzzle, and that it solves a grid, a slowed grid and a grid of the game of life optimally. The test also checks that the counter `reexpanded`, which `PASE` increments when it selects a state already expanded, is 0 on the slowed grid, i.e., that no closed state is reopened (the code never reopens one, as Lemma 8.21 requires). Finally, it checks that $\varepsilon_p = 1.5$ gives a cost within the bound of Theorem 8.22 on the 15-puzzle, that wPA*SE with $w = 1.3$ (taken as 333/256), 2 and 3, scanning 64 states or all of OPEN, stays within wC^* with no re-expansion, that a grid split by a wall is unsolvable, and the corner cases of `check_astar_edges` with 8 and with 0 workers.

8.9 Parallel A* for slow evaluations of the edges

When the cost of an expansion is that of evaluating its edges (a collision check, a simulation), the unit of the parallel work may be the edge rather than the state: a state with many successors then keeps several workers busy, and an edge whose state stops being the best is not evaluated at all. This is the idea of ePA*SE of Mukherjee, Aine and Likhachev [121], which the library implements in `EPASE.h` with its weighted variant w-ePA*SE, the generalized GePA*SE of Mukherjee and Likhachev [122] and the anytime A-ePA*SE (and A-GePA*SE) of Yang, Mukherjee and Likhachev [123], together with two baselines, the Parallel Weighted A* (PwA*, `PWAStar.h`) of [121] and [124], and the lazy search MPLP of Mukherjee, Aine and Likhachev [124] (`MPLP.h`).

The domains by edges

A search by edges needs a `Domain` that evaluates one edge at a time. The library states it with three concepts of `Domain.h`:

- `EdgeDomain` (`Domain.h:625`): `for_each_action( s , f )` calls `f( a )` for each action a of s, an index below $2^{32} - 1$, without evaluating the edge (s, a), which costs little; `evaluate_edge( s , h_s , a , f )` evaluates it, which may cost much, and calls `f( t , c , h_t )` if the edge exists, t being the state that a leads to and c the cost of the edge, and nothing otherwise. The successors given by `for_each_successor` are those that `evaluate_edge` gives on the actions of s, so that a search by edges and one by states search the same graph.
- `HeterogeneousEdgeDomain` (`Domain.h:641`): an `EdgeDomain` whose `expensive_edge( s , a )` tells, without evaluating it, whether the edge is expensive; GePA*SE evaluates the cheap edges of a state together, and each expensive one apart.
- `LazyEdgeDomain` (`Domain.h:657`): an `EdgeDomain` whose `lazy_edge( s , h_s , a , f )` calls `f( t , c , h_t )`, at little cost, if the edge may exist, c being a lower bound on its cost, and nothing if it surely does not; `evaluate_edge` then gives the same t and a cost not smaller than c, or no edge. The heuristic has to be consistent also with the costs c, as the lazy searches use them.

To reproduce the regimes of these papers on any domain of the library, the class template `SlowEdgeDomain< D >` of `SlowDomain.h` makes the edges of a domain D slow to evaluate, cheap or expensive, and some of them missing, with the work given by an `EdgeWork`:

```
struct EdgeWork {

 unsigned long cheap = 0;          ///< iterations of a cheap edge
 unsigned long expensive = 0;      ///< iterations of an expensive edge
 double expensive_share = 0;       ///< the share of the expensive edges
 double missing_share = 0;         ///< the share of the edges that, once
                                   ///< evaluated, turn out not to exist
 std::uint64_t seed = 1;           ///< the seed of the two choices
```

```
};  // end( struct( EdgeWork ) )
```

The actions of a state s are the indices $0, \ldots, k-1$ of its successors in the order in which D gives them; evaluating the edge (s, a) costs the work of `SlowDomain` for `cheap` or `expensive` iterations; an edge is expensive, or missing, if a hash of the hash of s, of a and of the seed, mapped to $[0, 1)$, falls below `expensive_share`, or below `missin g_share` for an independent hash, so that a given edge is always of the same kind. A missing edge is reported by `lazy_edge` with its cost, and found missing only by `evaluate_edge`, as a collision check would do; `for_each_ successor` evaluates all the edges of the state, paying their work, and gives the existing ones. The graph is that of D without the missing edges, on which the heuristics of D stay admissible and consistent (removing edges does not shorten any path), and the lower bounds of `lazy_edge` are the costs themselves. With cheap and expensive work in the ratio of 1 to 30 on half of the edges this is the regime of the 2D grids of GePA*SE (section "Evaluation" of [122]); the 2D navigation of their experiments is the domain `RobotNav2D` of Chapter 14.

ePA*SE and GePA*SE

`EPASE< D , Executor >` (`EPASE.h:246`, with D a `PairwiseDomain` and an `EdgeDomain`) keeps the states whose g has improved in OPEN, each standing for all its edges (the dummy edge of the paper). When a state of OPEN is safe it is *expanded*, i.e., moved to BE with its edges put in READY, which costs no evaluation; an edge of READY is evaluated by a worker outside the lock, and its successor is relaxed under the lock; when all its edges are evaluated, a state leaves BE. The keys of OPEN, READY and BE are $f' = g + wh$ of the state, with v, the g of the state at its expansion, in place of g for BE and READY, and the search takes the first safe state of OPEN or the first edge of READY, whichever has the smaller key. In the terms of Equations 2 and 3 of [121] an edge is safe when its state is independent of BE and of the states of OPEN of smaller key; the edges of a state in BE are always safe, since the state was safe when it was expanded and its g has not changed since. The pseudocode of the library (the Doxygen comment of `EPASE.h`) is the following, with $\varepsilon \geq w$ the inflation of the independence rule (8.4) and K the states of OPEN examined (all, if 0).

Algorithm 8.10 (ePA*SE and GePA*SE).

```
 1  procedure SELECT                                  ▷ under L
 2     r ← the first edge of READY, if any
 3     for each s among the first K states of OPEN with key(s) < key(r) do
 4        if SAFE(s) and (s is not a goal or ENDS(s)) then return s
 5     return r, if any
 6  procedure SAFE(s)
 7     for each t ∈ BE ∪ OPEN with f'(t) < f'(s) do
 8        if g(s) − g(t) > ε h(t, s) then return false
 9     return true
10  procedure EXPAND(s)                               ▷ under L
11     move s from OPEN to BE
12     for each action a of s do
13        if a is cheap (GePA*SE) then add a to the batch B of s
14        else add the edge (s, a) to READY
15     if B ≠ ∅ then add B to READY
16  procedure EVALUATE(s, A)          ▷ the edges A of s, outside of L
17     S ← the successors (v, c(s, v)) of the edges of A
18     lock L
19     for each (v, c(s, v)) ∈ S do
20        g' ← g(s) + c(s, v)
21        if v is new then add v to OPEN, with g(v) ← g', parent(v) ← s
22        else if v ∈ OPEN and g' < g(v) then
23           g(v) ← g';  parent(v) ← s;  reorder v in OPEN
24     if all the edges of s are evaluated then move s from BE to CLOSED
25     unlock L
26  procedure ePA*SE(s0, p)
27     OPEN ← {s0}, with g(s0) ← 0;  READY ← ∅;  BE ← ∅
28     on the worker 0 (the dispatcher), lock L
29     while not done do
30        x ← SELECT, an edge only if a worker is idle
31        if there is none then
32           if OPEN = READY = BE = ∅ then done ← true
```

```
33          else wait on L until a set changes or a worker is idle
34       else if x is a state and a goal then record x;  done ← true
35       else if x is a state then EXPAND(x)
36       else remove x from READY;  give it to the idle worker of
            least index, which runs EVALUATE on it and waits again
```

Here ENDS(s) is the test of PA*SE (Algorithm 8.9), with the g of the expansion for the states of BE, and SAFE checks the states of BE with their g at the expansion; the states of larger key are independent of s by (8.5). With the option `prune` (true by default), line 4 also skips the states s with $g(s) + \varepsilon h(s) \geq z$, z being the least g of a goal in OPEN, and line 2 drops unevaluated the edges of READY whose state has $v + \varepsilon h \geq z$ (EPASE.h:568-623). The options are the following.

```
struct EPASEOptions : public ParallelOptions {

 /// the inflation \f$ \varepsilon \geq 1 \f$ of the independence rule,
 /// raised to the weight \f$ w \f$ of SearchOptions if smaller: the cost
 /// is within a factor \f$ \varepsilon \f$ of the optimum; w is that of
 /// the keys, \f$ f' = g + w h \f$ (w-ePA*SE if above 1), and the first
 /// weight of the anytime search
 double eps = 1;

 /// GePA*SE: the cheap edges of a state evaluated together [see
 /// HeterogeneousEdgeDomain]; all the edges are expensive if the Domain
 /// does not tell them apart
 bool generalized = false;

 /// one worker selects and expands, the others evaluate, woken one at a
 /// time (the thread management of ePA*SE); otherwise every worker
 /// selects, expands and evaluates
 bool dispatcher = true;

 /// the states of OPEN examined for a safe one, from the first (0: all),
 /// those that the prune skips not counted
 std::size_t candidates = 64;

 /// once a goal of cost \f$ z \f$ is in OPEN, neither the states nor the
 /// edges of the states with \f$ g + \varepsilon h \geq z \f$ are expanded
 /// or evaluated, which keeps the bound [see the file]
 bool prune = true;

 /// the decrease \f$ \delta > 0 \f$ of the weight of the anytime search
 /// (SearchOptions::anytime), which lowers the weight by it down to 1, with
 /// \f$ \varepsilon \f$ equal to the weight, after each path found, and
 /// records in Solution::improvements the best path of each iteration with
 /// the bound \f$ c / w_i \f$ on the optimal cost
 double weight_step = 0.5;

 /// the seconds after which the anytime search stops with the best path
 /// found (0: no limit)
 double time_limit = 0;

};  // end( struct( EPASEOptions ) )
```

The weight w of `SearchOptions` is that of the keys (w-ePA*SE if above 1) and the first weight of the anytime search, and `eps` is raised to it when smaller (EPASE.h:262-271). With `dispatcher` (the default) only the worker 0 scans OPEN, and it wakes one worker per edge, the idle one of least index, so that the workers beyond those that the independent edges keep busy never run: this is the thread management of [121], whose threads are spawned as needed; the states are expanded by the dispatcher itself, which costs little, also while all the workers are busy. Without the dispatcher, each worker selects, expands and evaluates under the lock, as in PA*SE (the variant "without thread management" of the paper), and with one worker the search is the sequential eA* of the paper, which evaluates an edge only when its state is the best: when the successor reached by the first edge evaluated has a smaller key than its parent, the other edges of the parent wait, and with an informative heuristic

many of them are never evaluated. With `generalized` (GePA*SE) the cheap edges of a state, as `expensive_edge` tells them in a `HeterogeneousEdgeDomain`, are evaluated as one item of READY, given to one worker, and each expensive edge apart; without expensive edges it is PA*SE on the edges, and without cheap ones it is ePA*SE. Note that the lock is a `std::mutex` whatever the executor, since the workers that wait sleep on condition variables, while those of a spin lock would take the cores of the workers that evaluate the edges.

Let $g^*(s)$ be the cost of a cheapest path to s and $v(s)$ the g of s at its expansion, and let (H) hold (with h admissible and the pairwise heuristic admissible and forward-backward consistent) and $w \leq \varepsilon$. The proofs are those of the page on parallel A* of the documentation, which we follow.

Theorem 8.26 (Bounded expansions). With integer costs or the exact test, when a state s is expanded, $g(s) \leq \varepsilon\, g^*(s)$.

Proof. Suppose not, and let s be the first state expanded with $g(s) > \varepsilon\, g^*(s)$. Let π be a cheapest path from s_0 to s and s_m the first state of π that is in OPEN or in BE (there is one, s itself being in OPEN). The states of π before s_m are closed: each was expanded before s, hence with $v \leq \varepsilon g^*$, and its edges were evaluated, each relaxing its successor with a g not larger than v plus the cost of the edge. Thus, if $s_m = s_0$ then $g(s_m) = 0$, and otherwise, s_{m-1} being its predecessor on π, $g(s_m) \leq v(s_{m-1}) + c(s_{m-1}, s_m) \leq \varepsilon g^*(s_{m-1}) + c(s_{m-1}, s_m) \leq \varepsilon g^*(s_m)$; if s_m is in BE, also $v(s_m) \leq \varepsilon g^*(s_m)$ directly. Let $\bar{g}$ be $g(s_m)$ if s_m is in OPEN and $v(s_m)$ if it is in BE: s is independent of s_m with $\bar{g}$, either by the test at the expansion, if the key of s_m is smaller, or by (8.5). Hence $g(s) \leq \bar{g} + \varepsilon h(s_m, s) \leq \varepsilon g^*(s_m) + \varepsilon g^*(s_m, s) = \varepsilon g^*(s)$, where $g^*(s_m, s)$ is the cost of the part of π from s_m to s, which is at least $h(s_m, s)$ by (H1); this is a contradiction. □

The proof is that of Theorem 1 of [121], written for the test that skips the states of larger key in BE (the improvement of GePA*SE [122]) and with v for the states of BE. It gives the properties of the search.

Proposition 8.27 (The properties of ePA*SE). Under the assumptions of Theorem 8.26: (i) a state is expanded at most once and an edge is evaluated at most once; (ii) the goal s returned has $g(s) \leq \varepsilon C^*$, and $g(s) = C^*$ when $\varepsilon = 1$, with or without `prune`; (iii) on a finite graph the search ends, with a goal if one is reachable.

Proof. (i) A state in BE or CLOSED is never put back into OPEN (a better g for it is counted as a duplicate, `EPASE.h:657-691`), and its edges are made once, at its expansion, and evaluated once each. (ii) A goal s is returned when it is safe and ENDS(s) holds, i.e., no state t of OPEN or BE has $\bar{g}(t) + \varepsilon h(t) < g(s)$ (which holds for the states of key not smaller than $g(s)$, since $w \leq \varepsilon$). A cheapest path to a goal has a first state s_m in OPEN or BE, with $\bar{g}(s_m) + \varepsilon h(s_m) \leq \varepsilon g^*(s_m) + \varepsilon h^*(s_m) = \varepsilon C^*$ by the argument of Theorem 8.26 and the admissibility of h; hence $g(s) \leq \varepsilon C^*$. With `prune`, the workers expand no state x with $g(x) + \varepsilon h(x) \geq z$ and evaluate no edge of a state with $v + \varepsilon h \geq z$, but the argument uses only ENDS at the selection of the goal, which the pruning does not change; if the state s_m above is such a state, then $z \leq \varepsilon C^*$ already. (iii) Each step either expands a state, at most $|V|$ times by (i), or hands out an edge, at most $|E|$ times, or waits; it waits only while an edge is being evaluated or no worker is idle, since with no edge in READY and none being evaluated BE is empty (a state stays in BE only while one of its edges is in READY or being evaluated), and then the first state of OPEN not skipped is selected (the proof of Proposition 8.23). If a path exists, OPEN or BE holds a state of a cheapest path to a goal until a goal is returned, so that the search does not end without one. □

Safety alone gives $g(s) \leq \varepsilon g^*(s)$ for a goal s (Theorem 3 of [121]), which is εC^* only if s is the goal of a cheapest path; with a goal region, as in the 2D navigation, a safe goal may be returned while a goal of smaller f is still waiting. The documentation of the library reports that on the 50 instances of the map den520d with $w = \varepsilon = 1$ and 5 workers the test of safety alone returned paths up to 4% longer than the optimum on 8 instances with PA*SE, and that on den520d and hrt201n with 5 to 20 threads the code of [122] returned paths up to 24% longer on up to 11 instances out of 50, while with ENDS each path was optimal; we have not rerun these counts. The remark after Proposition 8.23 on the states skipped by `prune`, which do not count among the K examined and which SAFE does not check (`EPASE.h:584-587`), applies to EPASE as well.

The anytime search

With `anytime` (A-ePA*SE, and A-GePA*SE with `generalized`), the search repairs its path with the weights $w_0 > w_1 > \cdots > w_k = 1$, w_0 the weight of the options, $w_{i+1} = \max\{1, w_i - \delta\}$ with δ the `weight_step` (0.5 by default), and $\varepsilon = w_i$, as ARA* [48] does (Chapter 6). A state of BE or CLOSED whose g improves goes to INCON if it was expanded in the current iteration, and back to OPEN otherwise; a goal selected as safe ends the iteration, which waits for the edges being evaluated, publishes the best path with the bound c/w_i on C^* in `Solution::improvements`, and moves INCON and BE back into OPEN with the keys of w_{i+1} (`EPASE.h:846-888`); a state whose edges the pruning has left unevaluated goes to INCON as well, since its successors have not been relaxed. The edges already evaluated stay with their state and are relaxed again, without a new evaluation,

when the state is expanded again; thus each edge is evaluated at most once over all the iterations. The search stops after the iteration of weight 1, or at the first iteration that ends after `time_limit` seconds (0 for no limit), with the best path found.

Theorem 8.28 (The anytime bound). At the end of the iteration i, the best path costs at most $w_i C^*$, and C^* at the end of the last one.

Proof. At the start of an iteration each state whose successors may have been relaxed with a g larger than its present one is in OPEN (INCON and BE are moved there), so that each other state u not expanded in the iteration is *consistent*: its successors x have $g(x) \leq g(u) + c(u, x)$. The proof of Theorem 8.26 then holds within the iteration, with $\varepsilon = w_i$, if the states of π before s_m may also be consistent: for such a state $g(s_j) \leq g(s_{j-1}) + c(s_{j-1}, s_j)$, and by induction from s_0, or from the last state expanded in the iteration, $g(s_j) \leq w_i g^*(s_j)$. A state of INCON is not in OPEN, but it was expanded in the iteration with $v \leq w_i g^*$, and the induction uses v for it. Hence the goal selected at the end of the iteration, safe and with no state of OPEN or BE of key smaller than its g, has $g \leq w_i C^*$, as in Proposition 8.27 (Theorem 1 of [123]). □

Note that the iteration ends when a goal is safe, not when its g is below the least key of OPEN as in ARA*: the latter would also need the least key of BE, whose edges being evaluated may still lower the g of the goal. The counters of the `Solution` count in `expanded` the states expanded (more than once only in the anytime search, which counts the repeated ones in `reexpanded`), in `evaluated` the edges evaluated, in `generated` the edges that exist, in `iterations` the iterations of the anytime search, and in `expanded_by` the expansions and evaluations of each worker, which shows how many workers have run.

Two baselines

PwA* (`PWAStar< D , Executor >`, PWAStar.h:100, with the options `PWAStarOptions`, which add nothing to `ParallelOptions`) is the weighted A* whose expansion evaluates the edges of the state in parallel. The pseudocode of the library is the following.

Algorithm 8.11 (PwA*).

```
 1  procedure PwA*(s₀, p)
 2     OPEN ← {s₀}, with g(s₀) ← 0;  CLOSED ← ∅
 3     while OPEN ≠ ∅ do
 4        s ← the state of OPEN of least f' (larger g on ties)
 5        if s is a goal then return the path to s
 6        add s to CLOSED
 7        the p workers evaluate the edges of s, each taking the next
 8           action not yet taken                  ▷ in parallel
 9        for each (v, c(s, v)) found, in the order of the actions do
10           if v ∉ CLOSED and g(s) + c(s, v) < g(v) then
11              g(v) ← g(s) + c(s, v);  parent(v) ← s;  put v in OPEN
```

The worker 0 runs the weighted A* and expands; at each expansion the workers, the worker 0 among them, take the actions of the state from an atomic counter and evaluate their edges, and the worker 0 applies the results once all are in (PWAStar.h:201-345). Since the results are applied in the order of the actions, the order of the expansions, and hence the path, the expansions and the edges evaluated, are those of the sequential weighted A* without reopening, whatever the number of workers, which the test `edge-thread` checks; its cost is within wC^* for a consistent heuristic (the bound of the weighted A* without reopening, [48] and Chapter 6), and its speedup is bounded by the branching factor, since the workers evaluate the edges of one state at a time.

MPLP (`MPLP< D , Executor >`, MPLP.h:135, with D a `LazyEdgeDomain` and the options `MPLPOptions`, which add nothing to `ParallelOptions`, whose weight is that of the lazy searches) is a lazy search: the successors of a state are generated without evaluating their edges, each edge taking its lower bound $\underline{c}$, and the edges are evaluated by the other workers while the search goes on. With $c(e)$ the cost of the edge e if evaluated (∞ if it does not exist) and $\underline{c}(e)$ otherwise, the pseudocode of the library is the following.

Algorithm 8.12 (MPLP).

```
 1  procedure MPLP(s₀, p)                          ▷ on the worker 0
 2     z ← −∞;  Π ← ∅
 3     repeat
 4        π ← weighted A* from s₀ on the costs c, each edge read at the
```

```
 5            expansion of its state, the new edges queued for evaluation
 6        if there is no π then return no path
 7        z ← max(z, c(π));  if π ∉ Π then add π to Π, and its edges
 8            not evaluated to the front of the queue
 9        wait until an edge evaluated has changed its cost
10     until a path is accepted
11  procedure EVALUATE                          ▷ on the workers 1, ...
12     loop: take the first edge e of the queue, evaluate it
13        for each π ∈ Π through e do
14           if e does not exist then drop π
15           else if all the edges of π are evaluated and c(π) ≤ z then
16              accept π
```

The edges discovered go to the back of the queue, those of a path found to its front, which are the priorities 1 and 2 of the paper. The paper runs the search, the check of the paths and the hand-out of the edges on three threads; in the library the check of a path is done by the worker that evaluates its last edge, and the workers take the edges from the queue themselves, so that p workers evaluate on $p-1$ of them instead of $p-3$ (`MPLP.h:305-465`). The search does not repeat while no evaluation has changed a cost (line 9), since it would find the same path; the paper repeats it at once. With one worker the search evaluates the edges of each path it finds before searching again, which is the lazy shortest path of Dellin and Srinivasa [125].

Theorem 8.29 (MPLP). Let the heuristic be admissible and consistent with the lower bounds $\underline{c}$. If a lazy search finds no path, none exists; each path accepted costs at most wC^*; and on a finite graph MPLP ends, unless it reaches the node limit.

Proof. Since $\underline{c} \le c$, each path of the graph is a path of the graph searched at line 4, and it is not longer there; hence, if no path is found, none exists, and each π found costs, with the costs of line 4, at most w times the optimum of a graph whose edges are not longer, i.e., at most wC^*, by the bound of the weighted A* without reopening with a consistent heuristic (Lemma 3 of [124]). So $z \le wC^*$, and a path accepted at line 16, all of whose edges are evaluated, costs $c(\pi) \le z \le wC^*$ (Theorem 5 of [124]). For the termination, each edge is evaluated at most once, hence the changes of cost are finitely many, and after the last one the search finds a path π, whose cost at line 4 is its cost, since no evaluation will change it; its edges not evaluated are put at the front of the queue, and when the last of them is evaluated (with its lower bound as its cost, or a change would follow) π is accepted, since $c(\pi) \le z$. If the path found was already in Π, line 7 checks it again, since z may have grown. □

The paper proves the completeness on a finite graph through the paths without repetition, which are finitely many (its Theorem 2). The counters of MPLP count in `expanded` the expansions of all the lazy searches, in `iterations` the searches, in `generated` the edges discovered, in `evaluated` the edges evaluated, and in `expanded_by` the edges evaluated by each worker.

The test `edge` checks, without parallelism, that the actions and the evaluated edges of `SlowEdgeDomain` give the successors of an expansion, and that PwA*, eA*, GeA* and the lazy search with one worker find paths within the bound on 21 states of the 8-puzzle with 15% of the edges missing, PwA* evaluating at most the 4 edges of each expansion and eA* and GeA* expanding no state twice, and the same searches on a grid and on the 2D navigation of the map den520d; the test `edge-thread` runs ePA*SE, w-ePA*SE and GePA*SE with and without the dispatcher, A-ePA*SE with three iterations, PwA* and MPLP with 1, 2, 3 and 8 workers on the 8-puzzle, on a grid with 8 moves and missing edges and on a small 2D navigation with two walls.

8.10 The searches compared

The following table gathers, for each parallel search of the library, what the workers share, where the duplicates are detected, how the end is detected and in which regime the search may pay off. Its overheads are those that one may expect from the scheme; only measurements on a given machine and domain (e.g., by `AStarBench`, §8.11) can decide among the searches.

Search	Shared	Duplicates	End	Regime
SPA*	OPEN, CLOSED	in CLOSED	busy, under L	slow expansions
MQA*	$2p$ queues, CLOSED	in CLOSED	counter m	contended OPEN

Search	Shared	Duplicates	End	Regime
farm	nothing	at the master	at the master	slow generation
PA*SE	OPEN, BE, CLOSED	in CLOSED	safe goal	very slow expansions
ePA*SE, GePA*SE	OPEN, READY, BE, CLOSED	in CLOSED	safe goal	slow edges
PwA*	the edges of a state	in CLOSED	as A*	slow edges, wide states
MPLP	queue of edges, paths	in each lazy search	a path accepted	lazy successors
HDA*	batches	at the owner	deficit counting	balanced owners
HDA*, MPI	messages	at the owner	Mattern	cluster memory
Safe PBNF	abstract graph	in an nblock	no nblock left	small scopes

In the table, the end of PA*SE is a safe goal that passes ENDS, or OPEN and BE both empty, and that of ePA*SE the same with READY empty as well; the end of MPLP is a path all of whose edges are evaluated and whose cost does not exceed that of every path found, or a lazy search that finds no path; that of HDA* on MPI is the test of the four counters of Mattern, and that of Safe PBNF is the state in which no nblock is free and none is in use. The last column is the regime in which the search may pay off: expansions slow with respect to the time under the shared lock (SPA*), a lock of OPEN that is the bottleneck (MQA*), a generation of successors much slower than their bookkeeping (the farm), expansions much slower than the scans of the test of safety (PA*SE), edges much slower to evaluate than the bookkeeping, and fewer than the workers or of very different costs (ePA*SE, GePA*SE), a branching factor about the number of the workers (PwA*), successors known before their edges are evaluated (MPLP), a distribution that balances the load (HDA*), more states than the memory of one machine holds (HDA* on MPI), and an abstraction with many nblocks of small scope (Safe PBNF).

HDA* is treated in Chapter 9, Safe PBNF in Chapter 11, and §12.10 compares the further schemes of Chapter 12. In the comparison of section 3.2 of [5], on the 15-puzzle and on the 24-puzzle, HDA* with Zobrist hashing is faster than Safe PBNF on the "hard" instances, while Safe PBNF is faster on the "easy" ones, where the initial burst of HDA*, i.e., the expansions made while the OPEN lists of the workers are still nearly empty, weighs more. The outcome on other domains and machines is not necessarily the same. Since the shared-memory searches are templates over the executor and take the same `ParallelOptions`, one can measure the same search on each runtime, and all the searches on the same runtime.

8.11 Benchmarks

The tables of this section are written by `manual/build/tables.py` from the runs of `tools/bench/campaign.sh`, whose machine and compiler are stated in the note of each table. Each run is a separate process, started by `tools/AStarBench` with the search, the runtime and the number of workers as options, and pinned by `numactl` to cores of one NUMA node, with its memory on that node and one worker per core (`PBEFS_PIN`). Each parallel run is made twice, in two processes, and the gap between the two runs of a pair is the A/A noise reported in the note. On the 15-puzzle the costs are integers and exact, and each run reports the cost it found with its status; only the solved runs enter the tables. The clock is the `Stopwatch` started at the beginning of `solve()` and read at its end. It includes the allocation of the shared structures, the creation of the threads (the benchmark gives no executor, and each `solve()` builds one), the search and the rebuilding of the path. It does not include the reading of the file of the instances, the construction of the domain and of its tables, nor the printing of the results.

Table 8.1. The centralized searches on the 15-puzzle (campaign B): SPA* and MQA* with the C++ threads, and the farm on FastFlow, on the 6 instances of Korf on which campaign B runs SPA* and MQA* (1, 3, 10, 14, 15 and 26; the farm also runs on 6 more, which the table leaves out), with the Manhattan distance, for each number of cores of the campaign. Each cell holds the seconds summed over the 6 instances, each the median of its two runs, and in parentheses the speedup over the sequential A* with the bucket queue on 1 core, summed over the same instances. The farm runs its master and one worker less than the cores, i.e., one thread per core, except on 1 core, where its master and its single worker share the core. An empty cell is a search that did not solve every instance.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

Table 8.2. Slow expansions (campaign D): the instance 79 of Korf, an "easy" one, in two parts, one with 1000 and one with 10000 iterations of the work of `SlowDomain` added to each expansion. Each part gives SPA*, MQA*, the farm (on FastFlow, with one worker less than the cores), PA*SE and HDA* with Abstract Zobrist hashing on the C++ threads, for each number of cores, and the reference is the sequential A* with the bucket queue on the same slowed domain on 1 core; the columns are those of Table 8.1, on one instance.

work 1000 per expansion

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

work 10000 per expansion

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

Both tables are placeholders until the campaign is run on an idle machine. In the meantime, we remark that the experiment of Table 8.1 measures the regime of "fast" expansions, in which the time under the shared lock (SPA*), the queues (MQA*) or the master (the farm) is often a large part of an expansion. That of Table 8.2 measures the regime of "slow" expansions, in which it is a small part and in which PA*SE is meant to work ("slow" meaning, say, a collision check per successor), the sequential A* on the same slowed domain being the reference.

Table 8.3. Slow edges: the searches by edges on the 2D navigation of the maps of the data of the library (`data/robot-nav-2d`), with the expensive diagonal motions. The table gives PA*SE, ePA*SE and GePA*SE (each with $w = 1$ and with a large weight), A-ePA*SE, PwA* and MPLP on the C++ threads, for each number of workers of the campaign; each cell holds the seconds summed over the pairs, each the median of its two runs, and in parentheses the edges evaluated relative to those of the sequential eA* (the search with one worker), the reference being the sequential A* on the same domain on 1 core.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

In Table 8.3 the work is that of the collision checks of the domain, not an artificial one, and the counts of the edges evaluated do not depend on the machine for one worker; the table is a placeholder until the campaign is run.

8.12 Notes and references

The overheads of §8.2 and their trade-off are those of the survey of Fukunaga et al. [1], section 3.1, with the precise definitions of Jinnai and Fukunaga [5], section 2.1, who also identified the band effect (load imbalance) and the burst effect (the start of HDA*) as the sources of the search overhead of the decentralized searches (their section 3.1); Kishimoto, Fukunaga and Botea [2] measured the same overheads for HDA* (their section 4.2). The stopping criterion of Theorem 8.6 is the condition stated informally in section 3.3.1 of [1] for the decentralized searches. We have written it once, for all the searches of the library, as a property of the entries in flight; each search then only has to prove that its test implies it. Counting the entries in transit is classical in distributed termination detection (Dijkstra and Scholten [35], Mattern [11]).

SPA* is Algorithm 2 of [1], the baseline of that survey; Burns et al. [3] found that the shared OPEN list makes it slower than the sequential A* when the expansions are fast, even with lock-free data structures, which is the motivation of the decentralized searches. Our SPA* differs from Algorithm 2 of the survey in its termination (busy counted under the lock of OPEN, instead of a separate procedure of termination detection) and in its CLOSED, whose duplicates usually take no lock. The MultiQueue is due to Rihani, Sanders and Dementiev [37]; Postnikova et al. [38] studied it as the priority scheduler of parallel algorithms on graphs. We are not aware of a published A* on a MultiQueue with the termination of Lemma 8.14; MQA* should therefore be read as an engineering choice of the library, whose expansions are those of SPA* up to the order. Also, the farm is a master-worker form of the centralized approach of section 3.2 of [1], which goes back to Kumar, Ramesh and Rao [153], written on the farm of FastFlow [12] on that runtime. PA*SE is due to Phillips, Likhachev and Koenig [6], who proved its bounded suboptimality for one goal state and measured it on motion planning with expensive collision checks; the weighted wPA*SE is their section 4. The test ENDS for a goal region and the bound K on the scan are ours. The searches by edges are due to Mukherjee, Aine and Likhachev (ePA*SE [121], whose evaluation also introduces PwA*, and MPLP [124]), Mukherjee and Likhachev (GePA*SE [122]) and Yang, Mukherjee and Likhachev (A-ePA*SE [123]); the lazy shortest path of MPLP with one worker is that of Dellin and Srinivasa [125]. Our ePA*SE differs from the paper in the test of BE restricted to the states of smaller key (the improvement of GePA*SE), in ENDS, in the pruning by the incumbent and in the dispatcher, which expands the states itself; our MPLP checks a path on the worker that evaluates its last edge, and repeats a lazy search only after a change of cost.

Further parallel schemes of the literature that the library implements, i.e., PSDD and its forms, KBFS, the portfolio of A*, PRA*, the transposition-driven scheduling of a distributed table of states, the parallel IDA* below a frontier and the parallel greedy searches, are the subject of Chapter 12; A* on graphics processors is outside the library.

8.13 Exercises

Exercise 8.1 (`ctest -R PBeFS.spa-thread`). Remove the counter busy from Algorithm 8.1, so that a worker sets done as soon as it finds no entry of $f < z$ in OPEN. On the graph of Figure 8.3, whose optimal cost is 4, give an execution with two workers in which w_0 is slow on a while w_1 expands b, v and selects the goal t with $g = 5$, and the search stops with $z = 5$. Then check that the case `spa-thread`, which compares the cost of SPA* with the known costs of two instances of Korf with 1, 2, 3 and 8 workers, passes with the counter.

Exercise 8.2 (`ctest -R PBeFS.spa-thread`). Prove that SPA* with $p = 1$ makes the expansions of A* with the same OPEN list, and that with p workers and a consistent heuristic each re-expansion of a node v happens because a node u on an optimal path to v had not been expanded with its optimal g when v was selected. Then show that the bound of Lemma 8.11 is attained, i.e., give a graph and an interleaving in which a node is expanded once per value of its g.

Exercise 8.3 (`ctest -R PBeFS.mqa-thread`). In Algorithm 8.5, replace line 13 by "push the entries of F in OPEN; m ← m + |F| − 1" (the addition after the pushes). Show an interleaving of two workers in which a worker reads $m = 0$ while an entry is in OPEN, and explain why the version of the library, with the addition first, never does (Lemma 8.14). The case `mqa-thread` runs the library's version on 1 to 8 workers.

Exercise 8.4 (`ctest -R PBeFS.mqa-thread`). Using Lemma 3.6, show that the order of the packed key of §8.6 is that of the sequential A* with the ties of f broken towards the larger g. What goes wrong in the key if $f \geq 2^{32}$, or if $g < 0$? For which domains of Chapter 14 is the packed key used?

Exercise 8.5 (`ctest -R PBeFS.farm-thread`). Show that the farm with $p = 1$ and tasks of 1 node expands the nodes in the order of A*, and that with $p > 1$ the bound of Proposition 8.19 on the nodes out is attained. Explain why the master computes g at line 8 of Algorithm 8.6 from the current g of the parent rather than from

the g it had when the task left, and why both choices would be correct. The case `farm-thread` runs the farm with tasks of 1 and 4 nodes.

Exercise 8.6 (`ctest -R PBeFS.pase-thread`). Build a small graph with two goal states and a pairwise heuristic that satisfies (H) on which PA*SE with $\varepsilon_p = 1$, without the test ENDS, returns a goal that is not optimal, and show which state of OPEN or BE makes ENDS fail for that goal. Then show that ENDS holds for every goal of a domain with one goal state as soon as it is safe. The case `pase-thread` checks the optimality on the 8-puzzle and on grids, and the bound of Theorem 8.22 with $\varepsilon_p = 1.5$ and with the weights 1.3, 2 and 3.

Exercise 8.7 (`ctest -R PBeFS.pase-thread`). Give an example in which the only safe state of OPEN is the $(K+1)$-th, so that the bound K on the scan of SELECT (Algorithm 8.9) makes a worker wait although a state could be expanded in parallel, and explain why Proposition 8.23 still holds without `prune`. Then build an example, with `prune` and $\varepsilon_p > w$, in which the first K states of OPEN are all skipped while BE is empty, and show that the scan, which does not count them, still selects a state (the remark after Proposition 8.23). The case `pase-thread` checks the optimal cost on a slowed grid with 1, 2, 3 and 8 workers, and wPA*SE with $K = 64$ and $K = 0$.

Exercise 8.8 (`ctest -R PBeFS.pase-thread`). Which problems may parallelize well with the searches of this chapter? Relate the answer to the ratio between the cost of an expansion and the time spent under the shared lock (or at the master), and to the number of distinct values of f of a problem. Then use `SlowDomain`, as the case `pase-thread` does, to make a domain of the library slow, and say from which work per expansion one may expect SPA*, the farm and PA*SE to overtake the sequential A* on a given machine.

Exercise 8.9 (`ctest -R PBeFS.farm-fastflow`). Run the same search on the three runtimes (the cases `spa-openmp`, `mqa-openmp` and `farm-fastflow` do so for SPA*, MQA* and the farm) and check that the costs agree. Explain why the expansions may not, and which of the counters of `SearchStats` are deterministic for SPA* with one worker.

Exercise 8.10 (`ctest -R PBeFS.edge`). Show that a `SlowEdgeDomain` satisfies the requirements of `EdgeDomain` and `LazyEdgeDomain`, i.e., that its actions and its evaluated edges give the successors of its expansion, and that its heuristics stay admissible and consistent although some edges are missing. Then show that eA* (ePA*SE with one worker) expands each state at most once and evaluates at most the edges of the states it expands, and give an instance on which it evaluates fewer edges than A*.

Exercise 8.11 (`ctest -R PBeFS.edge-thread`). Prove that PwA* makes the same expansions and evaluates the same edges with any number of workers, as the test checks against its run with one worker, and that its speedup cannot exceed the largest number of actions of a state. Why does the test compare the cost of MPLP with that of A* only for $w = 1$?

Exercise 8.12 (`ctest -R PBeFS.edge-thread`). Follow Theorem 8.28 on the grid of the test, with $w_0 = 3$ and $\delta = 1$: how many iterations does A-ePA*SE make, which bound does it record at each, and why does no edge get evaluated twice? The test checks three iterations whose costs do not increase and whose bounds do not exceed C^*.

9. Hash-distributed search

[Source: include/PBeFS/astar/HDAStar.h (HDAStar, ParallelOptions), include/PBeFS/astar/HDAStarMPI.h (HDAStarMPI), include/PBeFS/astar/Distribution.h (the work distributions), include/PBeFS/astar/Domain.h (FeatureDomain, IncrementalFeatureDomain, AbstractFeatureDomain, AbstractStateDomain, PerfectHashDomain), include/PBeFS/core/SearchUtils.h (ParallelParams)]

We are concerned in this chapter with Hash Distributed A* (HDA*) of Kishimoto, Fukunaga and Botea [2], the parallel search in which each state has an *owner* among the p workers, fixed by a function of the state alone. Only the owner stores the state; hence, the workers share no OPEN and no CLOSED list, and yet the duplicates are detected as in the sequential search. In the searches of Chapter 8 (from the Parallel A* with one shared OPEN list onwards) each node costs a lock, or an atomic operation, on a structure that all the workers touch. HDA* replaces them by messages between pairs of workers, at the price of the nodes that change worker and of an order of expansion that is only (and often rather loosely) that of A*. We use the notation of Chapter 1 and the invariants of A* of Chapter 2; the search overhead (SO), the communication overhead (CO) and the load balance (LB) of a parallel search, and the general question of when a parallel search may stop, are those of Chapter 8. The structure of the chapter is as follows. §9.1 states the interface (the search, its options, the concept of a distribution and the refinements of the `Domain` that the distributions need), §9.2 gives the algorithm with its invariants, the proof of its optimality and the cost of its steps, and §9.3 the mailbox through which the messages travel, with the proof that it detects the end without error. §9.4 and §9.5 are devoted to the distributions and to the incremental Zobrist hash, §9.6 to the search over MPI with the termination detection of Mattern, and §9.7 to the engineering of the implementation; the chapter ends with the benchmark, the notes and the exercises.

9.1 The search and its interface

A search is an object of the class template `HDAStar`, built from a `Domain D`, a distribution `Dist` and the options, and run by `solve()`, which returns the `Solution` (the status, the cost, the path and the counters of Chapter 2); the following declarations are quoted from HDAStar.h:138-188, where they come with their bodies.

```
template< Domain D , class Dist = StateHashDistribution< D > ,
          WorkerExecutor Executor = ThreadPool ,
          class Open = /* buckets for integer costs, a heap otherwise */ >
class HDAStar {
 public:
 HDAStar( const D & domain , const Dist & dist ,
          const ParallelOptions & options = {} );   // both must outlive it
 void set_executor( Executor & ex ) noexcept;
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };
```

Its workers run on an `Executor`, i.e., on a pool of C++ threads (the default), on OpenMP or on FastFlow [12] (§1.6). One can give the search an executor that outlives it by `set_executor()`, which saves the start of the threads at each search; otherwise, `solve()` builds one of p threads. The options are a `ParallelOptions`, i.e., the options of the sequential search plus the parameters of the parallel ones (SearchUtils.h:98-115): the number p of workers (`threads`, 1 by default), the number of nodes packed in a message (`batch`, 64), the number of expansions between two flushes of the outboxes that are not full (`flush_period`, 64) and a `seed`, which `HDAStar` does not read (a distribution takes its seed in its constructor, 1 by default). A parameter equal to 0 is taken as 1 (`normalize()`), and therefore no setting of the options can stop the exchange of the nodes. The following example, from the case `manual` of the tester, which the build compiles and runs (`ctest -R PBeFS.m`

anual), solves an instance of the 8-puzzle of optimal cost 26 with Zobrist hashing and with Abstract Zobrist hashing (§9.4) on 4 workers and reads the counters of §8.2:

```
const EightPuzzle d( EightPuzzle::parse( "7 2 4 5 0 6 8 3 1" ) );
ParallelOptions o;
o.threads = 4;
o.batch = 64;           // nodes per message (the default)
o.flush_period = 64;    // expansions between two flushes of an outbox
// the seed changes the owners of the states, not the cost
const ZobristDistribution< EightPuzzle > z( d , 7 );
const auto a = HDAStar< EightPuzzle , decltype( z ) >( d , z , o )
               .solve();
const AbstractZobristDistribution< EightPuzzle > az( d );
const auto b = HDAStar< EightPuzzle , decltype( az ) >( d , az , o )
               .solve();
const double co = a.stats.communication_overhead();  // sent / generated
const double lb = a.stats.load_balance();
const auto sent = a.stats.sent;           // nodes sent to another worker
const auto messages = a.stats.messages;   // the batches they went in
```

The same search with one MPI process per worker is

```
template< Domain D , class Dist = StateHashDistribution< D > >
class HDAStarMPI {
 public:
 HDAStarMPI( const D & domain , const Dist & dist ,
             const ParallelOptions & options = {} ,
             MPI_Comm comm = MPI_COMM_WORLD );
 [[nodiscard]] Solution< state_t< D > , cost_t< D > > solve( void ) const;
 };
```

(HDAStarMPI.h:111-144), described in §9.6; there `threads` is not read, the number of workers being that of the processes of the communicator.

The distributions

A *distribution* is a class with a method `owner( s , p )` that returns the worker in $\{0, \ldots, p-1\}$ which owns the state s, and with a static method `name()` that gives its name in the notation HDA*[*hash*, *abstraction*] of Jinnai and Fukunaga (section 2.6 of [5]). Each distribution of the library has a constructor that takes the `Domain` and a seed, e.g.,

```
template< FeatureDomain D >
class ZobristDistribution {
 public:
 explicit ZobristDistribution( const D & domain , std::uint64_t seed = 1 );
 [[nodiscard]] std::uint64_t hash( const state_t< D > & s ) const;
 [[nodiscard]] std::uint32_t key( const state_t< D > & s ) const;
 [[nodiscard]] unsigned owner( const state_t< D > & s , unsigned p ) const;
 [[nodiscard]] std::uint32_t key( std::uint32_t k , const state_t< D > & s ,
                                  const state_t< D > & t ) const
  requires IncrementalFeatureDomain< D >;
 [[nodiscard]] static std::string name( void );
 };
```

(Distribution.h:134-187, declarations quoted without their bodies). We remark that the search on a shared memory requires no more of a distribution; in particular, the owner need not be deterministic there (the random distribution of §9.4 is not), and each worker keeps a copy of the distribution, so that a distribution with a mutable state (the generator of the random one) needs no lock. `HDAStarMPI`, on the contrary, needs a deterministic owner, since it rebuilds the path by asking the owner of each state (§9.6). A distribution that can follow the moves models the concept `IncrementalDistribution` (Distribution.h:631): it also gives a 32-bit *key* `key( s )` (for Zobrist hashing, the high 32 bits of the hash of §9.4), whose owner among p workers is `key_owner( key( s ) , p )`, and the key `key( k , s , t )` of a successor t of s computed from the key k of s, which must

equal `key( t )`. HDA* keeps the key of each node in the node, and the function `successor_owners()` (`Distribution.h:660`) returns, for the node being expanded, a function that maps each successor to its owner and its key, computed from the key of the parent when the distribution is incremental and from scratch otherwise (the key is then 0, as `distribution_key()` returns for a distribution that is not incremental).

What the distributions need from the Domain

Each distribution requires of the `Domain` the least that it uses, through a refinement of the concept `Domain` of Chapter 1 (all in `Domain.h`).

- `FeatureDomain` (`Domain.h:193`): a state is a set of propositions, the *features*, taken from a universe of `num_features()` of them, and `for_each_feature( s , f )` calls `f( i )` for each feature i that holds in s. Zobrist hashing needs it.
- `IncrementalFeatureDomain` (`Domain.h:209`): moreover, `for_each_feature_change( s , t , f )` calls `f( i )` for each feature that holds in exactly one of s and of its successor t, i.e., for the features of the symmetric difference of the two sets.
- `AbstractFeatureDomain` (`Domain.h:221`): moreover, a projection `abstract_feature( i )` of the features onto `num_abstract_features()` abstract features, which is the projection A of Abstract Zobrist hashing.
- `AbstractStateDomain` (`Domain.h:267`): a map `abstract_state( s )` of the states to the integers, i.e., a state abstraction.
- `PerfectHashDomain` (`Domain.h:408`): an injective map `perfect_hash( s )` of the states to the integers.

On the sliding-tile puzzles (`SlidingTile.h:395-480`) a feature is a pair (tile, cell), i.e., the index $t \cdot n + c$ for the tile $t \neq 0$ in the cell c of a board of n cells. An abstract feature is a pair (tile, half of the board), the half being the top rows or the bottom rows, and the abstract state of a state is the triple of the cells of the tiles 1, 2 and 3, which is the abstraction that Jinnai and Fukunaga chose among the candidates of footnote 4 of section 3.1.4 of [5]. Furthermore, the grids and the vacuum world give features and abstractions as well (cf. Chapter 14), and any `Domain` gives the hash of its closed list, which is all that the default distribution needs (one can therefore run HDA* on any `Domain`). Two more refinements, `FeatureGroupDomain` and `FeatureTransitionDomain`, give the groups of the features and the transitions between them, which the automatic distributions read (Definition 10.2), and `LevelDomain` gives the levels of the hyperplane distribution (Chapter 15).

9.2 Hash Distributed A*

We now describe the search. Each worker i runs A* on its lists OPEN_i and CLOSED_i, which hold only the states v with $\text{owner}(v) = i$. A successor generated by a worker is sent to its owner, which checks it against its CLOSED list and possibly inserts it in its OPEN list, as the sequential A* does with a state that it generates. The *incumbent* z, i.e., the cost of the best goal found so far, is shared by the workers, and each node with $f \geq z$ is pruned. We write a message as (v, g', u): it carries a state v with the cost g' of the path that reached it and its predecessor u on that path (and, in the implementation, the operator that generated v, its heuristic and its key).

Algorithm 9.1 (HDA*).

```
 1  procedure HDA*(s₀, p, owner)
 2     z ← ∞;  INSERT(s₀, 0, nil) on owner(s₀)
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        while the search is not over do
 5           for each message (v, g', u) received do INSERT(v, g', u)
 6           if OPEN_i has a node of f < z then
 7              u ← the first node of OPEN_i, removed (stale entries
                   dropped);  close u
 8              if u is a goal then
 9                 z ← min(z, g(u)), recording u if g(u) < z
10              else for each successor v of u do
11                 g' ← g(u) + c(u, v)
12                 if g' + h(v) ≥ z then continue
13                 if owner(v) = i then INSERT(v, g', u)
14                 else send (v, g', u) to owner(v)
15              every k expansions, send the outboxes not yet full
16           else
17              send all the outboxes;  wait for mail or for the end
```

```
18     return the path to the goal of cost z, if z < ∞
19  procedure INSERT(v, g', u)                          ▷ on the owner i of v
20     if g' + h(v) ≥ z then return
21     if v ∉ CLOSED_i then
22        add v to CLOSED_i, with g(v) ← g' and parent(v) ← u
23     else if g' < g(v) then
24        g(v) ← g';  parent(v) ← u;  v is not closed
25     else return                                      ▷ a duplicate
26     push (g' + h(v), g', v) in OPEN_i
```

This is the pseudocode of `HDAStar.h:18-46`. We comment on the lines that are not those of A*.

- Line 2: the root is inserted by its owner before the first expansion, and the other workers start with empty lists and wait for mail; the first expansions are therefore those of a single worker, which is the initial burst of the search (section 3.1.2 of [5]).
- Line 5: the messages are processed before each expansion; hence, a cheaper path that has arrived is used at once.
- Line 7: CLOSED holds all the nodes that a worker has stored, closed or not, and OPEN is a priority queue of entries with lazy deletion: when the g of a node decreases (line 24) a new entry is pushed, and the old one becomes *stale* and is dropped when it reaches the top, as in the sequential search (§3.1).
- Lines 8 and 9: the goal test is made at the expansion, not at the generation, and therefore a goal is recorded only when it is the best node of its owner; the incumbent is an atomic variable, and the goal is recorded under a mutex.
- Lines 12 and 20: a node with $f \geq z$ is pruned twice, by the worker that generates it (which saves the message) and by its owner (since z may have decreased while the node was travelling).
- Lines 13 and 14: a successor owned by the expanding worker does not go through the mailbox, while one owned by another worker is appended to the outbox of that worker, which leaves as one message when it holds `batch` nodes (cf. §9.3).
- Lines 15 and 17: an outbox that is not full leaves after each block of `flush_period` expansions, and all the outboxes leave before the worker waits; the wait ends when mail arrives or when the mailbox detects the end of the search, i.e., when no worker has a node of $f < z$ and no node is travelling (§9.3).
- Line 18: the parents cross the workers (a node records a pointer to the node of its predecessor, which may belong to another worker), and they are followed only after all the workers have returned.

As the sequential search, HDA* skips the move back to the parent in an `OperatorDomain`: the operator that generated a state travels in its message, and the successor that the inverse operator leads to is never generated, since it is never cheaper than the parent (cf. §2.9); on the sliding-tile puzzles this typically saves about one successor out of three and, in HDA*, its message as well.

Figure 9.1 shows an expansion. The code runs the loop of a worker as follows, with the operations SEND, FLUSH, POLL, QUIESCENT and STOP of the mailbox of §9.3 (Algorithm 9.3): the inbox is drained as a whole, and the hashes of all its nodes are computed and their places in the closed list prefetched before the first lookup, so that the cache misses of the lookups overlap instead of following one another (§3.8), and the owners of the successors are computed from the key of the parent (§9.5).

Algorithm 9.2 (The loop of a worker of HDA*, as coded).

```
 1  procedure WORKER(i)
 2     if i = owner(s₀) then INSERT(s₀, 0, nil)
 3     k ← 0
 4     while the mailbox is not stopped do
 5        M ← POLL(i)                                  ▷ the whole inbox
 6        for each (v, g', u) ∈ M do                   ▷ phase 1
 7           compute the hash of v;  prefetch its place in CLOSED_i
 8        for each (v, g', u) ∈ M do INSERT(v, g', u)        ▷ phase 2
 9        u ← NEXT()
10        if u ≠ nil then
11           EXPAND(u);  k ← k + 1
12           if k = flush_period then k ← 0;  FLUSH(i)
13           if i stores node_limit nodes then the limit is hit;  STOP()
14        else if QUIESCENT(i) then return
15        else wait a little
```

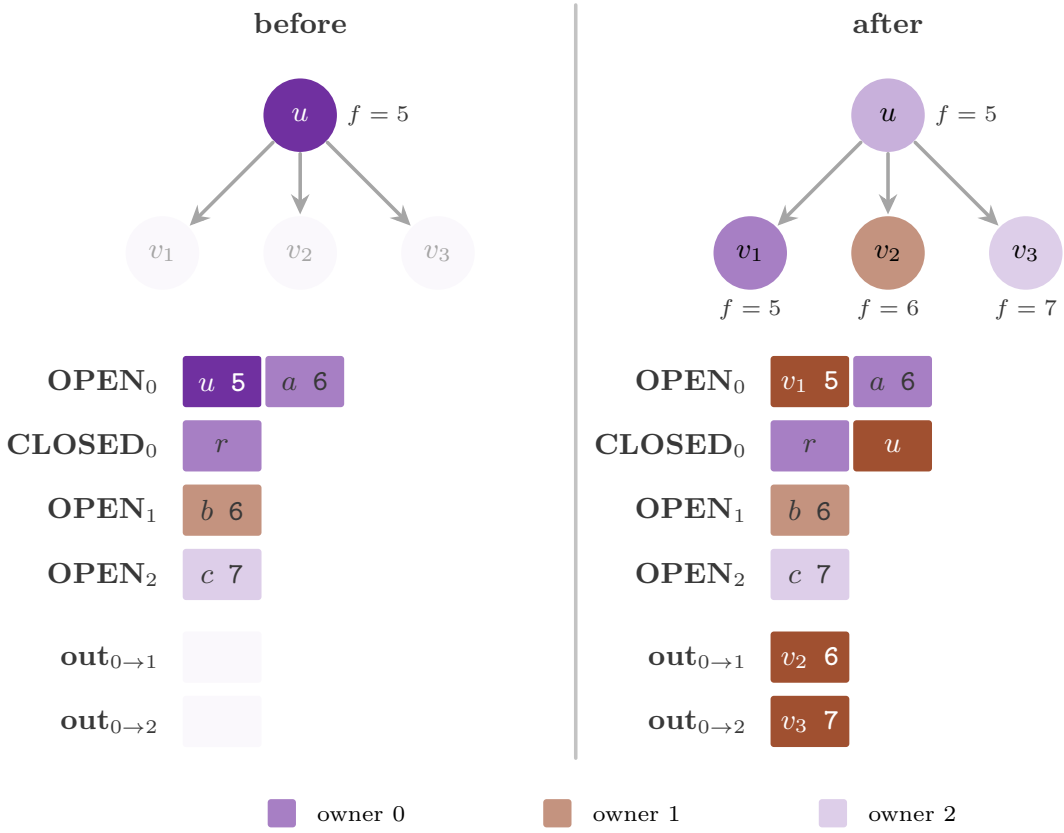


Figure 9.1. One expansion of HDA* with three workers. Before (left), worker 0 takes u, the first node of its OPEN list; after (right), u is closed, its successor v_1, which worker 0 owns, is in its OPEN list, and v_2 and v_3 wait in the outboxes of the workers 1 and 2 (the colour of a state is its owner, and the entries added by the step are in the darker rust colour).

```
16  function NEXT()
17     while OPEN_i is not empty do
18        (f, γ, u) ← the first entry of OPEN_i
19        if it is stale then remove it;  continue
20        if f ≥ z then return nil
21        remove it;  return u
22     return nil
23  procedure EXPAND(u)
24     close u
25     if u is a goal then
26        lock the goal record
27        if g(u) < z then z ← g(u);  record u
28        unlock the goal record;  return
29     O ← the map of the successors to their owners, from the key of u
30     for each successor v of u not reached by the inverse operator do
31        g' ← g(u) + c(u, v)
32        if g' + h(v) ≥ z then continue
33        j ← the owner given by O(v)
34        if j = i then INSERT(v, g', u)
35        else SEND(i, j, (v, g', u))
```

Figure 9.2 shows what the owner of a state does with the copies of it that the other workers send. All the copies of v reach the same worker, which keeps in its CLOSED list the cheapest of those that it has processed. Hence, the duplicate detection is that of the sequential search. The difference is that the copies arrive in an order that depends on the timing of the workers, and a copy that arrives after its state has been expanded with a larger g reopens it. This is why HDA* may reopen a state even when the heuristic is consistent (footnote 6 of section 3 of [2]; one can build a small example, cf. Exercise 9.5), and the reopenings are part of its search overhead.

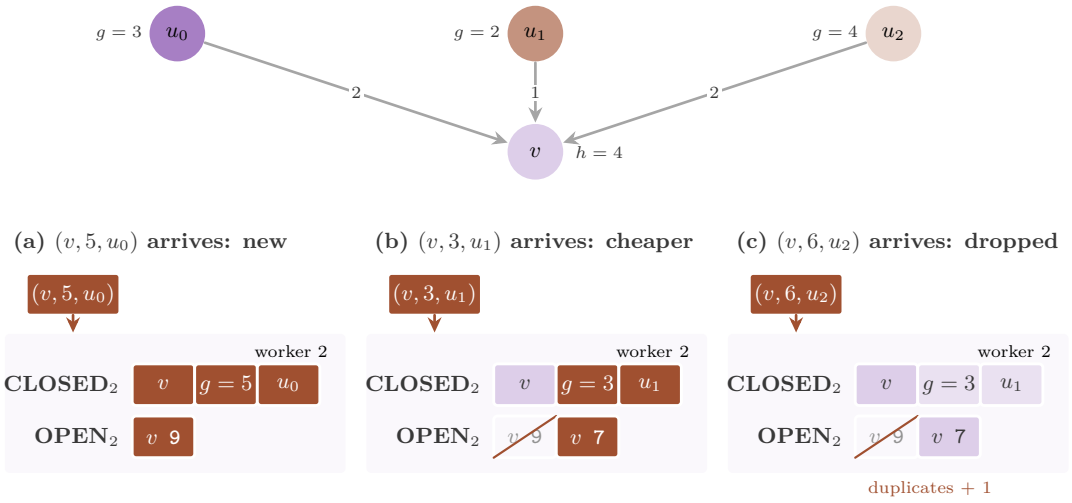


Figure 9.2. The copies of a state v, owned by worker 2, sent by three other workers. (a) The first copy creates the node of v, with $g = 5$ and parent u_0, and pushes it with $f = 9$; (b) a cheaper copy lowers g and changes the parent, and the old entry of OPEN becomes stale; (c) a copy that is not cheaper is dropped as a duplicate.

Invariants

We now state the invariants of Algorithm 9.1, from which its optimality follows. We assume that the costs are nonnegative and the heuristic is admissible, as in Chapter 2, and C^* is the cost of an optimal path from s_0 to a goal. The first invariant is the reason why the workers need no lock on the nodes.

Lemma 9.1 (Ownership). At any time, each node is created, stored and written only by the worker that created it, which is the owner of its state for the copy of the distribution of that worker; if the distribution is deterministic, at most one node of v exists, at owner(v), and each copy of v that is generated is compared with it.

Proof. A node is created only by INSERT (lines 21-22), which a worker calls only on a state that it owns: either

on the root, by its owner (line 2), or on a message, which was sent to the owner of its state (line 14), or on a successor that it owns (line 13). A node is written by INSERT (lines 22-24) and by the expansion (line 7), and a worker expands only the nodes of its OPEN list, which INSERT fills. The other workers only hold the address of a node, as the parent in a message, and they do not dereference it before the end (line 18). If the owner is a function of the state, all the copies of v reach the same worker, whose CLOSED list holds at most one node of v (line 21). □

Without a deterministic owner (the random distribution of §9.4) a state may have up to p nodes, one per worker that has received a copy of it; the duplicates are then detected only among the copies that happen to reach the same worker, a weakness that the hashing of the state was introduced to remove (cf. section 6 of [2]). The proofs below hold node by node, and therefore for any distribution.

Lemma 9.2 (Values and parents). At any time, (i) z is nonincreasing; (ii) for each stored node v with a parent u (i.e., each node but the one that the owner of s_0 creates at line 2), one has $g(v) \geq g(u) + c(u, v)$, and $g(v)$ only decreases; (iii) the parent pointers form a tree rooted at that node of s_0; and (iv) if $z < \infty$, the chain of the parents of the recorded goal t is a path from s_0 to t of cost at most $g(t) = z$.

Proof. (i) z changes only at line 9, to a smaller value, under the mutex. (ii) When INSERT sets $g(v) \leftarrow g'$ and parent$(v) \leftarrow u$, one has $g' = g(u) + c(u, v)$ with the value of $g(u)$ at the expansion of u (line 11). Afterwards $g(u)$ may only decrease (line 24 is reached only with $g' < g(v)$), which keeps the inequality, and the parent of v changes only together with a smaller $g(v)$, both written by the owner of v in the same step. (iii) The values of g are defined at each instant, whichever worker writes them, and (ii) holds at each instant. Suppose that setting parent$(v) \leftarrow u$ at an instant τ closes a cycle, i.e., that the chain of the parents of u at τ leads to v. Summing (ii) along that chain, at τ, gives $g(u) \geq g(v)$ for the value of $g(v)$ before the update, since the costs are nonnegative; since $g(u)$ at the expansion of u was not smaller than at τ, $g' \geq g(v)$, and line 24 is not reached, a contradiction. (iv) The goal recorded at line 9 has $g(t) = z$ at that moment; summing (ii) along the chain of its parents, which ends at s_0 by (iii), with $g(s_0) = 0$, gives a path of cost at most $g(t)$. □

Our next invariant is the parallel form of the one that makes A* optimal, i.e., that some node of an optimal path is always in OPEN with its optimal g (Lemma 2.7); here the node may also be in the hands of a worker or in a message, as Figure 9.3 shows.

Lemma 9.3 (A node of an optimal path is always at hand). Let $P = (s_0 = u_0, u_1, \ldots, u_k)$ be an optimal path to a goal. While $z > C^*$, some u_j of P with $g(u_j) = g^*(u_j)$ is (a) in the OPEN list of a worker that stores it, in an entry that is not stale, or (b) being expanded, or (c) in a message (in an outbox, in transit, or drained and not yet inserted) that carries $g' = g^*(u_j)$.

Proof. We take P simple (a cycle of an optimal path has cost 0 and can be removed). Note first that each u_j has $f(u_j) = g^*(u_j) + h(u_j) \leq C^* < z$, by admissibility; hence, no copy of it with its optimal g is ever pruned at line 12 or at line 20. At the start, s_0 is in case (a), or about to be inserted by its owner (line 2). Afterwards, let j be the largest index such that some node of u_j has been closed with $g(u_j) = g^*(u_j)$ (if there is none, s_0 has not been expanded yet and is in case (a) or (b)). If the expansion of u_j with that g is still in progress, u_j is in case (b). Otherwise, since u_j is not the goal (otherwise line 9 would have set $z \leq C^*$), its expansion generated u_{j+1} with $g' = g^*(u_j) + c(u_j, u_{j+1}) = g^*(u_{j+1})$ (line 11), and this copy was either given to INSERT at line 13 or sent at line 14. (In an `OperatorDomain` the expansion skips the successor that the inverse of the operator of u_j leads to, i.e., the parent w of u_j. If $w = u_{j+1}$, then, since the inverse move has the same cost, $g^*(u_j) = g(u_j) \geq g(w) + c(w, u_j) \geq g^*(u_{j+1}) + c(u_j, u_{j+1}) = g^*(u_j) + 2\,c(u_j, u_{j+1})$ by Lemma 9.2(ii), i.e., $c(u_j, u_{j+1}) = 0$ and w was expanded with $g(w) = g^*(u_{j+1})$ when it generated u_j, which contradicts the choice of j; hence u_{j+1} is generated.) In the first case, INSERT either stored u_{j+1} with $g^*(u_{j+1})$ and pushed it, or found $g(u_{j+1}) \leq g'$, i.e., $g(u_{j+1}) = g^*(u_{j+1})$ already; in both cases that node of u_{j+1} is open with its optimal g, or being expanded, or closed with its optimal g, and the last contradicts the choice of j. In the second case the copy is in a message, case (c), until the worker that receives it inserts it, after which the argument of the first case applies. □

The end of the search is detected by the mailbox (§9.3). For the moment we take its correctness as a hypothesis, i.e., we assume that the workers stop only when no worker has a node of $f < z$ in its OPEN list, no worker is expanding a node and no message exists (the condition of Theorem 8.6, with the entries in flight of Definition 8.5), and that they do stop once these hold; Proposition 9.8 proves it on a shared memory, for workers that are each scheduled infinitely often, and Theorem 9.18 over MPI.

Theorem 9.4 (Optimality of HDA*). Let the graph be finite, the costs nonnegative and the heuristic admissible, let the node limit be unset and let the detection of the end be correct. Then HDA* terminates, with any distribution; if a goal is reachable, it returns a path of cost C^*, and otherwise it reports that there is none.

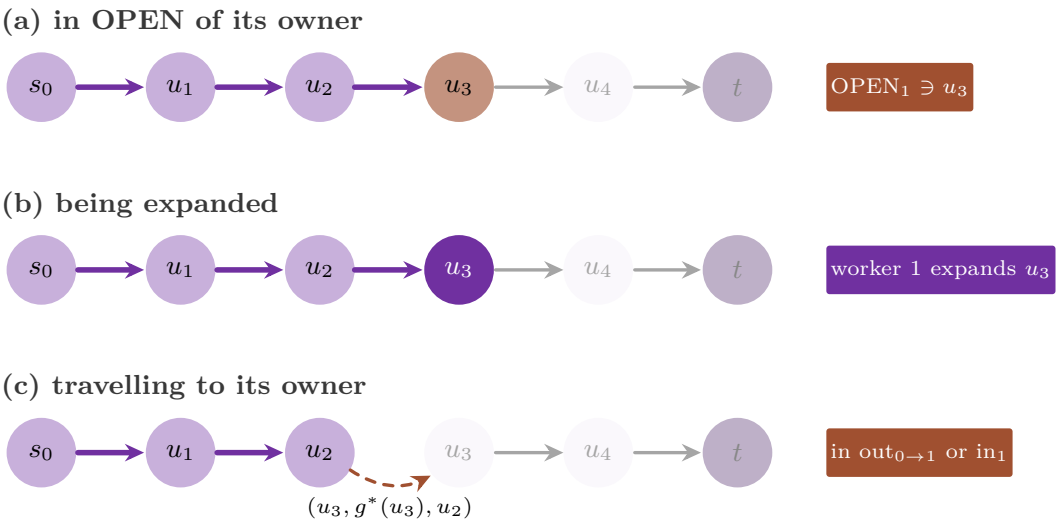


Figure 9.3. The three places of the node u_3 of an optimal path in Lemma 9.3, when its predecessors are closed with their optimal g (darker tint) and $z > C^*$: (a) in the OPEN list of its owner, (b) being expanded, (c) in a message of the worker that expanded u_2 to the owner of u_3.

Proof. Termination. Each insertion of a state with a given g is followed by at most one expansion of it with that g, and the g of a node only decreases (Lemma 9.2(ii)); a state has at most p nodes. Note that each value of g is the cost of a walk from s_0 that is not larger than the first value assigned. On a finite graph with nonnegative costs these costs form a finite set, since a walk of cost at most B has at most B/ε edges of positive cost, ε being the least positive cost, and its cost is a sum of at most that many positive costs taken from a finite set. Hence the expansions are finitely many, and so are the messages, each of which is generated by an expansion; after the last of them is processed no worker has work and no message is in transit, and the detection of the end stops the workers.

Optimality. At the end the conditions above hold; if a goal is reachable and $z > C^*$, Lemma 9.3 gives a node with $f \le C^* < z$ in an OPEN list, being expanded or in a message, which contradicts them. Hence $z \le C^*$, and $z \ge C^*$ since z is the cost of a path (Lemma 9.2(iv)). Note that the recorded goal t still has $g(t) = C^*$ at the end, since a later decrease of $g(t)$ would have pushed t with $f < z$ and recorded it again at its expansion. The chain of its parents is a path of cost at most $g(t)$ (Lemma 9.2(iv)), i.e., an optimal one. If no goal is reachable, z stays ∞, and the search reports the instance unsolvable. □

We now bound the scope of the theorem by two remarks. First, on an infinite (locally finite) graph with costs bounded below by some $\varepsilon > 0$ the argument of termination needs a reachable goal and also that each worker makes progress. Indeed, a worker that is never scheduled (in practice, one scheduled rarely, e.g., with more workers than cores, delays the incumbent in the same way) may hold the only node of an optimal path while the others expand nodes above the optimum that no incumbent prunes yet (on a finite graph the same situation costs time and memory rather than termination, and the tester runs the 15-puzzle with at most one worker per core for this reason, cf. `test_hda` in `test/test.cpp`). Second, the theorem does not need a consistent heuristic, since the reopenings are allowed; with a consistent one the sequential A* never reopens (Theorem 2.13), while HDA* still may, and its search overhead counts the reopenings. When the option `node_limit` of the sequential search (Chapter 2) is set, a worker that stores at least that many nodes after one of its expansions stops the mailbox (the test is made after each expansion only, so that the insertions since the previous test may overshoot the limit), all the workers return, and the status is `node_limit`, with no claim on the cost.

We end the section with the cost of the steps of the search.

Proposition 9.5 (The cost of a step and the memory of HDA*). An expansion of u that generates the successors S makes, for each successor not pruned, either one lookup in the closed list of the worker (and at most one insertion in its OPEN list) or one append to an outbox; it takes no lock but the mutex of the goal, when u is a goal, besides the operations of the channels of the executor, and it computes the owners in the time of Corollary 9.16 when the distribution is incremental. Each batch costs one atomic addition on the shared counter of the

mailbox at the sender (two if the queue refuses part of it), and at most one at the receiver, which subtracts all the messages of a drain at once (plus one addition when the drain ends a passive period). With a deterministic distribution the nodes stored by all the workers are at most one per state reached; with any distribution they are at most p per state.

Proof. The counts are read on Algorithms 9.2 and 9.3: INSERT makes one lookup and at most one push, SEND appends and pushes a full outbox as one batch (one addition, and one subtraction for the part that the channel refuses, cf. line 7 of Algorithm 9.3), and POLL subtracts the messages drained in one addition (with one more addition when the worker leaves the passive state). The bounds on the nodes are those of Lemma 9.1 and of the remark after it. □

9.3 The messages and the end of the search

Batches and flushes

Sending a node usually costs a push on a queue that another worker reads (on a shared memory) or a message of MPI (e.g., over a network); in both cases a rather large fixed part of the cost does not depend on the size of what is sent, and the nodes for the same worker are therefore (rather naturally) packed into one message, an idea that HDA* takes from the transposition-table-driven scheduling of Romein et al. (cf. section 3 of [2]). In the library, each worker has one outbox per other worker; a node is appended to the outbox of its owner, an outbox that holds `batch` nodes leaves at once as one message, and the outboxes that are not full leave after each block of `flush_period` expansions and before the worker waits (Figure 9.4). We remark that the flush also matters for the search overhead: a node of an optimal path that waits in an outbox for others to fill it is a node that its owner cannot expand, while the owner expands worse nodes and sends their successors around. Kishimoto et al. observed that packing "too many" nodes in a message raises the expansions, while packing "few" raises the cost of the communication (section 5.2 of [2]); hence, `batch` and `flush_period` trade one overhead for the other.

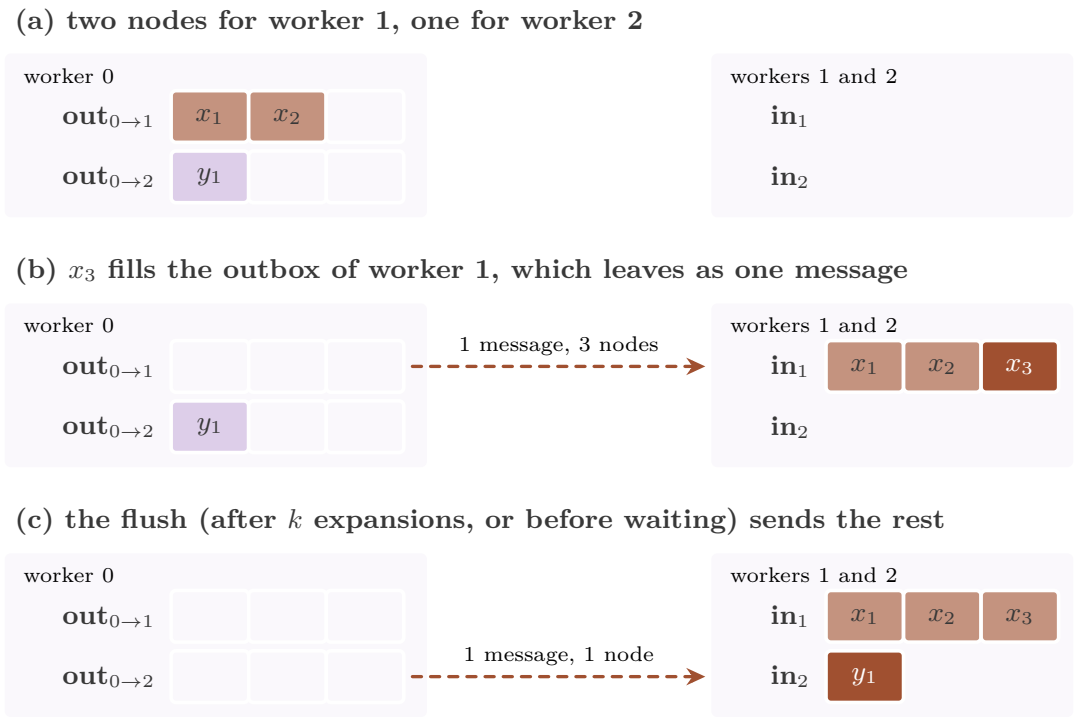


Figure 9.4. The outboxes of worker 0 with batches of 3 nodes. (a) Two nodes wait for worker 1 and one for worker 2; (b) the third node for worker 1 fills its outbox, which leaves as one message; (c) the flush, after k expansions or before the worker waits, sends the outbox of worker 2, which is not full.

The channels between the workers are, on every executor, a mesh of bounded queues with one producer and one consumer, one for each ordered pair of workers, and the queue of a pair may be full. The part of a batch that does not fit then stays in the outbox, and it is tried again when a further batch for that worker is ready or at the next flush. A worker whose flush has left something behind keeps polling its inbox (which empties the queues that the others are waiting on); thus, no worker waits for another.

The mailbox and its counter

A worker that finds no node of $f < z$ in its OPEN list cannot stop, since a node may be travelling to it, or another worker may still generate one. In fact, the mailbox detects the end by counting, with the deficit counting of Dijkstra and Scholten [35] on a single counter C in the shared memory instead of an acknowledgment of each message. A worker is *active* from the start until a call of QUIESCENT finds its outboxes empty, and it is *passive* from then until it drains a message. Each worker i has an outbox O_j for each worker j, with a threshold N_j, at first `batch`.

Algorithm 9.3 (The mailbox of the workers, on a shared memory).

```
 1  procedure SEND(i, j, x)                    ▷ i is active
 2     append x to O_j;  if |O_j| ≥ N_j then PUSH(i, j)
 3  procedure PUSH(i, j)                       ▷ nothing once stopped
 4     k ← |O_j|;  C ← C + k                     ▷ before they can be drained
 5     r ← the number of the first messages of O_j that the queue takes
 6     remove them from O_j;  N_j ← |O_j| + batch
 7     C ← C − (k − r)
 8  function FLUSH(i)
 9     for each j with O_j ≠ ∅ do PUSH(i, j)
10     return all the outboxes of i are empty
11  function POLL(i)                           ▷ nothing once stopped
12     M ← ∅
13     for each message x taken from the queues to i do
14        if i is passive then C ← C + 1;  i is active
15        M ← M ∪ {x}
16     C ← C − |M|;  return M
17  function QUIESCENT(i)                      ▷ i has no node of f < z
18     if the mailbox is stopped then return true
19     if i is active then
20        if not FLUSH(i) then return false
21        i is passive;  C ← C − 1
22     return C = 0
23  procedure STOP()                           ▷ from any worker
24     the mailbox is stopped
```

Definition 9.6 (The deficit counter). The counter C of Algorithm 9.3 starts at p (all the workers are active). A worker adds k to C before it pushes k messages and subtracts those that the queue refuses (lines 4 and 7); a passive worker that drains messages adds 1 (it becomes active) before it takes the first of them (line 14), and it subtracts the messages it has drained (line 16). A worker that becomes passive subtracts 1 (line 21), and a passive worker only reads C.

Lemma 9.7 (The counter of the mailbox). Let each worker send only while active and become active only by draining a message. Then (i) at any time C equals the number of the active workers plus the number of the messages that have been counted by their sender and are still to be subtracted (by their receiver, or by their sender if the queue refuses them), i.e., those in the queues and those being pushed or drained; (ii) the messages in an outbox belong to an active worker; (iii) $C = 0$ holds only when no worker is active and no message exists, and then it holds forever; (iv) if at some time no worker is active and no message exists, then $C = 0$ from then on, and every worker that calls QUIESCENT returns.

Proof. (i) By induction on the operations of Algorithm 9.3: each changes C by the change of the quantity on the right, since a message is counted before it is pushed (hence before it can be drained), uncounted if the queue refuses it, and subtracted after it is drained, and the activity of a worker is added when it leaves the passive state and subtracted when it enters it. (ii) A worker becomes passive only at line 21, after FLUSH has found its outboxes empty, and a passive worker does not send. (iii) Both terms of (i) are nonnegative, so $C = 0$ forces both to be 0, and by (ii) the outboxes are empty too. From such a state no operation raises C: an addition for a push needs an active sender, and an addition for an activity needs a message to drain. (iv) If no worker is active and no message exists, (i) gives $C = 0$, and by (iii) it stays 0; each passive worker reads it at its next call of QUIESCENT, at line 22. □

The drained messages of line 16 are subtracted before the worker inserts them (lines 5 to 8 of Algorithm 9.2); they are then held by an active worker, and the worker becomes passive only after their insertions and after its next selection has failed. These hypotheses hold for HDA*, which we now prove together with the progress of

the workers on bounded queues.

Proposition 9.8 (The end of HDA* on a shared memory). In Algorithm 9.2, without the node limit, (i) a worker returns only when no worker has a node of $f < z$, none is expanding and no message exists; (ii) if each worker is scheduled infinitely often, every message is eventually pushed and drained, and once the condition of (i) holds every worker returns. With the node limit, all the workers return after the first call of STOP.

Proof. A worker sends only at line 35 of Algorithm 9.2 and in FLUSH, both while it has work, hence while active. A passive worker has no node of $f < z$, since NEXT found none before QUIESCENT, and since z never increases (Lemma 9.2(i)) no node of its OPEN list becomes selectable again: only a message can give it work, and draining one makes it active. Thus Lemma 9.7 applies. (i) A worker returns at line 14 when QUIESCENT reads $C = 0$; by Lemma 9.7(iii) no worker is active, hence none has a node of $f < z$ or is expanding, and no message exists. (ii) A message left in an outbox by a full queue is tried again at the next FLUSH of its sender, which happens every `flush_period` expansions and at each call of QUIESCENT while the sender is active; the queue of the pair empties when its receiver polls, which it does at each iteration of its loop. Hence no message stays forever in an outbox or in a queue. Once the condition of (i) holds, Lemma 9.7(iv) makes every worker return. After STOP the loop of line 4 ends at its next test, and QUIESCENT returns true. □

Hence the end that Theorem 9.4 assumes is detected correctly: the workers stop when, and only when, no worker has a node of $f < z$, none is expanding and no node is in transit; Figure 9.5 follows C on a run of two workers.

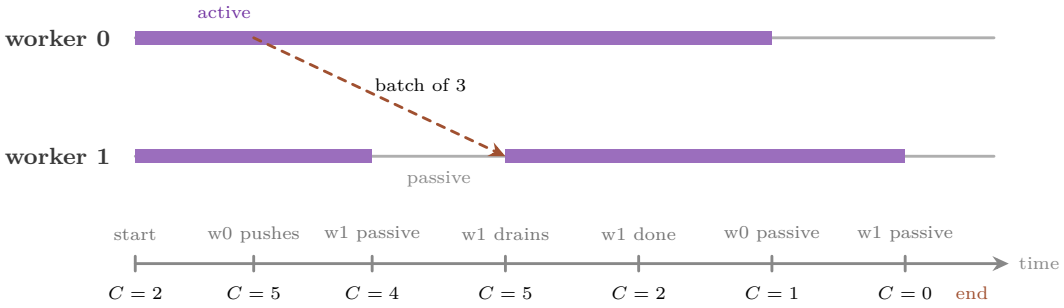


Figure 9.5. The counter C on a run of two workers (the thick segments are the active periods). Worker 0 pushes a batch of 3 nodes (C from 2 to 5); worker 1 becomes passive (4), becomes active again when it drains the batch (5) and subtracts the 3 nodes once they are drained (2), before it inserts them; the two workers become passive in turn, and the end is the moment at which C reaches 0.

Counting cannot be dropped. A scheme sometimes used in "simple" implementations is made of two rounds of idle flags without any count of the messages: a worker sets its first flag when it has no work, sets its second flag when it sees all the first flags set, and stops when it sees all the second flags set. This scheme is not sound, since a flag carries no information on a message that was pushed after its owner looked at its inbox.

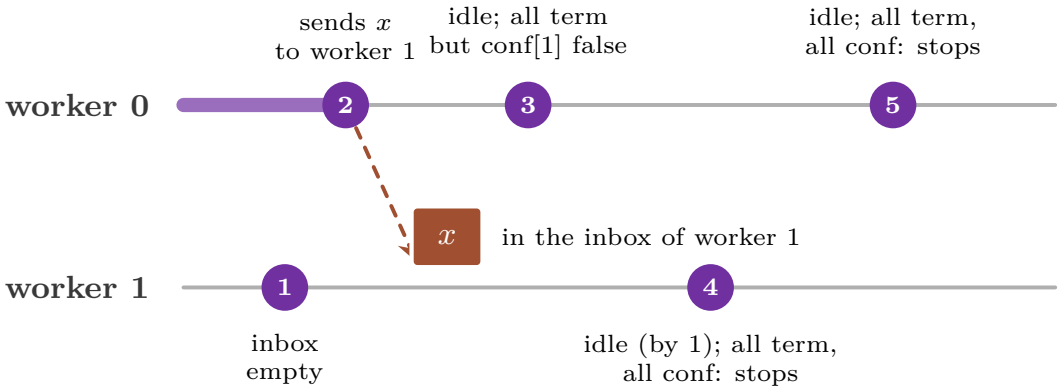
Proposition 9.9 (Idle flags are not enough). With two rounds of idle flags and no count of the messages, two workers may both stop while a node is in the inbox of one of them.

Proof. Figure 9.6 gives such a run. Worker 1 has set its first flag in an earlier round, while it had no work, and it looks at its inbox and finds it empty (event 1). Worker 0 then expands its last node and sends its successor x to worker 1 (event 2); having no work left, it sets its first flag, sees all the first flags set (that of worker 1 is still set), sets its second flag, and goes on, since the second flag of worker 1 is not set (event 3). Worker 1, acting on what it saw at event 1, has no work: it sees all the first flags and then all the second flags set, and stops (event 4); worker 0, at its next round, sees the same and stops (event 5). Thus, the node x is never processed; if it is the only node of an optimal path at hand in the sense of Lemma 9.3, the search returns a suboptimal cost, or reports an unsolvable instance. □

Nor does it help to let each worker look at its inbox again before it stops: a worker that finds a message there becomes active and may send a node to a worker that has already stopped, which only counting the messages in flight detects.

9.4 The work distributions

Much of the behavior of HDA* is decided by its distribution, which balances two overheads against each other: a distribution that keeps the successors of a node on the worker of the node lowers the communication, but it



	start	1	2	3	4	5
term[0]	F	F	F	T	T	T
term[1]	T	T	T	T	T	T
conf[0]	F	F	F	T	T	T
conf[1]	F	F	F	F	T	T

Figure 9.6. A run in which two rounds of idle flags stop both workers while the node x is still in the inbox of worker 1; the table gives the flags after each event, those that change in the rust colour. Counting the messages, as in Lemma 9.7 or in Theorem 9.18, excludes such a run.

tends to gather the nodes of least f on a few workers, while one that spreads the states uniformly balances the load at the price of sending almost all the nodes. The *band effect* of Jinnai and Fukunaga (section 3.1.1 of [5]) explains the (somewhat indirect) link between the load balance and the search overhead: if, with two workers, a worker owns a fraction q of the nodes of each value of f, its n-th expansion is roughly the (n/q)-th of A*, and the farther q is from $1/2$ the more the workers expand out of the order of A*, i.e., nodes that A* would not reach (an estimate of [5], not a theorem). We now describe the six distributions of `Distribution.h` that take their structure from the `Domain`, and then the Zobrist hashing on a table given by the caller, on which the hand-made projections and the automatic distributions of Chapter 10 are built, in the notation HDA*[*hash*, *abstraction*] of section 2.6 of [5] and by their names in the library; Figure 9.7 colours the 12 states of the 2×2 puzzle by their owner under three of them.

From a hash to a worker

A distribution that computes a 64-bit hash h maps it to a worker by the multiply-high reduction of Lemire [151],

$$\operatorname{reduce}(h, p) = \left\lfloor \frac{h\,p}{2^{64}} \right\rfloor ,$$

which uses the high bits of h. The code computes it by halves in 64-bit arithmetic, with the same result on every compiler (`Distribution.h:103-104`): with $h = a\,2^{32} + b$ and $b < 2^{32}$, $\lfloor hp/2^{64} \rfloor = \lfloor (ap + \lfloor bp/2^{32} \rfloor)/2^{32} \rfloor$, where $ap \le 2^{64} - 2^{33} + 1$ and $\lfloor bp/2^{32} \rfloor < 2^{32}$ for $p < 2^{32}$, so that no sum overflows. A 32-bit key k is mapped by `key_owner`, i.e., to $\operatorname{reduce}(2^{32}k, p) = \lfloor kp/2^{32} \rfloor$, the key standing for the high 32 bits of the hash.

Proposition 9.10 (The reduction is balanced). For $0 \le i < p \le 2^{64}$, the number of the values $h \in \{0, \dots, 2^{64}-1\}$ with $\operatorname{reduce}(h, p) = i$ is either $\lfloor 2^{64}/p \rfloor$ or $\lceil 2^{64}/p \rceil$; the same holds for `key_owner` with 2^{32} in place of 2^{64}.

Proof. $\operatorname{reduce}(h, p) = i$ if and only if $i \le hp/2^{64} < i + 1$, i.e., $h \in [\, i\,2^{64}/p, (i+1)\,2^{64}/p)$, a half-open interval of length $2^{64}/p$, which contains either $\lfloor 2^{64}/p \rfloor$ or $\lceil 2^{64}/p \rceil$ integers. □

Hence, for a uniform hash, the share of the states of each worker differs from $1/p$ by less than 2^{-64} (or 2^{-32} for a key), i.e., the reduction adds (almost) no imbalance; the modulo $h \bmod p$, which the library uses only where the comparison of [5] requires it (below), has the same property but costs a division.

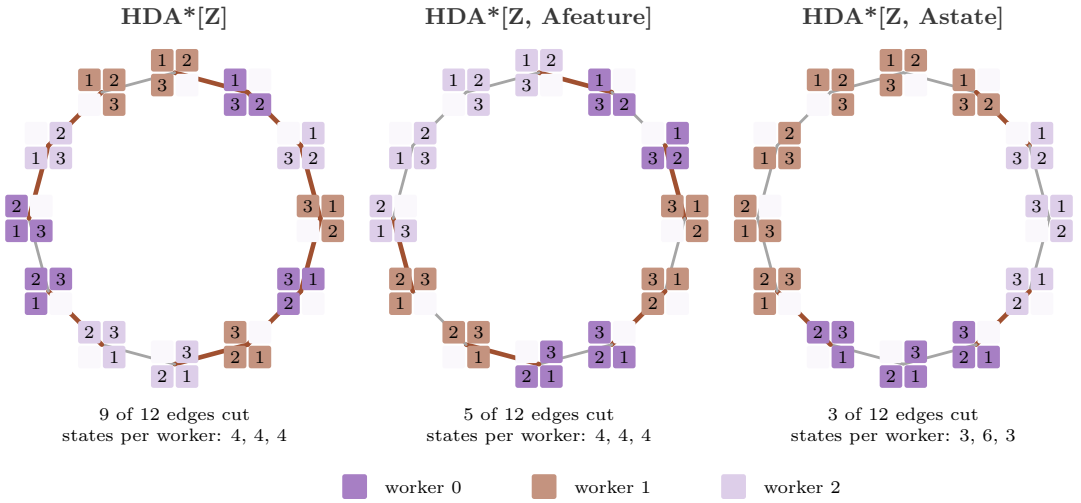


Figure 9.7. The 12 states of the 2×2 puzzle that are reachable from the goal, which form a cycle, coloured by their owner among $p = 3$ workers under Zobrist hashing (left), Abstract Zobrist hashing with the projection of a tile onto its row (middle) and the abstraction by the cell of tile 1 (right); the edges between two workers are thick and in the rust colour. The random tables were drawn for the figure, and on so small a space the counts change from one draw to another.

Zobrist hashing, HDA*[Z]

With the state s described by the set $F(s)$ of the features that hold in it and a table R of random 64-bit strings, one per feature, the Zobrist hash [152] is

$$Z(s) = \bigoplus_{x \in F(s)} R[x],$$

with $\oplus$ the exclusive or of the bits (equation (1) of [5]); `ZobristDistribution` (`Distribution.h:134`, for a `FeatureDomain`) draws R from the seed, takes as the key the high 32 bits of $Z(s)$ and as the owner that of the key. This is the original distribution of HDA* (section 3 of [2]). The following procedures, in which the table T is R for Zobrist hashing and the table R' of Abstract Zobrist hashing laid out by raw feature (see below), are those of both distributions; `xor` is the exclusive or of the bits.

Algorithm 9.4 (The Zobrist hash, its key and its owner).

```
 1  function HASH(s)
 2     y ← 0
 3     for each feature x of s do y ← y xor T[x]
 4     return y
 5  function KEY(s)
 6     return the high 32 bits of HASH(s)
 7  function KEY(k, s, t)              ▷ t a successor of s, k = KEY(s)
 8     for each feature x that holds in exactly one of s and t do
 9        k ← k xor (the high 32 bits of T[x])
10     return k
11  function OWNER(s, p)
12     return floor(KEY(s) · p / 2^32)
```

Zobrist hashing spreads the states over the workers regardless of the structure of the graph. Its statistical properties are those of the following proposition, in which we take, as the analysis of hashing does, the entries of R independent and uniform on $\{0,1\}^{64}$; the table of the library is drawn by a pseudo-random generator from the seed, to which the statement applies only as a model.

Proposition 9.11 (The independence of Zobrist hashing). Let the entries of R be independent and uniform. (i) For two distinct nonempty sets of features $S \neq T$, the pair $(Z(S), Z(T))$ is uniform on $\{0,1\}^{128}$, i.e., Z is pairwise independent on the nonempty sets. (ii) If each state has exactly one feature per *position*, i.e., the features are the pairs (position, value) and a state gives one value to each position (as the cell of each tile on a sliding-tile puzzle), then Z is 3-independent, i.e., the hashes of three distinct states are independent and uniform. (iii) On sets of varying size Z is not 3-independent in general.

Proof. (i) Some feature x lies in exactly one of the two sets, say in $S \setminus T$. Given all the entries of R but $R[x]$, $Z(T)$ is fixed and uniform over the choice of those entries (since T is nonempty and does not contain x), while $Z(S)$ is $R[x]$ xor a fixed string, hence uniform and independent of $Z(T)$. (ii) Here Z is the simple tabulation hashing of the tuple of the values. Let a, b and c be three distinct states. At a position where they do not all agree, either their three values are distinct or exactly two are equal, and in both cases one of the three states, say a, has a value that the other two do not have; such a position exists, since the states are distinct. Given all the entries but that of a at this position, $Z(b)$ and $Z(c)$ are fixed, and $Z(a)$ is uniform; hence $Z(a)$ is uniform and independent of $(Z(b), Z(c))$, which by (i) is uniform on pairs. (iii) With $S = \{x\}$, $T = \{y\}$ and $U = \{x, y\}$ one has $Z(U) = Z(S) \oplus Z(T)$. □

Corollary 9.12 (The communication of Zobrist hashing). Under the hypothesis of Proposition 9.11, for two states s and t with distinct nonempty sets of features, the probability that $\text{owner}(s) = \text{owner}(t)$ lies in $[1/p, 1/p + 2^{-32}]$; hence a successor not pruned is sent to another worker with probability between $1 - 1/p - 2^{-32}$ and $1 - 1/p$.

Proof. By Proposition 9.11(i) the keys of s and t, the high 32 bits of their hashes, are independent and uniform on $\{0, \ldots, 2^{32} - 1\}$. Let n_i be the number of keys that `key_owner` maps to i; the probability of equal owners is $\sum_i n_i^2 / 2^{64}$. Since $\sum_i n_i = 2^{32}$, the Cauchy-Schwarz inequality gives $\sum_i n_i^2 \geq 2^{64}/p$, and since $n_i \leq 2^{32}/p + 1$ by Proposition 9.10, $\sum_i n_i^2 \leq (2^{32}/p + 1)\, 2^{32}$. □

The corollary is the reason for $\text{CO} \approx 1 - 1/p$ under Zobrist hashing (section 2.4 of [5], where this is noted for an ideal uniform assignment). It concerns a fixed pair of states; the fraction measured in a run is an average over the successors that the search generates, which depend on the owners themselves, and since the counter `generated` also counts the pruned successors, the measured CO is typically somewhat below $1 - 1/p$. The hash of a successor can be updated from that of its parent in the time of the features that the move changes (§9.5).

Abstract Zobrist hashing, HDA*[Z, Afeature]

Jinnai and Fukunaga combine the hashing with a projection A of the features onto fewer *abstract* features (section 4 of [5], equation (2)):

$$AZ(s) = \bigoplus_{x \in F(s)} R'[A(x)],$$

with a table R' of random strings indexed by the abstract features. `AbstractZobristDistribution` (`Distribution.h:203`, for an `AbstractFeatureDomain`) lays its table out by raw feature, i.e., its constructor draws R' from the seed and stores $T[x] = R'[A(x)]$ at the index of each raw feature x (Algorithm 3 of [5]); thus, the hash costs as much as Z and is computed and updated by the same procedures of Algorithm 9.4. A move that changes no abstract feature keeps the successor on the worker of its parent, which lowers the communication; we make this precise.

Corollary 9.13 (The owners under AZ). If every feature that holds in exactly one of s and t has an abstract feature that appears an even number of times among those of the changed features (e.g., a move that changes (k, c) into (k, b) with $A(k, c) = A(k, b)$), then $AZ(t) = AZ(s)$, and t has the owner of s for every p.

Proof. By Proposition 9.15 below, $AZ(t) = AZ(s) \oplus \bigoplus_x R'[A(x)]$ over the changed features x, and the entries of an abstract feature that appears an even number of times cancel, since $y \oplus y = 0$. □

Also, the hashing of all the abstract features keeps the load (rather) balanced: $AZ(s)$ is the Zobrist hash, with the table R', of the set of the abstract features that appear an odd number of times in s, and the statement of Corollary 9.12 holds for two states whose such sets are distinct and nonempty; on the domains of section 4.1 of [5], the 15-puzzle and the 24-puzzle among them, AZ was the best of the distributions compared there. On the sliding-tile puzzles one can take as the abstract feature of a tile the half of the board that holds it, as the library does, and then a move changes it only when it slides a tile vertically across the middle line; on the 2×2 puzzle of Figure 9.7 (middle) the halves are the rows, a horizontal move always keeps the owner, and at most half of the edges are cut.

We remark that a state in which two raw features have the same abstract feature has the two entries cancelled in AZ, since $R'[a] \oplus R'[a] = 0$. The hash is still a function of the state (which is all that the correctness of §9.2 requires), but its values are less uniform. On the sliding-tile puzzles this never happens, since each tile is in one cell.

Abstraction, HDA*[Z, Astate]

`AbstractStateDistribution` (`Distribution.h:413`, for an `AbstractStateDomain`) sends all the states with the same abstract state to the same worker, hashing the abstract state with a seeded mix (the 64-bit mixing function SplitMix64 of the abstract state xor the mixed seed, not a Zobrist hash of its features), and reducing the result by reduce. This is AHDA* of Burns et al. [3] with a uniform hash in place of their perfect hash (hence the notation HDA*[Z, Astate] rather than HDA*[P, Astate], cf. section 2.5 of [5]). Since the owner is a function of the abstract state, a successor changes worker only when it changes the abstract state, and its communication is the least of these distributions; but no mechanism balances the load: the abstract states are usually "few", and those that contain the nodes of least f may belong to few workers, which raises the search overhead. On the 15-puzzle the abstraction is the triple of the cells of tiles 1, 2 and 3, i.e., $16 \cdot 15 \cdot 14 = 3360$ abstract states.

Perfect hashing, HDA*[P]

`PerfectHashDistribution` (`Distribution.h:449`, for a `PerfectHashDomain`) takes the rank of the state modulo p. This is the distribution called "HDA*" in [3], and Jinnai and Fukunaga found that on the sliding-tile puzzles it has both a high search and a high communication overhead (sections 3.2 and 4.1 of [5]); the library keeps it to reproduce that comparison. The reason, at least on the puzzles, lies in the following proposition, which states for the rank of the library what footnote 5 of section 3.1.4 of [5] observes for $p = 8$.

Proposition 9.14 (The perfect hash modulo p). Let the board have n cells, let $\mathrm{pos}(k)$ be the cell of tile k (the blank being tile 0), and let the rank be the one of `SlidingTile.h:590-603`, i.e.,

$$\mathrm{rank}(s) = \sum_{k=0}^{n-1} d_k \, (n-1-k)!, \qquad d_k = |\{\, j > k : \mathrm{pos}(j) < \mathrm{pos}(k) \,\}|.$$

If m is the least integer with $p \mid m!$, then $\mathrm{rank}(s) \bmod p$ depends only on the relative order of the cells of the tiles $n-m, \dots, n-1$.

Proof. The code computes $\mathrm{rank} \leftarrow \mathrm{rank} \cdot (n-k) + d_k$ for $k = 0, \dots, n-1$, which expands to the sum above. Also, d_k is the cell of tile k minus the number of the cells below it that the tiles $0, \dots, k-1$ occupy, i.e., the number of the tiles $j > k$ below it. For $k < n-m$ one has $n-1-k \geq m$, hence p divides $(n-1-k)!$ and the term vanishes modulo p; the remaining d_k, $k \geq n-m$, compare the cells of the tiles $n-m, \dots, n-1$ only. □

On the 15-puzzle with $p = 8$ one has $m = 4$, and the owner depends on the relative order of the cells of the tiles 12, 13, 14 and 15 alone: a move that changes this order may change the owner, any other move keeps it, and the classes of states that share an owner follow neither the load nor the locality of the search. We remark that over all the boards the residues are (almost) equally frequent, the rank being a bijection onto $\{0, \dots, n! - 1\}$; what the proposition rules out is a balance of the nodes that the search reaches at each value of f, which nothing in the distribution controls.

The hash of the closed list, and a random owner

`StateHashDistribution` (`Distribution.h:477`, the default of both searches) remixes the hash that any `Domain` gives to its closed list with a seed, by the same mix as above; it behaves as Zobrist hashing for any `Domain` to the extent that the hash of the `Domain` is uniform, with no feature to define, but it cannot be updated by a move. `RandomDistribution` (`Distribution.h:598`) sends each generated state to a worker drawn at random, as in the randomized strategies of Kumar, Ramesh and Rao and of Karp and Zhang (cf. section 6 of [2]). Hence, a state has no owner, a duplicate is detected only when its copies happen to reach the same worker, and each worker needs a separate generator (which the class declares by `per_worker`; HDA* gives each worker a copy of the distribution, all of them seeded alike). It is the baseline without detection of the duplicates across the workers; Kishimoto et al. report that on their planning benchmarks, with 16 cores and 2 GB per core, it ran out of memory on every instance, and that with 32 GB per core it expanded more than an order of magnitude more nodes than HDA* (section 6 and Table 8 of [2]).

Zobrist hashing on a given table

`TableZobristDistribution` (`Distribution.h:335`, for a `FeatureDomain`) is Zobrist hashing on a table R that the caller gives, one entry per feature (a missing entry being 0), together with a name that says how the table was built; its hash, its key and its owner are computed and updated by the procedures of Algorithm 9.4 with $T = R$, so that Proposition 9.15 and Corollary 9.16 hold for it, and it is an `IncrementalDistribution` when

the `Domain` is an `IncrementalFeatureDomain`. The table may be that of a projection written out as a `Feature Abstraction` (`Distribution.h:274`), i.e., the vector of the abstract feature of each feature (or `no_abstract_feature`, `Domain.h:334`, for a feature that the projection ignores) and the number of the abstract features; its `table( seed )` holds $R'[A(x)]$ at each feature x, with R' drawn from the seed, and 0 at the ignored ones, `of()` builds it from a vector, and `identity( n )` is the projection of Zobrist hashing. With the same seed, the table of the identity is that of `ZobristDistribution`, and the table of the projection of an `AbstractFeatureDomain` that of `AbstractZobristDistribution`, since the random words of a table are drawn one after the other from the seed (Proposition 10.1); the three classes then give every state the same owner. The table distribution is therefore Abstract Zobrist hashing with any projection, not only the one that the `Domain` provides: an ignored feature has the entry 0 and changes no key, so that ignoring all the features but those of a few tiles hashes the abstract state of AHDA* by Zobrist hashing (HDA*[Z, Astate]) instead of the mix of `AbstractStateDistribution`. The projections of the sliding-tile puzzles compared by Jinnai and Fukunaga (by rows, pairs of cells, blocks of 2×2 cells, halves of the board, and the regions of the 24-puzzle) are given by `SlidingTile::tile_abstraction()`, and the projections and the tables built from a sample of the instance by `AutoDistribution.h`; both are described in Chapter 10. Unlike the other classes, whose name is a static method, its `name()` is the member given at the construction.

The hyperplanes of a levelled graph

`HyperplaneDistribution` (`Distribution.h:531`, for a `LevelDomain`, `Domain.h:396`) is the hyperplane work distribution HDA*[Hyperplane] of Kobayashi, Kishimoto and Watanabe [43], which assigns to the workers, modulo p, the planes of consecutive levels of a levelled graph (e.g., the total length of the prefixes aligned in the multiple sequence alignment), so that the states reached by several paths meet on fewer workers. Its owner is a function of the level and of the hash of the state, it is not incremental, and it is described, with its thickness and the alignment on which it is used, in Chapter 15.

A summary

The following table gathers what each distribution needs and what it trades; the overheads are those that the scheme leads one to expect, and only measurements (cf. §9.8) decide among them on a given domain.

Name	Needs	Incr.	Det.	CO	Load
Z	`FeatureDomain`	yes	yes	about $1 - 1/p$	balanced
AZ	`AbstractFeatureDomain`	yes	yes	below Z	balanced
A	`AbstractStateDomain`	no	yes	lowest	not controlled
P	`PerfectHashDomain`	no	yes	high	not controlled
H	`Domain`	no	yes	about $1 - 1/p$	balanced
R	`Domain`	no	no	about $1 - 1/p$	balanced
T	`FeatureDomain` and a table	yes	yes	that of the table	that of the table
Hyperplane	`LevelDomain`	no	yes	low on a levelled graph	by planes

The column "Incr." gives whether the distribution is an `IncrementalDistribution` (Z, AZ and T when the `Domain` is an `IncrementalFeatureDomain`), "Det." whether the owner is a function of the state, and the load of AZ is balanced only if the abstract features are many with respect to p; the overheads of T are those of the projection or of the construction of its table, and those of the hyperplanes are discussed in Chapter 15. The first six distributions are made "by hand", the features, the projection or the abstraction coming from the `Domain`. The choice of the table of T from the problem (by partitioning the transition graphs of the features, by the operators, by a state abstraction, or by a subspace of the kernel of the key, each estimated on a sample) is the subject of Chapter 10.

9.5 Incremental Zobrist hashing

A hash computed from scratch reads all the features of the state, i.e., $n-1$ entries of the table on a puzzle of n cells; Zobrist hashing can instead be updated by a move, from the hash of the parent and the features that the move changes (section 3 of [2] and section 2.4 of [5]). We now state the identity on which the update (lines 7 to 10 of Algorithm 9.4) rests, for the sets of features $F(s)$ and $F(t)$ of a state and of its successor, whose symmetric difference $F(s) \triangle F(t)$ is enumerated by `for_each_feature_change()`.

Proposition 9.15 (Incremental Zobrist hash). For any two states s and t,

$$Z(t) = Z(s) \oplus \bigoplus_{x \in F(s) \triangle F(t)} R[x],$$

and the same holds for AZ with the table laid out by raw feature, and for the keys, i.e., for the high 32 bits of Z and of AZ.

Proof. The set $\{0,1\}^{64}$ with $\oplus$ is an abelian group in which each element is its own inverse. Hence

$$Z(s) \oplus Z(t) = \bigoplus_{x \in F(s)} R[x] \oplus \bigoplus_{x \in F(t)} R[x] = \bigoplus_{x \in F(s) \triangle F(t)} R[x],$$

since each feature of $F(s) \cap F(t)$ appears twice and cancels, and $\oplus$ with $Z(s)$ on both sides gives the claim. For AZ the same computation holds with the entry $R'[A(x)]$ stored at the index of x, which is how `AbstractZobristDistribution` computes both the hash and its update. Finally, taking the high 32 bits, i.e., $y \mapsto \lfloor y/2^{32} \rfloor$, commutes with $\oplus$, since the exclusive or acts bit by bit; hence the identity holds for the keys as well. □

Corollary 9.16 (Owners from the parent). For a distribution that is an `IncrementalDistribution`, the function that `successor_owners()` returns maps each successor t of a node of state s and key k to `( owner( t , p ) , key( t ) )`, and it costs $|F(s) \triangle F(t)|$ lookups in the table instead of the $|F(t)|$ of a computation from scratch.

Proof. `key( k , s , t )` applies the identity of Proposition 9.15 to the key k of s (lines 7 to 10 of Algorithm 9.4), with one lookup per changed feature, and the owner is `key_owner()` of the key, as for `owner( t , p )`. □

Our tester checks the corollary on random walks of the 8-, 15- and 24-puzzles, for $p \in \{1, 2, 3, 7, 64\}$, against the owners and the keys computed from scratch (`check_successor_owners` in `test/test.cpp`).

On the sliding-tile puzzles a move slides one tile k from a cell c to the cell b of the blank, and the symmetric difference is $\{(k,c),(k,b)\}$: two entries of the table instead of 15 on the 15-puzzle (and none for the state being expanded, whose key is stored in its node). Both cells are found without locating the blanks: the packed states s and t differ only in the fields of the cells b and c, both of which hold k in $s \oplus t$, and therefore two counts of trailing zeros of $s \oplus t$ give them (`SlidingTile.h:420-431`). Figure 9.8 shows the update for a move inside a half of the board and for one across the middle line.

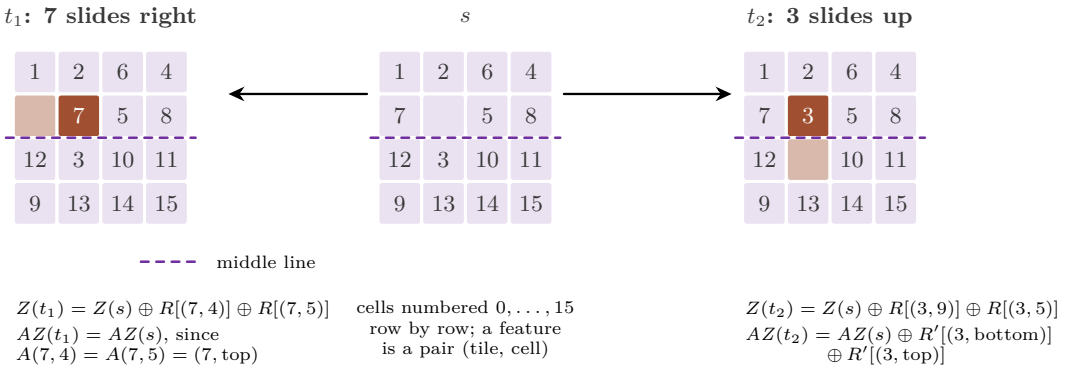


Figure 9.8. The updates of the Zobrist hash and of the abstract Zobrist hash by two moves from the state s (middle) of the 15-puzzle. Sliding 7 to the right (left) changes the features $(7,4)$ and $(7,5)$, in the rust colour, but not the abstract feature (7, top half), so that t_1 stays on the worker of s under AZ; sliding 3 up across the middle line (right) also changes its abstract feature, and t_2 may change worker.

9.6 HDA* over MPI

`HDAStarMPI` (`HDAStarMPI.h`) is the search of §9.2 with one MPI process per worker, as in the original implementation of Kishimoto et al., where the workers are MPI processes even on a single multicore machine (section 3 of [2]). Since the processes share no memory, a node records the *state* of its parent rather than a pointer, the incumbent travels as a message to all the other processes, and the state and the cost types must be trivially copyable, since the nodes travel as bytes (a static assertion checks the state type, `HDAStarMPI.h:114`). MPI is initialized by the caller (with at least `MPI_THREAD_FUNNELED` if the `Domain` uses threads), all the processes of the communicator call `solve()`, and all of them return the same `Solution`.

Algorithm 9.5 (HDA* over MPI, on the process i).

```
 1  procedure HDA*-MPI(s₀) on the process i
 2     z ← ∞;  if owner(s₀) = i then INSERT(s₀, 0, nil)
 3     loop
 4        for each message m received do
 5           if m is a batch of nodes then INSERT each of them
 6           else if m is an incumbent z' then z ← min(z, z')
 7           else if m is a stop then stop expanding
 8        if OPEN_i has a node of f < z then
 9           expand it as at lines 7 to 15 of HDA*, a goal that lowers z
               sending z to all the other processes
10        else send all the outboxes
11        if OPEN_i has no node of f < z and the mailbox detects the end
12           then break
13     the process of the least goal broadcasts it, and the path is rebuilt
         from it, each state broadcast with its parent by its owner
14     the sends in progress complete
```

This is the pseudocode of `HDAStarMPI.h:14-31`. In the library, the batches are sent with nonblocking sends (`MPI_Isend`), whose buffers are kept until MPI has completed them, and received with `MPI_Iprobe` and `MPI_Recv` from any process; the mailbox duplicates the communicator, and its messages never meet those of the program. The messages are a variant of a node and an incumbent, and the reception and the rebuilding of the path are made as follows, the parent of a node being a state, nil at the root.

Algorithm 9.6 (The reception and the path over MPI).

```
 1  procedure RECEIVE()                          ▷ on the process i
 2     M ← ∅
 3     for each message m drained from the mailbox do
 4        if m is a node then M ← M ∪ {m}
 5        else z ← min(z, the cost of the incumbent m)
 6     for each (v, g', u) ∈ M do                  ▷ phase 1
 7        compute the hash of v;  prefetch its place in CLOSED_i
 8     for each (v, g', u) ∈ M do INSERT(v, g', u)        ▷ phase 2
 9  procedure PATH()                             ▷ on every process
10     if some process hit the node limit then return node_limit
11     (c, r) ← the least cost of a recorded goal and its process
12     if c = ∞ then return unsolvable
13     (s, γ, u) ← the goal of r, its g and its parent, broadcast by r
14     π ← (s)
15     while u ≠ nil do
16        j ← owner(u)
17        (s, γ', u) ← u, the parent of its node at j, broadcast by j
18        π ← (s) followed by π
19     return the path π, of cost γ
```

Line 10 is a reduction (the maximum of the flags of the processes), and line 11 a reduction with `MPI_MINLOC` (`HDAStarMPI.h:383`), after the counters of all the processes have been gathered (`MPI_Allgather`). Over MPI the counter `messages` is the number of the batches sent, those that carry only incumbents included, while `sent` counts nodes only; hence, unlike on a shared memory, `messages` may exceed `sent`.

We let each process keep an incumbent z_i, the least cost that it has found or received; since z_i is never smaller

than the least cost z found by any process, a process prunes less than a shared z would, which may cost expansions (how many depends on the latency of the incumbent messages) but not correctness.

The termination detection of Mattern

Over MPI, the end is detected by the four counters of Mattern [11], in the form described in section 3 of [2] and in section 3.3.1 of [1]. Kishimoto et al. used the time algorithm of Mattern, a variation that needs one wave only, while the library uses the basic method with two waves. Each process i counts the messages that it has sent, s_i, and those that it has received, r_i, both nondecreasing. We say that a process is *passive* when it has no node of $f < z_i$ and has posted all its outboxes, and it becomes active only by receiving a message (an incumbent never gives work, and a node usually may). Note that a process with work never contributes to a wave.

Definition 9.17 (Waves). A *wave* is a nonblocking reduction (`MPI_Iallreduce`) of the pairs (s_i, r_i) of all the processes into their sums (S, R); a process contributes to a wave only when it is passive, i.e., from QUIESCENT, at an instant t_i, and it keeps receiving and working while the wave completes. A process starts the wave k only after the wave $k-1$ has completed for it. The end is declared at the first wave $k \geq 2$ whose sum of the sends S_k equals the sum of the receipts R_{k-1} of the wave before.

Algorithm 9.7 (QUIESCENT over MPI, on the process i of p, with no node of $f < z_i$). The wave also sums a flag stopped of each process, X is the number of the processes stopped, k counts the waves completed, and R' is the sum of the receipts of the last wave.

```
 1  function QUIESCENT()
 2     if the end is known then return true
 3     post all the outboxes, by nonblocking sends
 4     if a wave is in progress then
 5        if it has not completed then return false
 6        k ← k + 1;  (S, R, X) ← its sums
 7        if X = p then the end is known;  return true   ▷ all stopped
 8        if k > 1 and S = R' then the end is known;  return true
 9        R' ← R
10     start a wave, the nonblocking sum of (s_i, r_i, stopped)
11     return false
```

Since a reduction completes for a process only after all the processes have contributed to it, each contribution to the wave k comes after all the contributions to the wave $k-1$: if t_i^k is the instant of the contribution of i to the wave k, then $t_j^k > \max_i t_i^{k-1}$ for all j. A single wave is not enough, as the left half of Figure 9.9 shows. In fact, the counters of a wave are read at different instants. A message sent before the cut and received after it is counted by S but not by R, while one sent after the cut and received before it is counted by R only. The two errors may cancel while a message is in flight.

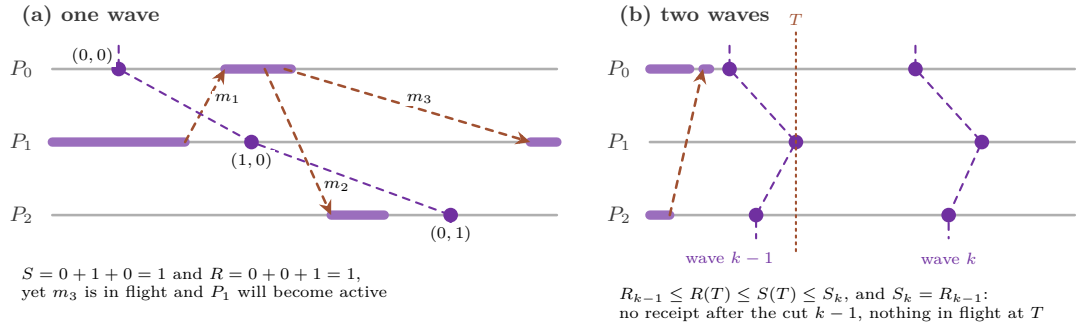


Figure 9.9. (a) One wave, with the contributions (s_i, r_i) of the three processes on a dashed cut: m_1 crosses the cut forwards and m_2 backwards, so that $S = R = 1$ although m_3 is in flight and will make P_1 active. (b) Two waves at the end of a computation: T is the last contribution to the wave $k-1$, and the equality of S_k and R_{k-1} forces each inequality of the chain to be an equality.

Theorem 9.18 (Mattern's four counters detect the end exactly). Let each process send only while active and become active only by receiving, and let $T = \max_i t_i^{k-1}$. (i) If $S_k = R_{k-1}$, then at the instant T all the processes were passive and no message was in transit; hence the computation had ended at T, and the end is never declared before it happens. (ii) If the computation ends at some instant T^* and each process keeps calling

QUIESCENT, then the end is declared, at the latest at the second wave all of whose contributions follow T^*, and all the processes declare it at the same wave.

Proof. (i) Let $S(t) = \sum_i s_i(t)$ and $R(t) = \sum_i r_i(t)$. A message is received only after it is sent; hence, $R(t) \le S(t)$ at all times, and the counters are nondecreasing. Since $t_i^{k-1} \le T$ for all i, one has $R_{k-1} = \sum_i r_i(t_i^{k-1}) \le R(T)$, and since $t_i^k > T$ for all i, one has $S(T) \le \sum_i s_i(t_i^k) = S_k$. Therefore

$$R_{k-1} \le R(T) \le S(T) \le S_k = R_{k-1},$$

and all are equalities. From $R(T) = R_{k-1}$ and the monotonicity of each r_i it follows that $r_i(T) = r_i(t_i^{k-1})$ for each i, i.e., no process received a message between its contribution to the wave $k-1$ and T. Since it was passive at its contribution and it can become active only by receiving, it was passive at T. From $S(T) = R(T)$, no message was in transit at T. A computation in which all the processes are passive and no message is in transit has ended, and it stays so, since nobody can send.

(ii) After T^* no message is sent or received; thus, S and R are constant and equal, say to N. Every process is passive and calls QUIESCENT, and therefore each wave completes in finite time; let W be the first wave all of whose contributions follow T^* (it exists, since the contributions to a wave follow those to the wave before). Then $R_W = N$ and $S_{W+1} = N$, and the end is declared at the wave $W+1$ at the latest. The sums of a reduction are the same on all the processes, and so is the decision. □

The end that Theorem 9.18 detects is the end assumed by Theorem 9.4. At that instant all the incumbents sent have been received; hence, each z_i equals the least cost found by any process; no process has a node of $f < z$, none is expanding, and no node is in transit. Hence, the proof of Theorem 9.4 applies word for word, with z_i in the pruning of the process i (which, as remarked above, prunes no more than z would). A wave runs while the processes keep working, and a process never waits for one; its latency is that of a reduction of MPI, typically logarithmic in the number of processes, which depends on the implementation of MPI and which we do not prove.

A process that reaches the node limit sends a note (an empty batch) to each other process; a process that sees it stops expanding and contributes to the waves as stopped, and the computation ends at the first wave to which all have contributed stopped (line 7 of Algorithm 9.7). The notes are not counted by the waves, which a stop ends by the flags alone. When the mailbox is destroyed, at the end of `solve()` and after the path has been rebuilt, the messages still in flight (which exist only after a stop) are received and dropped, through a collective exchange of the number of the messages that each process has sent to each other one (`MPI_Reduce_scatter_block`), and the pending sends are completed.

The path

We rebuild the path after the end, by the procedure PATH of Algorithm 9.6 (Figure 9.10): each process knows the next parent, hence its owner, and the owner looks it up in its closed list. This is why the distribution of `HDAStarMPI` must be deterministic: with the random owner, the process queried for a state need not be the one that stores it. We now prove that the rebuilding is sound.

Lemma 9.19 (The path over MPI). With a deterministic distribution and without the node limit, at the end of HDA* over MPI: (i) each state reached has one node, at its owner; (ii) the parent of each stored node other than the root is a state that has a node at its owner, so that the lookup of line 17 of Algorithm 9.6 never fails; and (iii) the chain of the parents of the goal is a path from s_0 of cost at most its g, which PATH rebuilds with one broadcast per state.

Proof. (i) is Lemma 9.1. (ii) The parent of a node is set from a message (v, g', u), created by the expansion of the node of u at the owner of u, and nodes are never deleted. (iii) By (i), a parent state names one node, and the proof of Lemma 9.2(ii) to (iv) holds with the parent states in the place of the pointers; hence the chain is acyclic, ends at s_0, whose parent is nil, and has cost at most $g(t)$. Each iteration of line 15 adds one state with one broadcast, and all the processes run the same iterations, since each broadcast gives them the same values. □

9.7 The engineering of the implementation

The algorithm of §9.2 leaves much freedom to the implementation, and we use it as follows.

- The inbox is drained as a whole, and the hashes of all its nodes are computed and their slots in the closed list prefetched before the first lookup (lines 5 to 8 of Algorithm 9.2, and lines 6 to 8 of Algorithm 9.6 over

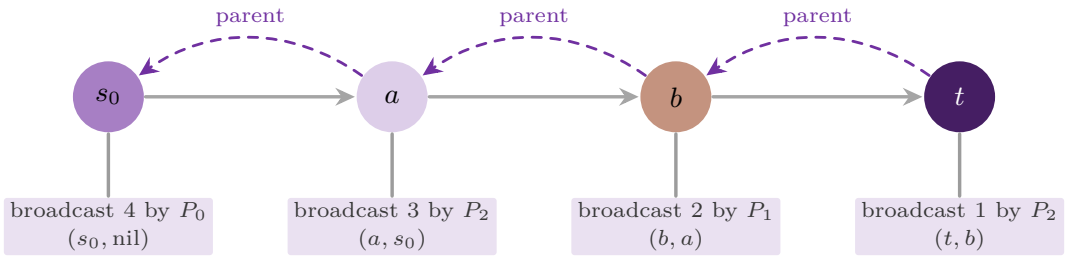


Figure 9.10. The path rebuilt over MPI: the process of the least goal broadcasts (t, b), i.e., the goal and its parent; the owner of b broadcasts (b, a); and so on to s_0, whose parent is nil. The colour of a state is its owner.

MPI). A message from another worker typically carries a state that the receiver has not touched recently, and its lookup is then likely to be a "cold" miss.

- The incumbent is an atomic variable on a separate cache line, and each worker is aligned to a cache line (`HDAStar.h:250` and `:260`), and the counters and the lists of one worker do not share a line with those of another. Also, the pruning tests read the incumbent with a relaxed load: a stale value is larger than the current one (the incumbent only decreases), and therefore it prunes less, never more.
- Each worker allocates its nodes from a private pool and keeps its closed list in a private table (cf. Chapter 3); the memory of a node is therefore written by one thread only, which is the content of Lemma 9.1.
- The key of a node, 32 bits, is stored in the node, where it fits in the padding of a node whose state is one or two words; the key of the state being expanded is never recomputed.
- With integer costs the OPEN list is a list of buckets, as in the sequential search; otherwise it is a binary heap.
- A batch costs one atomic addition at the sender and at most one at the receiver on the counter of Definition 9.6 (Proposition 9.5), and a passive worker only reads that counter.

9.8 Benchmark

Table 9.1. HDA* with the distributions Z, AZ and P on the threads, and with Z and AZ on OpenMP, on FastFlow and over MPI, on the 15-puzzle.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The experiment runs on 12 of the 100 instances of Korf [7] (the instances 1, 3, 10, 14, 15, 22, 26, 27, 32, 33, 37 and 52, as listed in `tools/bench/campaign.sh`), with the Manhattan distance, the default batches of 64 nodes and flushes every 64 expansions, on 1, 2, 4, 8, 16, 32, 48, 64 and 96 cores. A row is a search, i.e., HDA* with its distribution in brackets and its runtime (thread, openmp, fastflow or mpi), or the sequential A* with the OPEN list on buckets, against which the speedups are computed; a column is the number of cores given to the search (one worker per core, the workers pinned, and over MPI one process per core). A cell gives the seconds summed over the instances, each instance counting with the median of its two runs, and in parentheses the speedup over the sequential A* on 1 core; a cell is empty when the search did not solve all the instances. The machine, the compiler and the A/A noise (the gap between the two runs of the same search, which are made in two processes) are stated in the note under the table. All costs are integers; each run writes the cost it found in the records of the campaign, where it can be compared with that of the sequential search (the table does not show it, and a run counts in a cell as soon as it reports a solution). We measure the wall-clock time of `solve()`, i.e., from the creation of the workers to the solution, the tracing of the path included; since the benchmark gives no executor to the search, the start of the threads is included as well (over MPI, the time is that of the process of rank 0, the duplication of the communicator and the final gathering of the counters included). The time does not include the start of the process, the reading of the instance, the construction of the `Domain` and of the tables of the distribution, nor the release of the memory of the search. We remark that the table is a placeholder until the campaign is run on an idle machine.

9.9 Notes and references

The ownership of a state by a hash comes from PRA* of Evett et al., and the asynchronous exchange of the nodes, packed per destination, from the transposition-table-driven scheduling of Romein et al. HDA* of Kishimoto, Fukunaga and Botea [2] combined the two with Zobrist hashing and showed that the scheme scales on clusters and on multicore machines (we cite PRA* and TDS through section 2.3 of [5] and section 3 of [2], where they are summarized). Also, the survey of Fukunaga et al. [1] places HDA* among the decentralized parallel searches (section 4.1) and describes the termination detection of Mattern [11] (section 3.3.1); the deficit counting is that of Dijkstra and Scholten [35]. Burns et al. [3] compared HDA* with the perfect hash and AHDA* with SafePBNF (cf. Chapter 11). Jinnai and Fukunaga [5] showed that the perfect hash was a poor representative of HDA*, introduced the notation HDA*[*hash*, *abstraction*], Abstract Zobrist hashing and the automatic distributions of Chapter 10, and explained the search overhead of HDA* by the band and burst effects. In their comparison on the 15-puzzle and the 24-puzzle, HDA*[Z] was faster than SafePBNF on the instances that take more than about 10 seconds, and SafePBNF on the easier ones, where the initial burst weighs more (section 3.2 of [5]); the outcome on other domains and machines is not necessarily the same, and on grid pathfinding the abstraction-based searches were faster than HDA*[Z], whose communication overhead is high there (sections 3.3 and 3.4 of [5]). Finally, the Zobrist hash comes from the game-playing programs [152], and the reduction of a hash to an interval without a division from Lemire [151].

As to where the library stands, the following facts are stated from the code and the papers, not from measurements. The library offers all the distributions of the comparison of Jinnai and Fukunaga that need no automatic construction (Z, AZ, A and P, and the hyperplanes of Kobayashi et al. of Chapter 15), together with the hash of the closed list, the random owner and the Zobrist hashing on any table, on four runtimes and under the same search. The original implementation of [2] ran on MPI only, and the one of [5] for the 24-puzzle and the multiple sequence alignment used Pthreads with try locks for the asynchronous communication (section 4.1 of [5]), while the shared-memory search of the library exchanges batches through a mailbox whose end detection costs one atomic addition per batch at each side. Furthermore, the incremental key, the prefetching of whole inboxes and the skipping of the inverse move are engineering choices that these papers do not discuss (the last one is in [4], for the sequential search). The automatic distributions of Jinnai and Fukunaga (GAZHDA*, GRAZHDA*, FAZHDA*, OZHDA* and DAHDA*) are built on a sample of the instance in Chapter 10, on a branch that is not in a release yet. On the other hand, the library has no implementation of the time algorithm of Mattern, which detects the end with one wave instead of two at the price of a time stamp on each message.

9.10 Exercises

Exercise 9.1 (`ctest -R PBeFS.hda-thread`). Prove that the key of `ZobristDistribution` computed by `key ( k , s , t )` equals `key( t )` for each successor t of s, using Proposition 9.15, and that the same holds for `AbstractZobristDistribution`; the test checks it on random walks of the 8-, 15- and 24-puzzles for five values of p.

Exercise 9.2 (`ctest -R PBeFS.hda-owners`). On the 15-puzzle with the abstract features of `SlidingTile.h` (a tile and the half of the board that holds it), count, over the 16 cells of the blank, the pairs (cell of the blank, move) and those whose move changes an abstract feature. Show that the latter are 8 of 48, and discuss why this is not the fraction of the generated nodes that change worker under AZ (hint: the cells of the blank are not equally frequent among the generated states, and a node whose abstract features change still keeps its owner with probability about $1/p$).

Exercise 9.3 (`ctest -R PBeFS.hda-thread`). Check that the proof of Theorem 9.4 holds node by node with `RandomDistribution`, and say which of the statements of §9.2 and of §9.6 do need a deterministic owner. Its test runs it on 21 states of the 8-puzzle, whose $9!/2$ reachable states bound the memory whatever the timing.

Exercise 9.4 (`ctest -R PBeFS.hda-thread`). Show that the proofs of Theorem 9.4 and of Proposition 9.8 do not depend on the values of `batch` and `flush_period`, and that with both equal to 1 the search sends each node as soon as it is generated. The test runs the search with batches of 1 and with the parameters set to 0 (which `normalize()` turns into 1).

Exercise 9.5 (discussion, no test). Build a graph of four states with a consistent heuristic, a distribution on two workers and an order of the messages in which HDA* expands a state twice. Which counter of the `SearchStats` records the second expansion?

Exercise 9.6 (discussion, no test). Generalize the run of Figure 9.9(a): show that for every $p \geq 3$ there are runs in which a single wave has $S = R$ while a message is in transit, and find the least number of messages that such

a run needs. Does such a run exist with $p = 2$ processes?

Exercise 9.7 (discussion, no test). Bound the number of waves that complete after the end T^* of the computation before the end is declared (Theorem 9.18(ii)), counting the wave that may be in flight at T^*, and show that the bound is attained.

Exercise 9.8 (`ctest -R PBeFS.mpi`). Show that two processes that build `StateHashDistribution` with the same seed assign each state to the same owner, and describe how the rebuilding of the path of §9.6 would fail with `RandomDistribution`. The test runs the search over MPI with each deterministic distribution, with 1, 2 and 4 processes.

Exercise 9.9 (`ctest -R PBeFS.mpi`). Show that, after a process reaches the node limit, all the processes return from `solve()` with the status `node_limit`, even if some process found a goal in the meantime, and that no message is left pending (the test stops the search over MPI at 2000 nodes).

Exercise 9.10 (`ctest -R PBeFS.hda-owners`). Using Proposition 9.14, find the tiles on which the owner of a state of the 15-puzzle depends under `PerfectHashDistribution` for $p = 3$, $p = 5$, $p = 7$ and $p = 16$, and argue for which of these values the distribution is closest to a uniform one.

Exercise 9.11 (discussion, no test). Turn the two rounds of idle flags of Proposition 9.9 into a sound detection by adding counters of the messages, and show that the result is the counter of Definition 9.6 or the waves of Definition 9.17, depending on whether the workers share memory.

Exercise 9.12 (discussion, no test). Which problems are likely to parallelize well with HDA*, and which badly? Consider the cost of an expansion against that of a message, the number of states with $f < C^*$ against p, the branching factor, the number of states with the same f, and whether a good abstraction of the features exists; then place the domains of Chapter 14 in your classification.

Exercise 9.13 (`ctest -R PBeFS.hda-owners`). Show that the least key that `key_owner` maps to the owner i among p workers is $\lceil i\,2^{32}/p \rceil$, and derive from it the statement of Proposition 9.10 for the keys; the test checks, for $p \in \{3, 7, 64, 96\}$, the first key of each owner and that the numbers of the keys of the owners differ by at most one.

Exercise 9.14 (`ctest -R PBeFS.distributions`). Show that a `TableZobristDistribution` on the table of `FeatureAbstraction::of( FifteenPuzzle::tile_abstraction( TileRegions::half ) )` keeps on its worker exactly the moves that `AbstractZobristDistribution` keeps on the 15-puzzle (with probability 1, and the others with probability 2^{-32}), although the two do not give the same owners; which of the two would Proposition 10.1(iii) make equal, and why not this one? The test counts the regions of each hand-made projection.

10. Automatic work distribution

[Source: include/PBeFS/astar/AutoDistribution.h (FeatureModel, FeatureGraph, DistributionEstimate, Gf2Basis and the constructions), include/PBeFS/astar/Distribution.h (TableZobristDistribution, FeatureAbstraction), include/PBeFS/astar/Domain.h (FeatureGroupDomain, FeatureTransitionDomain), include/PBeFS/astar/SlidingTile.h (tile_abstraction(), TileRegions), include/PBeFS/astar/Solution.h (search_overhead(), storage_balance())]

> **Status — beta.** The code that this chapter describes is in `main` but not yet in a release, and its interface may still change.

We are concerned in this chapter with the choice of the work distribution of HDA* (Chapter 9) from the instance itself, without a projection or an abstraction written by hand for the domain. The distributions of §9.4 take their structure from the `Domain` (its features, its projection onto abstract features, its abstract state), and the quality of HDA* then rests on the author of the `Domain`; on the sliding-tile puzzles a good projection is known, but on a planning task, whose variables and operators are only given as data, there is none to write by hand. Jinnai and Fukunaga [141], [138], [5] build the distribution from the transition graphs of the variables of the task (the cuts of GAZHDA*, GRAZHDA* and FAZHDA*), from the operators (OZHDA*), or from a greedy state abstraction (AHDA* and DAHDA*), and they estimate the communication overhead (CO), the load balance (LB) and the search overhead (SO) of a distribution on a graph that stands for the part of the state space that the search explores, so that the distributions can be compared before HDA* runs. The library implements all of these constructions on a sample of the instance expanded by A*, for any `FeatureDomain`; every one of them yields a table of Zobrist hashing, i.e., a `TableZobristDistribution`, so that the correctness of HDA* and the incremental key of Chapter 9 hold for all of them at once. To these the library adds a few constructions of its own, namely the cuts weighed by the sample and by the estimated efficiency, the selection of a subspace of the kernel of the key (KZHDA*), and the choice among all the constructions of the one of largest estimated efficiency, which also tunes their parameters on the instance.

We use the notation of Chapter 9, the overheads of §8.2, and the notation HDA*[*hash*, *abstraction*] of section 2.6 of [5]. The structure of the chapter is as follows. §10.1 states the problem and the interface (the table distribution, the refinements of the `Domain` that the constructions read, and the functions that build them), and §10.2 the sample and the transition structure computed on it. §10.3 gives the estimates of the overheads and their measured counterparts, and §10.4 the view of the key as a linear map over GF(2), in which each distribution is the choice of a subspace of change vectors on which the key vanishes; this view is used by the sections that follow, which treat the cuts of the transition graphs (§10.5), the state abstractions (§10.6), the operator-based tables (§10.7) and the selection of the kernel (§10.8). §10.9 describes the automatic choice and the hand-made projections of the sliding-tile puzzles, and §10.10 the engineering of the implementation; the chapter ends with the benchmark, the notes and the exercises.

10.1 The problem and the interface

Let the `Domain` be a `FeatureDomain` (§9.1) with n features, so that a state s is described by the set $F(s)$ of the features that hold in it. A table R of n words of 64 bits defines the hash $Z_R(s) = \bigoplus_{x \in F(s)} R[x]$, the key of s (its high 32 bits) and its owner among p workers, the image of the key by `key_owner` (§9.4). The problem of this chapter is the choice of R, from the instance and before the search, such that HDA* with this owner has a small communication overhead and a small load balance, the two quantities that (through the search overhead that an imbalance causes) decide its efficiency. Zobrist hashing is the choice of a random R, and Abstract Zobrist hashing that of a table whose entries are equal on the features with the same abstract feature; the constructions of this chapter choose R by a program.

Zobrist hashing on a given table

`TableZobristDistribution` (Distribution.h:335) takes the table from the caller, one entry per feature, together with a name that says how the table was built:

```
template< FeatureDomain D >
class TableZobristDistribution {
 public:
 TableZobristDistribution( const D & domain , std::vector< std::uint64_t > r ,
                           std::string name = "T" );
 TableZobristDistribution( const D & domain , const FeatureAbstraction & a ,
                           std::uint64_t seed = 1 , std::string name = "AZ" );
 [[nodiscard]] std::uint64_t hash( const state_t< D > & s ) const;
 [[nodiscard]] std::uint32_t key( const state_t< D > & s ) const;
 [[nodiscard]] unsigned owner( const state_t< D > & s , unsigned p ) const;
 [[nodiscard]] std::uint32_t key( std::uint32_t k , const state_t< D > & s ,
                                  const state_t< D > & t ) const
  requires IncrementalFeatureDomain< D >;
 [[nodiscard]] const std::vector< std::uint64_t > & table( void ) const;
 [[nodiscard]] const std::string & name( void ) const;
 };
```

(declarations quoted from Distribution.h:335-402 without their bodies). The first constructor pads or cuts the table to `num_features()` entries, a missing entry being 0; the second one takes the table of a `FeatureAbstraction`, the written-out form of a projection A of the features (Distribution.h:274):

```
struct FeatureAbstraction {
 std::vector< std::uint32_t > abstract;   // A( x ), or no_abstract_feature
 std::uint32_t num_abstract = 0;          // all the A( x ) are below it
 [[nodiscard]] static FeatureAbstraction of( std::vector< std::uint32_t > a );
 [[nodiscard]] static FeatureAbstraction identity( std::size_t n );
 [[nodiscard]] std::vector< std::uint64_t > table( std::uint64_t seed = 1 )
  const;
 };
```

Here `of()` sets `num_abstract` to one more than the largest abstract feature of the table, `identity( n )` maps each feature to itself, and `table()` returns $R[x] = R'[A(x)]$ for each feature x, with R' a table of random words drawn from the seed, and $R[x] = 0$ for a feature that the projection ignores (whose abstract feature is the constant `no_abstract_feature` of Domain.h:334). Unlike the distributions of Chapter 9, whose name is a static method, the name of a table distribution is a member, since it depends on the construction.

Proposition 10.1 (Zobrist hashing on a table). Let R be any table and X the `TableZobristDistribution` of R. (i) The owner of a state is a function of the state; hence HDA* with X returns an optimal path under the hypotheses of Theorem 9.4, and `HDAStarMPI`, which needs a deterministic owner, may use it if every process builds the same table. (ii) If the `Domain` is an `IncrementalFeatureDomain`, X is an `IncrementalDistributio n`, i.e., `key( k , s , t )` equals `key( t )` for the key k of s and every successor t of s, and costs $|F(s) \triangle F(t)|$ lookups. (iii) With the same seed, the table of `FeatureAbstraction::identity( n )` is the table of `ZobristD istribution`, and, for an `AbstractFeatureDomain`, the table of `FeatureAbstraction::of()` of its projection is the table of `AbstractZobristDistribution`; hence the three classes then assign every state to the same owner.

Proof. (i) The hash is the exclusive or of the entries of the features of the state, and the owner is `key_owner` of its high 32 bits; neither depends on anything but the state and the table, which does not change after the construction. The optimality of HDA* requires of the owner only that it be a function of the state (the proof of Theorem 9.4 uses Lemma 9.1, i.e., that all the copies of a state reach one worker), and over MPI the path is rebuilt by asking the owner of each state (Lemma 9.19). (ii) Proposition 9.15 holds for any table, since its proof uses only that $\oplus$ is associative, commutative and its own inverse; `key( k , s , t )` adds to k the high 32 bits of the entries of the features that `for_each_feature_change()` enumerates, exactly as in lines 7 to 10 of Algorithm 9.4. (iii) The table of random words of the library is drawn word after word from a generator seeded once, so that its i-th word depends on the seed and on i only, not on the length of the table. `ZobristDistribu tion` takes as the entry of x the x-th word of a table of n words, and so does the identity projection; `AbstractZ obristDistribution` takes the $A(x)$-th word of a table of `num_abstract_features()` words, and the table of

`of()` takes the $A(x)$-th word of a table of `num_abstract` words, which is the same word. □

All the constructions of this chapter return either a `FeatureAbstraction`, which is then hashed by the second constructor, or a table, which is given to the first; by Proposition 10.1 every one of them may be used by HDA* on any runtime, by `HDAStarMPI` (the construction is deterministic, Proposition 10.23), and by the other searches that take a distribution (e.g., TDS, cf. Chapter 12), and the key of a successor is computed from that of its parent.

Groups and transitions

The constructions read the structure of the moves, and two refinements of the `Domain` let it give that structure instead of having it inferred from a sample.

Definition 10.2 (Feature groups and ground transitions). A `FeatureGroupDomain` (`Domain.h:238`) maps each feature i to a *group* `feature_group( i )` among the `num_feature_groups()` groups, the features of a group being mutually exclusive, as the values of a variable of SAS+ (an *atom group*): a move that deletes a feature of a group adds another one of the same group. A `FeatureTransitionDomain` (`Domain.h:255`) lists its ground transitions: `for_each_feature_transition( f )` calls `f( i , j )` for every ground operator and every feature i that it deletes and j that it adds in the same group. The *domain transition graph* (DTG) of a group is the undirected graph on its features with an edge $\{i, j\}$ for every such pair; an edge counts as many times as the ground operators that realize it, in either direction.

```
template< class D >
concept FeatureGroupDomain = FeatureDomain< D > &&
 requires( const D & d , std::size_t i ) {
  { d.num_feature_groups() } -> std::convertible_to< std::size_t >;
  { d.feature_group( i ) } -> std::convertible_to< std::size_t >;
  };

template< class D >
concept FeatureTransitionDomain = FeatureDomain< D > &&
 requires( const D & d ) {
  d.for_each_feature_transition( []( std::size_t , std::size_t ) {} );
  };
```

This is the DTG of SAS+ (Bäckström and Nebel [133]; section 6 of [5] defines it as a directed graph, and the constructions of the paper and of the library use its undirected form). On the sliding-tile puzzles (`SlidingTile.h:436-462`) the group of the feature "tile t in cell c" is the tile t, and the ground transitions are the moves of t between two adjacent cells, one per ordered pair of cells; hence the DTG of a tile is the grid graph of the board, each of its edges weighing 2. (The group 0 of the blank holds features that never hold, since the blank has no feature.) A planning task (Chapter 16) is both: its groups are its variables and its features its facts $v = d$, so that a state has exactly one feature per group, and an operator with an effect $v := d'$ gives the arc from each value $d \neq d'$ that its precondition on v allows (from every value if it has none) to d' (`Planning.h:250-275`). The test `distributions` checks that each group of a sample of a planning task is one of its variables and that its DTG edges join facts of one variable. A `Domain` that is neither has its groups inferred from the moves and its DTG weighed by the distinct operators of the sample (§10.2).

The interface of the constructions

A construction starts from a `FeatureModel`, built from the `Domain` and a number of expansions, which samples the instance and computes the transition structure (§10.2); its part that does not depend on the `Domain` is the base class `FeatureGraph`, on which every construction is written, so that the constructions are compiled once for all the domains. We quote the declarations from `AutoDistribution.h:168-672`, without their bodies and attributes.

```
class FeatureGraph {
 public:
 static constexpr std::uint32_t no_group = 0xFFFFFFFF;
 struct Arc { std::uint32_t to; double w; };
 struct Operator { std::vector< std::uint32_t > features; double weight; };
 std::size_t num_expanded( void ) const noexcept;
```

```
 const std::vector< std::pair< std::uint32_t , std::uint32_t > > &
  edges( void ) const noexcept;
 std::size_t num_features( void ) const noexcept;
 std::size_t num_groups( void ) const noexcept;
 std::uint32_t group( std::size_t i ) const noexcept;
 bool seen( std::size_t i ) const noexcept;
 double mass( std::size_t i ) const noexcept;
 const std::vector< std::uint32_t > & members( std::size_t g ) const noexcept;
 const std::vector< Arc > & arcs( std::size_t i ) const noexcept;
 const std::vector< Arc > & static_arcs( std::size_t i ) const noexcept;
 double fluency( std::size_t g ) const noexcept;
 const std::vector< Operator > & operators( void ) const noexcept;
 const std::vector< std::uint64_t > & words( void ) const noexcept;
 std::pair< const std::uint32_t * , const std::uint32_t * >
  state_features( std::size_t k ) const noexcept;
 DistributionEstimate estimate_table( const std::vector< std::uint64_t > & r ,
                                      unsigned p , double c = 1 ) const;
 };

template< FeatureDomain D >
class FeatureModel : public FeatureGraph {
 public:
 explicit FeatureModel( const D & domain , std::size_t expansions = 32768 );
 const D & domain( void ) const noexcept;
 const std::vector< state_t< D > > & states( void ) const noexcept;
 template< class X >
 DistributionEstimate estimate( const X & x , unsigned p , double c = 1 )
  const;
 };
```

The estimates are a `DistributionEstimate` (§10.3), and the objective of the cuts of GRAZHDA* a `CutObjective` (§10.5):

```
struct DistributionEstimate {

 double co = 0;              ///< the communication overhead
 double lb = 1;              ///< the load balance
 double so = 0;              ///< the search overhead, lb - 1
 double efficiency = 1;      ///< the estimated efficiency
 double sparsity = 0;        ///< the scaled sparsity

 };  // end( struct( DistributionEstimate ) )
```

```
enum class CutObjective {
 sparsity ,          ///< the largest \f$ |S_1| |S_2| / w( S_1 , S_2 ) \f$,
                     ///< with the static weights
 sampled_sparsity ,  ///< the same, with the weights of the sample
 efficiency          ///< the largest \f$ 1 / ( ( 1 + c \, CO ) \, LB ) \f$,
                     ///< with the weights of the sample
 };
```

The constructions are functions at namespace scope (`AutoDistribution.h:943-1545`):

```
FeatureAbstraction greedy_abstraction( const FeatureGraph & m );
FeatureAbstraction sparsest_cut_abstraction( const FeatureGraph & m ,
 CutObjective o = CutObjective::sparsity , double c = 1 );
FeatureAbstraction fluency_abstraction( const FeatureGraph & m ,
                                        double ignored = 0.3 );
FeatureAbstraction dynamic_state_abstraction( const FeatureGraph & m ,
                                              double ratio = 0.3 );
FeatureAbstraction state_abstraction( const FeatureGraph & m ,
                                      std::size_t max_states = 1000 );
```

```
std::vector< std::uint64_t > operator_table( const FeatureGraph & m ,
 double fraction = 0.2 , std::uint64_t seed = 1 , bool as_paper = false );
std::vector< std::uint64_t > kernel_table( std::size_t n ,
 const Gf2Basis & basis , std::uint64_t seed = 1 );
std::vector< std::uint64_t > kernel_selection_table( const FeatureGraph & m ,
 unsigned p , double c = 1 , std::uint64_t seed = 1 ,
 std::size_t candidates = 1024 , const FeatureAbstraction * start = nullptr );
const std::vector< std::string > & auto_distribution_names( void );
template< FeatureDomain D >
TableZobristDistribution< D > make_distribution( const FeatureModel< D > & m ,
 const std::string & name , double c = 1 , std::uint64_t seed = 1 ,
 unsigned p = 8 );
template< FeatureDomain D >
TableZobristDistribution< D > auto_distribution( const FeatureModel< D > & m ,
 unsigned p , double c = 1 ,
 const std::vector< TableZobristDistribution< D > > & extra = {} ,
 DistributionEstimate * est = nullptr , std::uint64_t seed = 1 );
```

together with the class `Gf2Basis` of §10.8. In all of them c is the cost of sending a node relative to that of generating it, p the number of workers and the seed that of the random table. The following table gives the names that `make_distribution()` accepts (`AutoDistribution.h:1417-1506`), the construction of each and its default parameter; a parameter other than the default is written after a colon (e.g., `OZ:0.1` or `AHDA:100`), and an unknown name throws `std::invalid_argument`.

Name	Construction	Notation of [5]	Default
`Z`	`FeatureAbstraction::identity()`	HDA*[Z]	
`GAZ`	`greedy_abstraction()`	HDA*[Z, Afeature/DTGgreedy]	
`GRAZ`	`sparsest_cut_abstraction()`, `sparsity`	HDA*[Z, Afeature/DTGsparsity]	
`GRAZS`	the same, `sampled_sparsity`	(ours)	
`GRAZE`	the same, `efficiency` with c	(ours)	
`FAZ`	`fluency_abstraction()`	HDA*[Z, Afeature/DTGfluency]	0.3 of the groups
`OZ`	`operator_table()`	HDA*[Zoperator]	0.2 of the operators
`OZP`	`operator_table()`, `as_paper`	HDA*[Zoperator]	all the operators
`DAHDA`	`dynamic_state_abstraction()`	HDA*[Z, Astate/SDDdynamic]	0.3 of the features
`AHDA`	`state_abstraction()`	HDA*[Z, Astate/SDD]	1000 abstract states
`KZ`	`kernel_selection_table()` for p	(ours)	from Z
`KZ:GRAZ`, `KZ:GRAZS`, `KZ:GRAZE`, `KZ:DAHDA`	the same	(ours)	from that projection

The search itself is unchanged: one builds the model and the distribution, and gives the distribution to HDA*, e.g., `HDAStar< D , TableZobristDistribution< D > >( d , auto_distribution( FeatureModel< D >( d ) , p ) , options )`; the model need only live while the distribution is built, since the distribution keeps a reference to the `Domain` and a copy of its table, not the model. The benchmark tool takes the same names in its option `--dist` (§10.9).

10.2 The sample and the transition structure

The estimates of Jinnai and Fukunaga are computed on the *workload graph* of section 5.1 of [5], whose nodes are the states with $f < f^*$ and the goals and whose edges are the edges of the state space among them: these are the

states that every parallel search must expand to prove a path optimal, whatever its distribution (cf. Theorem 2.14). The paper computes it by running A* to the end, which makes the method a model rather than a practical construction (footnote 11 there), and builds the distributions of its section 6 from the DTGs of the task instead. The library does both on a *sample*: A* is run from the initial state up to a bounded number L of expansions (32768 by default) or to the first goal, and the sample both stands for the workload graph, on which the estimates are computed, and gives the transition structure (the groups, their DTGs with weights, the operators and their frequencies) from which the distributions are built. Algorithm 10.1 is `FeatureModel::sample()` (`AutoDistribution.h:457-671`).

Algorithm 10.1 (The sample of a `FeatureModel`).

```
 1  function SAMPLE(L)                          ▷ E the expanded states, in order
 2     g(s₀) ← 0; push (h(s₀), 0, s₀) into OPEN; E ← ()
 3     while OPEN ≠ ∅ and |E| < L do
 4        pop the entry (f, −g', s) of least f, of largest g' on ties
 5        if s is closed or g' ≠ g(s) then continue
 6        close s; append s to E
 7        if s is a goal then break
 8        add 1 to the mass of each feature of F(s)
 9        for each successor t of s, of cost c, do
10           if t is new or g(s) + c < g(t) then
11              g(t) ← g(s) + c; reopen t; push (g(t) + h(t), −g(t), t)
12           add the edge (s, t) to the sample
13           D ← F(s) \ F(t); A ← F(t) \ F(s)
14           if D ∪ A = ∅ then continue
15           add 1 to the moves and to the weight of the operator D ∪ A
16           for each a ∈ D and b ∈ A do
17              if the Domain has groups and a, b are in two groups then continue
18              if the Domain has no groups then join a and b
19              add 1 to the sampled weight of the edge {a, b}
20              if D ∪ A is a new operator and the Domain lists no transitions then
21                 add 1 to the static weight of the edge {a, b}
22     for each ground transition (a, b) of the Domain, if it lists them, do
23        if the Domain has no groups then join a and b
24        add 1 to the static weight of the edge {a, b}
25     the groups ← those of the Domain, or the classes of the joins
```

The rest of the construction orders the states (the expanded ones first), assigns the groups to the features that a move or a transition changes, collects the members of each DTG, sorts the operators by decreasing weight and computes the fluency of each group, and stores for each state its features and, for each group, the exclusive or of random words of its features in that group (`words()`, used in §10.6). We name what it computes.

Definition 10.3 (The transition structure of a sample). Let E be the expanded states of the sample and M its *moves*, i.e., the edges (s, t) with $s \in E$ and $F(s) \neq F(t)$. (i) The *operator* of a move is its change set $F(s) \triangle F(t)$, and the weight of an operator the number of its moves; `operators()` lists them by decreasing weight, the ties in the lexicographic order of their sorted features. (ii) A feature has a group (`group()`) if some move or some ground transition changes it; the groups are those of the `Domain` if it is a `FeatureGroupDomain`, renumbered, and otherwise the connected components of the graph that joins every deleted feature of a move (and of a transition) to every added one; the features that nothing changes have the group `no_group`. (iii) The DTG of a group has as nodes its *members* (`members()`), i.e., its features that hold in a state of the sample (`seen()`) or that a ground transition changes, and an edge $\{a, b\}$ for each pair deleted and added by a move within the group; an edge has the *sampled weight* (`arcs()`) of the moves that cross it in either direction and the *static weight* (`static_arcs()`) of the ground operators that realize it, those of the `Domain` if it is a `FeatureTransitionDomain` and otherwise the distinct operators of the sample that realize it. (iv) The *mass* of a feature (`mass()`) is the number of states of E in which it holds, the goal that ends the sample excluded. (v) The *fluency* of a group is the fraction of the moves whose operator changes a feature of the group.

The fluency of section 4.2.2 of [5] counts the ground actions that change the variable over all the ground actions; (v) counts instead the moves of the sample, i.e., the ground actions weighed by how often the search applies them near the root. The following lemma states what the sample holds, which is the hypothesis under which its estimates stand for those of the workload graph.

Lemma 10.4 (What the sample holds). Let h be consistent, a goal be reachable, and the sample reach a goal

before L expansions. Then E holds every state u with $g^*(u) + h(u) < C^*$, each once, closed with $g(u) = g^*(u)$, followed by some states with $g^*(u) + h(u) = C^*$ and by the goal; hence E is the node set of the workload graph of [5] together with some states of the last layer $f = C^*$. If the sample stops at L expansions instead, E holds the L states of least f (of largest g on ties) that A* expands first.

Proof. Lines 3 to 11 are A* (Algorithm 2.2) with an OPEN list of entries $(f, -g, s)$ and stale entries skipped, which pops the entries of least f and breaks the ties by larger g, and which adds to A* only the bookkeeping of lines 8 and 12 to 21. By Theorem 2.13, each state is closed once, with its optimal g, and the keys of the selected entries do not decrease; by Theorem 2.14(ii), every state with $g^*(u) + h(u) < C^*$ is expanded before a goal is selected, and by (i) of the same theorem no state with $g^*(u) + h(u) > C^*$ is. The goal is appended to E at line 6 before the test of line 7. The second claim is the order of selection of A*. □

The goal is in E, and therefore in the load of its owner (§10.3), but it adds no edge and no mass. With an inconsistent heuristic a state may be reopened at line 11 and expanded again; it is then appended to E once per expansion (the loop at `AutoDistribution.h:563` keeps every copy), so that its load counts twice, which is also what HDA* pays for it. With the default bound, the sample of an instance whose layer $f < C^*$ holds more than 32768 states is a prefix of it, the states nearest to the root in the order of A*, and the estimates then extrapolate from that prefix; this is the reason for the estimate among $2p$ workers of KZHDA* (§10.8).

Lemma 10.5 (The inferred groups). Let the `Domain` not be a `FeatureGroupDomain`. (i) Two features changed by a move or a transition are in the same group if and only if they are connected in the graph J that joins each deleted feature of a move of M (or of a ground transition) to each added one. (ii) If there is a partition of the features such that each move and each transition deletes and adds features of one class only, every inferred group lies within a class. (iii) On a sliding-tile puzzle that gives its transitions and no groups, the inferred groups are the tiles.

Proof. (i) Line 18 and line 23 join exactly the pairs of J in a union-find structure, whose classes after the last join are the connected components of J, and line 25 numbers the classes that contain a changed feature. (ii) Every edge of J joins two features of one class, and so does every path. (iii) The features of a tile t are the pairs (t, c); a move of t, or a transition of t, deletes (t, c) and adds (t, b), so that by (ii) a group lies within a tile; the transitions join (t, c) to (t, b) for every pair of adjacent cells, and the grid graph of the board is connected, so that the features of t form one component. □

Part (iii) is what the test `distributions` checks on the 15-puzzle without its groups (the type `UngroupedFifteen` of `test/distributions.cpp`, which deletes `num_feature_groups()`), against the groups of the `Domain`. Part (ii) also shows the limit of the inference: a move that changes two variables, as many operators of a planning task do, joins their features into one group, whose DTG is then not the transition graph of a single variable; a planning task is therefore given with its groups.

10.3 The estimates

The estimates of a distribution are computed on the sample as on a workload graph, following equations (3) to (5) of [5]. Algorithm 10.2 is `FeatureGraph::estimate_owners()` (`AutoDistribution.h:321-352`), which `FeatureModel::estimate()` calls with the owners that a distribution gives to the states of the sample, and `FeatureGraph::estimate_table()` with those that the table r gives, computed from the stored features of the states without the `Domain`.

Algorithm 10.2 (The estimates of a distribution on a sample).

```
1  function ESTIMATE(own, p, c)    ▷ own(s) the owner of each state of the sample
2     if p = 0 or E = ∅ then return (co 0, lb 1, so 0, eff 1, sparsity 0)
3     load(i) ← |{s ∈ E : own(s) = i}| for each worker i
4     co ← |{(s, t) edge of the sample : own(s) ≠ own(t)}| / (the edges)
5     lb ← max_i load(i) / (|E| / p); so ← lb − 1
6     eff ← 1 / ((1 + c · co)(1 + so))
7     sparsity ← Π_i (load(i) / (|E| / p)) / co        ▷ ∞ if co = 0
8     return (co, lb, so, eff, sparsity)
```

Definition 10.6 (The estimates). For a distribution and p workers, let $S_i \subseteq E$ be the expanded states of the sample that the worker i owns, and c the cost of sending a node relative to that of generating it. The estimated communication overhead co is the fraction of the edges (s, t) of the sample ($s \in E$, t any successor generated, the moves back to a parent included) whose ends have different owners; the estimated load balance is

$\mathrm{lb} = \max_i |S_i|/(|E|/p)$; the estimated search overhead is $\mathrm{so} = \mathrm{lb} - 1$; the estimated efficiency is

$$\mathrm{eff} = \frac{1}{(1 + c\,\mathrm{co})(1 + \mathrm{so})} = \frac{1}{(1 + c\,\mathrm{co})\,\mathrm{lb}},$$

and the scaled sparsity is $\prod_i (p|S_i|/|E|)/\mathrm{co}$.

The estimated CO is the ratio of sent to generated nodes that HDA* measures (Definition 8.2) on the expansions of the sample, whereas equation (3) of [5] counts the undirected edges between the states of the workload graph only; the two differ by the edges to states of $f \geq C^*$, and HDA* also counts as generated (and sends) the successors that it then drops as duplicates. The estimated search overhead rests on the following model, which is that of section 5.1 of [5].

Proposition 10.7 (The search overhead of an imbalance). Let all the states of $S = \bigcup_i S_i$ have to be expanded, every worker expand at the same rate and keep expanding until the last state of S is expanded, and the states of S be expanded by their owners only. Then the workers expand $|S|(\mathrm{LB} - 1)$ states outside S, with $\mathrm{LB} = \max_i |S_i|/(|S|/p)$; hence the search overhead relative to the expansions $|S|$ of A* is $\mathrm{LB} - 1$, and the search efficiency $1/(1 + \mathrm{SO})$ is $1/\mathrm{LB}$.

Proof. The search ends when the most loaded worker has expanded its $\max_j |S_j|$ states, since all of them must be expanded and each worker expands one state per unit of time; by then the worker i has expanded $\max_j |S_j|$ states, of which $|S_i|$ are in S. The states outside S are therefore

$$\sum_{i=0}^{p-1} \left(\max_j |S_j| - |S_i| \right) = p \max_j |S_j| - |S| = |S| \left(\frac{\max_j |S_j|}{|S|/p} - 1 \right) = |S|(\mathrm{LB} - 1),$$

and the parallel search expands $|S| + |S|(\mathrm{LB} - 1) = |S|\,\mathrm{LB}$ states, i.e., $\mathrm{SO} = \mathrm{LB} - 1$ by Definition 8.1. □

Equation (4) of [5] writes the same sum as $p(\mathrm{LB} - 1)$, which is its value in units of the mean $|S|/p$ rather than of $|S|$, and equation (5) then adds it to 1 as a ratio to the expansions of A*; taken literally, that formula weighs an imbalance p times more than the model gives. The library uses $\mathrm{LB} - 1$, so that the estimated efficiency is the product of the communication efficiency $1/(1 + c\,\mathrm{CO})$ and of $1/\mathrm{LB}$; we remark that the two readings may rank two distributions differently, and that the correlation measured in section 5.2.1 of [5] (the actual efficiency about 0.86 times the estimate, on 48 processes, Figure 13 there) was obtained with the formula of the paper. The scaled sparsity is equation (6) of [5], whose sizes are fractions of $|S|$, multiplied by p^p and with the cut weight normalized by the edges, so that a balanced partition has scaled sparsity $1/\mathrm{co}$; `auto_distribution()` does not read it, and the benchmark prints it for comparison.

Proposition 10.8 (Properties of the estimates). (i) $\mathrm{lb} \geq 1$, $\mathrm{so} \geq 0$, $0 \leq \mathrm{co} \leq 1$ and $0 < \mathrm{eff} \leq 1$ for $c \geq 0$; with $p = 1$, $\mathrm{co} = 0$ and $\mathrm{lb} = 1$. (ii) eff is decreasing in co (for $c > 0$) and in lb. (iii) For any table r of `num_features()` entries, `estimate_table( r , p , c )` equals `estimate( TableZobristDistribution< D >( d , r ) , p , c )`. (iv) $\mathrm{lb} \leq p$, with equality if and only if one worker owns all of E.

Proof. (i) The largest load is not smaller than the mean, and the fraction of the edges cut lies in $[0, 1]$; with $p = 1$ every state has the owner 0. (ii) Both factors of the denominator are positive and increasing. (iii) `estimate_table()` computes for each state of the sample the exclusive or of the entries of r at its stored features, which is `hash()` of the table distribution, and takes `key_owner()` of its high 32 bits, which is `owner()`; then both call the same function on the same owners. (iv) $\max_i |S_i| \leq |E|$, with equality if and only if one S_i is E. □

The test `distributions` checks (i) on Zobrist hashing for $p = 1$ and $p = 8$ on an instance of the 15-puzzle, where the estimated co exceeds 0.8 and lb stays below 1.2 for $p = 8$ (cf. Corollary 9.12, which predicts a fraction of about 7/8), and (iii) on KZHDA*. Figure 10.1 shows the estimate on a small sample.

The estimates have measured counterparts in the `SearchStats` of a run (§8.2): `communication_overhead()` is the CO of Definition 8.2; `load_balance()` the LB of Definition 8.3 on the expansions of each worker; `storage_balance()` (`Solution.h:121`) the same ratio on the nodes stored by each worker at the end (`stored_by`, the states that it owns and has received), which is the LB of equation (3) of [5], measured on all the states generated rather than on those of $f < C^*$; and `search_overhead( sequential )` (`Solution.h:134`) the SO of Definition 8.1, expanded/sequential − 1, given the expansions of a sequential run (and 0 if they are 0). The storage balance returns 1, as `load_balance()` does, when the vector is empty or all 0 (`src/Solution.cpp:62-79`).

We remark on what the estimates do not see. They ignore the states of $f \geq C^*$ that a parallel search expands because of the order of the expansions (the band and burst effects of section 3.1 of [5]), the re-expansions, the latency of the messages and the cost of the hash itself; and the model of Proposition 10.7 assumes that the

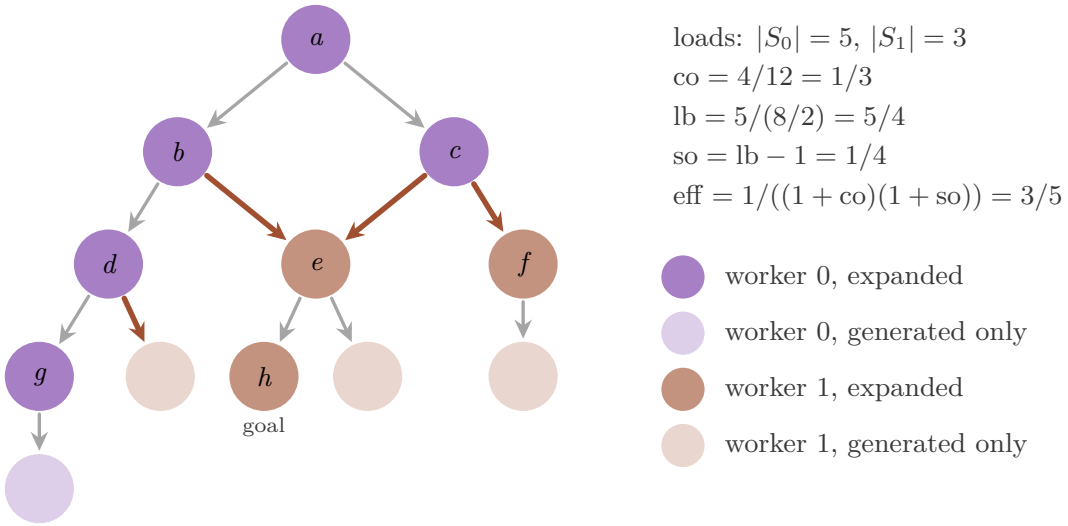


Figure 10.1. The estimate of a distribution among $p = 2$ workers on a sample of 8 expanded states (the dark discs, the two colours being the two owners), stopped at the goal h, and of the 4 states generated but not expanded (the light discs). Of the 12 edges, the 4 in the rust colour join two owners, so that co $= 1/3$; one worker owns 5 expanded states against 3, so that lb $= 5/4$, so $= 1/4$ and, with $c = 1$, eff $= 1/((4/3)(5/4)) = 3/5$. The states generated only count in the edges, not in the loads.

imbalance of the states of $f < C^*$ is the imbalance of the work, which holds only if the workers expand the other states at the same rate. A sample that is a prefix of the layer $f < C^*$ measures the balance of the states nearest to the root, which may fall into fewer classes than those of the whole layer (cf. Corollary 10.13). The estimates are therefore a means of ranking distributions on an instance, not a prediction of its speedup.

10.4 The key as a linear map

All the distributions of this chapter are Zobrist hashing on some table, and Zobrist hashing is linear over the field GF(2) of two elements; this gives a single statement of what each of them keeps on the worker of the parent, and of what it costs in balance. We identify a set of features with its indicator vector in $GF(2)^n$, so that $x(s) \in GF(2)^n$ is the vector of $F(s)$, the exclusive or of sets is the sum of vectors, and e_x is the vector of the single feature x.

Definition 10.9 (The key matrix and the change vector). The *key matrix* of a table R is the $32 \times n$ matrix H over GF(2) whose column x holds the high 32 bits of $R[x]$. The *change vector* of a move from s to t is $\delta = x(s) + x(t)$, i.e., the vector of $F(s) \triangle F(t)$. For a subspace V of $GF(2)^n$, a table *vanishes on* V if $Hv = 0$ for every $v \in V$.

Theorem 10.10 (The moves that keep the key). Let H be the key matrix of a table. (i) The key of s is $k(s) = Hx(s)$, and a move with change vector δ gives $k(t) = k(s) + H\delta$; hence it keeps the key if and only if $\delta \in \ker H$, and a move that keeps the key keeps the owner for every p. (ii) For $p = 2^b$, the owner of a key is its first b bits, and a move keeps the owner if and only if the first b rows of H vanish on δ.

Proof. (i) The key is the high 32 bits of $\bigoplus_{x \in F(s)} R[x]$, and taking the high bits commutes with $\oplus$; hence $k(s) = \sum_{x \in F(s)} He_x = Hx(s)$, and by linearity $k(t) = H(x(s) + \delta) = k(s) + H\delta$. The owner is a function of the key. (ii) `key_owner( k , 2^b )` is $\lfloor k\, 2^b / 2^{32} \rfloor$, i.e., k shifted right by $32 - b$ bits, which is its first b bits (the high ones). Two keys have the same first b bits if and only if their sum vanishes there, and $k(s) + k(t) = H\delta$. □

For other values of p the owners of two distinct keys may coincide, and a move outside $\ker H$ keeps the owner with probability about $1/p$ (Corollary 10.13 below). Each distribution of this chapter is the choice of a subspace V on which its table vanishes, together with a table drawn among those that vanish on V; the larger V, the more moves keep their owner, but also the fewer distinct keys the states take. The following lemma, in which we take the free entries of the table independent and uniform (a model of the table drawn from the seed, as in Proposition 9.11), states what a table drawn uniformly among those that vanish on V does.

Lemma 10.11 (Uniform tables vanishing on a subspace). Let H be uniform among the $32 \times n$ matrices that vanish on a subspace V, and let $V^\perp = \{r \in GF(2)^n : r \cdot v = 0 \text{ for all } v \in V\}$ be its annihilator. (i) The rows of

H are independent and uniform on $V^\perp$. (ii) For $\delta \in V$, $H\delta = 0$; for $\delta \notin V$, $H\delta$ is uniform on $GF(2)^{32}$. (iii) For δ, δ' with δ, δ' and $\delta + \delta'$ outside V, the pair $(H\delta, H\delta')$ is uniform on $GF(2)^{64}$.

Proof. (i) $Hv = 0$ means that each row r of H has $r \cdot v = 0$; hence the matrices that vanish on V are the matrices whose 32 rows lie in $V^\perp$, a product of 32 copies of $V^\perp$, and the uniform measure on a product is the product of the uniform measures. (ii) The first claim is the definition. If $\delta \notin V$, then, since $(V^\perp)^\perp = V$ in a finite-dimensional space, some $r \in V^\perp$ has $r \cdot \delta = 1$; the map $r \mapsto r \cdot \delta$ is then a nonzero linear form on $V^\perp$, which takes each value on half of $V^\perp$, so that each bit of $H\delta$ is uniform, and the bits are independent by (i). (iii) The map $r \mapsto (r \cdot \delta, r \cdot \delta')$ from $V^\perp$ to $GF(2)^2$ is linear; if it were not onto, its image would be annihilated by some nonzero (α, β), i.e., $r \cdot (\alpha\delta + \beta\delta') = 0$ for all $r \in V^\perp$, and $\alpha\delta + \beta\delta' \in V$, which is one of δ, δ', $\delta + \delta'$. Hence it is onto, each pair of bits is uniform, and the 32 pairs are independent by (i). □

Corollary 10.12 (The subspaces of Z and AZ). (i) Zobrist hashing is the uniform table vanishing on $V = \{0\}$. (ii) For a projection A, let

$$V_A = \operatorname{span}\left(\{e_x + e_y : A(x) = A(y)\} \cup \{e_x : x \text{ ignored}\}\right);$$

then $\delta \in V_A$ if and only if, for every abstract feature a, an even number of the features of δ have $A(x) = a$; and the table of `FeatureAbstraction::table()` is uniform among the tables that vanish on V_A. Hence a move keeps the key with probability 1 if every abstract feature appears an even number of times among its changed features, and with probability 2^{-32} otherwise.

Proof. (i) The entries of Zobrist hashing are independent and uniform, and every matrix vanishes on $\{0\}$. (ii) A generator of V_A changes by 0 or 2 the number of its features in each abstract feature, or adds an ignored feature; hence the counts of a sum of generators are even. Conversely, a vector with even counts is, in each abstract feature, a set of an even number of features $x_1, \ldots, x_{2m}$, the sum of the pairs $e_{x_{2j-1}} + e_{x_{2j}}$, plus a set of ignored features, a sum of the e_x. A matrix vanishes on V_A if and only if its columns are equal within each abstract feature and 0 at the ignored features; the table of the projection has the columns given by $R'[A(x)]$ and 0, with R' uniform, and the map from R' to these matrices is a bijection, so that the table is uniform among them. The last claim is Lemma 10.11(ii). □

This is the form of Corollary 9.13 for any projection. Figure 10.2 shows the key matrices of Z, of AZ and of a kernel table (§10.8) on six features.

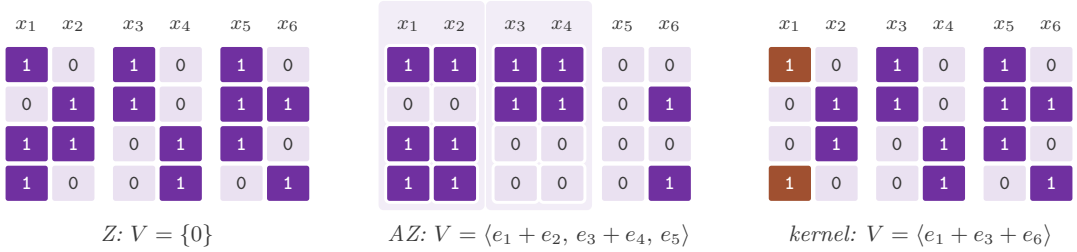


Figure 10.2. The key matrix H of three tables on six features, with a key of 4 bits instead of 32 (a column is the high bits of an entry, a dark cell a bit 1). Left: Zobrist hashing, $V = \{0\}$. Middle: Abstract Zobrist hashing with $A(x_1) = A(x_2)$, $A(x_3) = A(x_4)$ and x_5 ignored, whose equal columns are framed together and whose column x_5 is 0, so that $V = \operatorname{span}(e_1 + e_2, e_3 + e_4, e_5)$. Right: the kernel table of $V = \operatorname{span}(\delta)$ for the operator $\delta = e_1 + e_3 + e_6$, whose pivot column x_1 (in the triad colour) is the sum of the columns x_3 and x_6, so that $H\delta = 0$ while the other columns are those of Zobrist hashing.

The price of a larger V is in the balance. The key of a state depends only on its class $x(s) + V$, and the classes of the states of the sample are the units that the hash distributes.

Corollary 10.13 (The classes of a subspace). Let the table be uniform among those vanishing on V, and partition the expanded states E of the sample into the classes $C_1, \ldots, C_m$ of the relation $x(s) + x(t) \in V$. (i) All the states of a class have one key, and therefore one owner. (ii) Two states of distinct classes, neither of which has $x(s) \in V$, have independent uniform keys, and therefore the same owner with probability in $[1/p, 1/p + 2^{-32}]$. (iii) Whatever the table, $\text{lb} \geq p \max_j |C_j|/|E|$.

Proof. (i) By Theorem 10.10, $k(s) + k(t) = H(x(s) + x(t)) = 0$. (ii) Lemma 10.11(iii) with $\delta = x(s)$ and $\delta' = x(t)$, whose sum is outside V since the classes differ; the bound on the probability is then the proof of Corollary 9.12. (iii) The worker that owns the largest class owns at least $\max_j |C_j|$ states. □

The corollary is the trade-off of the chapter in its exact form: the communication falls with the moves whose change vectors enter V, and the balance is bounded by the largest class, whose size grows with V; in the limit $V = GF(2)^n$ every state has the key 0 and $\mathrm{lb} = p$. Of course, a distribution need not be uniform among the tables that vanish on its V to keep the moves of V; but all the constructions of the library are, as the following sections show, so that each of them is described by its subspace alone.

10.5 Cutting the transition graphs

GAZHDA*, GRAZHDA* and FAZHDA* cut the DTG of each group in two sides S_1 and S_2, which become two abstract features of the group (section 4.2.1 of [5] and section 6 there); for G groups the states are partitioned into at most 2^G classes, the patterns of the sides of their features, which Abstract Zobrist hashing maps to the workers. The projection is built by `detail::cut_abstraction()` (`AutoDistribution.h:700-727`): the members of a cut group get the abstract feature of their side, two new abstract features per group; the features of an ignored group are ignored; and every other feature (those of a group of fewer than two members, those with no group, and those of a group that hold in no state of the sample and that no transition changes) is its own abstract feature.

Proposition 10.14 (The moves that a cut keeps). Let the projection A be a cut abstraction and a move change, in each group, either nothing or one member x_g into another member y_g (as a move of a `FeatureGroupDomain` with one feature per group does). Then the move keeps the key with probability 1 if, in each changed group that is cut and not ignored, x_g and y_g lie on the same side, and with probability 2^{-32} if some such group has them on two sides.

Proof. The changed features of a group g are x_g and y_g. If g is ignored, both are; if g is cut, $A(x_g) = A(y_g)$ exactly when they are on the same side, and the two abstract features of g appear in no other group. Hence every abstract feature appears an even number of times among the changed features if and only if no cut group has its two features on two sides, and the claim is Corollary 10.12(ii). □

Hence the CO of a cut abstraction on the sample is, up to the factor of about $1 - 1/p$ of Corollary 10.13(ii), the fraction of the moves that cross the cut of some group, i.e., the sampled weight of the edges cut in all the DTGs over the moves, if the moves change one group each; this is why the sampled weights measure the communication of a cut.

GAZHDA*: the greedy halves

`greedy_abstraction()` (`AutoDistribution.h:943`) cuts each DTG by Algorithm 4 of [5], which `detail::greedy_cut()` implements with the degrees and the links of the static DTG (Algorithm 10.3); the degree of a node is the number of its neighbours among the members, and its link the number of its neighbours already in S_1.

Algorithm 10.3 (GAZHDA*: the greedy halves of a DTG).

```
 1  function GREEDY-CUT(X)                    ▷ X the n members of the group
 2     if n < 2 then return (∅, X)
 3     deg(x) ← the static neighbours of x in X, for each x ∈ X
 4     link(x) ← 0 for each x ∈ X
 5     x₀ ← the node of least deg, of least index on ties
 6     S₁ ← {x₀}; link(y) ← link(y) + 1 for each static neighbour y of x₀
 7     while 2 |S₁| < n do
 8        x ← the node of X \ S₁ of largest link, of least index on ties
 9        S₁ ← S₁ ∪ {x}; link(y) ← link(y) + 1 for each static neighbour y of x
10     return (S₁, X \ S₁)
```

Lemma 10.15 (The greedy halves are balanced). For $n \geq 2$, Algorithm 10.3 returns $|S_1| = \lceil n/2 \rceil$ and $|S_2| = \lfloor n/2 \rfloor$, i.e., $|S_2| \leq |S_1| \leq |S_2| + 1$, whatever the edges, in $O(n^2 + |E_X|)$ time for the static edges E_X of the DTG.

Proof. Each iteration of the loop adds one node, and the loop runs while $2|S_1| < n$, i.e., until $|S_1| \geq n/2$, the least such integer being $\lceil n/2 \rceil$. Each iteration scans the n nodes, and each node updates the links of its neighbours once, when it enters S_1. □

The cut is balanced in the number of members, not in the mass of the sample, and the paper notes (section 6.2.1 of [5]) that the balance is paid with the edges it cuts: on the logistics DTG of Figure 16 there, it cuts 21 edges

where the sparsest cut cuts one. On the grid graph of a tile of the 15-puzzle the greedy side grows from the corner cell 0 along the rows and stops after the first two rows, which is the projection of a tile onto the half of the board of §9.4 (Exercise 10.4).

GRAZHDA*: the cut of largest sparsity or efficiency

`sparsest_cut_abstraction()` (`AutoDistribution.h:987`) cuts each DTG at the two sides of best objective, by `detail::best_cut()`. For a cut with side masses $m_1 = q(S_1)$ and $m_2 = q(S_2)$ (of total M) and cut weight ω (of total W), the three objectives of `CutObjective` are:

- `sparsity`: $m_1 m_2/\omega$ with the static weights, a node weighing 1 and an edge the ground operators that realize it, which is equation (6) of [5] for two parts (with the sizes as fractions and the weights normalized, a constant factor that changes no maximizer);
- `sampled_sparsity`: the same ratio with the weights of the sample, a node weighing its mass and an edge the moves that cross it, so that a cut weighs the part of the state space that the search visits rather than the domain of the variable;
- `efficiency`: $1/((1+c\,\omega/W)\max(m_1, m_2)/(M/2))$ with the weights of the sample, the estimated efficiency of Definition 10.6 for this DTG alone and two workers.

A cut of weight 0 counts as a cut of weight $10^{-9}W$, so that among the cuts that cut nothing the most balanced wins; if no member has a mass (a group seen only in generated states), the nodes weigh 1. The paper proposes the sparsity, finds the objective that minimizes CO + LB comparable to it, and leaves the others open (section 6.2 there); the two variants weighed by the sample are ours. The DTGs of up to 20 nodes are cut optimally by Algorithm 10.4, in which the node 0 stays on side 0 and the other nodes visit all their 2^{n-1} assignments in the order of a Gray code.

Algorithm 10.4 (GRAZHDA*: the best cut of a DTG of at most 20 nodes).

```
 1  function BEST-CUT(X, q, w, value)   ▷ nodes 0, ..., n − 1; q, w their weights
 2     side(x) ← 0 for each x; m₁ ← 0; ω ← 0
 3     best ← side; v* ← value(0, 0)
 4     for i ← 1 to 2^(n−1) − 1 do
 5        k ← 1 + (the number of the trailing zero bits of i)
 6        for each neighbour u of k do
 7           ω ← ω + w(k, u) if side(u) = side(k), ω − w(k, u) otherwise
 8        side(k) ← 1 − side(k)
 9        m₁ ← m₁ + q(k) if side(k) = 1, m₁ − q(k) otherwise
10        if value(m₁, ω) > v* then best ← side; v* ← value(m₁, ω)
11     return best
```

Lemma 10.16 (The enumeration is exact). For a DTG of $n \geq 2$ nodes, Algorithm 10.4 evaluates every cut with node 0 on side 0 exactly once, the trivial cut (all the nodes on side 0) included, with its exact mass m_1 and weight ω, and returns the first cut of largest value in the order of the enumeration; it takes $O(2^{n-1}(\Delta + n))$ time for the largest degree Δ.

Proof. Let $g(i) = i \oplus \lfloor i/2 \rfloor$, the reflected Gray code of i, read as the set of the nodes $j+1$ for the bits j of $g(i)$, which are on side 1. Since $g(i) \oplus g(i-1) = (i \oplus (i-1)) \oplus \lfloor (i \oplus (i-1))/2 \rfloor$, and $i \oplus (i-1) = 2^{t+1} - 1$ for the number t of the trailing zeros of i, the two codes differ in the bit t only; hence the assignment after step i is $g(i)$, by induction from $g(0) = 0$. The map g is a bijection of $\{0, \ldots, 2^{n-1} - 1\}$ (its inverse is the prefix exclusive or of the bits), so that the steps visit every nonzero assignment once, and line 3 the zero one. When k changes side, an edge $\{k, u\}$ becomes cut if u was on the side of k and uncut otherwise, and no other edge changes, which is line 7; line 9 is the change of the mass of side 1. The strict comparison of line 10 keeps the first maximum. A step costs the degree of k, and a new best cut a copy of n sides. □

The header states the cost as $O(2^n \bar{d})$ for the mean degree $\bar{d}$; since the node k changes side 2^{n-1-k} times, the cost of line 7 is $\sum_k 2^{n-1-k} \deg(k)$, which is $O(2^n \bar{d})$ on the regular grid graphs of the puzzles but is governed in general by the degrees of the low-numbered nodes, hence the bound with Δ. The paper cuts DTGs of fewer than 10 nodes by a depth-first branch-and-bound (section 6.2.3 of [5]) and suggests the algorithm of Fiduccia and Mattheyses [142] for the larger ones, which is what the library does above 20 nodes (Algorithm 10.5), from the greedy halves of Algorithm 10.3 and from the halves grown in breadth-first order from up to 16 nodes, the j-th from the node $\lfloor jn/\min(n, 16) \rfloor$.

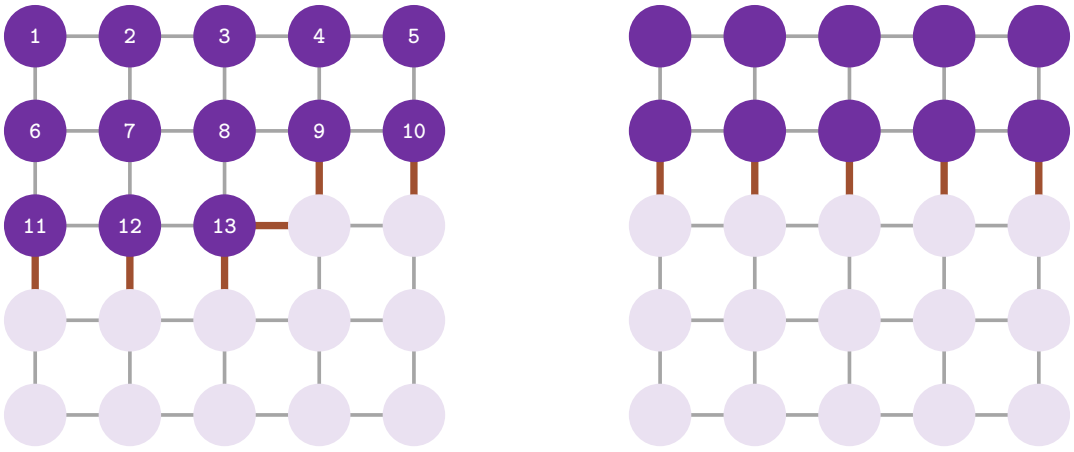
Algorithm 10.5 (GRAZHDA*: the passes of Fiduccia and Mattheyses from a cut s).

```
 1  function FM-PASSES(s)
 2     v ← value(s)
 3     repeat
 4        unlock every node; moved ← (); b ← v; l ← 0
 5        for step ← 1 to n do
 6           k ← the unlocked node whose move to the other side gives the largest value
 7           move k to the other side; lock k; append k to moved
 8           if value(s) > b · (1 + 10^−12) then b ← value(s); l ← |moved|
 9        move back the nodes moved after the l-th one
10        improved ← l > 0; if improved then v ← b
11     until not improved
12     return s
```

Proposition 10.17 (The passes). Algorithm 10.5 terminates, returns a cut whose value is at least that of s, and costs $O(n(n+|E_X|))$ time per pass; the cut that `best_cut()` returns is at least as good as each of its starts and as the trivial cut.

Proof. A pass ends with the cut after the best prefix of its moves (line 9), whose value b is at least the value v at the start of the pass, since $\ell = 0$ restores the start. A pass is repeated only if $b > v(1+10^{-12})$, so that the values at the starts of the passes strictly increase; as there are finitely many cuts, finitely many passes are made. A step evaluates every unlocked node with the change of the cut weight at that node, i.e., $O(n+|E_X|)$ for the whole step, and a pass has n steps. Finally, `best_cut()` starts from the trivial cut and keeps the result of a start only if its value is larger. □

No bound better than the finiteness of the cuts is known to us on the number of passes, and the result is a local optimum, not the optimum. The test `distributions` checks the enumeration against an independent enumeration of the 2^{15} cuts of three DTGs of the 15-puzzle with the weights of the sample, and the passes on the 24-puzzle, whose DTGs have 25 nodes, by the cut of the paper: with the static weights GRAZHDA* cuts the 15-puzzle in two halves of two rows and the 24-puzzle in two rows against three, the abstract features of Figure 19 of [5] (Exercise 10.6). Figure 10.3 compares the two cuts on a tile of the 24-puzzle.



GAZHDA: 6 edges cut* *GRAZHDA*: 5 edges cut*

Figure 10.3. The DTG of a tile of the 24-puzzle, the grid graph of its 25 cells, each edge weighing the 2 ground moves that cross it, cut by GAZHDA* (left) and by GRAZHDA* with the sparsity (right); the side S_1 is in the primary colour and the cut edges in the rust colour. GAZHDA* grows S_1 from the corner cell in the order of the numbers, up to 13 cells, and cuts 6 edges, of sparsity $13 \cdot 12/12 = 13$; GRAZHDA* cuts the 5 edges between the second and the third row, of sparsity $10 \cdot 15/10 = 15$, the largest of all the cuts of this graph.

FAZHDA*: the fluent groups ignored

A group that nearly every move changes makes nearly every move change its abstract feature, whatever the cut of the other groups (the hand of the blocks world, Figure 12 of [5]); `fluency_abstraction()` (`AutoDistributi on.h:1010`) ignores the $\lfloor \phi G \rfloor$ groups of largest fluency, for the fraction ϕ (0.3 by default, the best of 10% to

70% on the planning domains of section 5.2.1 of [5]), the ties going to the group of least index, and cuts the others by Algorithm 10.3. By Corollary 10.12, the subspace of FAZHDA* contains e_x for every feature of an ignored group, so that a move that changes only ignored groups, or crosses no cut elsewhere, keeps its owner; by Corollary 10.13 the classes grow by the combinations of the members of the ignored groups that the sample holds. The paper notes (section 6.1) that the fluency counts the changes of the features, whereas the owner changes with the abstract features: a fluent group whose moves stay on one side of its cut costs nothing, and ignoring it only coarsens the classes. On the 15-puzzle, with 15 groups and $\phi = 0.3$, the 4 most fluent tiles are ignored, as the test `distributions` checks.

10.6 State abstractions: AHDA* and DAHDA*

AHDA* of Burns et al. [3] keeps the states with the same abstract state on one worker, the abstract state being, on a planning task, the values of a set P of variables chosen by the greedy abstraction of Zhou and Hansen [139] for structured duplicate detection. That algorithm adds one group at a time, the one that minimizes the locality

$$\delta(P) = \max_{u \in S_P} \frac{|\text{Successors}(u)|}{|S_P|}$$

of the abstract graph S_P of P (equation (1) of [139], self loops omitted), until the abstract graph exceeds a bound $N_{\max}$. The library computes δ on the abstract graph of the sample: the abstract state of a state for P is the exclusive or of the words of its features in the groups of P (`words()`, Definition 10.3), and an arc joins the abstract states of the two ends of an edge of the sample that changes it. Algorithm 10.6 is `detail::zhou_hansen_groups()` (`AutoDistribution.h:1049-1106`).

Algorithm 10.6 (The greedy groups of Zhou and Hansen on the sample, with a stopping rule).

```
 1  function GREEDY-GROUPS(stop)
 2     P ← ∅; F ← 0; key(k) ← 0 for each state k of the sample
 3     loop
 4        for each group g ∉ P with members do
 5           cand(k) ← key(k) xor word(k, g) for each state k
 6           N(g) ← the distinct values cand(k)
 7           δ(g) ← (the most distinct cand(t) ≠ cand(s) over the edges (s, t) from one cand(s)) /
↪  |N(g)|
 8        if no group is left then break
 9        g* ← the group of least δ, then of least fluency, then of least index
10        if stop(F + |members(g*)|, |N(g*)|) then break
11        P ← P ∪ {g*}; F ← F + |members(g*)|
12        key(k) ← key(k) xor word(k, g*) for each state k
13        if F is the number of the members of all the groups then break
14     return P
```

`state_abstraction()` (`AutoDistribution.h:1154`) stops before the group whose addition would give more than $N_{\max}$ abstract states on the sample (1000 by default), and `dynamic_state_abstraction()` (`AutoDistribution.h:1125`), DAHDA* of [138], stops before the group whose addition would make the groups hold more than the fraction ρ of the members of all the groups (0.3 by default, the best of 10%, 30%, 50% and 70% in appendix A of [5]), but always keeps the first group; both ignore the features of the groups not chosen, and keep every other feature by itself.

Lemma 10.18 (The state abstractions). Let P be the groups chosen. (i) The key of a state depends only on its features in the groups of P and on its features without a group, and the subspace of the table contains e_x for every feature of a group outside P. (ii) For DAHDA*, P is not empty if some group has members, and the members of P are at most ρ times those of all the groups unless P is the first group alone. (iii) For AHDA*, the states of the sample take at most $N_{\max}$ abstract states for P; in particular, if the first group chosen exceeds $N_{\max}$ alone, P is empty and every state of the sample has the same key. (iv) Each round costs $O(G(N + |M|))$ expected time for G groups, N states and $|M|$ edges of the sample, and there are at most G rounds.

Proof. (i) The projection maps the features of the groups outside P to `no_abstract_feature`, i.e., to the entry 0, and the others to their own abstract features; Corollary 10.12 gives the subspace. (ii) The rule of DAHDA* returns false at its first call and then true exactly when the members of the union would exceed ρ times the total. (iii) The rule of AHDA* stops before any addition that would give more than $N_{\max}$ values of the abstract key on the states of the sample, which are its abstract states (the words of distinct sets of features being distinct, with

probability 1 in the model of random words); if the first candidate exceeds the bound, nothing is added, and the features that remain are those that no move changes, which are the same in every state of the sample, since each state of the sample is reached from the root by moves. (iv) A candidate costs a pass over the states and one over the edges, with hash sets of the abstract states and of the arcs. □

Thus the distribution of AHDA* is Zobrist hashing of the abstract state of P, i.e., HDA*[Z, Astate/SDD] of section 2.5 of [5], with the abstract graph of the sample in place of the abstract graph of the task. The difference between the two rules is the one of [138]: a fixed bound on the abstract states balances the load badly on instances of varying size (with the bound of 10^4, AHDA* ran out of memory on Blocks10-2 and Gripper8 on 48 cores, section 5 there), while a fraction of the features scales with the instance. The ties of line 9 go to the group of least fluency, which the papers do not specify; the abstraction of the nblocks of the planning tasks breaks them by the larger graph instead (Chapter 16). The test `distributions` checks that DAHDA* keeps at most 30% of the features of the 15-puzzle and that one more tile would exceed the bound, and that AHDA* with $N_{\max} = 200$ gives the states of the sample between 2 and 200 abstract states.

10.7 Operator-based Zobrist hashing: OZHDA*

Instead of projecting the features, OZHDA* (section 4 of [138] and section 2.4 of [5]) changes the table of Zobrist hashing so that the hash of an operator, the exclusive or of the entries of the features that it adds and deletes, is 0; a move by such an operator then keeps the hash of its parent. `operator_table()` (`AutoDistribution.h:1191-1226`) is Algorithm 10.7.

Algorithm 10.7 (OZHDA*: the operator-based table).

```
 1  function OPERATOR-TABLE(φ, paper)
 2     R ← the table of Zobrist hashing drawn from the seed
 3     ops ← the operators of the sample, by decreasing weight
 4     if paper then sort ops stably by increasing number of features
 5     mask ← all the 64 bits if paper, the high 32 bits otherwise
 6     for each of the first ceil(φ · |ops|) operators a of ops do
 7        if a has fewer than 2 features then continue
 8        x ← the first feature of a, by index, that is not fixed
 9        if there is none then continue
10        set the bits of mask of R[x] to the xor of those of R[y], y ∈ a \ {x}
11        fix every feature of a
12     return R
```

Proposition 10.19 (The operators that OZHDA* keeps). Call an operator *fixing* if it reaches line 10, and let x_a be the feature that it sets. (i) For every fixing operator a, the bits of mask of $\bigoplus_{y \in a} R[y]$ are 0 in the returned table; hence a move by a keeps its key, and its owner, for every p. (ii) The change vectors δ_a of the fixing operators are linearly independent, and the returned key matrix is uniform among those vanishing on their span V_O; hence, by Lemma 10.11, a move whose change vector is outside V_O keeps its key with probability 2^{-32}.

Proof. (i) Right after line 10 the masked bits of $R[x_a]$ are the sum of those of the other features of a, so that the masked bits of the sum over a vanish. An entry is changed only at line 10, and only for a feature that is not fixed; since line 11 fixes every feature of a, no entry of a feature of a changes afterwards, and the sum stays 0. (ii) Order the fixing operators as they are processed. An operator a cannot contain x_b for a fixing b processed after it, since every feature of a is fixed after a whereas x_b is not fixed when b is processed; hence x_a is in δ_a and in no earlier δ_b (it was not fixed before a), and the vectors are independent, since in a vanishing sum the latest operator would be the only one with its feature x_a. The key matrix has random columns at the features that no fixing operator sets, and at each x_a the sum of the columns of the other features of a, which are features set earlier or never set; this is a bijection from the free columns onto the matrices that vanish on V_O (in such a matrix the column x_a is determined by the others, by induction on the order), so that the key matrix is uniform among them. □

The paper processes all the actions, by increasing number of preconditions, and sets the whole entry; the library processes the operators of the sample by decreasing weight up to the fraction ϕ (0.2 by default) and sets only the 32 bits of the key, and with `as_paper` (the name `OZP`, all the operators by default) it sets the whole entry and orders the operators by increasing number of changed features, which stands for the number of preconditions that a `FeatureDomain` does not give, the ties keeping the order of frequency. An operator whose features are all fixed is skipped (line 9) even if its change vector is outside V_O, which Algorithm 10.7 cannot see; KZHDA* (§10.8)

tests membership in the subspace instead. Since each fixing operator joins its new feature to features already joined, the operators of a DTG join whole paths of it into one class: on a tile of a puzzle the operators are the edges of its grid, and a large fraction of them makes most of the cells of a tile one class, which by Corollary 10.13 balances the load badly (the paper's criticism of the bottom-up clustering of OZHDA*, section 6.1 of [5]). The test `distributions` checks (i) on the most frequent operator of a sample of the 15-puzzle, and that the estimated CO of OZHDA* is below that of Zobrist hashing for $p = 8$.

10.8 Kernel selection: KZHDA*

By Theorem 10.10 and Corollaries 10.12 and 10.13, a distribution is, up to the draw of its table, the subspace V on which its key vanishes: Z is $V = \{0\}$, AZ is V_A, the cuts of the DTGs and the state abstractions are particular projections, and OZHDA* is V_O. KZHDA* chooses V directly: the change vectors of the most frequent operators of the sample are inserted into a basis of V by Gaussian elimination over GF(2), an operator being kept if the table of the larger subspace has a larger estimated value, and the table is drawn uniformly among those that vanish on V. This construction is ours; its starting point is the observation of section 6.1 of [5] that all the earlier distributions are clusterings followed by a partition of the DTGs.

The basis

`Gf2Basis` (`AutoDistribution.h:1237-1295`) keeps a basis of a subspace of $GF(2)^n$ as rows of 64-bit words, each with a pivot:

```
class Gf2Basis {
 public:
 using Vector = std::vector< std::uint64_t >;
 explicit Gf2Basis( std::size_t n );
 std::size_t rank( void ) const noexcept;
 bool insert( const std::vector< std::uint32_t > & bits );
 std::pair< std::vector< Vector > , std::vector< std::uint32_t > >
  reduced( void ) const;
 };
```

Algorithm 10.8 (The basis over GF(2): the rows $\rho(1), \ldots, \rho(\mathrm{r})$ with the pivots $\pi(1), \ldots, \pi(\mathrm{r})$).

```
 1  function INSERT(v)                          ▷ v the vector of the given bits
 2     for j ← 1 to r do
 3        if v has the bit π(j) then v ← v xor ρ(j)
 4     if v = 0 then return false
 5     r ← r + 1; ρ(r) ← v; π(r) ← the least bit of v
 6     return true
 7  function REDUCED()                           ▷ on a copy of the rows
 8     for j ← r down to 1 do
 9        for q ← j + 1 to r do
10           if ρ(j) has the bit π(q) then ρ(j) ← ρ(j) xor ρ(q)
11     return the rows and the pivots
```

Lemma 10.20 (The echelon form). (i) After each insertion, every row $\rho(j)$ has the bit $\pi(j)$ and none of the bits $\pi(q)$, $q < j$; the rows are independent and span the subspace spanned by the vectors inserted. (ii) `insert( v )` returns true if and only if v is not in the span of the earlier vectors, i.e., the rank grows. (iii) `reduced()` returns rows that span the same subspace, each with the bit of its own pivot and none of the pivots of the other rows (reduced row echelon form, up to the order of the rows). An insertion costs $O(rn/64)$ and the reduction $O(r^2 n/64)$.

Proof. (i) and (ii), by induction on the insertions. At step j of lines 2 and 3, v loses the bit $\pi(j)$ if it had it; a later step $q > j$ adds $\rho(q)$, which by induction lacks $\pi(j)$, so that the bit stays at 0. Hence the final v lacks every pivot; if it is not 0, its least bit is a new pivot, and the new row satisfies (i). The vector given is the final v plus a sum of rows, so that the span grows by it exactly when $v \neq 0$, which is (ii). The rows are independent: in a vanishing sum of rows, the row of least index j in it is the only one with the bit $\pi(j)$, since the rows after it lack it. (iii) We show, for j from r down to 1, that after its loop the row j has its own pivot and no other one, and that every row keeps the property of (i). The rows $q > j$ already have only their own pivots; adding $\rho(q)$ to

$\rho(j)$ clears the bit $\pi(q)$ and changes no other pivot bit, so that after the loop $\rho(j)$ has no $\pi(q)$, $q > j$, still none of the $\pi(q)$, $q < j$ (the rows added lack them), and still $\pi(j)$ (the rows $q > j$ lack it). Row operations keep the span. The costs are those of the loops on vectors of $\lceil n/64 \rceil$ words. □

The kernel table

`kernel_table()` (AutoDistribution.h:1310-1326) draws the table of Zobrist hashing from the seed and overwrites the high 32 bits of the entry of each pivot.

Algorithm 10.9 (The kernel table of a basis B).

```
1  function KERNEL-TABLE(n, B)
2     R ← the table of Zobrist hashing on n features, drawn from the seed
3     (ρ, π) ← REDUCED() of B
4     for j ← 1 to r do
5        y ← the xor of the high 32 bits of R[x], for x ∈ ρ(j), x ≠ π(j)
6        replace the high 32 bits of R[π(j)] by y
7     return R
```

Theorem 10.21 (The kernel table). Let V be the span of B. (i) The table vanishes on V; hence every move whose change vector is in V keeps its key and its owner. (ii) If the entries drawn from the seed are independent and uniform, the key matrix is uniform among the matrices that vanish on V, and its rows are independent and uniform in $V^{\perp}$; hence a move whose change vector is outside V keeps its key with probability 2^{-32}, and the owners of the classes of V are those of Corollary 10.13. (iii) The low 32 bits of every entry, and the high bits of every non-pivot feature, are those of Zobrist hashing with the same seed.

Proof. (i) By Lemma 10.20(iii) the features of $\rho(j)$ other than $\pi(j)$ are not pivots, so that their entries are not changed by line 6, for any j; hence after line 6 the high bits of $\bigoplus_{x \in \rho(j)} R[x]$ are $y \oplus y = 0$, i.e., $H\rho(j) = 0$, and by linearity H vanishes on the span of the rows, which is V. Theorem 10.10 gives the moves. (ii) A matrix that vanishes on V has, at each pivot $\pi(j)$, the column equal to the sum of the columns of the other features of $\rho(j)$, all non-pivots; conversely any choice of the non-pivot columns gives, by lines 4 to 6, a matrix that vanishes on V. Hence the map from the non-pivot columns to the result is a bijection onto the matrices that vanish on V, and uniform non-pivot columns give a uniform matrix; Lemma 10.11 and Corollary 10.13 apply. (iii) Line 6 changes only the high bits of the pivots. □

The test `distributions` checks (i) on a basis of 8 features with a dependent vector (Exercise 10.10), and on the subspace of the 40 most frequent operators of a sample of the 15-puzzle.

The selection of the subspace

`kernel_selection_table()` (AutoDistribution.h:1362-1402) grows V greedily over the candidate operators of largest weight (1024 by default), by decreasing weight, from $\{0\}$ or from the subspace V_A of a projection A given by the caller (Algorithm 10.10). The value of a table is the smaller of its estimated efficiencies among p and among $2p$ workers (`estimate_table()`): the sample holds the states of least f, which A* expands first, and a subspace that balances them among p workers only through a few classes balances the later states badly, which the estimate among $2p$ workers exposes. The table of every subspace is drawn from the same seed, so that two steps differ only in the entries of the pivots, and an operator already in V costs nothing.

Algorithm 10.10 (KZHDA*: the selection of the kernel).

```
 1  function KERNEL-SELECTION(p, c, K, A)
 2     B ← the basis of {0}
 3     if a projection A is given then
 4        for each feature x, by index, do
 5           if A ignores x then INSERT(e_x) into B
 6           else if some y < x has A(y) = A(x) then INSERT(e_y + e_x) into B, y the last one
 7     T ← KERNEL-TABLE(n, B); v ← min(eff(T, p), eff(T, 2p))
 8     for each of the first K operators a, by decreasing weight, do
 9        B' ← a copy of B
10        if not INSERT(δ_a) into B' then continue        ▷ already in the span
11        T' ← KERNEL-TABLE(n, B')
12        if min(eff(T', p), eff(T', 2p)) > v then B ← B'; T ← T'; v ← that value
```

```
13     return T
```

Proposition 10.22 (The selection). (i) The vectors inserted at lines 4 to 6 span V_A, so that the start is a table uniform among those that vanish on V_A, i.e., Abstract Zobrist hashing with A in law (not entry by entry). (ii) The returned table vanishes on the span of the start and of the accepted operators, so that every move by an accepted operator keeps its owner. (iii) Its value $\min(\mathrm{eff}(T,p), \mathrm{eff}(T,2p))$ is at least that of the start, and so is its estimated efficiency among p workers, i.e., $\mathrm{eff}(T,p)$ is at least the value of the start. (iv) A step costs $O(r^2n/64 + n + N_F)$ time, for the rank r and the number N_F of the features of all the states of the sample.

Proof. (i) In each abstract feature, the vectors $e_y + e_x$ of consecutive members form a path, whose sums are the even subsets of the class, and the e_x of the ignored features span the rest; this is the set of Corollary 10.12(ii). Theorem 10.21(ii) gives the law. (ii) Theorem 10.21(i) for the final basis, which contains the start and the accepted vectors. (iii) v only increases at line 12, and the returned table is the one of the last increase, or the start; and $\mathrm{eff}(T,p) \geq \min(\mathrm{eff}(T,p), \mathrm{eff}(T,2p)) = v \geq v_0$. (iv) A copy and an insertion cost $O(rn/64)$, the reduction $O(r^2n/64)$, the table $O(n)$, and two estimates read the features of the states of the sample. □

We remark that (iii) does not say that the estimated efficiency of the result among p workers is at least that of the start among p workers: the start may be better among p workers and worse among $2p$. The test `distributions` compares the efficiency of KZHDA* among 8 workers with that of Zobrist hashing among 8 workers, which holds on its instance but does not follow from the code (Exercise 10.11). The names `KZ:GRAZ`, `KZ:GRAZS`, `KZ:GRAZE` and `KZ:DAHDA` start from the subspace of that projection, built on the same model; with `KZ` the start is Zobrist hashing. With p a power of 2, Theorem 10.10(ii) shows that a table need only vanish with its first b rows to keep the owners; the library makes all 32 rows vanish, so that the key, and not only the owner, is kept, and the moves that keep their owner are the same for every p.

10.9 The automatic choice

`auto_distribution()` (AutoDistribution.h:1522-1545) builds every distribution of `auto_distribution_names()` (AutoDistribution.h:1409), the methods of this chapter with their parameters at a few values around the default,

```
Z  GAZ  GRAZ  GRAZS  GRAZE  FAZ:0.1  FAZ:0.3  FAZ:0.5  OZ:0.1  OZ:0.2  OZ:0.3
DAHDA:0.2  DAHDA:0.3  DAHDA:0.5  AHDA:100  AHDA:1000  AHDA:10000
KZ  KZ:GRAZS  KZ:GRAZE  KZ:DAHDA
```

(21 names; `OZP` and `KZ:GRAZ` are accepted by `make_distribution()` but are not among them), adds the distributions of `extra`, estimates each one on the sample among p workers with the cost c, and returns the one of largest estimated efficiency.

Algorithm 10.11 (The automatic choice).

```
1  function AUTO-DISTRIBUTION(m, p, c, extra)
2     C ← MAKE-DISTRIBUTION(m, name, c, seed, p) for each name, then extra
3     best ← the first X ∈ C of largest eff of ESTIMATE(X, p, c)
4     if est is given then est ← the estimate of best
5     return the table distribution of the table of best, named "auto:" and its name
```

Proposition 10.23 (The automatic choice). The returned distribution has the table of a candidate of largest estimated efficiency among p workers, the first in the order of the names and then of `extra` on ties, and therefore the owners of that candidate; HDA* with it returns an optimal path (Proposition 10.1). The construction draws nothing but the table from the seed, so that two processes that build it on the same instance, with the same parameters and the same order of the successors, obtain the same table.

Proof. Line 3 keeps the first candidate of strictly largest efficiency. The returned distribution is built from the table of that candidate, and the owner is a function of the table (Proposition 10.1(i)). The sample is A* with a deterministic OPEN list; the maps that the constructions iterate are ordered, the hash tables are only searched or counted, and every tie is broken by index or by order, so that the result depends on the instance, the parameters and the seed only. □

The choice rests on the correlation of the estimated with the actual efficiency found in section 5.2.1 of [5]; beyond

the choice of the method, it tunes the fractions of FAZHDA*, OZHDA* and DAHDA* and the bound of AHDA*, which the papers fix once for all the instances. Its cost is that of building the 21 candidates and of 21 estimates, of which the 4 kernel selections, each of up to 1024 steps of Proposition 10.22(iv), are by that bound the most expensive. The test `distributions` checks that the chosen name begins with `auto:` and that no candidate has a larger estimate, and solves logistics-4-0 with the automatic distribution on 1 and 3 workers; on blocks-6-2 it solves the task with every candidate.

The hand-made projections of the sliding-tile puzzles

The projections of the sliding-tile puzzles compared by Jinnai and Fukunaga (footnote 4 of section 3.1.4, footnote 6 of section 3.2 and Figure 8 of [5], and [141]) are given by `SlidingTile::tile_abstraction()` (`SlidingTile.h:493-515`), which returns, for a `TileRegions` and a list of tiles, the table of a `FeatureAbstraction`:

```
enum class TileRegions { single , pair , line , block , half , quad , cross ,
                         odd };
[[nodiscard]] static std::vector< std::uint32_t > tile_abstraction(
 TileRegions r , const std::vector< int > & tiles = {} );
```

The abstract feature of "tile t in cell c" is the pair of t and of the region of c for the tiles of the list (all if it is empty), and the other tiles are ignored; the regions are the following (`SlidingTile.h:844-875`).

`TileRegions`	The regions of the cells of a board of $W \times H$ cells
`single`	each cell by itself, i.e., Zobrist hashing
`pair`	the cells $2k$ and $2k+1$, in the order of the rows
`line`	the rows
`block`	the blocks of 2×2 cells (smaller on the last row or column of an odd board)
`half`	the first $\lfloor WH/2 \rfloor$ cells and the others
`quad`	four blocks of 6 cells, the center ignored (5×5 only)
`cross`	five regions of 5 cells around the center (5×5 only)
`odd`	the middle column and four regions of 5 cells (5×5 only)

Ignoring all the tiles but a few gives the AHDA* of Burns et al. as Zobrist hashing of the abstract state (e.g., the tiles 1 to 3 on the 15-puzzle, the abstraction chosen for AHDA* in footnote 4 of [5]). A region that the board has not, or a tile outside $[1, WH)$, throws `std::invalid_argument`. These projections can be passed to `auto_distribution()` in `extra`, as `TableZobristDistribution< D >( d , FeatureAbstraction::of( D::tile_abstraction( r ) ) , seed , name )`, so that the automatic choice compares them with the constructions. The test `distributions` counts the regions of each projection on the 15-puzzle and on the 24-puzzle, and solves two instances of the 24-puzzle with HDA* on 2 workers under every construction and under the projections `quad`, `cross`, `odd`, `line` and `half` on the tiles 1 to 5.

The benchmark tool

The option `--dist` of `tools/AStarBench.cpp` takes, besides the distributions of Chapter 9 (`Z`, `AZ`, `A`, `P`, `H`, `R`) and the hyperplanes (`HWD`, Chapter 15), every name of `make_distribution()`, `AUTO` for `auto_distribution()` (whose choice is printed in the column `chosen`), and `tile:<regions>[:<tiles>]` for a hand-made projection, e.g., `tile:block:1-4`; `--sample` sets the expansions of the sample (32768) and `--cost` the cost c of a message (1). The constructions are offered on the domains that are an `AbstractFeatureDomain` or a `FeatureGroupDomain` (the puzzles, the grids and the planning tasks), and run on the C++ threads or over MPI (`tools/AStarBenchSolve.h:473-553`). Each run prints the estimates of its distribution on the sample (the columns `estCO`, `estLB`, `sparsity` and `effesti`) next to the measured `CO`, `LB` and `LBstored` (the storage balance), and, with `--so 1`, the search overhead `SO` against a run of A* on the same instance; with `--estimate 1` the distributions of Chapter 9 are estimated as well.

10.10 The engineering of the implementation

- The constructions read only a `FeatureGraph`, which holds no state of the `Domain`: the features of each state are stored in one array with offsets (`state_features()`), and the words of the states in the groups in one array of NG words. Hence the constructions, the estimates of a table and the kernel selection are compiled once, and only the sample and `FeatureModel::estimate()` are instantiated per `Domain`.
- The sample keeps each state in a hash map indexed by the hash of the `Domain`, its values in parallel arrays, and the entries of OPEN as triples in a binary heap; the states are copied once more into the final order, the expanded ones first, so that the loads of an estimate are the first `num_expanded()` owners.
- The operators are counted in an ordered map from their change sets, and the DTG edges in ordered maps from their pairs, which makes the construction deterministic (Proposition 10.23) at the price of a logarithmic factor per move.
- `estimate_table()` computes the hash of each state from its stored features, i.e., in N_F lookups, which is what makes the two estimates of each step of the kernel selection affordable; the estimate of a distribution that is not a table calls its `owner()` on the stored states.
- The cuts use the static weights for the greedy halves and for the sparsity of the paper, and the sampled weights otherwise, both kept per feature as adjacency lists sorted by neighbour; the enumeration of the cuts reads the adjacency of the node that changes side only.
- The kernel table is recomputed from the seed at each step instead of being updated, which keeps every step independent of the rejected operators (Theorem 10.21(iii)).

10.11 Benchmark

Table 10.1. The distributions of this chapter and of Chapter 9, and the hand-made projections, on the 24-puzzle: estimated and measured overheads, and the speedup of HDA* on the threads.

distribution	estCO	estLB	effesti	CO	LB	LBstored	SO	seconds (speedup), 8 cores	seconds (speedup), 32 cores

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

Table 10.2. The distributions of this chapter and of Chapter 9 on the planning tasks of `data/planning`.

distribution	estCO	estLB	effesti	CO	LB	LBstored	SO	seconds (speedup), 8 cores	seconds (speedup), 32 cores

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The experiment runs HDA* on the C++ threads with each distribution, on the instances of the 24-puzzle distributed with the code of Jinnai and Fukunaga (Table 10.1) and on the largest planning tasks of the tester (Table 10.2), with batches of 64 nodes and flushes every 64 expansions, on 8 and 32 cores, with samples of 32768 expansions and $c = 1$. A row is a distribution, by its name in `--dist` (for `AUTO`, the constructions chosen on the instances are listed in the note under the table), and a column gives, for a number of cores, the estimates on the sample (co, lb and eff), the measured CO, LB and storage balance, the search overhead against A*, and the time summed over the instances with the speedup over A* in parentheses. The time of `solve()` is measured as in §9.8; the time of the sample and of the construction of the distribution, which it does not include, is given in a separate column, since it is the price that an automatic distribution pays over a hand-made one. We remark that the tables are placeholders until the campaign is run on an idle machine.

10.12 Notes and references

Abstract Zobrist hashing, and the greedy cut of the DTGs (GAZHDA*), are from Jinnai and Fukunaga [141]; the fluency filter (FAZHDA*), the operator-based Zobrist hashing (OZHDA*) and the dynamic threshold of AHDA* (DAHDA*) from [138]; and the workload graph, the estimates of equations (3) to (5), the sparsity of equation (6), GRAZHDA* and the reading of the earlier methods as clusterings followed by a partition of the DTGs from the journal paper [5], which also gathers the experiments of the two conference papers (on planning with the merge-and-shrink heuristic [140], on a cluster and on 128 virtual cores, and on the 24-puzzle). AHDA* is from Burns et al. [3], who took the greedy abstraction of Zhou and Hansen [139] for structured duplicate detection; the DTG is that of SAS+ [133], as the planner Fast Downward [134] computes it; and the local search of the cuts is that of Fiduccia and Mattheyses [142]. In section 6.2.2 of [5] the correlation between the sparsity and the estimated efficiency is shown on partitions of the workload graph made by a graph partitioning package, which the library does not use.

The following facts are stated from the code and the papers, not from measurements. The papers implement these distributions in Fast Downward, for planning tasks, with the DTGs of the task and with the workload graph computed by a full run of A*; the library builds them on a bounded sample of any `FeatureDomain`, for which the DTGs, the operators and the fluency are those of the moves of the sample, and the groups may be inferred. The weights of the sample in the cuts, the objective of efficiency, KZHDA* and the choice by the estimate among all the constructions are ours, and we have no measurement of them yet (§10.11). The reading of every distribution as a subspace of the kernel of a linear map (§10.4) makes precise the remark of section 6.1 of [5] that all the earlier methods are instances of GRAZHDA* with a clustering; we are not aware of a statement of it in this form. Finally, the hyperplane work distribution of Kobayashi et al. [43] for the multiple sequence alignment, which uses the levels of the alignment rather than its features, is treated in Chapter 15.

10.13 Exercises

Exercise 10.1 (`ctest -R PBeFS.distributions`). Using Proposition 10.1(iii), show that `make_distribution ( m , "Z" )` assigns every state to the same owner as `ZobristDistribution` with the same seed, and say what would change if the generator drew the table backwards; the test checks the owners of the states of a sample of the 15-puzzle for $p = 7$.

Exercise 10.2 (`ctest -R PBeFS.distributions`). Prove that the DTG of every tile of the 15-puzzle in a `FeatureModel` is its grid graph, i.e., that every edge joins two cells at Manhattan distance 1, and that the groups inferred without `num_feature_groups()` are the tiles (Lemma 10.5). Would the inference still give the tiles if the puzzle listed no ground transitions and the sample were small?

Exercise 10.3 (`ctest -R PBeFS.distributions`). Show that for $p = 1$ the estimates of every distribution are $\mathrm{co} = 0$ and $\mathrm{lb} = 1$, and explain by Corollary 9.12 why the estimated CO of Zobrist hashing among 8 workers exceeds 0.8 on a sample of the 15-puzzle. Why is it not exactly 7/8?

Exercise 10.4 (`ctest -R PBeFS.distributions`). Run Algorithm 10.3 by hand on the grid graph of a tile of the 15-puzzle and show that it adds the cells $0, 1, \ldots, 7$ in this order, so that GAZHDA* is there the projection onto the halves of the board of §9.4; then run it on the 5×5 grid and check the cut of Figure 10.3. The test checks that each cut has two sides whose sizes differ by at most one (Lemma 10.15).

Exercise 10.5 (`ctest -R PBeFS.distributions`). Prove that the enumeration of Algorithm 10.4 is exact (Lemma 10.16) by writing out its seven steps for $n = 4$, with the side of each node and the weight of the cut. The test compares the cut of GRAZHDA* with the weights of the sample against all the 2^{15} cuts of three DTGs of the 15-puzzle.

Exercise 10.6 (`ctest -R PBeFS.distributions`). With the static weights, show that the sparsest cut of the 4×4 grid is two rows against two and that of the 5×5 grid two rows against three (hint: a cut of k edges that does not cross the board from side to side isolates a corner of at most $k^2/4$ cells, and one that crosses it cuts at least w edges of the $w \times w$ grid). The test checks both cuts, the second one found by the passes of Algorithm 10.5.

Exercise 10.7 (`ctest -R PBeFS.distributions`). Show that `fluency_abstraction( m , 0.3 )` ignores exactly 4 of the 15 groups of the 15-puzzle, those of largest fluency, and describe by Corollary 10.13 how the classes of the states grow when one more tile is ignored.

Exercise 10.8 (`ctest -R PBeFS.distributions`). Show that DAHDA* with $\rho = 0.3$ keeps on the 15-puzzle at most 4 tiles, i.e., 64 of the 240 members of the 15 groups, and that a fifth tile would exceed the bound; then

find the conditions under which AHDA* with $N_{\max} = 200$ keeps no group at all (Lemma 10.18(iii)), and what HDA* does with that distribution.

Exercise 10.9 (`ctest -R PBeFS.distributions`). Prove Proposition 10.19(i) for the most frequent operator of a sample of the 15-puzzle, and give an operator of the 15-puzzle that Algorithm 10.7 may skip although its change vector is not in V_O. Why does a fraction close to 1 give a poor load balance?

Exercise 10.10 (`ctest -R PBeFS.distributions`). Insert the vectors $\{0,1\}$, $\{1,2\}$, $\{0,2\}$ and $\{5\}$ into a `Gf2Basis` of 8 features by Algorithm 10.8, and show that the third insertion returns false and the rank is 3. Compute the reduced rows and the high bits of the entries 0, 1, 2 and 5 of the kernel table, and the probability that the high bits of the entry 3 are 0 (Theorem 10.21).

Exercise 10.11 (discussion, no test). Build a sample and two tables T_0 and T_1 such that $\min(\mathrm{eff}(T_1,p),\mathrm{eff}(T_1,2p)) > \min(\mathrm{eff}(T_0,p),\mathrm{eff}(T_0,2p))$ but $\mathrm{eff}(T_1,p) < \mathrm{eff}(T_0,p)$, and conclude which part of the check of KZHDA* in the test `distributions` is guaranteed by Proposition 10.22(iii) and which is not.

Exercise 10.12 (`ctest -R PBeFS.distributions`). Show that `auto_distribution()` returns the table of a candidate of largest estimated efficiency (Proposition 10.23), and that HDA* with it, or with any construction, returns an optimal plan on a planning task (Proposition 10.1); the test solves blocks-6-2 with every candidate and logistics-4-0 with the automatic choice, on 1 and 3 workers.

Exercise 10.13 (`ctest -R PBeFS.hda-owners`). Using Theorem 10.10(ii), show that for $p = 64$ the owner of a key is its first 6 bits, that each owner has 2^{26} keys, and that a move keeps its owner among 64 workers if and only if the first 6 rows of the key matrix vanish on its change vector; the test checks the owners of the keys for $p = 64$ among other values (cf. Exercise 9.13).

Exercise 10.14 (discussion, no test). Write the subspace V of the projection `quad` of the 24-puzzle (the center ignored) and that of `line`, and compare by Corollary 10.13 the number of classes that each gives to the positions of a tile. Which of them can keep a move of a tile into or out of the center on its worker?

11. Parallel best-nblock-first search

[Source: `include/PBeFS/astar/SafePBNF.h` (`SafePBNF`, `PBNFOptions`), `include/PBeFS/astar/NBlockMap.h` (`NBlockMap`), `include/PBeFS/astar/Domain.h` (`NBlockDomain`), and the nblocks of `include/PBeFS/astar/Grid.h`, `SlidingTile.h` and `VacuumWorld.h`]

We are concerned in this chapter with a parallel A* in which the workers never lock the lists of the search while they expand nodes: the Parallel Best-NBlock-First search (PBNF) of Burns, Lemons, Ruml and Zhou [3], in the livelock-free form that the authors call Safe PBNF, which is the one of the library. Whereas the searches of Chapter 8 share one OPEN list under a lock, and HDA* (Chapter 9) sends each generated node to the worker that owns it, PBNF partitions the state space by an abstraction into *nblocks*, each with an OPEN and a CLOSED list, and lets a worker expand the nodes of an nblock only when no other worker can touch the lists that these expansions may write (we say that the worker holds the nblock). Hence, the synchronization is confined to a graph of counters (the abstract graph of the nblocks), which is usually (but not always) much smaller than the state space and which the workers lock only to choose the next nblock. The price is twofold: the order of expansion is best-first only approximately, which often costs some search overhead (cf. Chapter 8), and the `Domain` has to supply an abstraction of the problem.

The idea comes from the Structured Duplicate Detection (SDD) of Zhou and Hansen, developed for the search on external memory and turned into the Parallel Structured Duplicate Detection (PSDD) for breadth-first heuristic search (both summarized in section 2.4 of [3], pp. 694-696). We describe PSDD, with its synchronization by layers, in Chapter 12. Here we use the notation of Chapter 1. A* and its invariants are in Chapter 2, the OPEN and CLOSED lists of a single nblock are those of Chapter 3, and the overheads by which a parallel search is measured, as well as the general condition under which a parallel search may stop, are in Chapter 8.

The structure of the chapter is as follows. Section 11.1 states the interface, i.e., the concept `NBlockDomain`, the abstractions of the domains of the library and the options of the search. Section 11.2 gives the duplicate detection scope and the interference between nblocks, on which the search is built, and section 11.3 the search PBNF. Section 11.4 shows how plain PBNF may livelock and how the hot nblocks of Safe PBNF prevent it. In section 11.5 we give the pseudocode of the library line by line, and in section 11.6 we prove its properties (exclusive access, absence of deadlock, optimality, termination, what can be proved of the livelock freedom, and the detection of the end). Section 11.7 describes the implementation (the creation of the nblocks, the free list and the expansion, in pseudocode) with the cost of its operations and its memory, section 11.8 the benchmark, and sections 11.9 and 11.10 give the references and the exercises.

11.1 The problem and the interface

We consider the problem of Chapter 1: a `Domain` gives the initial state s_0, the successors of a state u with the costs $c(u, v)$ of the edges, the goal test and an admissible heuristic h, and the search returns a path of least cost C^* from s_0 to a goal, or proves that none exists. Throughout the chapter the costs are positive, and a statement that needs them bounded away from zero states it. As in the other parallel searches (§8.3), the workers share the *incumbent* z, i.e., the least g of a goal selected so far ($z = +\infty$ before the first one), and they prune the nodes with $f \geq z$. PBNF needs, besides this, an abstraction of the state space, which we now define.

Definition 11.1 (Abstraction and nblocks). An *abstraction* is a map ϕ from the states to a finite set $B = \{0, \ldots, N-1\}$ of *abstract states*, together with a relation $\mathrm{succ}(b) \subseteq B \setminus \{b\}$, the *abstract successors* of b, such that (i) for each edge (u, v) of the state space, either $\phi(v) = \phi(u)$ or $\phi(v) \in \mathrm{succ}(\phi(u))$, and (ii) the relation is symmetric, i.e., $a \in \mathrm{succ}(b)$ if and only if $b \in \mathrm{succ}(a)$. The *nblock* b is the set of the states u with $\phi(u) = b$ (and, by the usual abuse, the abstract state b itself), and the *abstract graph* has the nodes B and the edges $\{a, b\}$ with $a \in \mathrm{succ}(b)$.

Condition (i) means that the abstraction is a homomorphism of the state space onto the abstract graph with its self-loops, i.e., that an edge never jumps over an nblock. Burns et al. state it as "if two states are successors in the state space, then their images are successors in the abstract graph" ([3], section 2.4, p. 694), with a self-loop on an nblock when one of its states has a successor in it. Condition (ii) is not required by Burns et al., whose appendix A also covers directed abstract graphs. We assume it in the library, since `SafePBNF` builds the interference scope of an nblock from its successors alone (the comment "with an undirected abstract graph, the predecessors are the successors" of `NBlockMap.h:15-16`, which builds the scopes). Both conditions hold for the domains of the library, and we check neither of them; a `Domain` that breaks them gives a search with data races (see Exercise 11.9).

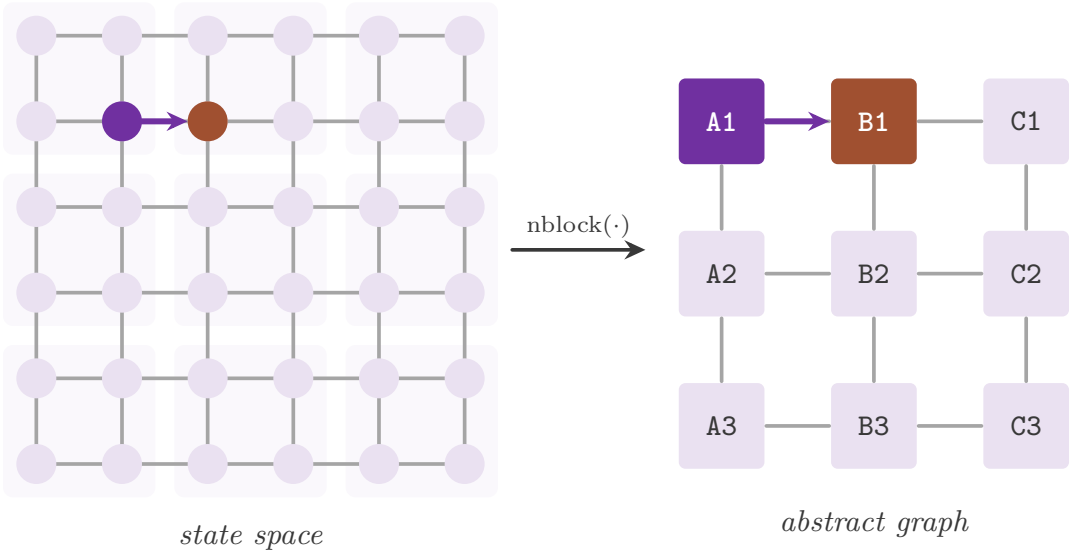


Figure 11.1. An abstraction of a 6×6 grid of states into nine nblocks of 2×2 states (left, the shaded squares) and the abstract graph of the nine nblocks (right). The edge of the state space from the state in the primary colour to the state in the accent colour crosses from the nblock A1 to the nblock B1, and it is mapped onto the abstract edge between them, as condition (i) of Definition 11.1 requires.

In the library, a `Domain` provides an abstraction by modelling the concept `NBlockDomain` (`Domain.h:281`), i.e., by three members: `num_nblocks()`, the number N of nblocks; `nblock( s )`, the nblock $\phi(s) \in \{0, \ldots, N-1\}$ of a state; and `for_each_nblock_successor( b , f )`, which calls `f(a)` for each $a \in \mathrm{succ}(b)$, without b (the search adds b to the scope, see below). Figure 11.1 gives the picture on a grid; in the library, three domains model the concept, as follows (one can of course add the three members to any other `Domain` that has a "natural" abstraction).

- `Grid< Moves >` (`Grid.h:721-747`): the nblocks are the rectangles of `nblock_width()` $\times$ `nblock_height()` cells given by the `GridNBlocks` of the constructor (squares of $s \times s$ cells, $s = 20$, by default, and 20 in `AStarBench`; `GridNBlocks::coarse` gives a number of nblocks per side instead), and the abstract successors of a block are the 4 blocks next to it with 4 moves, and the 8 blocks around it with 8 moves, since a diagonal move may cross a corner of a block.
- `SlidingTile< W , H >` (`SlidingTile.h:527-563`): the nblock of a board is the triple of the cells of the blank and of the tiles 1 and 2, which is the abstraction used by Burns et al. for the 15-puzzle ([3], section 4.4, p. 712); its ids are $(p_0 \cdot WH + p_1) \cdot WH + p_2$, i.e., $16^3 = 4096$ ids for the 15-puzzle, of which only the $16 \cdot 15 \cdot 14 = 3360$ with three distinct cells are abstract states (the others have no successors and are never reached). A move of the blank to a cell that holds the tile 1 or 2 swaps two cells of the triple, and any other move changes only the cell of the blank.
- `VacuumWorld` (`VacuumWorld.h:176-197`): a row of n cells with a robot that moves left or right or sucks the dirt of its cell. The nblock of a state is the cell of the robot, and the abstract graph is the path $0 - 1 - \cdots - (n-1)$, which is the abstract graph of the figures of this chapter from Figure 11.3 on (where, for readability, the nblocks are numbered from 1).

In the library, the search is the class template `SafePBNF` (`SafePBNF.h:244-249`), whose parameters are the `Domain` (a model of `NBlockDomain`), the executor that runs the workers (the default thread pool unless another one is given; the executors are described in §1.6 and §8.1) and the OPEN list of the nblocks (the bucket queue for

integer costs and the binary heap otherwise, Chapter 3). Its constructor takes the `Domain` (which must outlive the search) and a `PBNFOptions`, and its `solve()` returns the `Solution` of Chapter 2, with the status `solved`, `unsolvable` or `node_limit` and the counters of the search. Each of the p workers runs on a separate thread of the executor (e.g., on the threads of C++ or on OpenMP, the two executors that the tests and the benchmark use). One can lend an executor that already exists by `set_executor()`; without it, `solve()` builds one of p threads. The members that a user calls are declared as follows (the declarations of `SafePBNF.h`, without their bodies; `SolutionT` is `Solution< state_t< D > , cost_t< D > >`):

```
explicit SafePBNF( const D & domain , const PBNFOptions & options = {} );
void set_executor( Executor & ex ) noexcept;
[[nodiscard]] SolutionT solve( void ) const;
```

`PBNFOptions` (`SafePBNF.h:186-203`) derives from the `ParallelOptions` of the other parallel searches (§8.1) and adds three fields,

```
struct PBNFOptions : public ParallelOptions {

 /// expansions in an nblock before a worker may switch to a better one
 unsigned min_expansions = 32;

 /// the weights of the iterations of ARPBNF, decreasing, the iterations
 /// after the last one having the weight 1; empty for (w)PBNF, whose weight
 /// is that of the SearchOptions
 std::vector< double > schedule;

 /// true for the hot nblocks that prevent the livelock; false for plain
 /// PBNF, in which a worker leaves its nblock only for a better free one,
 /// so that a better nblock that its scope blocks may wait for ever
 bool safe = true;

 };  // end( struct( PBNFOptions ) )
```

whose meaning, with that of the inherited fields, is the following:

- `min_expansions`, the number m of expansions that a worker makes in an nblock before it may switch to another one (32 by default; 0 is taken as 1, `SafePBNF.h:271-272`);
- `safe`, true by default for Safe PBNF, and false for plain PBNF, in which no nblock is ever hot and a worker leaves its nblock only for a better free one (section 11.4);
- `schedule`, the decreasing weights, not below 1, of the iterations of Anytime Restarting PBNF (ARPBNF), empty by default; the constructor throws `std::invalid_argument` if they do not decrease or fall below 1 (`SafePBNF.h:273-277`);
- of the inherited fields, `threads` is the number p of workers (0 is taken as 1), and `node_limit` stops the search with the status `node_limit` when a worker has allocated that many nodes; the fields `batch`, `flush_period` and `seed`, which serve the messages and the owners of HDA*, and `expected_nodes` are not used, since the closed lists are per nblock and grow from a "small" size (16 slots); the weight w, the dropping of the duplicates and the anytime search of `SearchOptions` make the search the weighted PBNF (wPBNF) of section 6 of [3] and the anytime weighted PBNF, which Chapter 6 describes with ARPBNF (section 11.5 gives the changes to the pseudocode).

The following example, from the case `manual` of the tester, which the build compiles and runs (`ctest -R PBeFS.manual`), solves a random grid of 300×200 cells with 4 moves, whose nblocks are squares of 10×10 cells (the last argument of the constructor of the `Grid`), with $p = 4$ workers and $m = 32$ (the default, set explicitly in the options `o`); the case then checks the `Solution r` against the cost of the sequential A* on the same grid.

```
const auto m = GridMap::random( 300 , 200 , 0.25 , 2 );
// from the bottom-left to the bottom-right corner, with nblocks of
// 10 x 10 cells (100 is the side of the blocks of the other searches)
const Grid< 4 > g( m , 0 , 199 , 299 , 199 , 100 , 10 );
PBNFOptions o;
o.threads = 4;
o.min_expansions = 32;   // expansions in an nblock before a switch
```

```
const auto r = SafePBNF< Grid< 4 > >( g , o ).solve();
// r.status, r.cost, r.path and r.stats as for A*
```

The declaration needs only the `Domain` type, since the executor and the OPEN list have defaults, and the `Solution` is that of every search of the library.

11.2 Structured duplicate detection

When a worker expands a node u of the nblock b, the children may fall only in b or in an abstract successor of b, by condition (i) of Definition 11.1. Hence, a worker that has exclusive access to the lists of these nblocks can check each child against the CLOSED list of its nblock and insert it in the OPEN list of its nblock with no lock at all. This is the observation behind SDD, which the following definitions make precise.

Definition 11.2 (Scopes and interference). The *duplicate detection scope* of the nblock b is $S(b) = \{b\} \cup \mathrm{succ}(b)$. Two distinct nblocks a and b *interfere* if $S(a) \cap S(b) \neq \emptyset$, and the *interference scope* of b is the set $I(b) = \{a \in B \setminus \{b\} : S(a) \cap S(b) \neq \emptyset\}$ of the nblocks that interfere with b.

Note that the definition of Burns et al. reads $\mathrm{IntScope}(b) = \{l : \exists x \in \mathrm{Successors}(b),\ l \in \mathrm{Predecessors}(x)\}$ ([3], section 3.1, p. 698), with the self-loops of the abstract graph among the successors. On an undirected abstract graph it is the set above plus b, which plays no role. The following two facts are immediate, and the rest of the chapter uses them throughout.

Lemma 11.3 (The children stay in the scope). If u is a state of the nblock b and v is a successor of u, then $\phi(v) \in S(b)$.

Proof. By condition (i) of Definition 11.1, $\phi(v) = b$ or $\phi(v) \in \mathrm{succ}(b)$, i.e., $\phi(v) \in S(b)$. □

Lemma 11.4 (Form and symmetry of the interference scope). For each nblock b,

$$I(b) = \Big(\mathrm{succ}(b) \cup \bigcup_{a \in \mathrm{succ}(b)} \mathrm{succ}(a)\Big) \setminus \{b\},$$

and $a \in I(b)$ if and only if $b \in I(a)$. In particular, $\mathrm{succ}(b) \subseteq I(b)$.

Proof. We obtain the symmetry from that of the condition $S(a) \cap S(b) \neq \emptyset$. For the form, let $a \neq b$. If $a \in \mathrm{succ}(b)$, then $a \in S(a) \cap S(b)$; if $a \in \mathrm{succ}(x)$ for some $x \in \mathrm{succ}(b)$, then $x \in S(a) \cap S(b)$. Conversely, let $x \in S(a) \cap S(b)$: if $x = b$, then $b \in \mathrm{succ}(a)$, and hence $a \in \mathrm{succ}(b)$ by condition (ii); if $x = a$, then $a \in \mathrm{succ}(b)$; otherwise, $x \in \mathrm{succ}(b)$ and $x \in \mathrm{succ}(a)$, and therefore $a \in \mathrm{succ}(x)$ by condition (ii). □

We remark that this is how `SafePBNF`, through the class template `NBlockMap`, builds the interference scope of an nblock the first time that the nblock is touched: the successors of b and their successors, sorted, without repetitions and without b (`NBlockMap.h:114-142`, Algorithm 11.4 in section 11.7). Figure 11.2 gives the scopes on a grid of nblocks. With 4 moves, $S(b)$ is the cross of the nblock and its four neighbours and $I(b)$ the diamond of the 12 nblocks within Manhattan distance 2. With 8 moves, $S(b)$ is the 3×3 square and $I(b)$ the 24 other nblocks of the 5×5 square around b (fewer on the border of the grid); Lemma 11.19 proves these counts and the general bound $|I(b)| \leq d^2$ for an abstract graph of largest degree d. Yet, the interference scope is usually a tiny part of the abstract graph, e.g., 24 nblocks out of the 62 500 of the benchmark of section 11.8.

Figure 11.3 gives the interference on the path of the vacuum world. The scopes of the nblocks 3 and 5 share the nblock 4, and a worker expanding the nodes of 3 and another expanding those of 5 could both write the lists of 4. On the contrary, the scopes of 3 and 6 are disjoint, and the two nblocks can be searched in parallel.

A worker that holds the nblock b thus expands a node u of b much as the sequential A* does (cf. Chapter 2), except that each child v is looked up in the CLOSED list of $\phi(v)$ and inserted in the OPEN list of $\phi(v)$, both of which lie in $S(b)$ by Lemma 11.3. In the expansion of Figure 11.4, one of the three children of u falls in the nblock 2, one in the nblock 3 of u, and the third is a state already closed in the nblock 4 with a path that is not longer, which is dropped as a duplicate.

PSDD (section 2.4 of [3]) uses these scopes to expand in parallel the nodes of one layer of a breadth-first heuristic search. In each layer, the workers acquire nblocks whose scopes are disjoint, expand their nodes of the current depth, and synchronize all together before the next layer. PSDD and the variants of Burns et al. that order the layers by f (BFPSDD) or deepen them iteratively (IDPSDD) are described in Chapter 12. PBNF keeps the scopes and drops the layers, whose synchronization is often "costly" when they are small. Indeed, Burns et

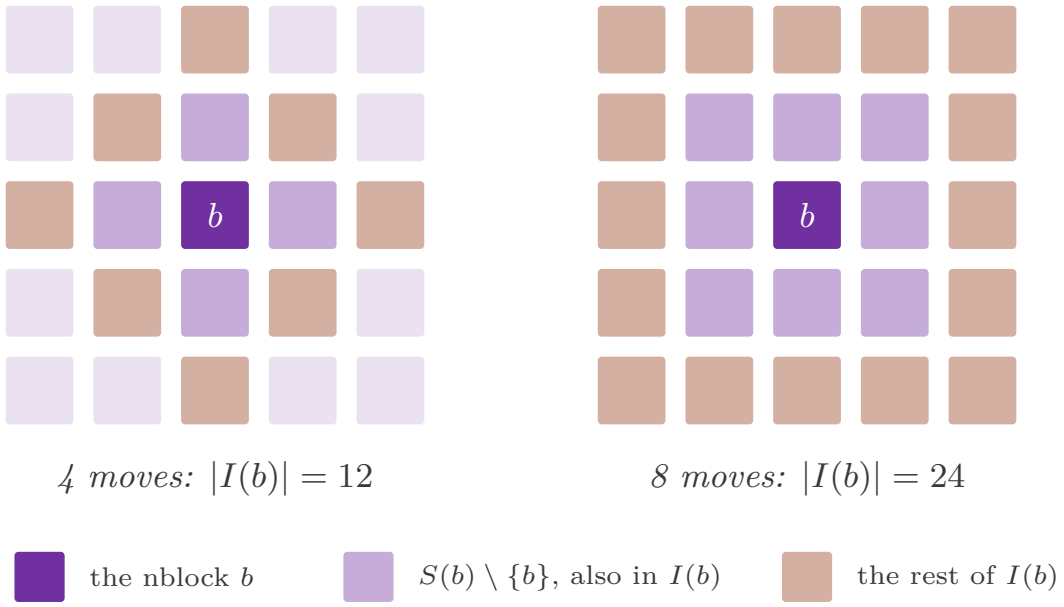


Figure 11.2. The scopes of an nblock b (dark) in the middle of a grid of nblocks, with 4 moves (left) and with 8 moves (right): the other nblocks of the duplicate detection scope $S(b)$ (light violet), which belong to the interference scope $I(b)$ as well, and the rest of $I(b)$ (accent colour). While b is in use, none of the coloured nblocks can be acquired.

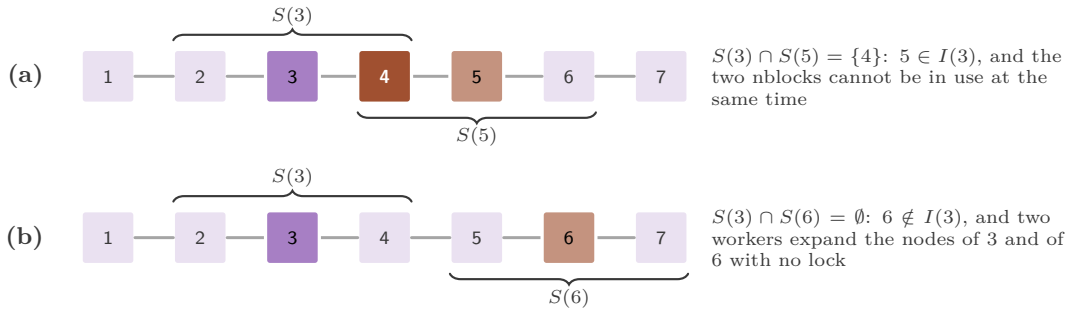


Figure 11.3. Interference on the abstract graph of the vacuum world, a path. (a) The scopes of the nblocks 3 and 5 share the nblock 4: the two interfere, and at most one of them can be in use. (b) The scopes of the nblocks 3 and 6 are disjoint: two workers may hold them at the same time, and they expand their nodes with no lock.

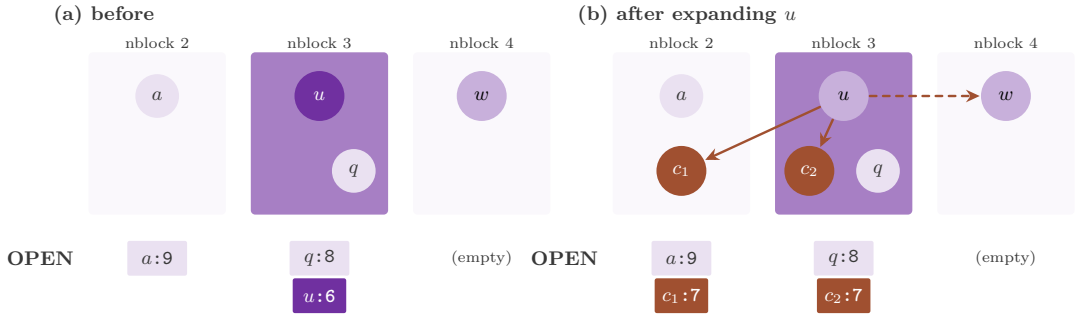


Figure 11.4. An expansion by the worker that holds the nblock 3 (in the colour of worker 0). (a) The node u, the first of OPEN(3), is selected. (b) Its children c_1 and c_2 go into OPEN(2) and OPEN(3), and the third child, the closed state w of the nblock 4, is found in CLOSED(4) with a value of g that is not larger, and it is dropped. All three lists lie in $S(3)$, and the worker writes them with no lock.

al. observe that BFPSDD "may incur excessive synchronization overhead if there are few nodes in each f layer" (section 2.4.1, p. 696).

11.3 Parallel best-nblock-first search

PBNF approximates the order of A* by a heap of nblocks. Each worker expands the nodes of the nblock with the best OPEN list among those that can be taken, and keeps it while it is not worse than the others (section 3 of [3], pp. 696-697, which borrows the heap of sets of nodes from the Localized A* of Edelkamp and Schrödl). Besides the lists of the nblocks, we call the state of the search the following.

Definition 11.5 (The nblock graph). At each instant, an nblock is *in use* if a worker holds it. For each nblock b, $f(b)$ is the least f of the nodes in OPEN(b) ($+\infty$ if none), $\sigma(b)$ is the number of the nblocks in use whose interference scope contains b, and, in Safe PBNF, $\sigma_h(b)$ is the number of the *hot* nblocks (section 11.4) whose interference scope contains b (in plain PBNF, $\sigma_h \equiv 0$). The nblock b is *free* if it is not in use, $\sigma(b) = \sigma_h(b) = 0$, and $f(b) < z$, where z is the incumbent. The *free list* holds the free nblocks ordered by $f(b)$.

The counters, the hot flags and the free list form the *nblock graph*, which one lock protects; its critical sections are rather short, since each of them typically touches only the interference scopes of a few nblocks. A worker that holds the nblock b searches it as follows. It takes the best node of OPEN(b), stops if its f is not smaller than z, records it as the incumbent if it is a goal, and otherwise expands it as in Figure 11.4. After each m expansions, it *checks* whether it should switch, i.e., whether the best free nblock or the best nblock of $I(b)$ is better than b. If it should, it locks the graph, *releases* b (it decrements σ on $I(b)$ and puts in the free list the nblocks that become free) and *acquires* the first nblock b' of the free list (it takes b' out of the free list and increments σ on $I(b')$, whose nblocks leave the free list). Figures 11.5 and 11.6 give the two operations on the path of the vacuum world, with $f(b)$ above each nblock and σ and σ_h below it.

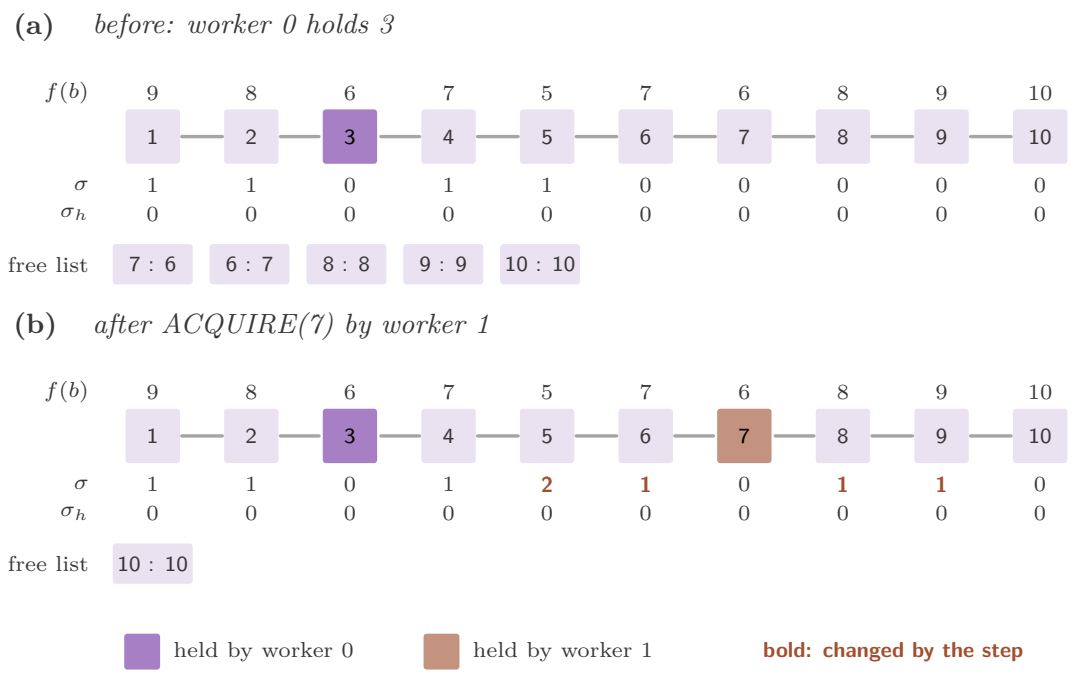


Figure 11.5. ACQUIRE on the path of the vacuum world, with 10 nblocks. (a) Worker 0 holds the nblock 3, and hence $\sigma = 1$ on $I(3) = \{1, 2, 4, 5\}$, and the free list holds the nblocks 6 to 10, ordered by f (each cell gives the nblock and its f). (b) Worker 1 acquires 7, the first of the free list: σ grows by one on $I(7) = \{5, 6, 8, 9\}$, and 6, 8 and 9 leave the free list. The nblock 5, which interferes with both, has $\sigma(5) = 2$, and only 10 stays free.

Three choices of Burns et al. reduce the overhead of this scheme, and we keep all of them. First, the minimum number m of expansions between two checks bounds the frequency of the lock of the graph, at the price of "speculative" expansions, i.e., of nodes that may be worse than the best node of another nblock (section 3, p. 697). Their tuning on twenty 5000×5000 grids with 62 500 nblocks found that the time spent on the lock decreased as m grew from 1 to 64, and that the benefit levelled off at about $m = 8$, while the number of expanded nodes grew with m and with the number of threads (section 4.1 and Figure 4, pp. 702-703). In their implementation

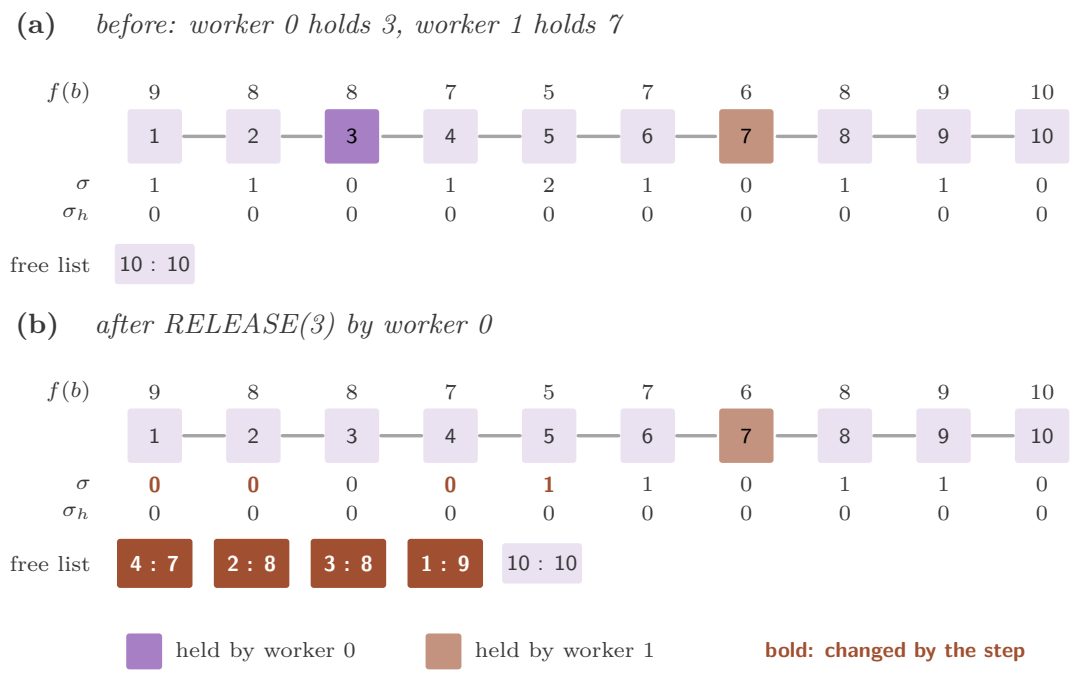


Figure 11.6. RELEASE on the state reached in Figure 11.5, after some expansions in the nblock 3. (a) Workers 0 and 1 hold 3 and 7. (b) Worker 0 releases 3, and σ drops by one on $I(3)$: 1, 2 and 4 become free, and so does 3, while 5 stays out, since $\sigma(5) = 1$ because of the nblock 7. The new entries of the free list (in the accent colour) keep the order of f, ties broken by the smaller nblock.

without instruments, somewhat larger values of m gave the best times, and $m = 32$, the default of the library, is the value of their experiment on the size of the abstraction (Figure 5, p. 704). Second, a worker whose nblock is not empty does not wait for the lock when it would switch: it tries the lock and, if another worker holds it, it usually goes on with its nblock (line 25 of Algorithm 11.2, in section 11.5). Third, the first goal found is not necessarily optimal (with several workers it may well not be), and the search goes on until no OPEN list holds a node better than the incumbent. It ends when no nblock is in use and the free list is empty (line 32), which section 11.6 proves to be the right condition.

11.4 Livelock and the hot nblocks

In plain PBNF a worker chooses its next nblock greedily among the free ones, and nothing forces the nblocks of $I(b)$ to be out of use at the same time. Hence, the nblock b that holds the best nodes may never become free. Burns et al. found such an execution with a model checker ([3], section 3.2.3, p. 699), and the following proposition gives one explicitly.

Proposition 11.6 (Plain PBNF may livelock). There are an infinite state space with an admissible and consistent heuristic, an abstraction, and an execution of plain PBNF with two workers in which a node of least f is never expanded and the search never ends, although a goal is reachable.

Proof. We take the state space of Figure 11.7 with $h \equiv 0$ (and hence $f = g$), the abstract graph being the path of the nblocks $1, \dots, 9$: the root s_0 in the nblock 4, the states v, t and the goal γ in 5, x in 6, and two infinite chains $u_1, u_2, \dots$ in 3 and $y_1, y_2, \dots$ in 7, with the costs of the figure. Condition (i) of Definition 11.1 holds, since each edge stays in an nblock or joins two adjacent ones. We let $m = 1$ and consider the following execution. Worker 1 acquires 4 (the only free nblock) and expands s_0, which puts u_1 into OPEN(3) and v into OPEN(5). It releases 4 and acquires 5, the better of the free 3 and 5, while worker 0 finds no free nblock (3 is in $I(5)$) and waits. Worker 1 expands v, which puts x into OPEN(6), releases 5 and acquires 6, and worker 0, woken by the release, acquires 3, whose scope $\{2, 3, 4\}$ is disjoint from $S(6) = \{5, 6, 7\}$. Worker 1 expands x, which puts t (with $f = 5$) into OPEN(5) and y_1 into OPEN(7). It then releases 6 and acquires 7, since 5 is not free ($\sigma(5) = 1$ because of the nblock 3) and 7 is. From now on $f(5) = 5$, while $f(3), f(7) \geq 12$ grow by one at each expansion. At each of its checks, worker 0 sees the nblock 5 of $I(3)$ better than its own, releases 3, which brings $\sigma(5)$ from 2 to 1, and, in the same critical section, acquires the first free nblock, which is 3 again (the nblocks 1, 2 and 4 are

empty, and 5 is not free). Worker 1 does the same with 7 (Figure 11.8). Since each release is followed by an acquisition under the same lock, $\sigma(5) \geq 1$ at every instant, the nblock 5 is never free, t is never expanded, and the chains are expanded forever. □

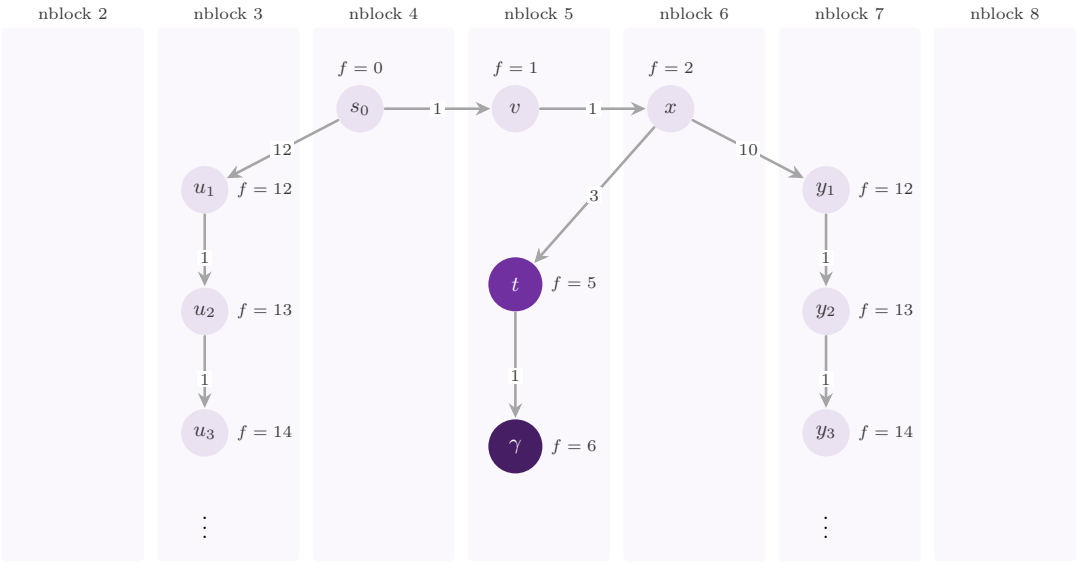


Figure 11.7. The state space of Proposition 11.6, drawn in the columns of its nblocks (the path $1 - 2 - \cdots - 9$; the empty nblocks 1 and 9 are not drawn). With $h \equiv 0$, $f = g$ is written next to each state. The node t (in the primary colour) has the least f among the nodes generated after x, and the goal γ is reached only through it. The chains of the nblocks 3 and 7 are infinite.

Note that one worker alone cannot livelock in this way, since when it releases its nblock σ drops to 0 on the whole interference scope. Also, the execution above needs a particular interleaving of the two workers (which a scheduler may, or may not, produce). We remark that a finite state space cannot livelock (Theorem 11.13 below), although it may well pay some search overhead for the same reason, since the "best" nblock may stay blocked for rather long.

The library keeps plain PBNF as an option, `safe` false of `PBNFOptions`, for the comparisons with Safe PBNF: a worker then switches only when the best free nblock is better than its own, and it never looks at the nblocks of its interference scope, which spares the scan of $I(b)$ at each check but leaves the livelock of Proposition 11.6 possible on an infinite state space. The changes to the pseudocode are given in section 11.5, and the results of section 11.6 that hold for plain PBNF are listed there.

Safe PBNF (section 3.1 of [3], pp. 697-699) prevents the livelock by what they call the hot nblocks. At its check, a worker that holds b looks also at the best nblock b' of $I(b)$. If b' is better than b and better than the best free nblock, the worker sets b' *hot*, i.e., it increments σ_h on $I(b')$, and no nblock of $I(b')$ is free any more and none can be acquired. Then it releases b, and each other worker that holds an nblock of $I(b')$ releases it at its next check. Hence $\sigma(b')$ can only decrease, and it eventually reaches 0. When it does, b' is set cold and freed, with the least key of the free list. To keep this from blocking too much of the graph, at most one nblock of an interference scope is hot. Indeed, an nblock is not set hot if a better hot nblock interferes with it, and setting it hot cools the worse hot nblocks that interfere with it. These are the three cases of Burns et al. for undirected abstract graphs (section 3.1, p. 698), whose two further cases concern directed graphs, which the library does not support. Also, a worker that keeps its nblock at a check, because no nblock of $I(b)$ is better than b, cools the hot nblocks of $I(b)$, which would otherwise wait for an nblock that is not worse than them.

In Figures 11.9 and 11.10 we run Safe PBNF from the state of Figure 11.8 (a). The first check of worker 0 sets 5 hot, and two releases later worker 1 acquires 5. Section 11.6 proves what holds in general.

11.5 The pseudocode

The pseudocode of the library (the Doxygen comment of `SafePBNF.h`) follows appendix A of [3] (pp. 737-739), with the conventions of §1.4. We split it here into three parts, which keep the numbers of its lines. The nblock graph is protected by one lock, m is `min_expansions`, and $f(b)$ of an nblock in use is the value that its worker publishes after each expansion.

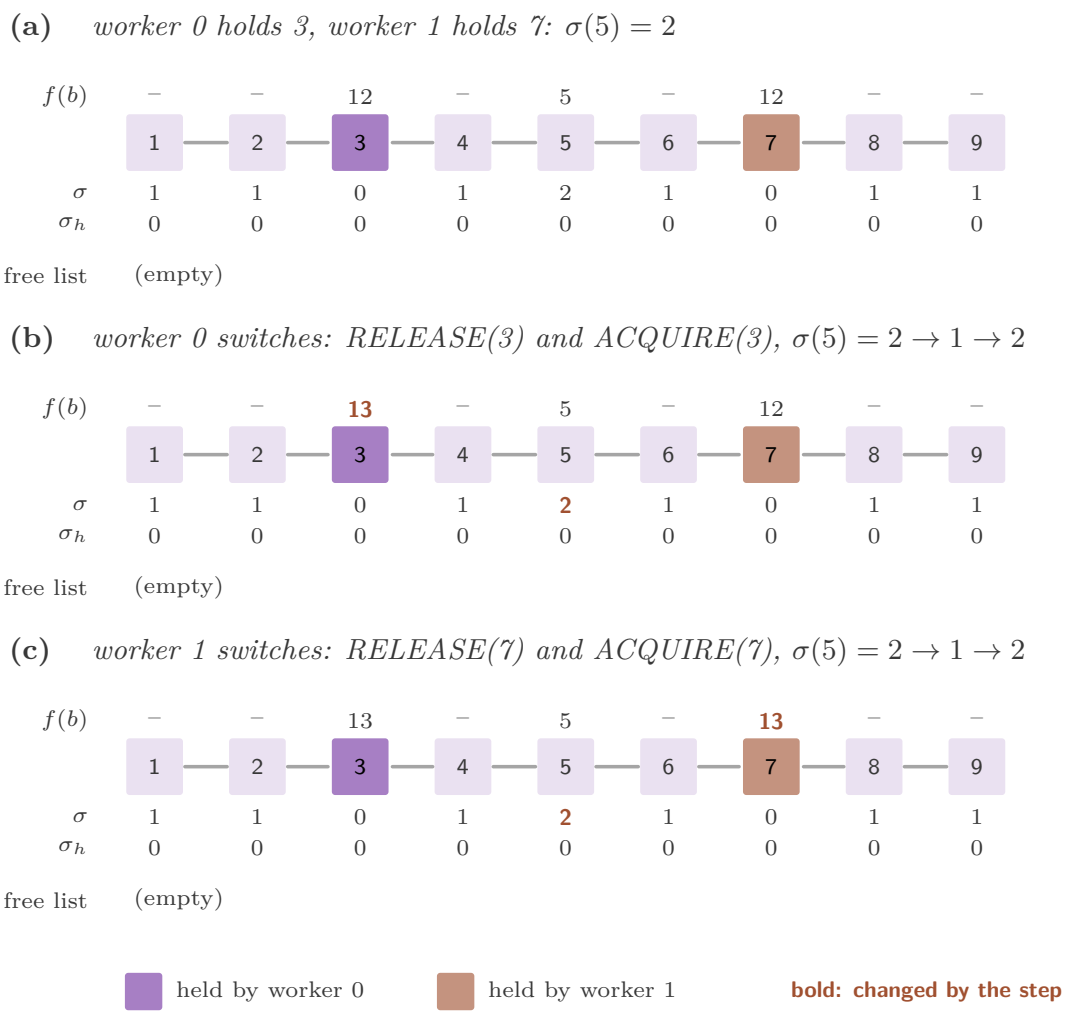


Figure 11.8. The livelock of plain PBNF on the state space of Figure 11.7, from the instant at which worker 1 acquires 7 ("–" is an empty OPEN list). (a) Worker 0 holds 3 and worker 1 holds 7, and $\sigma(5) = 2$. (b) Worker 0 switches, since $f(5) < f(3)$, but the release of 3 and the acquisition of the first free nblock, which is 3 again, happen under one lock, and $\sigma(5)$ never reaches 0. (c) Worker 1 does the same with 7, and the state is that of (a) with larger values of f.

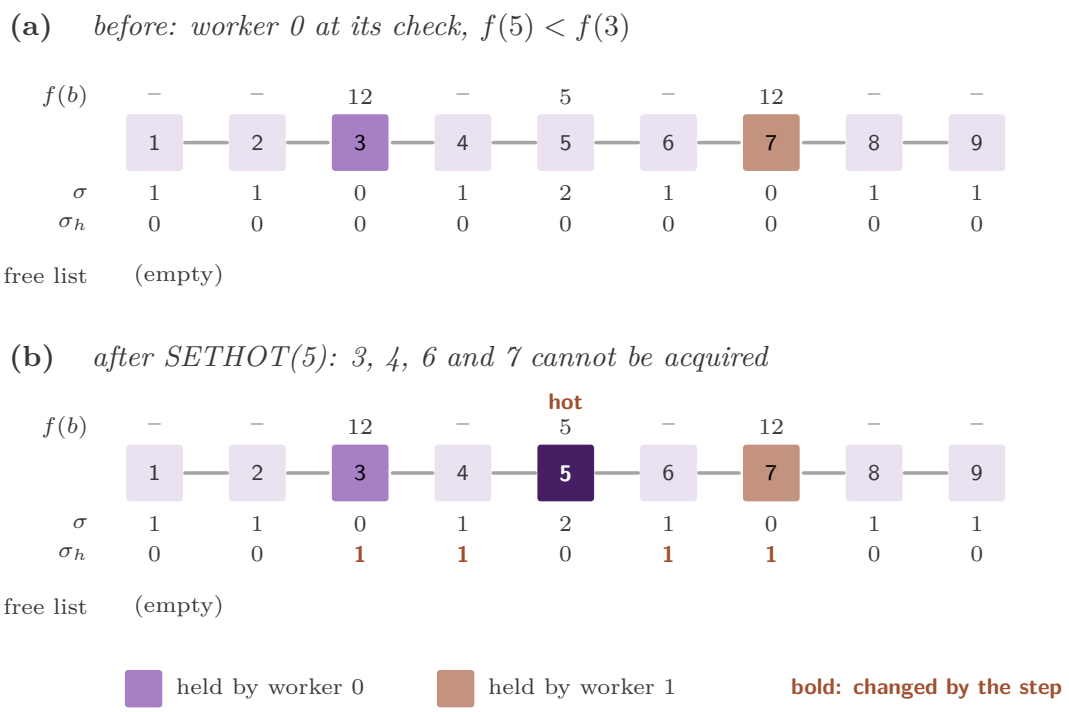


Figure 11.9. SETHOT on the state of Figure 11.8 (a). (a) Worker 0, at its check, finds $f(5) < f(3)$, and the free list is empty. (b) It sets 5 hot: $\sigma_h = 1$ on $I(5) = \{3, 4, 6, 7\}$, and none of these nblocks can be acquired, even when it is not in use, until 5 is cold again.

Algorithm 11.1 (Safe PBNF: the worker).

```
 1  procedure THREADSEARCH()                          ▷ on each worker
 2     b ← nil
 3     loop
 4        b ← NEXTNBLOCK(b);  if b = nil then return
 5        k ← 0
 6        while not SHOULDSWITCH(b, k) do
 7           u ← the first node of OPEN(b), removed (stale entries dropped)
 8           if f(u) ≥ z then OPEN(b) ← ∅;  break
 9           close u
10           if u is a goal then z ← min(z, g(u)), recording u if g(u) < z
11           else for each successor v of u with g(u) + c(u,v) + h(v) < z do
12              insert v in the lists of its nblock, as A* does
13           k ← k + 1;  publish f(b)
```

Before the first call, the root s_0 is inserted in OPEN($\phi(s_0)$) and its nblock is put in the free list (SafePBNF.h:299-306). Line 7 drops the stale entries of OPEN, i.e., those whose node has been closed or has received a smaller g since the entry was pushed (the lazy deletion of §3.1). Line 8 empties OPEN(b) when its first node is not better than the incumbent, since then no node of the list is (e.g., after a goal of cost close to the optimum has been found, for an nblock that holds only worse nodes). We make the goal test at the selection (line 10), as in the other searches of the library, under a separate lock that protects the incumbent and its node. Line 11 prunes at the generation the children that cannot be better than z. Line 12 is the insertion of A* (a lookup in the CLOSED list of $\phi(v)$, a new node or a reopened one if the path is shorter, and a push on OPEN($\phi(v)$)), made with no lock, since $\phi(v) \in S(b)$ by Lemma 11.3. Line 13 publishes $f(b)$, an atomic value that the other workers read without the lock at lines 18 and 28. The insertions into an nblock $a \neq b$ lower its published value as well.

Algorithm 11.2 (Safe PBNF: the switch).

```
14  procedure SHOULDSWITCH(b, k)
15     if OPEN(b) = ∅ then return true
16     if k < m then return false
```

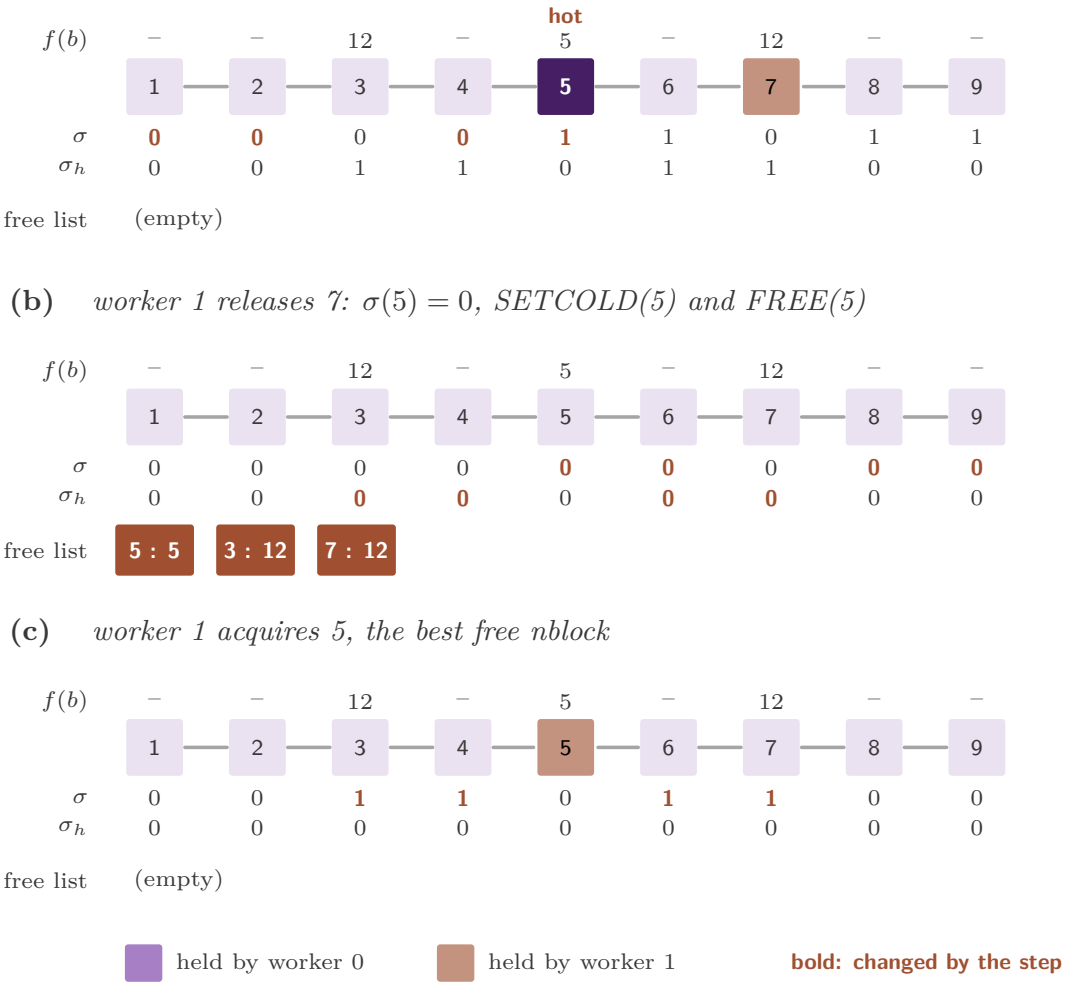


Figure 11.10. Safe PBNF after Figure 11.9 (b). (a) Worker 0 releases 3, which is not free ($\sigma_h(3) = 1$). The free list is empty, and worker 0 waits. (b) At its check, worker 1 releases 7: $\sigma(5)$ drops to 0, 5 is set cold, σ_h drops to 0 on $I(5)$, and 5, 3 and 7 enter the free list, 5 first. (c) Worker 1 acquires 5, and the nodes t and γ of Figure 11.7 are expanded next.

```
17     k ← 0;  lock the graph
18     b' ← the nblock of I(b) of least published f
19     if f(free list) < f(b) or f(b') < f(b) then
20        if f(b') < f(free list) then SETHOT(b')
21        return true
22     SETCOLD each hot nblock of I(b);  return false
23  procedure NEXTNBLOCK(b)
24     if b ≠ nil, OPEN(b) ≠ ∅ and b was not just set hot then
25        if the graph is locked by another worker then return b
26     lock the graph
27     if b ≠ nil then
28        if f(b) < z, f(b) ≤ f(b'), f(b) ≤ f(free list) and b was not
             just set hot then return b
29        RELEASE(b)
30     drop from the free list the nblocks with no node of f < z
31     while the free list is empty do
32        if no nblock is in use then done ← true;  wake all;  return nil
33        sleep until woken
34     b ← the first nblock of the free list;  ACQUIRE(b);  return b
```

The check (lines 17 to 22) is made after m expansions at least, and always when OPEN(b) is empty (line 15). Line 19 is the switching rule of PBNF, and line 20 is the addition of Safe PBNF to it. The best interfering nblock b' is set hot if no free nblock is as "good", since otherwise the worker can switch to a free nblock that is at least as good, since a free nblock can be acquired at once, while a hot one waits for the releases of the others. "Just set hot" in lines 24 and 28 means that the check before the call has called SETHOT (the code sets the flag even when the call returns at once because b' was already hot). Then the worker always takes the lock and releases b, and this release eventually frees the hot nblock. Line 25 is the try-lock of PBNF, and line 28 tests again, under the lock, the condition of line 19 on the values of that moment: a worker keeps b if, in the meantime, it has become the best again. Line 30 drops from the free list the nblocks whose key is not below the incumbent (they were free when z was larger), which is a scan of the entries with a key not below z only. Lines 31 to 33 are the end and the wait: a worker that finds no free nblock sleeps on a condition variable until FREE wakes it, and the last worker to find no nblock in use declares the end.

With `safe` false (plain PBNF), line 18 reads no scope, the test of line 19 becomes $f(\text{free list}) < f(b)$, lines 20 and 22 are not executed, and the test of line 28 drops $f(b) \le f(b')$; a worker therefore never sets an nblock hot, $\sigma_h \equiv 0$, and it keeps its nblock unless a free one is better (`SafePBNF.h:655-725`). The check takes the lock of the graph as often as in Safe PBNF, but reads no interference scope under it.

The weight $w > 1$ of `SearchOptions` makes the search wPBNF (section 6 of [3]): the OPEN lists, the values $f(b)$ and the free list are ordered by $f' = g + wh$, line 8 empties OPEN(b) when its first node has $f' \ge z$ and skips a node of $wf \ge z$, line 11 prunes the successors of $wf \ge z$, and lines 28, 30 and 60 compare $f(b)$, now the least f', with z; the path found costs at most wC^*. The anytime search (anytime weighted PBNF) skips at line 8 the nodes of $f \ge z$ only, prunes by $f \ge z$ at line 11, keeps free every nblock with a node, and ends with an optimal z. With a `schedule` of weights the search is Anytime Restarting PBNF (ARPBNF, section 7.1 of [3]): each iteration is a wPBNF with the weight of the schedule that prunes by $f \ge z$ only and keeps the nodes of $f' \ge z$ for the next iterations, a closed node whose g decreases goes to a list INCONS of its nblock instead of OPEN, as in ARA*, and the iteration ends at line 32; then the worker that finds no nblock in use, with the graph locked, orders OPEN ∪ INCONS of every nblock by the next weight, empties CLOSED, records the incumbent with the bound $\min(z, \min f)$ over OPEN ∪ INCONS, and frees the nblocks again, until the bound reaches z (`SafePBNF.h:528-574`). Burns et al. re-sort at each new incumbent, in parallel; the library re-sorts at the end of each iteration, when no worker holds an nblock, which needs no protocol. These searches, their bounds and their proofs belong with the bounded-suboptimal and anytime searches of Chapter 6, and in the rest of this chapter $w = 1$ and the schedule is empty.

Algorithm 11.3 (Safe PBNF: the operations on the nblock graph, all under its lock).

```
35  procedure ACQUIRE(b)
36     b leaves the free list and is in use
37     for each n ∈ I(b) do σ(n) ← σ(n) + 1;  n leaves the free list
38  procedure RELEASE(b)
39     b is not in use
```

```
40     for each n ∈ I(b) do
41        σ(n) ← σ(n) − 1
42        if σ(n) = 0 then
43           if σ_h(n) = 0 and n is hot then SETCOLD(n)
44           FREE(n)
45     FREE(b)
46  procedure SETHOT(b)
47     if b is hot, σ(b) = 0, or a hot nblock of I(b) has a smaller f
          then return
48     b is hot
49     for each n ∈ I(b) do
50        if n is hot then SETCOLD(n)
51        n leaves the free list;  σ_h(n) ← σ_h(n) + 1
52  procedure SETCOLD(b)
53     b is not hot
54     for each n ∈ I(b) do
55        σ_h(n) ← σ_h(n) − 1
56        if σ(n) = 0 and σ_h(n) = 0 then
57           if n is hot then SETCOLD(n)
58           FREE(n)
59  procedure FREE(n)
60     if n is not in use, σ(n) = σ_h(n) = 0 and f(n) < z then
61        put n in the free list, with the key f(n);  wake all
```

Note that FREE is the only way into the free list, and it tests exactly the condition of Definition 11.5, computing $f(n)$ from OPEN(n) (no worker can touch OPEN(n) at that moment, Theorem 11.8). SETHOT refuses an nblock with $\sigma(b) = 0$ (which no nblock in use keeps from being free) and one that interferes with a better hot nblock (case 2 of Burns et al.). Otherwise, it cools the worse hot nblocks of $I(b)$ (case 3) and blocks the whole of $I(b)$. Line 43 cools a hot nblock when the last nblock in use of its interference scope is released, which is the event that the hot flag waits for, and line 57 is a guard that Lemma 11.7 shows never to fire.

Our pseudocode differs from appendix A of [3] in six details, none of which changes the arguments of section 11.6. First, appendix A prunes a single node with f not better than the incumbent, while line 8 empties the whole list, which is equivalent since OPEN is ordered by f. Second, an nblock enters the free list only if $f(n) < z$, and line 30 removes those that the incumbent has made useless, while appendix A admits any nblock that is not empty. Third, line 28 keeps b also on ties ($\leq$ instead of $<$). Fourth, ACQUIRE takes the nblocks of $I(b)$ out of the free list explicitly. Fifth, $I(b)$ does not contain b (in appendix A, whose interference scope is built with the self-loops, it does): being in use is a flag of its own, and RELEASE puts b itself back with FREE at line 45. Finally, the end is detected by the number of nblocks in use instead of the condition $\sigma \equiv 0$ of appendix A; the former implies the latter by Lemma 11.7 (a), and it is the right one in the library, since an nblock in use with an empty interference scope (e.g., the only nblock of a trivial abstraction) leaves $\sigma \equiv 0$.

11.6 Analysis

We now prove the properties of the search. Our arguments are about the states of the nblock graph between two critical sections, i.e., at the instants at which no worker holds its lock. The operations under the lock (Algorithms 11.2 and 11.3, and the insertion of the root) are atomic with respect to each other, while the expansions (lines 7 to 13) run in parallel with the rest.

Lemma 11.7 (Invariants of the nblock graph). At each instant at which the lock of the graph is free, the following hold.

- (a) $\sigma(n)$ is the number of the nblocks in use in $I(n)$.
- (b) $\sigma_h(n)$ is the number of the hot nblocks in $I(n)$.
- (c) Each nblock n of the free list is not in use and has $\sigma(n) = \sigma_h(n) = 0$, and its key is the least f of OPEN(n).
- (d) No two nblocks in use interfere, i.e., their scopes are disjoint.
- (e) Each hot nblock b has $\sigma(b) \geq 1$ and $\sigma_h(b) = 0$, i.e., it interferes with an nblock in use and with no hot nblock.
- (f) Each nblock n that is not in use, has $\sigma(n) = \sigma_h(n) = 0$ and has a node of $f < z$ in OPEN(n) is in the free list.

Proof. We proceed by induction on the critical sections. At the start no nblock is in use or hot, all the counters are 0, and the only nblock with a node is that of s_0, which is put in the free list; hence (a) to (f) hold.

(a) and (b). ACQUIRE(b) and RELEASE(b) change σ by one on $I(b)$ exactly when b starts and stops being in use, and SETHOT(b) and SETCOLD(b) change σ_h by one on $I(b)$ exactly when b becomes hot and cold (SETHOT returns at line 47 when b is already hot, and SETCOLD is called only on a hot nblock, at lines 22, 43, 50 and 57). Hence, we have that $\sigma(n)$ counts the nblocks c in use with $n \in I(c)$, which by Lemma 11.4 are the nblocks in use of $I(n)$. The same holds for σ_h.

(c). An nblock enters the free list only through FREE, which checks the condition and takes the key from OPEN(n). It leaves the list at its own ACQUIRE, at the ACQUIRE of an nblock of $I(n)$ (line 37, when $\sigma(n)$ becomes positive), at the SETHOT of an nblock of $I(n)$ (line 51, when $\sigma_h(n)$ becomes positive), and at line 30. While it is in the list, $\sigma(n) = 0$ and n is not in use, and therefore, by (a), no nblock in use has n in its scope (if $n \in S(c)$ with $c \neq n$, then $n \in \mathrm{succ}(c) \subseteq I(c)$ by Lemma 11.4); hence, no worker writes OPEN(n), and the key stays exact.

(d). ACQUIRE takes the first nblock b of the free list, for which $\sigma(b) = 0$ by (c), i.e., no nblock of $I(b)$ is in use, by (a).

(e). SETHOT(b) makes b hot only if $\sigma(b) \geq 1$ (line 47), and it cools the hot nblocks of $I(b)$ (line 50), after which $\sigma_h(b) = 0$ by (b). A hot b keeps $\sigma_h(b) = 0$, since an nblock $x \in I(b)$ becomes hot only through SETHOT(x), which cools b first at line 50, unless it returns at line 47. Also, $\sigma(b)$ decreases only at line 41 of a RELEASE, and when it reaches 0 the nblock, hot with $\sigma_h(b) = 0$, is cooled at line 43. By (e), the condition of line 57 (a hot nblock with $\sigma = 0$) never holds at the start of a critical section, and SETHOT and SETCOLD do not create it; hence, line 57 is never executed.

(f). The nodes of OPEN(n) are written only by the worker that holds n or an nblock c with $n \in S(c)$ (lines 7 to 13), i.e., while n is in use or $\sigma(n) \geq 1$, by (a) and Lemma 11.4. Each event that brings n to "not in use and $\sigma(n) = \sigma_h(n) = 0$" calls FREE($n$): the release of n (line 45), the release of the last nblock in use of $I(n)$ (line 44), and the cooling of the last hot nblock of $I(n)$ (line 58). FREE puts n in the list if $f(n) < z$. If not, n has no node of $f < z$, and it gets none until it is in a scope again, since z only decreases. Finally, n leaves the list only when one of the three conditions fails, or at line 30 when its key, which is exact by (c), is not below z. □

Indeed, Lemma 11.7 (e) is Lemma 1 of Burns et al. ("If an nblock n is hot, there is at least one other nblock in its interference scope that is in use. Also, n is not interfering with any other hot nblocks", [3], p. 700), proved here on the pseudocode of the library. As a first consequence, we can show that the lists of an nblock have one writer at a time, which is the purpose of the whole construction.

Theorem 11.8 (Exclusive access). At each instant, the OPEN and CLOSED lists of an nblock n are read or written by at most one worker outside the lock of the graph, namely the worker that holds the nblock c with $n \in S(c)$, if there is one. Under the lock, they are read only by that worker or when no such worker exists.

Proof. We first observe that a worker that holds c reads and writes, in lines 7 to 13, the lists of c (lines 7 and 8) and those of the nblocks $\phi(v)$ of the children of the nodes of c, which lie in $S(c)$ by Lemma 11.3. If two workers held $c \neq c'$ with $n \in S(c) \cap S(c')$, then c and c' would interfere, against Lemma 11.7 (d); hence, at most one worker touches the lists of n. Under the lock, OPEN(n) is read by FREE (line 60), which requires n not in use and $\sigma(n) = 0$, and therefore no nblock in use with n in its scope, by Lemma 11.7 (a), and at line 28 by the worker that holds n. □

Corollary 11.9 (Duplicate detection). Each state has at most one node in the search, and the test of line 12 that compares a child with the CLOSED list of its nblock is the test of the sequential A*: a child is a duplicate if and only if a node of its state exists, and it is reopened if and only if its new path is shorter.

Proof. We store the node of a state v only in the CLOSED list (the table of the nodes) of $\phi(v)$, which by Theorem 11.8 has one writer at a time; the lookup and the insertion of line 12 are therefore those of a sequential table. □

One can therefore say that the duplicate detection is "perfect" in the sense of SDD, and no node is sent from a worker to another: the only shared writes outside the lock of the graph are the published values $f(b)$ (atomic), the incumbent (under a separate lock) and the creation of an nblock the first time that it is touched (by a compare-and-swap, section 11.7). Since the search has more than one lock, the absence of deadlock needs a (short) argument.

Proposition 11.10 (No deadlock). No worker waits forever for a lock, and a worker that sleeps at line 33 is woken when an nblock enters the free list or the search ends.

Proof. We recall that the two locks of the search are those of the graph and of the incumbent; the creation of an nblock takes no lock (Algorithm 11.4). The second lock is taken only in the goal test of line 10, which holds no other lock and takes no other, while the node limit takes the lock of the graph and holds no other; the end of an iteration of ARPBNF (section 11.5) takes the lock of the incumbent under that of the graph, and no worker takes the lock of the graph while it holds that of the incumbent. Hence the locks are always taken in the order graph, incumbent, and a cycle of waits is impossible. The condition variable of line 33 is waited on under the lock of the graph and notified under it by FREE and at line 32, so that no notification is lost. Also, a worker that sleeps has released its nblock at line 29, or had none. □

This is section 3.2.2 of Burns et al. for the one lock of their pseudocode, which "the thread that currently holds it never attempts to acquire a second time" (p. 699). We now come to the optimality, for which one has to read the argument of A* on the union of all the OPEN lists.

Lemma 11.11 (An optimal path is always represented). Let h be admissible, let $P = (s_0 = w_0, w_1, \ldots, w_k)$ be an optimal path to a goal, and let $z > C^*$ at some instant. Then some w_j has a node with $g(w_j) = g^*(w_j)$ that is either in OPEN($\phi(w_j)$) or being expanded, and therefore $f(w_j) \le C^* < z$.

Proof. We let j be the largest index such that w_j has a node with $g(w_j) = g^*(w_j)$; it exists, since s_0 has $g = 0$ from the start, and g of a node never increases. Suppose that the node of w_j is neither in OPEN nor being expanded: then its expansion with $g = g^*(w_j)$ has completed. If $j = k$, the goal w_k was selected with $g = C^*$ at line 10, and $z \le C^*$. If $j < k$, the expansion generated w_{j+1} with $g = g^*(w_{j+1})$, which line 11 did not prune, since $g^*(w_{j+1}) + h(w_{j+1}) \le C^* < z$ by admissibility; then the node of w_{j+1} received $g = g^*(w_{j+1})$ at line 12, or had it already (Corollary 11.9), against the choice of j. Hence, we conclude that the node of w_j is in OPEN or being expanded, and $f(w_j) = g^*(w_j) + h(w_j) \le C^*$. Line 8 does not remove it, since it empties OPEN only when its first node, and hence each node, has $f \ge z$. □

Theorem 11.12 (Optimality). Let h be admissible. If `SafePBNF` ends with the status `solved`, the path that it returns is optimal. If it ends with the status `unsolvable`, no goal is reachable.

Proof. We recall that the search ends at line 32, when no nblock is in use and the free list is empty (line 31, after line 30). Then $\sigma \equiv 0$ by Lemma 11.7 (a), no nblock is hot by (e), and therefore $\sigma_h \equiv 0$ by (b); by (f), each nblock with a node of $f < z$ in OPEN is in the free list, which is empty. No node is being expanded, since an expansion happens only in an nblock in use. Hence, if a goal were reachable and $z > C^*$, the node of Lemma 11.11 would be in some OPEN with $f < z$, a contradiction. Therefore, $z \le C^*$ if a goal is reachable, and $z = +\infty$ (the status `unsolvable`) only if none is. The incumbent z is the value g of a goal node, and the path is rebuilt from that node through the parents after all the workers have returned. Along this chain $g(v) \ge g(\text{parent}(v)) + c(\text{parent}(v), v)$, since the parent's g can only have decreased after it was used; hence g decreases strictly along the chain (the costs are positive), the chain is acyclic, and its cost is at most $z \le C^*$, i.e., it is optimal. □

This is Theorem 3 of Burns et al. (p. 701), in whose short proof one reads that "the search continues to expand nodes until the minimum f value among all frontier nodes is greater than or equal to the incumbent". The part that needs the invariants is that the end condition of line 32 implies it, i.e., implies the condition of the end of a parallel search of Theorem 8.6: no entry in flight that is not stale has $f < z$, i.e., no such entry in an OPEN list, no worker expanding a node, and no node in transit, the last being trivial here, since a node is never in transit.

Theorem 11.13 (Termination on a finite state space). Let the state space reachable from s_0 be finite, with positive costs. Then `SafePBNF` ends, with any number of workers and any $m \ge 1$, under any scheduling of the threads in which a worker that can proceed eventually does (the assumption (A1) below).

Proof. A node is expanded once for each value that its g takes, and g takes strictly decreasing values, each the cost of the path of its parents at the time, which is simple since g grows strictly along it; as the graph is finite, the simple paths are finitely many, and so are the expansions and the insertions into OPEN. Between two acquisitions, a worker expands at least one node or empties a list: after ACQUIRE, $k = 0 < m$ and OPEN(b) holds a node of $f < z$ by Lemma 11.7 (c) (line 30 has just removed the keys not below z), and line 6 lets it expand (or, if a goal has lowered z meanwhile, which happens under the lock of the incumbent and not under that of the graph, line 8 empties the list). Also, a worker that keeps its nblock at line 25 or 28 returns to line 6 with $k = 0$. Each emptying of a list at line 8 removes at least one entry, and the entries are finitely many. Hence, we have finitely many critical sections as well, and each of them ends. A worker that sleeps at line 33 holds no nblock, and if no other worker holds one, the last one to look at the free list finds it empty with no nblock in use and ends the search (line 32), and by Proposition 11.10 no worker is left asleep while an nblock is free. □

Burns et al. prove completeness for an admissible heuristic or a finite space (Corollary 1, p. 701), relying for the former on their Theorem 2. On an infinite space the argument above fails at its first step (and a node limit

is then the only "safeguard" that one may rely on), and termination needs that the node of Lemma 11.11 be expanded in finite time, which is the progress property that the hot nblocks are meant to give. It also needs, as usual, costs bounded below by some $\varepsilon > 0$, as the sequential A* does (Chapter 2). In the following theorem we state what the pseudocode of the library guarantees for a hot nblock. It needs three assumptions on the execution, which do not follow from the pseudocode (although they are, arguably, rather mild):

- (A1) each worker is scheduled, and each expansion takes a finite time, so that a worker that holds an nblock reaches its next check (line 14 with $k \geq m$, or line 15 with an empty list) in finite time;
- (A2) the lock is fair, and the try-lock of line 25 does not fail at infinitely many consecutive attempts of a worker;
- (A3) at each check, the published value of each nblock is the least f of its OPEN list, and the test of line 28 gives the same answer as the test of line 19 of the same check.

The first two concern the threads and the lock, and they usually hold for the threads of C++ and of OpenMP (although neither standard promises a fair lock). Instead, (A3) closes the window between the check, whose lock is released at its end, and NEXTNBLOCK, which takes the lock again (the same window is in appendix A, whose SHOULDSWITCH reads the values without the lock).

Theorem 11.14 (A hot nblock is freed or overtaken). Assume (A1) to (A3), and let the nblock b be hot at some instant. Then, after finitely many checks, one of the following happens.

- (i) $\sigma(b)$ reaches 0, and b is set cold and freed (lines 42 to 44), entering the free list if it has a node of $f < z$.
- (ii) b is set cold at line 22 by a worker whose nblock $c \in I(b)$ has $f(c) \leq f(b)$.
- (iii) b is set cold at line 50 by SETHOT(x) for some $x \in I(b)$ with $f(x) \leq f(b)$.

Proof. We first show that, while b is hot, $\sigma(b)$ does not increase: each $n \in I(b)$ has $\sigma_h(n) \geq 1$ by Lemma 11.7 (b), and hence it is not in the free list, by (c); and an nblock is acquired only from the free list; hence no nblock of $I(b)$ starts being in use. Let $c \in I(b)$ be in use, held by the worker i. By (A1), i reaches a check in finite time. If OPEN(c) is empty, i calls NEXTNBLOCK, which takes the lock (lines 24 and 26) and releases c, since $f(c) = +\infty$ fails line 28. Otherwise, let b' be the best nblock of $I(c)$; since $b \in I(c)$ by Lemma 11.4, $f(b') \leq f(b)$. If the test of line 19 fails, then $f(c) \leq f(b') \leq f(b)$, and line 22 cools b, which is case (ii). If it succeeds and $f(b') < f(\text{free list})$, line 20 calls SETHOT($b'$), which either returns at once ($b' = b$, or b' already hot, or a better hot nblock interferes with b') or makes b' hot, cooling b at line 50 if $b \in I(b')$ (case (iii), since $f(b') \leq f(b)$). In all the other sub-cases the flag "just set hot" makes NEXTNBLOCK take the lock and release c. If it succeeds and $f(\text{free list}) \leq f(b')$, NEXTNBLOCK obtains the lock by (A2), and line 28 fails by (A3); hence c is released. Each release of an nblock of $I(b)$ decreases $\sigma(b)$ by one, and $\sigma(b) \leq p$, since at most p nblocks are in use; hence, unless (ii) or (iii) happens first, after at most $\sigma(b)$ checks of these workers $\sigma(b) = 0$, at which point line 43 cools b (which has $\sigma_h(b) = 0$ by Lemma 11.7 (e)) and line 44 frees it, which is case (i). Lines 43 and 57 are the only other places where b could be cooled, and the second is never executed, by Lemma 11.7. □

Note that in cases (ii) and (iii) the hot flag passes to, or is overridden by, an nblock that is not worse than b and is in use or hot. When b holds the unique best nodes of the search, neither can happen, and the theorem gives the progress for which Safe PBNF is designed.

Corollary 11.15 (Progress with a unique best nblock). Assume (A1) to (A3) and a consistent h. Let $\varphi < z$ be the least f of the nodes in the OPEN lists at some instant, and let all the nodes with $f = \varphi$ lie in one nblock B, not in use. Then the following hold.

- (a) If B is free, a worker acquires B at the next acquisition, and one happens after finitely many checks.
- (b) If B is hot, a worker acquires B after finitely many checks.
- (c) If $\sigma(B) \geq 1$, then after finitely many checks B is hot, or it is free, or $\sigma(B) = 0 < \sigma_h(B)$.

Proof. We first note that, since h is consistent, a child has an f not smaller than that of its parent; hence, until B is acquired, no node of $f \leq \varphi$ is generated outside B, and the other nblocks have $f > \varphi$. (a) While B is free it is the first of the free list, and it can leave the list only when it is acquired: an nblock of $I(B)$ cannot be acquired before it (it is behind B in the list), and SETHOT(x) with $B \in I(x)$ needs $f(x) < f(\text{free list}) = \varphi$ at line 20, which no $x \neq B$ has. A worker that sleeps was woken when B entered the list, and a worker that holds an nblock c finds at its next check, by (A1), $f(\text{free list}) = \varphi < f(c)$, does not set anything hot ($f(b') > \varphi$, since $\sigma(B) = 0$ means that no nblock in use interferes with B), and releases c by (A2) and (A3), reaching line 34. (b) By Theorem 11.14, since cases (ii) and (iii) need an nblock other than B with $f \leq \varphi$, B is freed in case (i); the worker that releases the last nblock in use of $I(B)$ goes on, under the same lock, to line 34, where B is the first of the free list. (c) Let $c \in I(B)$ be in use. At its check, the best of $I(c)$ is B (by Lemma 11.4, $B \in I(c)$), with

$\varphi < f(c)$, and $f(\text{free list}) > \varphi$ since B is not free; hence line 20 calls SETHOT(B), which makes B hot unless $\sigma(B)$ has dropped to 0 in the meantime (no hot nblock of $I(B)$ has a smaller f). In the latter case B is free if $\sigma_h(B) = 0$, and the third alternative holds otherwise. □

Remark 11.16 (What is taken from Burns et al.). The case that we leave open in Corollary 11.15 is that of an nblock B with the best nodes, no nblock in use around it, and a worse hot nblock x in $I(B)$: by Theorem 11.14, x is freed or overtaken in finite time, which decreases $\sigma_h(B)$, but new hot nblocks may appear in $I(B)$ meanwhile, and we have no bound on such a sequence. Burns et al. state the progress property as their Theorem 2 ("A node n with minimum f value will eventually be expanded", pp. 700-701), whose third case (an nblock kept from being free by an nblock in use) is reduced to their Theorem 1 ("If an nblock n becomes hot, it will eventually be added to the free list and will no longer be hot", p. 700). The latter is proved, in TLA$^+$, on a model in which a search step only chooses an nblock to set hot and then releases the nblock that it holds (section 3.2.3, p. 699, and appendix B, pp. 740-743). In the pseudocode of appendix A, as in ours (line 22), a worker that keeps its nblock may also cool a hot nblock, which is why Theorem 11.14 has the cases (ii) and (iii). Burns et al. also checked with the model checker TLC that their model of Safe PBNF has no livelock with up to three threads and 12 nblocks on an undirected ring, and up to three threads and eight nblocks on a directed graph (p. 699). We take the livelock freedom of Safe PBNF beyond Corollary 11.15 from these results, within the limits of their model, and we do not claim more. In particular, no test of the library can check it, since a livelock needs an infinite state space.

It remains to state how the workers learn that the search is over. The flag `done` of the nblock graph is the only signal: it is set at line 32, or by a worker that reaches the node limit (under the lock of the graph, `SafePBNF.h:642-649`), and a worker returns only when it has read it, or at line 32. The following proposition shows that the detection is both sound (the flag is never set too early) and live (once it is set, every worker returns).

Proposition 11.17 (The detection of the end). (i) When `done` is set at line 32, no worker holds an nblock or expands a node, and no OPEN list holds an entry that is not stale with $f < z$; hence the incumbent is final, and Theorem 11.12 applies. (ii) Under (A1) and (A2) of section 11.6, once `done` is set (at line 32 or by the node limit), every worker returns in finite time. (iii) A worker that sleeps at line 33 when the search is not over is woken by the next FREE that puts an nblock in the free list, or by the setting of `done`.

Proof. (i) Line 32 runs under the lock of the graph, with the free list empty after line 30 and no nblock in use. No worker expands a node outside an nblock in use, and a worker inserts nodes only during an expansion; hence, no OPEN list changes until a worker acquires an nblock, which needs a free one. By Lemma 11.7 (f), as in the proof of Theorem 11.12, no nblock has a node of $f < z$, and the entries that are not stale are the nodes of the lists. (ii) A worker is in one of three places: asleep at line 33, in a check or a call of NEXTNBLOCK, or in the expansions of lines 7 to 13. The flag is set under the lock and followed by a notification of all the sleepers under the same lock (`wake all`); a sleeper wakes, takes the lock and returns at its next test of `done` (`SafePBNF.h:707-708`), and a worker that waits for the lock does the same once it has it. A worker in its expansions reaches a check in finite time by (A1); there, either OPEN(b) is empty and NEXTNBLOCK takes the lock, or SHOULDSWITCH takes it after m expansions and returns true on reading `done` (`SafePBNF.h:666-667`). Then NEXTNBLOCK takes the lock (a try-lock that fails only returns to the expansions, which reach the next check, and by (A2) it does not fail forever), does not keep b (its test reads `done`), releases it and returns at the test of `done`. (iii) The waits and the notifications are under the same lock, and the notification of FREE (line 61) follows the insertion into the free list; a worker tests the free list and the flag under the lock before each wait, and a spurious wakeup only repeats the test, so that no notification is lost. □

Note that the node limit leaves the worker that reaches it holding its nblock (it returns at once), so that the counter of the nblocks in use does not reach 0 afterwards; this is harmless, since every other worker returns on reading `done`, and the status is then `node_limit` whatever the incumbent (`SafePBNF.h:338-340`), as Exercise 11.5 asks to show.

The results of this section hold for plain PBNF (`safe` false) as well, but those on the hot nblocks. With no nblock ever hot, $\sigma_h \equiv 0$, Lemma 11.7 holds with (b) and (e) trivial, and its proof, Theorem 11.8, Corollary 11.9, Proposition 11.10, Lemma 11.11, Theorem 11.12, Theorem 11.13 and Proposition 11.17 use nothing else: the exclusive access, the optimality, the termination on a finite state space and the detection of the end are those of Safe PBNF. Theorem 11.14 and Corollary 11.15 do not hold, and Proposition 11.6 shows that the progress they give may fail on an infinite state space. The tests `pbnf-thread` and `pbnf-openmp` run plain PBNF on two instances of Korf and on a grid with 4 moves, and on the corner cases of `check_astar_edges`.

11.7 The implementation

`SafePBNF::solve()` (SafePBNF.h:290-343) builds the nblock graph and the workers, inserts the root, runs the workers on the executor, and then collects their counters and rebuilds the path. The timer of the `Solution` starts at the beginning of `solve()` and stops at its end. We now go through the parts of the code that are specific to PBNF, i.e., the creation of the nblocks, the free list and the expansion, with the cost and the memory of each; the OPEN lists, the tables of the nodes and the node arena are those of the sequential A* (Chapter 2 and Chapter 3). The pseudocode below continues the numbering of the lines of Algorithms 11.1 to 11.3.

The nblocks

An nblock (SafePBNF.h:354-369), which typically groups many states (400 cells on the grids of the benchmark), holds its OPEN list (the bucket queue for integer costs and the binary heap otherwise), its table of nodes (the CLOSED list, which starts with 16 slots and grows by doubling), its scope and its interference scope as vectors of ids, the counters σ and σ_h, the flags hot, in use and in the free list, its key in the free list, and the published value $f(b)$, an atomic read and written with relaxed order (i.e., with no ordering w.r.t. the other memory operations). The nblocks are created the first time that a worker touches them, so that only the visited part of the abstract graph costs memory (Burns et al. do the same, section 4, p. 702). This matters somewhat for the 15-puzzle, whose 4096 ids include 736 that are not abstract states, and much more for "large" grids, of which the search often reaches only a part. The nblocks are kept by the class template `NBlockMap< D , NB >` of NBlockMap.h, which the searches by layers of §12.2 share with Safe PBNF: an array A of N atomic pointers, all nil at the start (NBlockMap.h:88-92), the member `find( b )`, which returns $A[b]$ read with acquire order (nil if the nblock has not been built), `block( b )`, which builds the nblock at its first use, with no lock, as follows (NBlockMap.h:114-142), and `for_each( f )`, which calls `f` on every nblock built, by increasing index. The parameter `NB` is the nblock of the search, default constructible, with the vectors `scope` and `interferes`, which `block()` fills, and the index `id`, when it has one.

Algorithm 11.4 (`NBlockMap`: the creation of an nblock).

```
62  function BLOCK(b)                    ▷ the nblock b, built at its first use
63     x ← A[b], read with acquire order;  if x ≠ nil then return x
64     n ← a new nblock, private to the caller
65     T ← succ(b);  J ← T
66     for each a ∈ T do J ← J ∪ succ(a)      ▷ predecessors = successors
67     I(n) ← J \ {b}, sorted, each nblock once;  S(n) ← T, then b
68     OPEN(n) ← ∅;  CLOSED(n) ← a table of 16 slots;  σ(n) ← σ_h(n) ← 0
69     if A[b] = nil then A[b] ← n, by a compare-and-swap with
          acquire-release order;  return n
70     destroy n;  return A[b], as the compare-and-swap read it
```

Line 66 is Lemma 11.4, which holds only on an undirected abstract graph (condition (ii) of Definition 11.1); sorting $I(b)$ once makes the loops of Algorithm 11.3 visit the nblocks in a fixed order, which (arguably) makes an execution somewhat easier to follow in a debugger. Two workers that build the same nblock at once build the same one, and the first compare-and-swap publishes it while the other copy is dropped; hence `block()` may be called under any lock of the caller, and takes none.

Lemma 11.18 (The creation of the nblocks). For each b, at most one nblock is published in $A[b]$, every call of BLOCK(b) returns that nblock, and a worker that obtains it sees its scopes, counters and empty lists as line 68 wrote them.

Proof. We first note that $A[b]$ changes only at line 69, by a compare-and-swap from nil, which succeeds at most once; once not nil, it never changes, since the nblocks are destroyed only with the map, when `solve()` returns. A call returns at line 63 the pointer that it read, at line 69 the nblock that it has published, or at line 70 the pointer that its failed compare-and-swap read, which is the published one. Finally, the successful compare-and-swap of line 69 is a store with release order that follows all the writes of lines 64 to 68 in its thread, and the reads of line 63 and of a failed compare-and-swap at line 70 are loads with acquire order: by the memory model of C++, a load with acquire order that reads the value of a store with release order sees all the writes that precede the store in its thread. □

After the creation, the scopes of an nblock are only read, and its lists are written only by the worker of Theorem 11.8; a worker that loses the race of line 69 has spent the building of the scopes of one nblock, which happens at

most once per worker and nblock, since the nblock is published when its call returns. The size of the scopes, on which the cost of every operation of the graph depends, is bounded as follows.

Lemma 11.19 (The size of the scopes). Let d be the largest number of abstract successors of an nblock. Then $|S(b)| \le d+1$ and $|I(b)| \le d^2$ for each b. On a grid of nblocks with 4 moves $|I(b)| \le 12$, with 8 moves $|I(b)| \le 24$, both with equality for an nblock at distance at least 2 from the border, and in the vacuum world $|I(b)| \le 4$.

Proof. By definition $|S(b)| = 1 + |\mathrm{succ}(b)| \le 1 + d$. By Lemma 11.4, $I(b)$ is made of the at most d successors of b and, for each successor a, of the successors of a other than b, which are at most $d-1$ since $b \in \mathrm{succ}(a)$ by condition (ii); hence $|I(b)| \le d + d(d-1) = d^2$. On a grid of nblocks with 4 moves the successors of b are the nblocks at Manhattan distance 1, and their successors those at distance 0 or 2 from b; hence $I(b)$ is the set of the nblocks at distance 1 or 2, i.e., at most $4 + 8 = 12$, all of them when none falls outside the grid. With 8 moves the successors are the nblocks at Chebyshev distance 1, and the nblocks reached by two of them are those at Chebyshev distance at most 2; hence $I(b)$ is the 5×5 square around b without b, i.e., at most 24 nblocks. In the vacuum world the abstract graph is a path, $d = 2$, and $I(b)$ is made of the nblocks at distance 1 or 2 along it. □

For the sliding-tile puzzles with the abstraction of section 11.1 one has $d \le 4$ (one abstract successor for each move of the blank), and hence $|I(b)| \le 16$, a bound that we have not checked to be tight.

The free list and the graph operations

ACQUIRE and RELEASE are lines 35 to 45 of the pseudocode, and SETHOT and SETCOLD are lines 46 to 58, with the same names in lowercase in the code (SafePBNF.h:447-510); SETHOT reads the published values of the hot nblocks of $I(b)$ to apply case 2 of Burns et al. We keep the free list F in an ordered map (a B-tree of the default data structures) keyed by the pairs $(f(b), b)$, so that the first entry is the best nblock, the ties go to the smaller id, the removal of a given nblock is exact, and line 30 is a single scan from the first key not below z. An nblock knows whether it is in F and with which key, and the operations of the pseudocode that touch F are the following (`make_free`, `make_busy`, `prune_free`, `best_free` and `best_scope`, SafePBNF.h:407-444 and 514-522).

Algorithm 11.5 (Safe PBNF: the free list, under the lock of the graph).

```
72  procedure FREE(n)                          ▷ line 59 of Algorithm 11.3
73     if n ∈ F, n is in use, σ(n) ≠ 0 or σ_h(n) ≠ 0 then return
74     drop the stale entries at the top of OPEN(n);  φ ← their least f, or ∞
75     if φ ≥ z then return
76     key(n) ← φ;  publish f(n) ← φ;  F ← F ∪ {(φ, n)};  wake all
77  procedure BUSY(n)                          ▷ "n leaves the free list"
78     if n ∈ F then F ← F \ {(key(n), n)}
79  procedure PRUNEFREE()                      ▷ line 30
80     for each (φ, n) ∈ F with φ ≥ z, in increasing order do F ← F \ {(φ, n)}
81  function BESTFREE()                        ▷ f(free list)
82     return the key of the first pair of F, or ∞ if F = ∅
83  function BESTSCOPE(b)                      ▷ line 18
84     return the least published f(a) over the built nblocks a ∈ I(b),
          with its nblock, or (∞, nil)
```

Line 74 computes $f(n)$ from OPEN(n), not from the published value, as the proof of Lemma 11.7 (c) requires; line 76 publishes it, so that the value read by the others at line 18 is exact for a free nblock. BESTSCOPE skips the nblocks not yet built, whose OPEN lists are empty. We can now bound the cost of the critical sections.

Proposition 11.20 (The cost of the graph operations). Let F also denote the number of the entries of the free list ($F \le N$), let Δ be the largest $|I(b)|$ of an nblock built, and let each insertion, removal and lower bound of the B-tree cost $O(\log F)$. Then, under the lock of the graph, (i) BUSY and BESTFREE take $O(\log F)$ time, and FREE takes $O(\log F)$ time plus one extraction for each stale entry that it drops; (ii) ACQUIRE and SETCOLD take $O(\Delta \log F)$ time, RELEASE $O(\Delta^2 \log F)$, and SETHOT $O(\Delta^2 \log F)$, the stale entries dropped by FREE apart; (iii) PRUNEFREE takes $O((r+1)\log F)$ time, r being the number of the pairs that it removes; (iv) the check of lines 17 to 22 takes $O(\Delta)$ time besides its calls of SETHOT and SETCOLD. Over the whole search, each entry of an OPEN list is dropped at most once, and each pair removed by PRUNEFREE was inserted by an earlier FREE.

Proof. (i) is the cost of one operation of the B-tree, plus the constant work of the tests. (ii) ACQUIRE makes one BUSY and constant work for each $n \in I(b)$. SETCOLD makes, for each $n \in I(b)$, one decrement and possibly

one FREE, and no recursive call, since line 57 is never executed (Lemma 11.7). RELEASE makes, for each $n \in I(b)$, one decrement, possibly one SETCOLD (line 43) and one FREE, and one more FREE for b. SETHOT reads the published values of the hot nblocks of $I(b)$ in $O(\Delta)$, and then makes, for each $n \in I(b)$, possibly one SETCOLD and one BUSY. (iii) Each step of the scan removes one pair, and the scan starts with one lower bound. (iv) BESTSCOPE reads one atomic value for each nblock of $I(b)$, BESTFREE is (i), and line 22 tests the flag of each nblock of $I(b)$. Finally, an entry dropped by FREE or at line 7 leaves its list, and a pair enters F only at line 76. □

By Lemma 11.19, $\Delta \leq 24$ on the grids and $\Delta \leq 4$ in the vacuum world; hence the critical sections cost $O(\log N)$ on the domains of the library. The number of the critical sections is bounded as well. A worker takes the lock of the graph at a check only after m expansions in its nblock, or when its OPEN list is empty: in the first case at most twice (lines 17 and 26), in the second once (line 26), and the counter k of line 5 starts again from 0 after each check. Hence, if E is the number of the expansions of the search and Y the number of the times that a worker finds its OPEN list empty (or empties it at line 8), the lock is taken at most $2E/m + Y + p$ times (p for the first call of NEXTNBLOCK of each worker), the waits at line 33 apart (the creations of the nblocks take no lock). Note that this bound says nothing on the time that a worker spends waiting for the lock, which depends on the scheduling; the try-lock of line 25 is there to keep that time "small".

The worker

The check and the choice of the next nblock follow Algorithm 11.2 (`should_switch` and `next_nblock`, SafePBNF.h:655-725). The lock of the graph is the one of the executor (`Executor::lock_type`), so that the same code runs on the threads of C++ and on OpenMP, and a worker waits on a `std::condition_variable_any`, which works with any lock. The expansion is that of the sequential A* of the library, in two phases: the successors are generated first (without the move back to the parent, in a `Domain` that knows its operators), each with its nblock and the slot of its table computed and prefetched, i.e., requested from memory without waiting for it, and the lookups follow, so that the cache misses of the tables overlap (the design of [4]; cf. Chapter 3). Lines 9 to 12 of Algorithm 11.1 are thus the following (`expand` and `insert`, SafePBNF.h:729-802).

Algorithm 11.6 (Safe PBNF: the expansion of u, in the nblock b held by the worker, with no lock but that of the incumbent).

```
85  procedure EXPAND(u)
86     close u;  count an expansion, and a re-expansion if u was expanded
87     if u is a goal then
88        lock the incumbent;  if g(u) < z then z ← g(u);  record u
89        unlock the incumbent;  return
90     K ← ()                                          ▷ phase 1: generate, prefetch
91     for each successor v of u, but its parent if the Domain knows it, do
92        γ ← g(u) + c(u, v);  if γ + h(v) ≥ z then continue
93        a ← BLOCK(nblock(v));  hv ← hash(v)
          prefetch the slot hv of CLOSED(a)
94        append (v, γ, hv, a) to K
95     for each (v, γ, hv, a) ∈ K do                   ▷ phase 2: the lookups
96        if INSERT(a, v, γ, hv) then publish f(a) ← min(f(a), γ + h(v))
97  function INSERT(a, v, γ, hv)
98     x ← the node of v in CLOSED(a), found from hv, or a new node of v
          with g = γ and parent u, taken from the arena of the worker
99     if x is not new then
100       if γ ≥ g(x) then count a duplicate;  return false
101       g(x) ← γ;  parent(x) ← u;  x is not closed      ▷ a reopening
102    push the entry (x, γ) on OPEN(a);  return true
```

By Lemma 11.3, $a \in S(b)$ at line 93, and therefore lines 98 to 102 write only lists that Theorem 11.8 reserves to the worker. The read of z at line 92 is relaxed, which may only prune less (cf. §8.3, where pruning by a past value of z is shown to be sound). The insertion into a neighbouring nblock lowers its published value with a relaxed atomic store at line 96, which is safe since only the worker that holds the scope writes it (a reader may see a "stale" value, as discussed below). Each worker allocates its nodes from a private arena (`f_pool`, SafePBNF.h:820), whose nodes are never moved. Hence, the parent pointers that cross the workers stay valid until the end, and they are followed only after all the workers have returned. The worker object is aligned to a cache line (SafePBNF.h:603), and the counters of two workers never share a line. After each expansion, a worker publishes the least f of its OPEN list (line 13) and compares the size of its arena with `node_limit` (SafePBNF.h:639-64

9).

The cost of an expansion is thus that of the sequential A* with the same lists: the generation of the successors, one call of BLOCK per child (one atomic load once the nblock exists), one lookup of the table and one push of OPEN per child that is not pruned, i.e., the costs of Chapter 3 for the table with open addressing and for the bucket queue or the binary heap, the latter on the list of one nblock, which is shorter than the single OPEN list of A* (cf. section 11.9). The memory is the following.

Proposition 11.21 (The memory of the search). Let B be the number of the nblocks created, V the number of the states reached and J the number of the pushes on the OPEN lists. Then the search uses $O(N + B\Delta + V + J)$ memory, Δ being as in Proposition 11.20, and `solve()` takes $\Omega(N)$ time to initialize the array A.

Proof. We note that A has N atomic pointers, set to nil one by one (`NBlockMap.h:88-92`). Each nblock created holds $S(b)$ and $I(b)$, i.e., $O(\Delta)$ ids by Lemma 11.19, and constant data besides its lists. By Corollary 11.9 each state has at most one node, allocated once in the arena of a worker; the tables of the nodes hold one pointer per node, with the load factor of Chapter 3, and an OPEN list holds at most one entry per push, the stale ones included, until they are dropped. The free list has at most B entries. □

The term N is the reason why the abstraction should not be made "huge" without need: for the grids of the benchmark it is 62 500 pointers, for the 15-puzzle 4096, while an abstraction into, say, 10^8 ids would cost about 800 MB before the first expansion.

Relaxed values

In fact, we read the published values $f(b)$ without the lock and with relaxed order (lines 18 and 28, and line 47 for the hot nblocks), and therefore a check may see a value slightly out of date (typically by a few expansions). This changes the order of expansion and the choice of the hot nblock, but none of Lemma 11.7, Theorem 11.8, Theorem 11.12, Theorem 11.13 and Proposition 11.17, whose proofs read only the counters, the flags and the lists under the lock (FREE computes $f(n)$ from OPEN(n), line 74, not from the published value). It is one of the reasons for the assumption (A3) of Theorem 11.14. Note also that the value published at line 13 is the key of the first entry of OPEN(b), which may be a stale entry; since that key is the least of all the entries of the list, the live ones included, the published value is a lower bound on $f(b)$, never an overestimate of it.

11.8 The benchmark

We compare Safe PBNF with HDA* on grids of the kind used by Burns et al., which have a "natural" abstraction into nblocks. The instances are three random grids of 5000 × 5000 cells, each cell blocked with probability 0.35 and the free cells of cost 1, generated by `GridMap::random` with the seeds 1, 2 and 5 (whose corners are connected); the start is the bottom-left corner and the goal the bottom-right one (the defaults of `AStarBench`). The moves are 8, a diagonal move costing $\sqrt{2}$ times a straight one, with the octile heuristic. Note that Burns et al. blocked a cell with probability 0.45 on their grids with 8 moves ([3], section 4.3, p. 708), and the instances are not theirs. The nblocks are blocks of 20 × 20 cells (62 500 nblocks, as in section 4.1 of [3]) and $m = 32$, the defaults of `AStarBench` (`--nblock 20 --minexp 32`).

We run each search on 1, 2, 4, 8, 16, 32, 48, 64 and 96 cores, one worker per core, pinned to the cores of one NUMA node with the memory on that node, each run twice in separate processes, the gap between the two runs being the A/A noise given in the note of the table. The rows are Safe PBNF on the threads of C++ and HDA* with the distributions of Chapter 9 (Z, AZ and P on the threads, Z and AZ on OpenMP, on FastFlow and over MPI), as `tools/bench/campaign.sh` runs them. Since the costs are not integers, the searches keep their OPEN lists in binary heaps, and the reference labelled "A* buckets", the sequential A* of `AStarBench` on 1 core, runs on the heap as well (`astar()` takes the bucket queue only for integer costs). The costs are those of the domain, in double precision.

Table 11.1. Safe PBNF and HDA* on three random grids of 5000 × 5000 cells with 8 moves. Each cell gives the seconds summed over the three instances (each the median of its two runs) and, in parentheses, the speedup over sequential A* on 1 core; an empty cell is a search that did not solve every instance. The machine and the compiler are given in the note of the table.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The clock is the timer of the `Solution`: it starts at the beginning of `solve()` and stops at its end, and therefore it includes the creation of the nblock graph (the nblocks themselves are created during the search), the start of the threads of the executor (`AStarBench` does not lend one), the search and the rebuilding of the path. It does not include the generation of the grid, the construction of the `Domain`, the start of the process and the printing of the results. We leave the table as a placeholder until the campaign is run on an idle machine.

11.9 Notes and references

PBNF and Safe PBNF are due to Burns, Lemons, Ruml and Zhou [3], who also brought the abstraction into the hash-based distribution of PRA* and HDA* (AHDA*, treated in Chapter 9) and compared the two families on grids, sliding-tile puzzles and STRIPS planning; their appendix A is the pseudocode followed by the library, and their appendix B the TLA$^+$ model of the hot nblocks. The scopes come from the SDD [139] and the PSDD [128] of Zhou and Hansen, which Burns et al. summarize in their section 2.4 (PSDD is the subject of §12.2), and the ordering of the nblocks by their best node from the Localized A* of Edelkamp and Schrödl, which grouped the nodes by the page of memory where they reside (cited there, and not in the bibliography of this manual, since we could not verify a doi for it). We refer to the survey of Fukunaga et al. [1] for the place of PBNF among the parallel best-first searches.

Jinnai and Fukunaga [5] revisited the comparison of Burns et al. with HDA* on Zobrist hashing, on "easy" and "hard" instances, and with both the heap and the bucket queue as OPEN list (section 3.2, pp. 507-510). They observed that HDA*[Z] is faster than Safe PBNF on the instances of the 15-puzzle and of the 24-puzzle that need "much" search, and Safe PBNF on those solved quickly, because of the start-up burst of HDA* (Observation 3 and Figure 5, p. 509). On grid pathfinding, instead, Safe PBNF outperformed HDA*[Z] although it expanded more nodes, because of the communication overhead of the latter (Observation 4, p. 512). They also remarked that PBNF gains less than HDA* from a bucket queue in place of a binary heap, since each nblock has its own queue: the difference is $O(\log(N/B))$ against $O(1)$ per insertion, with N states and B nblocks (p. 509). Finally, they attributed the smaller burst of Safe PBNF to the exclusive access to the scopes (section 3.1.2, p. 505). We implement both searches with the same OPEN lists and the same tables, on the same executors, so that the comparison of Table 11.1 is not biased by the data structures in the way that Jinnai and Fukunaga pointed out.

Two limits of the method are stated by Burns et al. in their discussion (section 9.1, p. 735): the abstraction is chosen by hand, and "it seems likely that abstraction size should change when number of threads changes or perhaps even based on features of the domain or problem instance". In the library the abstraction is given by the `Domain`, and its size by its parameters (e.g., `nblock_side` for the grids), which one has to choose by hand as well. The automatic construction of abstractions for HDA* is the subject of Chapter 10, and it does not cover the nblocks. Also, the library supports only undirected abstract graphs (condition (ii) of Definition 11.1), while Burns et al. treat directed ones with two further cases of SETHOT. The bounded-suboptimal and anytime variants of their sections 6 and 7 (wPBNF, anytime weighted PBNF and ARPBNF), which the library has (section 11.5), belong with Chapter 6; the library re-sorts the lists of ARPBNF at the end of each iteration, instead of at each new incumbent as Burns et al. do.

11.10 Exercises

Exercise 11.1 (`ctest -R PBeFS.pbnf-options`). In the vacuum world of n cells, write $S(b)$ and $I(b)$ for each nblock b, and show that $|I(b)| \leq 4$ and that at most $\lceil n/3 \rceil$ workers can hold nblocks at the same time. Check that `SafePBNF` returns the cost of A* on the vacuum world with 1, 3 and 4 workers and each value of m of the test.

Exercise 11.2 (`ctest -R PBeFS.pbnf-thread`). On a grid of $W \times H$ nblocks with 4 moves, show that the nblocks whose two coordinates are multiples of 3 have pairwise disjoint scopes, and hence that $\lceil W/3 \rceil \lceil H/3 \rceil$ workers can be busy at once, and that the same holds with 8 moves. Then count $|I(b)|$ on the border and in a corner of the grid. The test runs `SafePBNF` on grids of 300×200 cells with nblocks of 10×10 cells and 1, 2, 3 and 8 workers.

Exercise 11.3 (`ctest -R PBeFS.pbnf-thread`). Prove directly from Algorithm 11.3 that no reachable state of the nblock graph has two hot nblocks that interfere, and that line 57 is never executed. Which lines would have to change for the library to accept a directed abstract graph?

Exercise 11.4 (`ctest -R PBeFS.pbnf-thread`). The test runs `SafePBNF` with 4 workers on a 60×60 grid split by a wall of blocked cells, with nblocks of 7×7 cells, and expects the status `unsolvable`. Follow the proof of Theorem 11.12 on this instance: which nblocks are ever acquired, and why does the search end at line 32 rather than sleep forever at line 33?

Exercise 11.5 (`ctest -R PBeFS.pbnf-thread`). The node limit of `SafePBNF` is per worker, and the worker that reaches it sets the flag of the end under the lock of the graph and wakes all the others. Show that all the workers then return, whatever it was doing, and that the status is `node_limit` even if a goal was found before. The test checks a limit of 5 nodes on an instance of the 8-puzzle that needs at least 6 moves.

Exercise 11.6 (`ctest -R PBeFS.pbnf-options`). Show that the cost returned does not depend on m (Theorem 11.12), and explain why `SafePBNF` takes $m = 0$ as $m = 1$: which step of the proof of Theorem 11.13 fails if a worker may switch before its first expansion in an nblock? Discuss the trade-off of m between the time spent on the lock and the "speculative" expansions, as measured by Burns et al. (section 4.1 of [3]).

Exercise 11.7 (`ctest -R PBeFS.pbnf-thread`). Show that plain PBNF with one worker never livelocks (which is rather easy), and build an execution of plain PBNF with three workers on a cycle of nblocks in which the nblock with the best node is never free. Then run Safe PBNF on your execution by hand, as in Figures 11.9 and 11.10.

Exercise 11.8 (`ctest -R PBeFS.pbnf-openmp`). List the parts of `SafePBNF` that depend on the executor, and show that the proofs of section 11.6 hold for any executor whose lock provides mutual exclusion and a try-lock. The test runs the cases of `pbnf-thread` on OpenMP.

Exercise 11.9 (`ctest -R PBeFS.pbnf-thread`). Give a `Domain` whose abstraction violates condition (ii) of Definition 11.1 (a directed abstract graph), and two nblocks that interfere although the interference scope built by `SafePBNF` (Lemma 11.4) does not record it. Show an interleaving in which two workers write the same table at the same time, and explain why the tests, which use only undirected abstractions, cannot detect the defect.

Exercise 11.10 (`ctest -R PBeFS.pbnf-thread`). Run the execution of Proposition 11.6 with `safe` false and with `safe` true, and say at which check of Figure 11.8 the two diverge. Then list which lines of Algorithms 11.2 and 11.3 plain PBNF never executes, and show that its check reads no interference scope. The test runs plain PBNF on two instances of Korf and on grids with 4 moves, with 1, 2, 3 and 8 workers.

Exercise 11.11 (`ctest -R PBeFS.pbnf-thread`). Two workers call BLOCK(b) of Algorithm 11.4 at once, for an nblock not yet built. Show that both return the same nblock, that the copy of the loser is destroyed before anyone else can see it, and that neither needs a lock; then explain why a store with relaxed order at line 69 would not suffice, by an execution in which a worker sees the pointer but not the scopes. Why may `block()` be called under the lock of the graph, as SETHOT does?

12. Further parallel schemes

[Source: include/PBeFS/astar/PSDD.h (LayeredNBlockSearch, PSDD, BFPSDD, IDPSDD, DynamicBoundedPSDD, PSDDOptions), include/PBeFS/astar/NBlockMap.h, include/PBeFS/astar/FarmAStar.h (FarmOptions), include/PBeFS/astar/MultiAStar.h (MultiAStar), include/PBeFS/astar/HDAStar.h (HDAOptions, LockedChannelsOf), include/PBeFS/astar/TDS.h (TDS, TDSOptions), include/PBeFS/astar/Deepening.h (DeepeningOptions, CutoffHistogram), include/PBeFS/astar/ParallelIDAStar.h (ParallelIDAStar, PIDAOptions, parallel_idastar), include/PBeFS/astar/ParallelIDAStarMPI.h (ParallelIDAStarMPI), include/PBeFS/astar/ParallelGreedy.h (ParallelGreedy, ParallelGreedyOptions, GreedyTrace, HDGBFS), include/PBeFS/astar/BreadthFirst.h (BreadthFirst, breadth_first) and cost_bounded_dfs of include/PBeFS/astar/IDAStar.h]

We are concerned in this chapter with the parallel searches of the library that divide the work in ways other than those of the previous chapters, i.e., other than a shared OPEN list (Chapter 8), an owner for each state (Chapter 9) or an nblock held while it is the best (Chapter 11). They are of four kinds. The first synchronizes the workers by layers: Parallel Structured Duplicate Detection (PSDD) of Zhou and Hansen [128], which expands in parallel the nodes of one layer of a breadth-first heuristic search in the disjoint scopes of an abstraction, with its best-first form and its two bounded forms of Burns et al. [3], and K-Best-First Search (KBFS) of Felner, Kraus and Korf [130], which expands the K best nodes of OPEN at each round. The second changes little in the searches of the previous chapters: a portfolio of independent A* searches that differ only in their ties (Multi-A*), the synchronous communication of the Parallel Retracting A* (PRA*) of Evett et al. [127] and of its abstract variant APRA* on the code of HDA*, and the lending of nodes to the workers left behind. The third parallelizes iterative deepening, either by distributing the transposition table among the workers, as in the Transposition-table Driven Scheduling (TDS) of Romein et al. [131], or by searching in parallel the subtrees below a frontier of units, as in the AIDA* of Reinefeld and Schnecke [55]. The fourth gives up the optimality of the path: the parallel greedy best-first searches with a shared OPEN list (KPGBFS, PUHF and its variants, SPUHF and OBAT) and with hash distribution (HDGBFS), whose guarantees are stated with respect to the sequential greedy search. Two sequential baselines, the breadth-first search and the cost-bounded depth-first search, complete the chapter.

We use the notation of Chapter 1, the invariants of A* of Chapter 2 and the stopping criterion of the parallel searches of §8.3 (Theorem 8.6); the nblocks, the scopes and the interference of PSDD are those of Chapter 11, the owners and the messages of PRA* and TDS those of Chapter 9, the sequential IDA* and its bounds those of §4.4, and the sequential greedy search, with its breaking of the ties, is in Chapter 2. Throughout the chapter the costs are positive and the heuristic h is admissible, unless stated otherwise; C^* is the optimal cost, z the incumbent of a search and p the number of its workers.

The structure of the chapter is as follows. Section 12.1 gives the interface of the searches. Sections 12.2 to 12.5 treat the searches by layers and the variants of A* and HDA* (PSDD and its forms, KBFS, Multi-A*, PRA* and the lending), sections 12.6 and 12.7 the two parallel forms of iterative deepening (TDS, and the search below a frontier with its version over MPI), section 12.8 the parallel greedy searches, and section 12.9 the two baselines. Sections 12.10 to 12.13 compare the searches and give the benchmark, the references and the exercises.

12.1 The searches and their interface

Each search of the chapter is a class template over the `Domain` D (a model of `NBlockDomain`, §11.1, for the searches of `PSDD.h`) and over the executor that runs its workers (the default thread pool, OpenMP or FastFlow, §1.6), with the interface of the parallel searches of §8.1: a constructor that takes the `Domain`, which must outlive the search, and the options; `set_executor()`, which lends an executor that must outlive the searches (otherwise each `solve()` builds one of as many threads as workers); and `solve()`, which returns the `Solution` of Chapter

2. The searches by owners (`TDS`, `HDGBFS`, and `HDAStar` with the options of section 12.5) take in addition the distribution of the states, as `HDAStar` does (§9.4). The class templates and the members that a user calls are declared as follows (quoted by hand from the headers, where they come with their bodies; the parameter `Open` defaults to the bucket queue for integer costs and to the binary heap otherwise, and the executor to the thread pool, as in the previous chapters).

```
template< bool BestFirst , NBlockDomain D , WorkerExecutor Executor ,
          class Open > class LayeredNBlockSearch;            // PSDD.h
template< NBlockDomain D , ... > using PSDD = LayeredNBlockSearch< false , D , ... >;
template< NBlockDomain D , ... > using BFPSDD = LayeredNBlockSearch< true , D , ... >;
 explicit LayeredNBlockSearch( const D & domain , const PSDDOptions & options = {} );
 [[nodiscard]] SolutionT solve( void ) const;                // no bound
 [[nodiscard]] SolutionT solve( Cost bound , Cost * pruned = nullptr ) const;
 SolutionT solve( Cost bound , CutoffHistogram< Cost > & cut , bool buckets ) const;
template< NBlockDomain D , ... > class IDPSDD;               // PSDDOptions
template< NBlockDomain D , ... > class DynamicBoundedPSDD;
 DynamicBoundedPSDD( const D & domain , double weight ,
                     const ParallelOptions & options = {} );
template< Domain D , WorkerExecutor Executor , class Open > class MultiAStar;
 explicit MultiAStar( const D & domain , const ParallelOptions & options = {} );
template< Domain D , class Dist , WorkerExecutor Executor > class TDS;
 TDS( const D & domain , const Dist & dist , const TDSOptions & options = {} );
template< Domain D , WorkerExecutor Executor > class ParallelIDAStar;
 explicit ParallelIDAStar( const D & domain , const PIDAOptions & options = {} );
 [[nodiscard]] std::uint64_t iterations( void ) const noexcept;
 [[nodiscard]] std::size_t units( void ) const noexcept;
template< Domain D > class ParallelIDAStarMPI;               // no executor
 ParallelIDAStarMPI( const D & domain , const PIDAOptions & options = {} ,
                     MPI_Comm comm = MPI_COMM_WORLD );
template< Domain D , WorkerExecutor Executor > class ParallelGreedy;
 explicit ParallelGreedy( const D & domain ,
                          const ParallelGreedyOptions & options = {} );
 void set_record( bool r ) noexcept;
 [[nodiscard]] SolutionT solve( void );                      // not const
 [[nodiscard]] const GreedyTrace< State > & trace( void ) const noexcept;
template< Domain D , class Dist , WorkerExecutor Executor > class HDGBFS;
 HDGBFS( const D & domain , const Dist & dist ,
         const ParallelGreedyOptions & options = {} );
template< Domain D > class BreadthFirst;                     // sequential
 explicit BreadthFirst( const D & domain , const SearchOptions & options = {} );
template< Domain D > Solution< ... > breadth_first( const D & domain ,
                       const SearchOptions & options = {} );
template< Domain D > Solution< ... > cost_bounded_dfs( const D & domain ,
                       cost_t< D > bound , const SearchOptions & options = {} );
template< Domain D > Solution< ... > parallel_idastar( const D & domain ,
                       const PIDAOptions & options = {} );
```

KBFS is the farm of §8.7, `FarmAStar`, with the option `synchronous` of `FarmOptions`, and PRA*, APRA* and the lending are `HDAStar` with the options of `HDAOptions` and the channels `LockedChannelsOf`; their interfaces are those of Chapter 8 and Chapter 9. `ParallelIDAStarMPI` has no `set_executor()`, since its workers are the processes of the communicator, and `ParallelGreedy::solve()` is not `const`, since it keeps the trace of its last search. The options of each search derive from the `ParallelOptions` of §8.1 (`threads`, `batch`, `flush_period`, `seed`, and the fields of `SearchOptions`, the node limit among them), and add the fields given in the section of the search. We remark that, of the fields of `SearchOptions`, the weight is read by `MultiAStar` (whose workers are then weighted A* searches) and by the farm and HDA*, as Chapter 6 describes, and by the first search of `DynamicBoundedPSDD`, an `AStar` with the options given; PSDD, BFPSDD, IDPSDD, `TDS` and the parallel greedy searches ignore it, and `ParallelIDAStar` and `ParallelIDAStarMPI` throw `std::invalid_argument` when it is not 1 (`ParallelIDAStar.h:243`).

12.2 Parallel structured duplicate detection

The searches of `PSDD.h` acquire nblocks as Safe PBNF does (Chapter 11), i.e., an nblock whose interference scope holds no nblock in use, and they expand its nodes and insert the children in the lists of its duplicate detection scope with no lock at all. They do not keep a free list ordered by f, however: they synchronize all the workers at the end of each *layer*, a set of nblocks fixed when the layer starts, which the workers acquire one at a time; when no nblock of the layer is left and none is in use, the last worker to release an nblock builds the next layer, while the others wait. Since the workers of a layer never wait for a better nblock, the search needs no hot nblocks and cannot livelock; the price is the barrier at the end of each layer, which costs most when the layers are small.

Two kinds of layers

In PSDD (Zhou and Hansen [128]) each nblock has two OPEN lists, OPEN for the current layer and NEXT for the following one, and each child generated in a layer goes into NEXT; hence the layers are those of a breadth-first search, as in the breadth-first heuristic search of Zhou and Hansen [129], on which PSDD is built, and the nodes of f above a bound β are pruned. A node reached again with a smaller g goes into NEXT as well, its old entry becoming stale, and it is therefore expanded again in the next layer. In BFPSDD (section 2.4.1 of [3]) NEXT is OPEN, and a layer holds the nblocks whose least f is the least of all, λ, together with others up to mp nblocks in all, m being the multiplier (4 by default, the value of [3]); a worker leaves its nblock when the least f of its OPEN list exceeds λ, after a minimum number e of expansions in it. Figure 12.1 draws a layer of PSDD on the abstract graph of the vacuum world (§11.1).

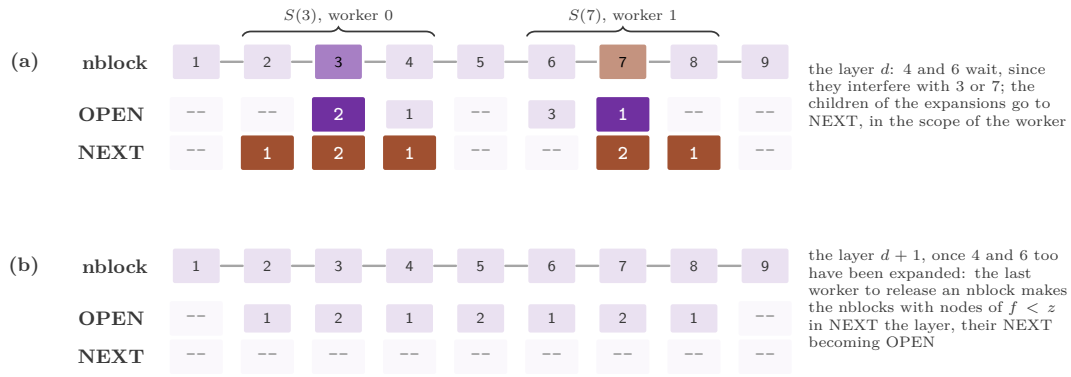


Figure 12.1. A layer of PSDD on the path of the nblocks of the vacuum world. (a) In the layer d worker 0 holds the nblock 3 and worker 1 the nblock 7, whose duplicate detection scopes are disjoint; the nblocks 4 and 6, which also have nodes in OPEN, cannot be acquired until 3 and 7 are released. The children of the expansions (in the accent colour) go into the lists NEXT of the scope of their worker. (b) Once each nblock of the layer has been expanded, the last worker to release one builds the layer $d+1$ from the nblocks with nodes of $f < z$ in NEXT.

The two searches are the class template `LayeredNBlockSearch`, whose first parameter `BestFirst` selects BFPSDD, with the two aliases `PSDD` and `BFPSDD` (`PSDD.h:191`, `664` and `673`). Their options are the following.

```
struct PSDDOptions : public ParallelOptions {

 /// BFPSDD: the nblocks of a layer are multiplier times the workers, at
 /// least, the ties on the least f all included; 4 is the value of
 /// section 2.4.1 of Burns et al. (JAIR 2010)
 double multiplier = 4;

 /// BFPSDD: the expansions in an nblock before its worker may leave it
 /// for a node of f above the value of the layer
 unsigned min_expansions = 32;

 /// IDPSDD: the growth of IDA*-CR, the bounds being the least f pruned if
 /// not above 1 [see Deepening.h]; every path found is optimal anyway,
 /// since PSDD finds the best path within its bound
 double growth = 0;

 PSDDOptions( void ) = default;
```

```
/// the options of the parallel searches, with the defaults of BFPSDD
PSDDOptions( const ParallelOptions & o ) : ParallelOptions( o ) {}

};  // end( struct( PSDDOptions ) )
```

The fields `multiplier` (m, taken as 4 if not positive) and `min_expansions` (e, taken as 1 if 0) are read by BFPSDD only, and `growth` by IDPSDD (below); of the inherited fields, `threads` is p and `node_limit` bounds the nodes stored by each worker, while `batch`, `flush_period` and `seed` are not used.

The algorithm

The pseudocode of the library (the Doxygen comment of `PSDD.h`) is the following, with z the incumbent, β the bound given to `solve()` (∞ by default), λ the value of the layer of BFPSDD, $b(v) = \phi(v)$ the nblock of v, and the conventions of §1.4.

Algorithm 12.1 (PSDD and BFPSDD).

```
 1  procedure THREADSEARCH()                          ▷ on each worker
 2     b ← nil
 3     loop
 4        b ← NEXTNBLOCK(b);  if b = nil then return
 5        k ← 0
 6        while OPEN(b) has a node of f < z do
 7           if BFPSDD, k ≥ e and the least f of OPEN(b) is above λ then
               break
 8           u ← the first node of OPEN(b), removed;  close u;  k ← k + 1
 9           for each successor v of u, g' ← g(u) + c(u, v) do
10              if g' + h(v) > β or g' + h(v) ≥ z then continue
11              if v ∉ CLOSED(b(v)) or g' < g(v) then
12                 g(v) ← g';  parent(v) ← u;  v is not closed
13                 if v is a goal then z ← min(z, g'), recording v
14                 else push v in NEXT(b(v))
15  procedure NEXTNBLOCK(b)
16     lock the graph
17     if b ≠ nil then release b, freeing the nblocks of the layer
          that no nblock in use interferes with any more
18     while no nblock of the layer is free do
19        if no nblock is in use then NEXTLAYER()
20        if done then return nil
21        if some nblock is in use then sleep until woken
22     b ← a free nblock of the layer;  acquire b;  return b
23  procedure NEXTLAYER()
24     if PSDD then
25        the layer: the nblocks with nodes of f < z in NEXT, whose
             NEXT becomes OPEN
26     else
27        λ ← the least f < z of the OPEN lists
28        the layer: the m p nblocks of least f < z in OPEN, and all
             those whose least f is λ
29     if the layer is empty then done ← true
30     wake all
```

Line 10 prunes by the bound and by the incumbent; the f of the nodes pruned by the bound are kept, for IDPSDD, in a `CutoffHistogram` (section 12.6). The goal test is made at the generation (line 13), and a goal is never pushed: it only lowers z, and the search goes on until no OPEN list holds a node of $f < z$ (line 29), which gives a path of least cost within the bound whatever the costs of the edges (Theorem 12.4). This is a departure from the scheme of Zhou and Hansen, in which the first goal ends the search and the path is optimal only when the edges have the same cost (cf. section 2.4.1 of [3]). The acquisition and the release of lines 17 and 22 are those of Algorithm 11.3 without the hot nblocks: an nblock of the layer is *free* when it is not in use and $\sigma = 0$; the release decrements σ on the interference scope and frees the nblocks of the layer that reach $\sigma = 0$ with nodes in their list of the layer; and the acquisition increments σ on the interference scope of the nblock taken, whose

nblocks stop being free (PSDD.h:369-400). In PSDD, line 25 swaps the roles of the lists OPEN and NEXT of each nblock (the code keeps two lists per nblock and the index of the layer, PSDD.h:424-468), and empties the lists NEXT that hold only nodes of $f \geq z$.

Analysis

We first show that the lists have one writer at a time, as in Safe PBNF, and then that the end of the search is exact; the optimality follows.

Definition 12.1 (The best path within a bound). For a bound β, a path from s_0 is *within* β if each of its nodes v has $g_P(v) + h(v) \leq \beta$, $g_P(v)$ being the cost of the path up to v, and C^*_β is the least cost of a path within β from s_0 to a goal (∞ if there is none).

Note that $C^*_\beta \geq C^*$, and that $C^*_\beta = C^*$ when $\beta \geq C^*$, since by admissibility each node v of an optimal path P has $g_P(v) + h(v) \leq g_P(v) + h^*(v) = C^*$.

Lemma 12.2 (Exclusive access). At each instant at which the lock of the graph is free, (a) $\sigma(n)$ is the number of the nblocks in use in $I(n)$, (b) each free nblock is not in use and has $\sigma = 0$, and (c) no two nblocks in use interfere. Hence the lists OPEN, NEXT and CLOSED of an nblock n are written by at most one worker at a time, the one that holds the nblock c with $n \in S(c)$.

Proof. (a) is the argument of Lemma 11.7 (a): the acquisition and the release of b change σ by one on $I(b)$ exactly when b starts and stops being in use. (b) An nblock becomes free only at line 17, when its σ reaches 0 and it is not in use, or at line 25 or 28, when no nblock is in use and hence $\sigma \equiv 0$ by (a); it stops being free when it is acquired, or when an nblock of its interference scope is acquired, since each acquisition clears the flag free of the nblocks whose σ it raises. (c) An nblock is acquired only when it is free, i.e., by (b) and (a), when no nblock of its interference scope is in use. The last claim is Theorem 11.8, whose proof uses only (a), (c) and the fact that the children of the nodes of c lie in $S(c)$ (Lemma 11.3). □

Lemma 12.3 (The end of a layered search). When `done` is set at line 29, no OPEN or NEXT list holds an entry that is not stale and has $f < z$, and no worker holds an nblock.

Proof. Line 29 runs within line 19, i.e., when no nblock is in use, under the lock of the graph. We first show that each nblock with an entry that is not stale in the list from which the next layer is built (NEXT in PSDD, OPEN in BFPSDD) is among the *candidates* that line 25 or 28 examines. An entry pushed into such a list that was empty, of an nblock that is not a candidate, makes the worker record the nblock, and the worker makes it a candidate under the lock of the graph at its next call of NEXTNBLOCK (PSDD.h:549-555 and 628-631); since no nblock is in use at line 19, each worker has made that call since its last insertion. In BFPSDD, moreover, an nblock released with nodes left in OPEN, and an nblock with nodes of $f < z$ that line 28 leaves out of the layer, are made candidates again (PSDD.h:383-391 and 455-464). NEXTLAYER then empties the lists whose least f, the stale entries dropped first, is not below z, and makes a layer of all the others, unless there is none. In PSDD each list OPEN of the finished layer is empty as well: each nblock of the layer with nodes in it is free or blocked by an nblock in use, a blocked one is freed by the release that brings its σ to 0 (line 17), and the layer ends only when none is free and none is in use, i.e., when each of them has been acquired and its list consumed, or emptied at line 6 because its least f is not below z (PSDD.h:519-545). Hence, when the layer is empty, no list holds an entry that is not stale with $f < z$. □

The goal test of PSDD is at the generation, unlike that of the searches of Chapter 8, and we therefore prove the optimality directly, along the lines of Theorem 8.6.

Theorem 12.4 (Optimality within the bound). Let h be admissible and let `solve( β )` of PSDD or BFPSDD end without reaching the node limit. Then it returns a path of cost C^*_β within β, or the status `unsolvable` if $C^*_\beta = \infty$. In particular, with $\beta \geq C^*$ (e.g., with no bound) the path is optimal.

Proof. If $h(s_0) > \beta$ or s_0 is a goal, `solve()` returns at once with the status `unsolvable` or the path (s_0) (PSDD.h:268-274), which is right. Otherwise, we first show that $z \leq C^*_\beta$ at the end. Suppose not, and let $P = (w_0 = s_0, \ldots, w_m)$ be a path within β to a goal, of cost C^*_β, with no goal before w_m (a goal w_j with $j < m$ would give a cheaper path within β). Call w_i *reached* if at some instant its node has $g \leq g_P(w_i)$. Clearly, w_0 is reached, and if w_m is reached then an insertion of line 11 gave it $g \leq C^*_\beta$ and line 13 set $z \leq C^*_\beta$, or line 10 skipped it because $g' + h(w_m) \geq z$, which with $h(w_m) = 0$ (by admissibility) again means $z \leq C^*_\beta$. Let $i \geq 1$ be the least index such that w_i is not reached, and let $\gamma \leq g_P(w_{i-1})$ be the least value that the g of w_{i-1} ever takes. The node of w_{i-1} received γ at line 12 (or is the root, with $\gamma = 0$) and was pushed with it at line 14 (it is not a goal), and that entry becomes stale only by the expansion of the node with $g = \gamma$, since g never decreases below γ. Its key is $\gamma + h(w_{i-1}) \leq g_P(w_{i-1}) + h^*(w_{i-1}) \leq C^*_\beta < z$, where $h^*(w_{i-1}) \leq C^*_\beta - g_P(w_{i-1})$ since the

rest of P is a path to a goal. By Lemma 12.3 the entry is not in a list at the end, and a list is emptied at line 6 or 25 only when its least f is not below the incumbent of that instant, which is at least the final z; hence the entry was expanded. Its expansion generated w_i with $g' \le \gamma + c(w_{i-1}, w_i) \le g_P(w_i)$, which line 10 did not prune, since $g' + h(w_i) \le g_P(w_i) + h(w_i) \le \beta$ (P is within β) and $g' + h(w_i) \le C^*_\beta < z$; then line 11 found its node with $g \le g'$ or gave it g', and w_i is reached, a contradiction.

Conversely, z is the g of the recorded goal, and the path rebuilt from it through the parents is within β. Indeed, each node of the chain was pushed or recorded with a value of g that passed line 10, and its g has only decreased since; along the chain $g(v) \ge g(\text{parent}(v)) + c(\text{parent}(v), v)$ holds at the end, since the g of a parent only decreases after it is used (cf. the proof of Theorem 11.12). Hence the cost of the chain up to each of its nodes v is at most $g(v)$, and its f along the chain is at most $g(v) + h(v) \le \beta$; the chain is a path within β to a goal, of cost at most z, and therefore $C^*_\beta \le z$. Hence $z = C^*_\beta$, and the path returned costs exactly C^*_β. If no goal is recorded, $C^*_\beta = \infty$ by the first part, and the status is `unsolvable`. □

With unit costs PSDD ends with the layer of the first goal, as the scheme of Zhou and Hansen does, which the following corollary makes precise.

Corollary 12.5 (Unit costs). If each edge costs 1, the layer d of PSDD holds the nodes of $g = d$, no node is reopened, and a goal generated in the layer d ends the search when the layer $d + 1$ is built.

Proof. By induction on d: the layer 0 holds the root, and the children of the nodes of $g = d$ have $g = d + 1$ and go into NEXT; a node reached again has a g not smaller than its first one, which with unit costs is its depth in breadth-first order, so that no node is reopened. A goal generated in the layer d gives $z = d + 1$, and each node pushed into NEXT in the layer d has $f \ge g = d + 1 = z$; hence line 25 finds no node of $f < z$, and the layer is empty. □

We remark that the scheme of Zhou and Hansen stops at the goal, before the rest of the layer d, while PSDD also expands the rest of that layer. The page on parallel A* of the documentation of the library (`docs/theory/parallel-astar.md`) reports that on the instance 55 of Korf with $\beta = 41$ the two expand the same 199 299 nodes, a count that we have not rerun.

The bounds of PSDD

A bound is what makes PSDD practical: without it, PSDD with an admissible h expands each node of $f < z$ layer by layer, and z is known only when a goal is generated, i.e., late; the bound prunes the nodes of large f from the start. The library gives the bound in two ways, both of Burns et al. (section 2.4.1 of [3]).

`IDPSDD` (`PSDD.h:694`) starts from $\beta = h(s_0)$ and, after each iteration that finds no path, raises β to the least f that the iteration pruned (the bounds of IDA*, §4.4), or, with `growth` $r > 1$, to the bound of IDA*-CR of Sarkar et al. [132], which `CutoffHistogram` computes from the f of the pruned nodes (section 12.6). The `Solution` sums the counters of the iterations, but `stored`, which is the largest number of nodes of an iteration, and one executor serves all the iterations.

Theorem 12.6 (IDPSDD). Let h be admissible and the state space finite. Then IDPSDD ends, if it does not reach the node limit, with an optimal path or with the status `unsolvable` when no goal is reachable, whatever the rule of its bounds (the least f pruned or IDA*-CR).

Proof. Each iteration is a `solve( β )` of PSDD, which ends on a finite space (Proposition 12.8 below). An iteration with $\beta < C^*$ finds no path, since a path within β to a goal costs at most β (h is 0 at a goal); and if a goal is reachable, it prunes a node of $f \le C^*$: the argument of the proof of Theorem 12.4, applied to an optimal path P, gives a first node of P that is not reached, which is generated with $f \le g_P + h \le C^*$ and pruned by the bound, since $z = \infty$ in that iteration. Both rules give a next bound not smaller than the least f pruned, which exceeds β (`Deepening.h:200`); since the values of f of a finite space are finitely many, after finitely many iterations either $\beta \ge C^*$, and then Theorem 12.4 gives a path of cost $C^*_\beta = C^*$, or no node is pruned and the iteration ends with no path, which IDPSDD returns as `unsolvable`. By the remark above, the latter happens only when no goal is reachable. □

Unlike IDA* (section 12.6), IDPSDD needs no completion of the iteration that finds the first path, since PSDD already finds the best path within its bound, and a bound above C^* costs only the nodes of f between C^* and the bound. This is why IDA*-CR is safe for IDPSDD with any growth, and why its option `growth` is described as giving optimal paths anyway.

`DynamicBoundedPSDD` (`PSDD.h:795`) takes the bound from a first search. A* (Chapter 2) on the `Domain` whose heuristic is wh ($\lfloor wh \rfloor$ with integer costs; the class `Inflated` of `PSDD.h:857`, which hands to the successors of

the original `Domain` the original h of the parent, as a `Domain` that computes its heuristic incrementally needs), with the weight $w \geq 1$ given to the constructor (1 if smaller), finds a path of cost c, and PSDD with $\beta = c$ then finds an optimal path.

Proposition 12.7 (DynamicBoundedPSDD). Let h be admissible. If the first search ends with a path, DynamicBoundedPSDD returns an optimal path (unless PSDD reaches the node limit); otherwise it returns the status of the first search (`unsolvable` or `node_limit`).

Proof. The cost c of the path of the first search is the cost of a path of the graph, hence $c \geq C^*$, and PSDD with $\beta = c \geq C^*$ returns a path of cost $C^*_c = C^*$ by Theorem 12.4. Since the two searches search the same graph, a first search that proves no goal reachable is right. □

Note that the proof uses only $c \geq C^*$; the bound $c \leq wC^*$ of weighted A* (Pohl [46] and Chapter 6) serves only to keep the second search small, since the nodes of $f \leq c$ grow with c, and hence with w. The documentation reports that on the instance 55 of Korf $w = 1.5$ gives $c = 47$ against $C^* = 41$ (a count that we have not rerun). The counters of the `Solution` are those of the two searches summed, but `stored`, the larger of the two.

Proposition 12.8 (Termination). On a finite state space, `solve(` β `)` of PSDD and BFPSDD ends, under any scheduling in which a worker that can proceed eventually does.

Proof. A node is pushed only when its g takes a new, strictly smaller value, the cost of a simple path (cf. Theorem 11.13); hence the pushes, and the expansions, are finitely many. Each acquisition is followed by at least one expansion or by the emptying of a list, since an nblock of a layer is free only with nodes in its list of the layer, and in BFPSDD $e \geq 1$ lets the worker expand at least one node; hence the acquisitions are finitely many, and so are the layers, each of which removes at least one entry. A worker sleeps at line 21 only while an nblock is in use, whose worker reaches its release in finite time and wakes the sleepers when it frees an nblock or builds a layer, under the same lock; hence no worker sleeps forever, and the last layer, the empty one, sets `done` and wakes all. □

The engineering

The nblocks are those of Safe PBNF, built at their first use by the class template `NBlockMap` of `NBlockMap.h`, which the two searches share (§11.7 describes it); an nblock of PSDD holds its two lists OPEN and NEXT (one list in BFPSDD), its table of nodes, its scopes, σ, and the flags in use, free, in the layer and candidate (`PSDD.h :332-342`). The free nblocks of a layer are a vector, which a worker scans from its end, skipping the entries whose flag free an acquisition has cleared (an acquisition clears the flags of the nblocks of the interference scope without searching the vector); the candidates of the next layer are a vector of their own, and BFPSDD sorts them by their least f at line 28. The lock of the graph is a `std::mutex` rather than the lock of the executor (`PSDD.h:325`): at the end of a layer all the workers but one wait while the last one builds the next layer, and with the spin lock of an executor they would spin for the whole of that time. The expansion is that of Safe PBNF (Algorithm 11.6): the successors are generated and their slots prefetched first, and looked up afterwards; the nodes are allocated from an arena of each worker; the goal is recorded under a separate mutex; and the incumbent is read with relaxed order at line 10, which may only prune less. The cost of a layer is that of its expansions, plus one critical section per acquisition and the barrier, i.e., the time between the first worker that finds no free nblock and the last release. When the layers are small, as the layers of f of BFPSDD on a domain with many distinct values of f, the barrier typically dominates, which is why Burns et al. introduced PBNF (section 2.4.1 of [3], p. 696, and §11.2).

12.3 K-best-first search

KBFS of Felner, Kraus and Korf [130] expands, at each step, the K best nodes of OPEN, and inserts all their children before the next step; with K processors the expansions of a step run in parallel. In the library it is the farm of §8.7 with the option `synchronous` of `FarmOptions`:

```
struct FarmOptions : public ParallelOptions {

 /// true for the synchronous rounds of KBFS: the master sends a task to
 /// each worker, waits for all of them and inserts their successors in the
 /// order in which it sent them before the next round
 bool synchronous = false;
```

```
FarmOptions( void ) = default;

/// the options of the parallel searches, without rounds
FarmOptions( const ParallelOptions & o ) : ParallelOptions( o ) {}

};  // end( struct( FarmOptions ) )
```

With it the farm runs in *rounds*: the master sends a task of up to `batch` nodes to each of the p workers, which are all free at the start of a round, waits until each task is back, and only then inserts the successors, task after task in the order in which it sent them, before it dispatches the next round; thus $K = p \cdot \text{batch}$. DISPATCH (Algorithm 8.6) is unchanged, and the return of a task (Algorithm 8.7) becomes the following (FarmAStar.h:261-279).

Algorithm 12.2 (The return of a task in the rounds of KBFS, on the master; R is the list of the tasks of the round, in the order of the sends, to which DISPATCH appends).

```
1  function BACK(T)
2     out ← out − 1
3     if out > 0 then return true           ▷ the round is not over
4     for each task T' of R, in order, do
5        insert the successors of the nodes of T' as Algorithm 8.7 does
6     R ← ∅
7     if the master stores at least node_limit nodes then the limit is hit
8     DISPATCH()                            ▷ the next round, appended to R
9     return out > 0 or (the limit is not hit and OPEN has a node of f < z)
```

Figure 12.2 draws one round. The search is deterministic: the order of the insertions, and hence the breaking of the ties in OPEN, no longer depends on the order in which the workers finish.

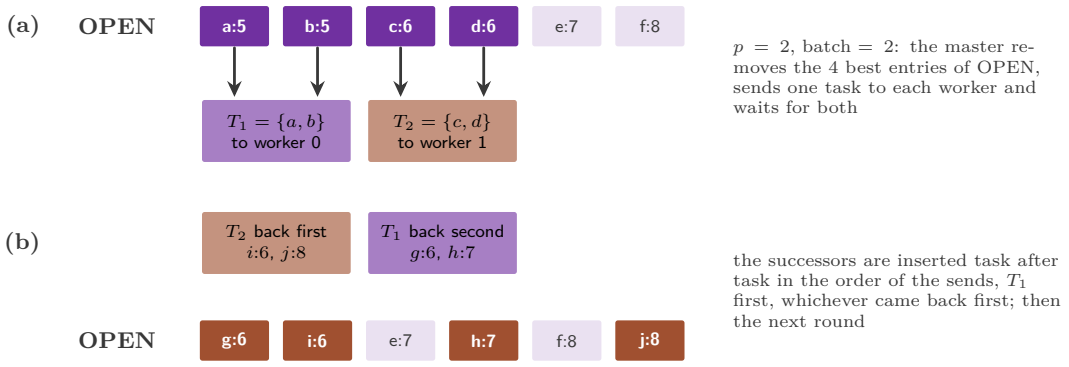


Figure 12.2. A round of KBFS with $p = 2$ workers and tasks of 2 nodes ($K = 4$), each entry with its f. (a) The master removes the 4 best entries of OPEN and sends $T_1 = \{a, b\}$ to worker 0 and $T_2 = \{c, d\}$ to worker 1. (b) T_2 comes back first, but the master waits for T_1 and inserts the successors of T_1 before those of T_2, in the order of the sends; then it dispatches the next round.

Proposition 12.9 (KBFS). Let h be admissible and the generation of the successors by the `Domain` deterministic. (i) KBFS returns an optimal path if it ends without reaching the node limit, and it ends on a finite graph. (ii) Its expansions, its counters and its path are the same at each run, whatever the scheduling of the workers. (iii) With one worker and tasks of one node it expands the nodes that A* expands with the same OPEN list, in the same order.

Proof. (i) The rounds only delay the insertions: the invariant of out (Lemma 8.17) holds whenever the master is between two of its steps, with the tasks of the round that have come back counted among those that are out until their successors are inserted (the master keeps them in R and inserts nothing before the round is over); hence the proof of Theorem 8.18 applies. (ii) The master is the only thread that changes OPEN, CLOSED, z and the nodes, and its steps are DISPATCH, which depends only on its state, and the insertion of a round, which depends only on the tasks of R, in the order of the sends, and on their successors, which are a function of the nodes of each task. By induction on the rounds, the state of the master after each round is the same at each run. (iii) With $p = 1$ and batch 1, each round sends the first node of OPEN of $f < z$, after recording as the

incumbent the goals that come first (the goal test of DISPATCH), and inserts its successors before the next selection, which is the loop of A* with the goal test at the selection. A* stops at the first goal it selects, of cost C^*; the master records it as $z = C^*$, and since that goal was the first entry of OPEN, no entry left has $f < z$, so that the master stops there too. □

The test `farm-thread` checks (iii) on an instance of Korf (the counters `expanded` of KBFS and of `astar()` are equal) and (ii) with 2, 3 and 8 workers and tasks of 2 nodes (two runs with the same expansions and generations). The price of the determinism is the barrier of each round, which leaves idle the workers that finish first; Burns et al. found that KBFS does not beat serial A* on their grids (section 2.1 of [3]), and we expect it to pay, if at all, only when the expansions are slow and of similar cost.

12.4 A portfolio of A* searches

The simplest parallel search shares nothing but a flag: `MultiAStar` (`MultiAStar.h:73`) runs p independent A* searches, with the options given (the weight included), and the first one to end gives the `Solution`. The worker 0 runs the plain A*; the worker $i > 0$ runs it on a view of the `Domain` (the struct `View`, `MultiAStar.h:143`) that generates the successors of each state in an order drawn by a generator of its own (a Fisher-Yates shuffle on a xorshift generator, seeded by `splitmix64` of the `seed` of the options and of i), so that the nodes of equal f and g enter its OPEN list in another order and the ties are broken otherwise. The view also makes each state a goal once the flag is set, so that the next selection of each other worker ends its search, whose result is dropped.

Algorithm 12.3 (Multi-A*; winner is atomic, −1 at the start).

```
1  procedure MULTI-A*(s_0, p)
2     winner ← −1
3     in parallel, on each worker i = 0, ..., p − 1 do
4        S_i ← A* on the view V_i, whose goal test is
                "winner ≥ 0 or the state is a goal"
5        if winner = −1 then winner ← i      ▷ a compare-and-swap
6     return S_winner, with the counters of all the workers summed
```

Figure 12.3 draws the portfolio.

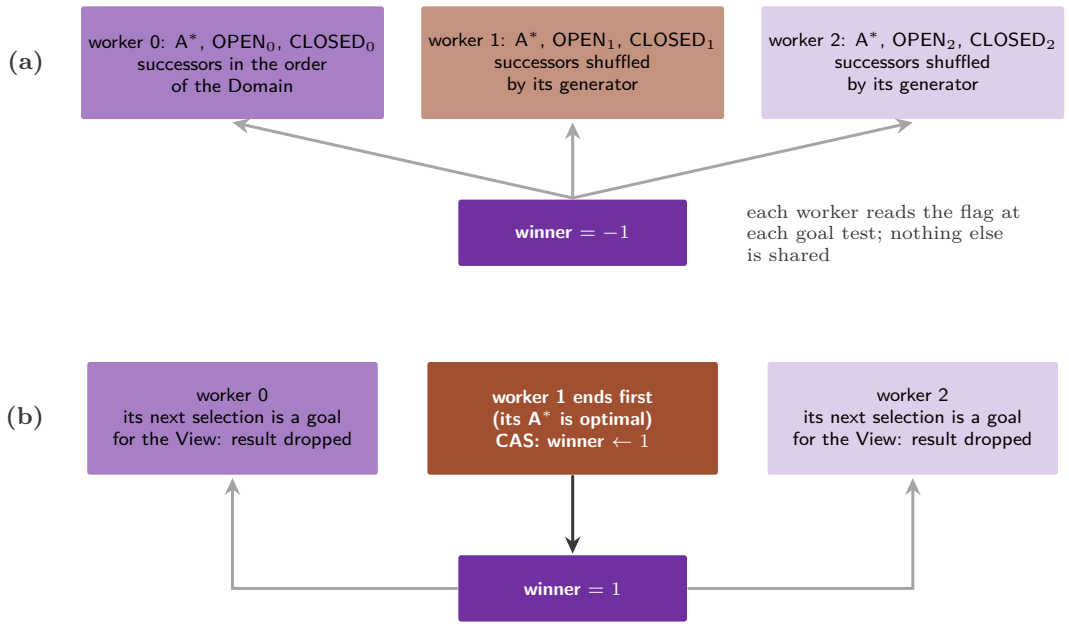


Figure 12.3. Multi-A* with three workers. (a) Each worker runs A* with its own lists; the workers 1 and 2 generate the successors in the order of their generators, the worker 0 in that of the `Domain`, and they share only the flag winner. (b) The worker 1 ends first and sets winner by a compare-and-swap; the others end at their next selection, which their view takes as a goal, and their results are dropped.

Proposition 12.10 (Multi-A*). The `Solution` of `MultiAStar` is that of an A* search on the `Domain` with the

options given, and hence it has the guarantees of A* (Chapter 2, or Chapter 6 with a weight); with one worker its expansions are those of `astar()`.

Proof. The winner is the worker whose compare-and-swap of line 5 found -1. Its search never read winner ≥ 0, since winner never returns to -1 once set, and a worker that has read it set finds it set at line 5; hence each goal test of the winner was the goal test of the `Domain`, and its search is A* on the graph of the `Domain` with the order of the successors of its view, which changes nothing else. With $p = 1$ the view of the worker 0 forwards the successors in the order of the `Domain`. □

We expect the portfolio to pay, if at all, only when the expansions of A* in the last layer of f depend much on the ties, which the preference for the larger g already reduces (§2.8), at the price of p times the memory and of a memory bus shared by p searches that usually do much the same work. The node limit holds for each worker, and the status `node_limit` of the first worker to reach it ends the portfolio, even if another worker would have solved the instance. The counters `expanded_by` and `stored_by` hold those of each worker, and the test `multi-thread` checks that one worker expands as `astar()`.

12.5 Synchronous messages and the lending of nodes

Two options of `HDAStar` change the communication of HDA* (Chapter 9), and a family of channels changes the way in which a worker receives:

```
struct HDAOptions : public ParallelOptions {

 /// true to push every node to its owner as soon as it is generated, as
 /// in PRA*, instead of packing the nodes in batches
 bool sync_send = false;

 /// if positive, a worker lends the node it is about to expand to the
 /// worker whose best f is the least among those not smaller than its f
 /// plus outsource, which expands it at once; 0 for none
 double outsource = 0;

 HDAOptions( void ) = default;

 /// the options of the parallel searches, with no synchronous send and no
 /// lending
 HDAOptions( const ParallelOptions & o ) : ParallelOptions( o ) {}

 };  // end( struct( HDAOptions ) )
```

```
template< ADS::WorkerExecutor Executor >
struct LockedChannelsOf {
 /// the family for the messages Msg
 template< class Msg >
 using type = ADS::LockedChannels< Msg , typename Executor::lock_type >;
 };  // end( struct( LockedChannelsOf ) )
```

PRA* and APRA*

PRA* of Evett et al. [127] is the ancestor of HDA*: each state has an owner by a hash, and a node is sent to its owner when it is generated. On the massively parallel machine for which it was written the communication was synchronous at both ends, and a worker whose memory filled up *retracted* its worst nodes, i.e., removed them and kept in their parents the least f of what it removed. Burns et al. separated the two halves of the communication, compared the four combinations of synchronous and asynchronous sends and receives, and called APRA* the variant whose owner is given by an abstraction [3]. The library has the two synchronous halves, without the retraction. With `sync_send` the mailbox of `HDAStar` has batches of one node (`HDAStar.h:267`), so that each node is pushed to its owner as soon as it is generated; with the channels `LockedChannelsOf< Executor >` an inbox is a vector under the lock of the executor, which the receiver takes, waiting for it, to receive. The two together are PRA*, and with the `AbstractStateDistribution` of Chapter 9 they are APRA*. Since only the transport of the messages changes, HDA* keeps its correctness and its termination (§9.3): the deficit

counting counts each message, whatever its size, and the test `hda-thread` runs PRA*, APRA* and the two halves alone on an instance of Korf.

The lending of nodes

A distribution of the states balances the nodes among the workers, not the nodes of small f: a worker may hold the best nodes of the search while another expands nodes that A* would never select (the band effect of §8.2). With `outsource` $= \delta > 0$, each worker publishes, before each expansion, the f of the node that it is about to expand (∞ when it has none), and the worker i that is about to expand u, not a goal, *lends* it to the worker j whose published value is the least among those not smaller than $f(u) + \delta$, if there is one (`HDAStar.h:434-466`). The borrower expands the node as soon as it receives it and sends its successors to their owners, while the node stays in the CLOSED list of its owner; a lent node is never lent again.

Algorithm 12.4 (The lending of HDA*, on the worker i that has selected u; F_j is the value published by the worker j).

```
1  procedure STEP(u)                  ▷ u selected at line 7 of HDA*
2     F_i ← f(u)
3     j ← the worker j ≠ i of least F_j among those with F_j ≥ f(u) + δ
4     if there is none or u is a goal then expand u as HDA* does;  return
5     close u;  send (u, g(u), lent) to j;  flush the outbox to j
6  procedure RECEIVE(u, g, lent)       ▷ on the borrower j
7     if g + h(u) ≥ z then return
8     for each successor v of u, g' ← g + c(u, v), with g' + h(v) < z do
9        if owner(v) = j then INSERT(v, g', u) else send (v, g', u) to owner(v)
```

Proposition 12.11 (HDA* with lending). With `outsource` positive, with or without `sync_send` and the locked channels, HDA* is well-formed in the sense of §8.3; hence, if it ends without reaching the node limit, it returns an optimal path.

Proof. We check the conditions of §8.3 for the entry of a lent node, the others being those of HDA*. The entry of u, selected by its owner, leaves OPEN at line 7 of HDA* and is closed at line 5; from then on it is in the message to j (condition (iii) of Definition 8.5), and then held by j until j has inserted or sent each successor (condition (ii)); the mailbox counts the message, so that the end of the search is not detected while it travels. The borrower satisfies condition (b): it prunes by a value of z of the past (lines 7 and 8) and creates, by the insertion or the message of line 9, an entry of each successor with $g' = g + c(u, v)$, the cost of a path to v through u, whose parent pointer is the node of u at its owner. A goal is never lent (line 4), so that condition (c) is that of HDA*, and Theorem 8.6 gives $z = C^*$ at the end. Since the borrower only reads the fields of the message and stores the pointer to the node of u, which only its owner writes, the lending adds no data race. □

The lending moves work away from a worker whose nodes are much better than those of the others, which the distribution alone usually cannot do, at the price of one message per node lent and of the published values, one atomic per worker on a cache line of its own (`HDAStar.h:325`). An idle worker publishes ∞, and therefore receives a node only when no busy worker is far enough behind; the gap δ keeps the workers whose nodes are about as good from lending to each other. The borrower counts the expansion of a lent node in its `expanded`, and the owner counts a later expansion of the same node, after an improvement of its g, as a re-expansion. The test `hda-thread` runs the lending with a gap of 2 on an instance of Korf and with a gap of 3 and synchronous sends on a grid with 8 moves, and the corner cases of `check_astar_edges` with PRA* and a gap of 1.

12.6 Transposition-table driven scheduling

IDA* (§4.4) stores only its path, and on a graph with many transpositions it searches the same subtree once for each path that reaches its root within the bound. TDS of Romein, Plaat, Bal and Schaeffer [131] parallelizes IDA* by a transposition table split among the workers by owners, as the CLOSED list of HDA* is: each node is sent to the worker that owns the entry of its state, which looks it up there and, if the state is not already being searched in the iteration with a budget at least as large, pushes it on its own stack; each worker searches depth first from its stack. The work moves with the data, so that a lookup never crosses the workers, and the subtree of a state is searched at most once in an iteration.

The algorithm

The pseudocode of the library (the Doxygen comment of `TDS.h`) is the following, with β the bound of the iteration, T_i the part of the table of the worker i, each entry with the g with which its state was last pushed in the iteration, and a message (v, g', u) a state with the cost and the predecessor of its path.

Algorithm 12.5 (TDS).

```
 1  procedure TDS(s₀, p, owner)
 2     β ← h(s₀)
 3     loop
 4        β' ← ∞;  INSERT(s₀, 0, nil) on owner(s₀)
 5        in parallel, on each worker i, until no worker has work and
             no message is in transit, or a goal is found, do
 6           for each message (v, g', u) received do INSERT(v, g', u)
 7           if the stack of i is not empty then
 8              u ← pop the stack (stale entries dropped)
 9              if u is a goal then record u;  stop all
10              for each successor v of u, g' ← g(u) + c(u, v) do
11                 if g' + h(v) > β then β' ← min(β', g' + h(v))
12                 else if owner(v) = i then INSERT(v, g', u)
13                 else send (v, g', u) to owner(v)
14        if a goal was recorded then return the path to it
15        if β' = ∞ then return unsolvable
16        β ← β'
17  procedure INSERT(v, g', u)                        ▷ on the owner i of v
18     if T_i has v from this iteration with g(v) ≤ g' then return
19     g(v) ← g';  parent(v) ← u;  push v on the stack of i
```

Line 18 is what TDS adds to IDA*: since the bound is the same for the whole iteration, a state already pushed in it with a g not larger has been, or is being, searched with a remaining budget $\beta - g$ at least as large, and its subtree is not searched again by any worker (Romein et al., p. 3 of [131]). An entry of the stack is the pair of the node and of the g with which it was pushed, and it is *stale* if the g of the node has changed since (line 8, `TDS.h:326-335`). The entries of the table are nodes kept from an iteration to the next and marked with the iteration that last pushed them (`TDS.h:383-408`), so that the table holds each state generated within the bounds, as the CLOSED list of A* does, and no subtree is searched twice in an iteration (the case of a table "large enough" in [131], which does not replace its entries); the node limit, per worker, bounds the table. Each iteration is a computation of the mailbox of Chapter 9, whose end, i.e., no worker with work and no message in transit, is detected by its deficit counting (§9.3); the nodes travel in batches, the outboxes being flushed each `flush_period` expansions and before a worker waits, and the least f pruned is the least over the workers, read after the iteration. The class template is `TDS< D , Dist , Executor >`, the distribution `Dist` being one of those of §9.4 (by default the hash of the state), and its options are the following.

```
struct TDSOptions : public ParallelOptions , public DeepeningOptions {

 TDSOptions( void ) = default;

 /// the options of the parallel searches, with the classical bounds
 TDSOptions( const ParallelOptions & o ) : ParallelOptions( o ) {}

 };  // end( struct( TDSOptions ) )
```

The fields of `DeepeningOptions` (`Deepening.h`) are those of the bounds of IDA*-CR, which IDA* shares (§4.4 describes them for the sequential search): `growth` r (0 by default, the classical bounds) and `optimal` (true by default). Of the inherited fields, `threads`, `batch`, `flush_period` and `node_limit` are read, while the weight is ignored.

Analysis with the classical bounds

We call an iteration *classical* when it is not completed as a branch-and-bound, i.e., when the first goal found stops it (line 9); with `growth` at most 1 each iteration is classical.

Lemma 12.12 (The optimal path in an iteration). Let h be admissible, let $P = (w_0 = s_0, \ldots, w_m)$ be an

optimal path to a goal with no goal before w_m, and consider an iteration of bound β in which no goal is found and the node limit is not reached. Then for some $j \geq 1$ the state w_j is generated with $g' \leq g_P(w_j)$ and pruned at line 11, with $f = g' + h(w_j) \leq C^*$, while $w_0, \ldots, w_{j-1}$ are pushed in the iteration with g not larger than g_P; and if $\beta \geq C^*$, a goal is found in the iteration.

Proof. We show by induction on i that, as long as w_i is not pruned, it is pushed in the iteration with $g \leq g_P(w_i)$, and expanded with the least g with which it was pushed. For $i = 0$ the root is inserted at line 4 with $g = 0$. If w_i is pushed with $g \leq g_P(w_i)$, let γ be the least g with which it is pushed in the iteration; the entry of γ is never stale, since within an iteration the g of a node only decreases (line 18 rejects the larger values), and since the iteration ends only when no worker has work and no message is in transit, the entry is popped and expanded with $g = \gamma$ (the incumbent does not prune in a classical iteration). Its expansion generates w_{i+1} with $g' \leq \gamma + c(w_i, w_{i+1}) \leq g_P(w_{i+1})$ and $f = g' + h(w_{i+1}) \leq g_P(w_{i+1}) + h^*(w_{i+1}) = C^*$; if $f > \beta$, it is pruned at line 11, and otherwise it reaches its owner, which either finds it pushed in the iteration with $g \leq g'$ (line 18) or pushes it with g' (line 19). If no state of P were pruned, the goal w_m would be expanded, and a goal found. Finally, if $\beta \geq C^*$, no state of P is pruned, since each is generated with $f \leq C^* \leq \beta$; hence a goal is found. □

Theorem 12.13 (TDS with the classical bounds). Let h be admissible and the costs positive, and let the bounds be the least f pruned (`growth` at most 1). If TDS ends without reaching the node limit, the path it returns is optimal; if it returns `unsolvable`, no goal is reachable.

Proof. A state is pushed only with $g + h \leq \beta$, so that a goal found in an iteration of bound β has $g \leq \beta$ (h is 0 at a goal). By Lemma 12.12, while $\beta < C^*$ no goal is found (a goal has $g \geq C^*$) and the least f pruned is at most C^*; hence the bounds increase strictly and never skip C^*, and the first iteration in which a goal is found is the one with $\beta = C^*$, in which each goal found has $g \leq C^*$, i.e., $g = C^*$. The path is rebuilt from the goal through the parents after the workers have returned; within the iteration a node's g and parent are set together at line 19 and the g only decreases, so that along the chain the g of a parent at the end is at most the value with which it generated its child, and the chain is a path of cost at most C^* whose nodes were all pushed in the iteration. Its cost is recomputed from the `Domain`, each edge being the cheapest between two consecutive states (`TDS.h:425-438`), which can only lower it; hence it is C^*. Finally, if no goal is reachable, TDS returns `unsolvable` only when an iteration prunes nothing ($\beta' = \infty$), which is right. □

Note that termination needs, as for IDA*, that each iteration be finite and that the bounds reach C^* in finitely many steps, which holds on a finite graph (the table makes each iteration finite even on a graph with cycles, since a state is pushed in an iteration only when its g decreases) and with integer costs on an infinite graph with finite branching; the test `tds-thread` checks the optimal cost on two instances of Korf, on 7 states of the 8-puzzle, on a grid with costs from 1 to 3 and on the map of Romania, and the status `unsolvable` on a grid split by a wall.

The bounds of IDA*-CR

On a grid with 8 moves, or with costs that are not integers, the values of f are many and close, the iterations are as many as the values of f, and each iteration expands again the nodes of the ones before. IDA*-CR of Sarkar, Chakrabarti, Ghose and De Sarkar [132] chooses instead a bound beyond which enough pruned nodes lie. The class template `CutoffHistogram` (`Deepening.h:117`) collects the f of the nodes pruned by an iteration of bound β_k in buckets of the binary exponent of the excess $f - \beta_k$ (the excesses in $[2^{x-1}, 2^x)$ in the bucket x, so that the histogram has a fixed size, 2100 buckets, whatever the costs, and is fine near the bound and coarse far from it), and its member `next( r , E_k )` returns, with E_k the expansions of the iteration and ℓ_k the least f pruned:

- ℓ_k, if $r \leq 1$ (the classical bound);
- otherwise, the largest f of the first buckets that hold at least $(r-1)E_k$ pruned nodes;
- when even all the pruned nodes are fewer than that and E_k is less than r times the expansions of the iteration before, as on a grid, where the children of the last layer of f are all just above it, the bound plus twice its last step, $\beta_k + 2(\beta_k - \beta_{k-1})$, so that the steps grow geometrically while the iterations stay small;

and never less than ℓ_k (`Deepening.h:181-204`). In TDS each worker keeps its histogram, and the histograms are merged after the iteration (`merge()`); IDPSDD does the same with the histograms of the workers of PSDD. The price of a bound $\beta_{k+1} > \ell_k$ is that the first goal found need not be optimal, which the option `optimal` corrects.

Theorem 12.14 (TDS with the bounds of IDA*-CR). Let h be admissible, $r > 1$, and let TDS end without reaching the node limit with a path of cost c, found in the iteration $k+1$. If `optimal` is false, then $c - C^* \leq \beta_{k+1} - \ell_k$, where ℓ_k is the least f pruned by the iteration k (and $c = C^*$ if the goal was found in the first iteration). If `optimal` is true, $c = C^*$.

Proof. As in the proof of Theorem 12.13, a goal found in the iteration of bound β has $g \le \beta$, so $c \le \beta_{k+1}$; and the iteration k, which found no goal, pruned by Lemma 12.12 a node of $f \le C^*$, so $\ell_k \le C^*$. Hence $c - C^* \le \beta_{k+1} - \ell_k$. With `optimal`, an iteration whose bound exceeds the least f pruned before is completed as a depth-first branch-and-bound (`TDS.h:195`): a goal lowers the incumbent z (initially ∞) instead of stopping the workers, the nodes of $f \ge z$ are not pushed and their entries are dropped when popped, and the iteration ends when no worker has work. In the iteration $k+1$ the argument of Lemma 12.12 applies to an optimal path as long as $z > C^*$, since its states have $f \le C^* < z$ and are not pruned by z, and $\beta_{k+1} \ge c \ge C^*$ prunes none of them by the bound; hence the iteration does not end before $z \le C^*$, i.e., $z = C^*$. The path is rebuilt from the goal of cost z as in Theorem 12.13. An iteration whose bound is the least f pruned before (e.g., the first one) is classical, and its first goal is optimal by the argument of Theorem 12.13, since its bound does not exceed C^*. □

The same bounds serve IDPSDD (Theorem 12.6), which needs no completion, and the sequential IDA*, for which §4.4 proves the bound of the cost and the completion. The test `tds-thread` checks TDS with $r = 2$ on a grid with 8 moves and costs from 1 to 3, with `optimal` (the cost of A*) and without it (a valid path not cheaper than the optimum); the test `psdd-thread` checks IDPSDD with $r = 2$ on a grid of 40×30 cells.

The engineering

A worker of TDS (`TDS.h:286-419`) keeps its part of the table, a table of nodes with open addressing, its arena of nodes and its stack, a vector of pairs (node, g). It polls its inbox, inserting each node received, and then pops and expands one entry of its stack, inserting the successors that it owns at once and sending the others, as HDA* does; the parents cross the workers, and they are followed only after all the workers have returned. The goal is recorded under a mutex, and a classical iteration stops the mailbox at the first goal. Since each iteration is a new computation of the mailbox and the table survives it, the cost of an iteration is that of its expansions and messages, and the memory is that of the table, which grows with the states generated within the bounds; TDS is thus a parallel IDA* that uses the memory of A* to search each subtree once, and its iterations still expand the nodes of the iterations before. With one worker the expansions are the same at each run, which the test checks.

12.7 Parallel iterative deepening below a frontier

`ParallelIDAStar` (`ParallelIDAStar.h`) parallelizes the depth-first iterations of IDA* in a different way: it first builds a *frontier*, a set of open nodes, the *units*, and then runs the iterations below the units on the p workers. Each unit u keeps a *threshold* t_u, initially $f(u) = g(u) + h(u)$, and its cost-bounded search with a bound $\beta \ge t_u$ returns, besides a goal of cost at most β if one exists below u, the least f of the nodes that it cuts, which becomes t_u. The frontier is built by one of three rules (`FrontierRule`):

- `astar`: A*, with reopening, until OPEN holds N live nodes, as in the AIDA* of Reinefeld and Schnecke [55] and in the A*+IDA* of Bu and Korf [56];
- `breadth`: uniform-cost search, by g, until the same, a breadth-first frontier with unit costs;
- `depth`: the tree of the least depth with N leaves, without duplicate detection (a goal is a leaf, never expanded), the work units of the parallel IDA* of Sturtevant and of Siag et al. [57].

If the first two rules expand a goal, or empty OPEN, the frontier ends the search, with the optimal path (A* with reopening and an admissible h, or uniform-cost search with positive costs, select a goal only with its optimal cost) or with the status `unsolvable` (`ParallelIDAStar.h:369-463`). Two schemes share the units.

Algorithm 12.6 (Parallel IDA* below a frontier, the synchronous scheme and the asynchronous one).

```
1  procedure PIDA*(s₀, p, N)
2     build the units U by the frontier rule (return if it ends the search)
3     t_u ← f(u) for each u ∈ U
4     loop                                          ▷ the synchronous scheme
5        β ← min_u t_u;  if β = ∞ then return unsolvable
6        A ← {u ∈ U : t_u ≤ β}        ▷ all the units without unit bounds
7        in parallel, the workers take the units of A
            (by work stealing, or from a shared cursor):
8           (goal, t'_u) ← DFS(u, g(u), β);  t_u ← t'_u
9           if goal then stop all the workers;  return its path

1  procedure AIDA*(s₀, p, N)                     ▷ the asynchronous scheme
2     build U;  z ← ∞;  Q ← U by increasing t_u
```

```
3     in parallel, each worker repeats until Q = ∅ and no worker searches:
4        u ← the first unit of Q, removed;  if t_u ≥ z then drop it
5        (goal, t'_u) ← DFS(u, g(u), t_u) cutting also f ≥ z
6        if goal then z ← min(z, its cost), recording its path
7        t_u ← t'_u;  if t_u < z then put u back in Q
8     return the path of cost z, or unsolvable
```

In the synchronous scheme each iteration ends at a barrier, whose last worker computes the thresholds, the next bound and the units of the next iteration (ParallelIDAStar.h:641-770); in the asynchronous one there is no barrier, the workers take the units by increasing threshold, each searched with its own threshold, and a goal of cost z prunes each node with $f \geq z$. The DFS of a unit is that of IDA* (§4.4) with an explicit stack of the successors not yet tried, the moves back to the parent skipped (by the operator in an `OperatorDomain`, with the redundant pairs of moves of a `TreeOperatorDomain`, otherwise by comparing the successor with the state two places back on the path), and the goal test when a node is reached within the bound (ParallelIDAStar.h:904-1013). The options are the following.

```
enum class FrontierRule : std::uint8_t {
 astar ,     ///< A* until OPEN holds the units wanted
 breadth ,   ///< uniform-cost search (by g) until the same
 depth       ///< the tree of the least depth with that many leaves
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// how the units of an iteration are given to the workers

enum class UnitSchedule : std::uint8_t {
 steal ,     ///< a range of units per worker, half stolen when it is empty
 shared      ///< one cursor over the units, shared by all the workers
 };
```

```
struct PIDAOptions : public ParallelOptions {

 /// the units wanted (0: 128 per worker)
 std::size_t frontier = 0;

 /// the rule that builds the frontier
 FrontierRule rule = FrontierRule::astar;

 /// the scheduling of the units of an iteration
 UnitSchedule schedule = UnitSchedule::steal;

 /// a unit is searched only when the bound reaches its threshold; false
 /// searches every unit at every bound
 bool unit_bounds = true;

 /// no barrier between the iterations: the units are searched by
 /// increasing threshold, each with its own (AIDA*)
 bool asynchronous = false;

 /// with a value s > 0, a worker that sees an idle one gives it half of
 /// the successors not yet tried at the shallowest level of its stack,
 /// those whose f is at least s below the bound; 0 disables the splitting
 double split = 0;

 /// the levels below a unit whose nodes are looked up in the table of the
 /// frontier, and pruned if found there with a g not larger
 unsigned table_depth = 0;

 /// the path found is that of the first goal in the order of the units,
 /// the same at every run
 bool deterministic = false;
```

```
};  // end( struct( PIDAOptions ) )
```

The field `frontier` is N (0 gives 128 per worker); `unit_bounds` selects line 6 as written, and false searches each unit with a finite threshold at each bound, the classical scheme; `asynchronous` selects the second scheme; `split` and `table_depth` are described below. Of the inherited fields, `threads` and `node_limit` are read (the latter bounds the expansions of each worker in the iterations), while `batch` and `flush_period` are not; the constructor throws `std::invalid_argument` for a weight other than 1, for `deterministic` with the asynchronous scheme or with splitting, for a table with the rule `depth`, which has no table, and for a negative `split` (ParallelIDAStar.h:238-256). `parallel_idastar( d , o )` runs the search on the default executor, and `iterations()` and `units()` give the iterations (in the asynchronous scheme, the searches of units) and the units of the last `solve()`.

Optimality

The proofs rest on one unit, which keeps a threshold not above C^* until an optimal goal is found below it. We assume that the frontier was built by the rule `astar` or `breadth` and did not end the search, and that the tree pruning of the DFS (the moves back to the parent and the redundant pairs of moves of a `TreeOperatorDomain`) leaves, from each state, an optimal path to a goal, as §4.4 shows for IDA*; for the rule `depth`, see the remark after Theorem 12.16.

Lemma 12.15 (The best unit). Let h be admissible, the costs positive, and $C^* < \infty$. There is a unit u^* whose node has $g(u^*) = g^*(u^*)$ and lies on an optimal path, such that (i) no state of an optimal path from u^* to a goal, after u^*, is in the table of the frontier with its optimal g; hence the lookups in the table, which prune a state found there with a g not larger, never prune such a path; and (ii) as long as no goal of cost C^* has been found below u^*, its threshold is at most C^*.

Proof. We use the invariant of A* with reopening (Chapter 2), which holds for both rules, whatever their order: a node closed has been expanded with its current g, and when a node with its optimal g is expanded, each successor gets a g not larger than the cost through it. The first state of an optimal path that is not closed with its optimal g is open with it; hence there are open nodes with their optimal g on optimal paths, and we let u^* be one of them with $g^*(u^*)$ largest. It is a unit, since each live node of OPEN is one. (i) Let S be an optimal path from u^* to a goal, and suppose a state w of S after u^* is in the table with $g(w) = g^*(w)$. If w is open, it contradicts the choice of u^*, since $g^*(w) > g^*(u^*)$. If it is closed, then, by the invariant, the successor of w on S has its optimal g, and so on, so that the first state of S after w that is not closed with its optimal g is open with it (the goal of S is not closed, since a goal expanded by the frontier ends the search); this again contradicts the choice of u^*. A state of S reached by the DFS of u^* along S has its optimal g, and the table holds it, if at all, with a larger g; hence it is not pruned. (ii) Initially $t_{u^*} = f(u^*) \le g^*(u^*) + h^*(u^*) = C^*$. A search of u^* with a bound $\beta < C^*$ follows S (by (i) and by the assumption on the tree pruning, S chosen among the paths that the pruning leaves), whose states have $f \le C^*$, and since the goal of S has $f = C^* > \beta$, the search cuts the first state of S with $f > \beta$, of $f \le C^*$; the least f cut, the new threshold, is then at most C^*. With splitting, the pieces of the unit report their least cut f to it, and the threshold is the least of them, so that the cut of the piece that holds S counts. □

Theorem 12.16 (The synchronous scheme). Let h be admissible and the costs positive. If the synchronous scheme ends without reaching the node limit, the path it returns is optimal; if it returns `unsolvable`, no goal is reachable below the units (on a graph whose cycles the tree pruning removes, no goal is reachable).

Proof. If the frontier ends the search, it is right, as remarked above. Otherwise, by Lemma 12.15 (ii), $\beta = \min_u t_u \le t_{u^*} \le C^*$ at each iteration while no optimal goal has been found, and a goal found in an iteration has $g \le \beta$, since the DFS reaches only states with $f \le \beta$; hence the first goal found has $g \le C^*$, i.e., $g = C^*$, and each worker may stop. The path is the path to the node of the unit, from the frontier, followed by the path of the DFS, whose cost is the g of the goal. If $\beta = \infty$, each threshold is ∞, i.e., the last search of each unit cut nothing and found no goal, and the subtree below each unit has been searched in full. □

For the rule `depth` each path from s_0 of at least the depth of the tree passes through a leaf (a goal reached earlier is a leaf as well), and the search below the leaf with its threshold is the search of IDA* from there; the argument of IDA* then gives the optimality, since the unit on an optimal path keeps a threshold not above C^* as in (ii), and there is no table to prune. The scheme without unit bounds (`unit_bounds` false), which searches each unit at each bound as the parallel IDA* of Sturtevant and of Siag et al. [57] does, has the same proof. The unit bounds save work without changing the result.

Lemma 12.17 (Unit bounds). In the synchronous scheme, let the unit u have $t_u > \beta$, its last search having had

the bound $\beta' < t_u$. Then its search with the bound β would expand the same states as its last one and find no goal; hence skipping it (line 6) changes neither the result nor the thresholds, and the expansions of the scheme with unit bounds are, in each iteration, at most those of the classical scheme.

Proof. The last search of u cut the states of $f > \beta'$, the least of which had $f = t_u$; hence no state reached below u has f in (β', t_u), and a search with $\beta \in [\beta', t_u)$ reaches the same states, which have $f \leq \beta'$, cuts the same ones, and finds no goal, since the last search found none (otherwise the scheme would have stopped). The table and the tree pruning are the same at each iteration. □

Theorem 12.18 (The asynchronous scheme). Let h be admissible and the costs positive. If the asynchronous scheme ends without reaching the node limit, the incumbent is optimal, and the status is `unsolvable` only if no goal is reachable below the units.

Proof. The scheme ends when Q is empty and no unit is being searched. Suppose $z > C^*$ at the end. A search of u^* with its threshold $t \leq C^* < z$ follows an optimal path S from u^* (whose states have $f \leq C^* < z$, so that the incumbent does not cut them) until it finds the goal of S, which sets $z \leq C^*$, or cuts a state of S of $f \leq C^*$, so that its new threshold is at most $C^* < z$, and line 7 puts u^* back in Q. Since a unit is dropped from Q only when its threshold is not below z (line 4), u^* is in Q or being searched at each instant while $z > C^*$, against the end. Hence $z \leq C^*$, and z is the cost of a path. If no goal is found, each unit left Q with an infinite threshold, and its subtree was searched in full. □

In the asynchronous scheme the threshold of a unit is thus a lower bound on the cost of each goal below it not found yet, and z is optimal once each threshold is at least z, which is when the search ends; the scheme saves the idle time at the barriers, and it may waste the work of the units searched with a threshold above C^* before the goal is found.

Scheduling and splitting

In each iteration of the synchronous scheme the units of A are sorted by the decreasing number of the expansions of their last search, the *longest processing time first* of Graham [66], since the expansions of a unit grow by a similar factor at each bound, and given to the workers in one of two ways (`UnitSchedule`): `steal`, which deals the sorted units to the workers in turn, each worker taking first from its own range (the units that it searched before, in the order of the deal) and, when its range is empty, stealing the upper half of the range of another worker by a compare-and-swap on a word that packs the two ends of the range (`ParallelIDAStar.h:808-833`); and `shared`, one cursor over the sorted units, taken by a fetch-and-add. A single unit can still hold most of an iteration, and with `split` $= \sigma > 0$ the splitting of the stacks of Rao and Kumar [58] balances it: a worker that sees an idle one (a counter of the workers that want work, above the number of the pieces queued) gives it half of the successors not yet tried at the shallowest level of its stack that has some, those whose f is at least σ below the bound, as *pieces*, each with the states from the unit to it (`ParallelIDAStar.h:1055-1094`). A piece is searched as a unit, with the bound of the unit, and reports its least cut f to the unit, whose threshold is the least over its pieces; a counter of the searches of the unit not ended tells the last one, which in the asynchronous scheme puts the unit back in Q (`ParallelIDAStar.h:882-896`). The proofs above hold with splitting, since a piece searches a part of the tree of its unit with the same bound, and the threshold is the least f cut over all the parts (Lemma 12.15 (ii)). The slack σ keeps a worker from giving away the successors that are almost cut, whose subtrees are small.

With `table_depth` $= d > 0$ (rules `astar` and `breadth`), the states of the first d levels below a unit are looked up in the table of the frontier, and a state found there with a g not larger is pruned; this removes many transpositions near the frontier at the price of a lookup per node, and Lemma 12.15 (i) shows that it never prunes the optimal suffix below u^*. With `deterministic` (synchronous scheme, no splitting), a goal found in the unit u (in the order of the frontier) abandons the units after u, which keep their thresholds, and the path returned is that of the first unit, in that order, below which a goal is found; it is the same at each run, with any number of workers, since the frontier, the sorting of the units by their work and the DFS of a unit do not depend on the scheduling, and each unit before the first one with a goal is searched in full in the last iteration.

The counters of the `Solution` add those of the frontier (`expanded`, `generated`, and the duplicates of its table) to those of the iterations; `duplicates` counts also the states pruned by the table in the iterations, `sent` the units stolen and the states given away by splitting, `messages` the steals and the splits, and `stored` the nodes of the frontier plus the longest path of each worker. With one unit (`frontier` $= 1$) and one worker the search is IDA*, with the same expansions, generations and path, which the test `pida-thread` checks on two instances of Korf.

Over MPI

`ParallelIDAStarMPI` (`ParallelIDAStarMPI.h`) runs the synchronous scheme on the processes of an MPI communicator, one worker per process. Each process builds the same frontier, since the rules are deterministic, so that the units need not travel, and an iteration gives each process a set of units and ends with two reductions over all the units: the minimum of the thresholds and the maximum of the expansions (a unit is searched by one process, and the others contribute ∞ and 0), after which each process knows the next bound and the next assignment. The pseudocode of the library is the following.

Algorithm 12.7 (Parallel IDA* over MPI, on the process i of p).

```
1  procedure PIDA*-MPI(s_0) on the process i of p
2     build the units U by the frontier rule;  t_u ← f(u), w_u ← 0
3     loop
4        β ← min_u t_u;  if β = ∞ then return unsolvable
5        A ← {u : t_u ≤ β}, by decreasing w_u (then by u)
6        give each unit of A in turn to the process of least load,
            the load growing by max(w_u, 1)
7        for each unit u given to i: (goal, t_u, w_u) ← DFS(u, β),
            a goal telling all the other processes to stop
8        t ← min-reduce(t);  w ← max-reduce(w)  ▷ others: ∞ and 0
9        if some process has a goal then broadcast its path; return it
```

The assignment of line 6 is the longest processing time first of Graham [66] on the expansions of the last search of each unit, which the processes compute alike from the same reduced vectors. A process that finds a goal sends a message to the others, which poll for it each 1024 expansions and stop (a unit not searched keeps its threshold); after the reductions, the process with the goal of least cost broadcasts its path, and each process returns the same `Solution`, with the counters summed over the processes. The options are those of `ParallelIDAStar`, of which `threads` (the processes of the communicator), `schedule`, `asynchronous`, `split`, `table_depth` and `deterministic` are not used; MPI must be initialized by the caller, each process of the communicator calls `solve()`, and `State` and `Cost` must be trivially copyable, since the path travels as bytes.

Proposition 12.19 (Over MPI). Under the assumptions of Theorem 12.16, `ParallelIDAStarMPI` returns on each process the same optimal path, or `unsolvable` when no goal is reachable below the units.

Proof. The iterations are those of the synchronous scheme, whose units are distributed by line 6 instead of being stolen; the reductions of line 8 give each process the thresholds that the barrier of the synchronous scheme computes, since a unit is searched by one process only (or keeps its threshold, if a stop arrived first), and Theorem 12.16 applies. The stops are drained before the next search (each process learns by a reduction how many stops were sent, and receives them), so that a stop of an iteration never ends a search of the next one; the winning path is that of the least cost, which is C^*, chosen by the reduction `MPI_MINLOC`, and the broadcast gives it to each process. □

We remark that the thresholds travel as `double`, so that integer costs above 2^{53} would be rounded; the costs of the domains of the library are far below that. The test `pida-mpi` runs it with each frontier rule on the 15-puzzle, on the pancakes, on the Towers of Hanoi and on the map of Romania.

12.8 Parallel greedy best-first search

The greedy best-first search (GBFS, Chapter 2) selects the node of least h and gives up the optimality of the path for speed. Unlike A*, for which a parallel search must expand each node of $f < C^*$ and need expand no other, GBFS fixes no set of states that a correct parallelization has to expand, and the question is how far a parallel GBFS may stray from the sequential one. We first recall the characterization of the states that GBFS may expand, due to Heusner, Keller and Helmert [112], and then give the searches of `ParallelGreedy.h` and what they guarantee with respect to it.

The bench transition system

Throughout the section the state space is finite, a goal has no successor and $h = 0$ (as in the definitions 3 to 6 of Shimoda and Fukunaga [110]), and GBFS has no reopening and tests the goal when it selects a node. A *breaking of the ties* is any rule, possibly depending on the past of the search, that chooses among the nodes of

OPEN of least h; GBFS with a given breaking of the ties is deterministic, but different breakings expand very different sets of states.

Definition 12.20 (Benches). The *high-water mark* of a state s is $\mathrm{hwm}(s) = \min_P \max_{s' \in P} h(s')$ over the paths P from s to a goal (∞ if there is none). A state s is a *progress state* if it is a goal or $\mathrm{hwm}(s) > \min_{t \in \mathrm{succ}(s)} \mathrm{hwm}(t)$. The *bench* of a progress state s has the level $\ell = \min_{t \in \mathrm{succ}(s)} \mathrm{hwm}(t)$, the *inner states* reachable from s through non-progress states of $h \leq \ell$, and the *exits*, the progress states of $h = \ell$ among the successors of s and of its inner states. The *bench transition system* (BTS) is the set of the states of the benches reached from the bench of s_0 through their exits that are not goals.

Theorem 12.21 (Heusner, Keller and Helmert). A state is expanded by GBFS with some breaking of the ties if and only if it lies in a bench of the BTS (theorem 1 of [112]).

We do not prove the theorem, which is the subject of [112]. We remark that, since $\mathrm{hwm}(s) = \max(h(s), \min_t \mathrm{hwm}(t))$, the high-water marks of a finite space are the distances of a shortest-path problem in which a path costs the largest h along it, solved by Dijkstra's algorithm backward from the goals with max in place of +; the tester computes the BTS in this way on random graphs and on the 8-puzzle, and checks it against GBFS with the breakings FIFO and LIFO before it uses it on the parallel searches (`pgbfs-rounds`). We call a parallel search *BTS-constrained* if it expands only states of the BTS, and *bounded* if its expansions are at most those of GBFS, with some breaking of the ties, plus a quantity that does not depend on the instance but on its path and on the workers.

The searches

All the parallel searches of `ParallelGreedy` share one OPEN and one CLOSED list, take the first node of OPEN only under a condition of their rule, and insert the successors of a node all together, so that they are ordered among themselves as GBFS orders them; this is the schema of the constrained parallel GBFS of algorithm 1 of Shimoda and Fukunaga [111]. Let s_j be the node that the worker j expands, $h(s_j) = \infty$ for an idle worker, and $\underline{h} = \min_j h(s_j)$ the least h of the nodes in expansion. The rules (`GreedyRule`) are the following.

- KPGBFS, the K-parallel GBFS of Vidal, Bordeaux and Hamadi [106]: no condition.
- PUHF of Kuroiwa and Fukunaga [108]: only a node marked *certain*, which a successor s' of a node s is when $h(s') = \min_{t \in \{s\} \cup \mathrm{succ}(s)} h(t)$, and the first node of OPEN is when no worker is expanding.
- PUHF2 of Shimoda and Fukunaga [109]: a certain node, or one with $h \leq \underline{h}$, which makes it certain; PUHF3 marks certain, in addition, all the nodes of OPEN with $h = \underline{h}$ after each removal; PUHF4 adds the criterion of PUHF to PUHF3 and prefers the certain nodes among those of equal h.
- SPUHF [108]: PUHF, and a worker that has no certain node expands a node of a *speculative* OPEN list, whose states share only their evaluations with the main search; a goal found by the speculation ends the search.
- OBAT, One Bench At a Time, of Shimoda and Fukunaga [110]: $h \leq \underline{h}$; a node with a successor of smaller h (a *b-state*, the only kind of node that can be a progress state; the others are *a-states*) does not insert its successors, but enters a queue DEFERRED, ordered by h, and its successors enter OPEN only when it leaves DEFERRED, which happens when the least h of DEFERRED is at most that of OPEN and at most $\underline{h}$ (algorithm 2 of [110]).

Each rule may run with the separate generation and evaluation of Shimoda and Fukunaga (SGE, [111]), in which the successors of a node are generated by the worker that removes it and evaluated by any idle worker, the node staying in expansion until all its successors are evaluated and inserted. The options and the record of a search are the following.

```
enum class GreedyRule {
 kpgbfs ,     ///< no constraint
 puhf ,       ///< the certain nodes of PUHF
 puhf2 ,      ///< certain, or h at most that of every node in expansion
 puhf3 ,      ///< PUHF2, the nodes of the least h in expansion certain
 puhf4 ,      ///< PUHF3 with the criterion of PUHF, certain ones first
 spuhf ,      ///< PUHF, with a speculative search for the idle workers
 obat         ///< One Bench At a Time
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// the options of ParallelGreedy
```

```
struct ParallelGreedyOptions : public ParallelOptions {

 GreedyRule rule = GreedyRule::kpgbfs;   ///< the rule

 /// the successors of a node are generated by the worker that removes it
 /// and evaluated by any worker (SGE)
 bool sge = false;

 /// the order among the nodes of equal h: fifo or lifo (the others are
 /// taken as fifo)
 TieBreak tie = TieBreak::fifo;

 /// the certain nodes come first among those of equal h (always with
 /// PUHF4)
 bool certain_first = false;

 /// the workers are emulated in synchronous rounds in the calling thread
 bool rounds = false;

 };  // end( struct( ParallelGreedyOptions ) )
```

```
template< class State >
struct GreedyTrace {

 /// the states removed from OPEN by the main search, in the order of
 /// their removals
 std::vector< State > expanded;

 /// the states completely expanded, in the order of their commits, i.e.,
 /// when they left OPEN or (a b-state of OBAT) DEFERRED for the last time;
 /// neither the goal nor the states still in expansion at the end are
 std::vector< State > committed;

 /// the states left in DEFERRED at the end (OBAT)
 std::vector< State > deferred;

 std::uint64_t speculative = 0;      ///< the speculative expansions
 bool goal_by_speculation = false;   ///< the goal was found by them
 std::uint64_t rounds = 0;           ///< the rounds, if emulated

 };  // end( struct( GreedyTrace ) )
```

The field `tie` (`TieBreak`, Chapter 2) orders the nodes of equal h by their arrival (FIFO) or the opposite (LIFO; the other breakings of the sequential search are taken as FIFO); `certain_first`, always set with PUHF4, puts the certain nodes before the others of equal h; and `rounds` emulates the $k = p$ workers in the calling thread, in synchronous rounds, in each of which every idle worker in turn takes its work under the rule, then all of them do it, then they insert their results in turn, so that each expansion (or, with SGE, each generation and each evaluation) takes one round. The search is then replicable, and its counts do not depend on the machine, which the tests of the bounds below need; the bounds hold for each interleaving, hence for this one. Of the inherited fields, `threads` is k and `node_limit` bounds the states stored; the weight is not read. After `solve()` , `trace()` gives the `GreedyTrace` of the search, with the states only if `set_record( true )` was called: the states removed by the main search in the order of their removals, the states *committed* (see below) in their order, the states left in DEFERRED at the end, the speculative expansions, whether the goal was found by them, and the rounds. The counter `expanded` of the `Solution` counts the removals of the main search plus the speculative expansions, `evaluated` the states evaluated, and `sent` is 0.

The code of a worker is the following, under one lock that protects OPEN, DEFERRED, the multiset of the values h of the nodes in expansion and the successors waiting for their evaluation (`ParallelGreedy.h:632-729` and `796-873`).

Algorithm 12.8 (The step of a worker w of the parallel greedy searches; hmin is $\underline{h}$, the least h in expansion, ∞

if none).

```
 1  function ACQUIRE(w)                              ▷ under the lock
 2     if SGE and a successor waits for its evaluation then
          w takes it;  return
 3     if OBAT and DEFERRED ≠ ∅ and (OPEN = ∅ or the least h of
          DEFERRED ≤ the least h of OPEN) then
 4        if the least h of DEFERRED ≤ hmin then
             commit its first node d: LIST(d)
 5        return
 6     if OPEN = ∅ then
 7        if nothing is in expansion, DEFERRED = ∅ and no successor
             waits then the search is over
 8        else if SPUHF then w takes a node of the speculative OPEN
 9        return
10     t ← the first node of OPEN
11     PUHF, SPUHF:  if nothing is in expansion then t is certain;
                     ok ← t is certain
       PUHF2-PUHF4:  if h(t) ≤ hmin then t is certain;  ok ← t is certain
       OBAT:         ok ← h(t) ≤ hmin;   KPGBFS: ok ← true
12     if not ok then (SPUHF: w speculates);  return
13     remove t from OPEN;  t is in expansion
14     PUHF3, PUHF4: if h(t) = hmin then every node of OPEN with h = h(t)
          is certain
15     if t is a goal then the search is over, with the path to t
16  procedure WORK(w)                                ▷ without the lock
17     generate the successors of t, and evaluate them unless SGE
18  procedure FINISH(t)        ▷ under the lock, all the successors evaluated
19     if OBAT and a successor of t has h < h(t) then
          put t in DEFERRED;  t is not in expansion;  return
20     LIST(t);  t is not in expansion
21  procedure LIST(t)
22     for each successor v of t not in CLOSED, with h(v) < ∞, do
23        put v in CLOSED, with parent(v) ← t
24        PUHF, SPUHF, PUHF4: if h(v) = min over {t} ∪ succ(t) of h then
             v is certain
25        put v in OPEN
```

A state *committed* is one that leaves OPEN, or (a b-state of OBAT) DEFERRED, for the last time: the a-states at their removal (line 13) and the b-states at line 4; the goal and the states in expansion at the end are not counted among the committed ones. CLOSED is the set of the states that have entered OPEN (a flag of the node), while the table of all the states generated serves also as the cache of their evaluations, each state being evaluated once.

What the rules guarantee

We first note that one worker is GBFS.

Proposition 12.22 (One worker). With one worker, each rule but PUHF4, with or without SGE, removes the states that GBFS with the same breaking of the ties expands, in the same order, and returns its path.

Proof. With one worker, $\underline{h} = \infty$ whenever the worker acquires a node of OPEN, except while the node is waiting for its evaluations under SGE, when the worker evaluates them (line 2) and acquires no node. Hence KPGBFS, PUHF2, PUHF3 and OBAT accept the first node of OPEN, and PUHF and SPUHF find it certain, since nothing is in expansion; SPUHF never speculates. In OBAT a b-state t, removed as the first node of OPEN, enters DEFERRED, and at the next acquisition the least h of DEFERRED, $h(t)$, is at most that of OPEN, so that t is committed at once and its successors inserted, as GBFS does. PUHF4 orders the nodes of equal h otherwise, by `certain_first`. □

The test `pgbfs-rounds` checks the proposition, by the counters `expanded` and the costs, on 6 instances of Korf with FIFO and LIFO. With more workers, KPGBFS is *pathological*, i.e., for each t there is an instance on which it expands more than t times the states of GBFS (Kuroiwa and Fukunaga [108]); its simplest witness is a state

outside the BTS, which a worker takes while another one expands the state that GBFS would take.

Proposition 12.23 (KPGBFS leaves the BTS). On the graph with s_0 ($h = 3$), its successors a ($h = 1$) and b ($h = 2$), the goal as the only successor of a, and a chain of states of $h = 2$ leading to no goal below b, KPGBFS with two workers in rounds expands b, which is not in the BTS, while PUHF does not.

Proof. GBFS with any breaking of the ties expands s_0, then a ($h = 1 < 2$), then selects the goal; hence b is not in the BTS by Theorem 12.21. In the first round one worker removes s_0, and the other finds OPEN empty; in the second, the first worker removes a and the second, under no constraint, removes b. Under PUHF, a is certain ($h(a) = 1$ is the least of $\{3, 1, 2\}$) and b is not, so that in the second round the second worker, which finds b first in OPEN with a node in expansion, removes nothing; in the third round the goal, certain, is removed. □

The constrained rules expand only states of the BTS. For PUHF, the argument for the successors is elementary.

Lemma 12.24 (The certain successors). If s is in the BTS and s' is a successor of s with $h(s') = \min_{t \in \{s\} \cup \mathrm{succ}(s)} h(t)$, then s' is in the BTS.

Proof. By Theorem 12.21, some GBFS, with a breaking of the ties T, expands s; when it selects s, each node of its OPEN has $h \geq h(s)$. If that GBFS has already expanded s', then s' is in the BTS. Otherwise, after the expansion of s, the node s' is in OPEN (it was there, or is inserted now), and $h(s') \leq h(s)$ and $h(s') \leq h(t)$ for each successor t of s, so that s' has the least h of OPEN; the breaking of the ties that follows T up to s and selects s' next is a breaking of the ties, and the GBFS that uses it expands s'. □

That the first node of OPEN is in the BTS when no worker is expanding is theorem 2 of [108], whose proof we do not repeat; with Lemma 12.24 it gives, by induction on the removals, that PUHF, and the main search of SPUHF, are BTS-constrained (a state is removed only when certain). For the other rules we rely on lemma 1 of [109], which we use in the following form: in a BTS-constrained parallel search on an instance with a reachable goal, a node n of OPEN of least h that is not in the BTS has a path to a goal of high-water mark below $h(n)$ that leaves the bench of its parent, and a state of that path, of $h < h(n)$, lies in OPEN or is in expansion.

Proposition 12.25 (PUHF2, PUHF3, PUHF4, OBAT). On an instance with a reachable goal, the searches PUHF2, PUHF3, PUHF4 and OBAT, with or without SGE, are BTS-constrained.

Proof. By induction on the removals. A node n removed under PUHF2 or OBAT is the first node of OPEN, hence of least h in OPEN, and has $h(n) \leq \underline{h}$, so that no state in expansion has $h < h(n)$; by lemma 1 of [109], n is in the BTS (the search being BTS-constrained before the removal). A node marked certain by PUHF3 at line 14 has $h = h(t) = \underline{h}$, and $h(t)$ is at most the h of each node of OPEN, since t was the first of OPEN; hence it is a node of OPEN of least h with no state of smaller h in expansion, and the same lemma applies. PUHF4 adds the criterion of PUHF, Lemma 12.24, and in OBAT the commits of line 4 insert successors but remove no node from OPEN. Under SGE a node is in expansion until all its successors are evaluated and inserted, so that $\underline{h}$ is computed over the same set. □

BTS-constrained does not mean bounded: Shimoda and Fukunaga ([110], section 3) show two benches entered at once by two workers, both in the BTS but never both explored by one GBFS, and nest the construction to make the ratio to GBFS unbounded. The first level of their construction is the following.

Proposition 12.26 (PUHF explores two benches). Let s_0 ($h = 10$) have the successors A and B ($h = 5$), each the head of a line of x states of $h = 4$ that leads to a common goal. GBFS expands $x + 3$ states with each breaking of the ties, while PUHF with two workers in rounds removes $2x + 4$ states, and OBAT with two workers removes $x + 3$.

Proof. GBFS expands s_0, then A or B, then the x states of its line (of $h = 4 < 5$), and selects the goal. Under PUHF, the first round removes s_0, whose successors A and B are both certain ($h = 5$ is the least of $\{10, 5, 5\}$); in the second round the two workers remove A and B, and in each of the next x rounds the next state of each line, certain since its $h = 4$ is the least of its parent and of itself. In the round after the last states of the lines one worker removes the goal: $1 + 2 + 2x + 1 = 2x + 4$ removals. Under OBAT, s_0 is a b-state, deferred and then committed; one worker removes A, a b-state that is deferred, and when A is committed its first successor ($h = 4$) enters OPEN, after which each removal of OPEN is of a state of $h = 4$ while one of them is in expansion, so that B ($h = 5 > \underline{h} = 4$) is never removed, and neither is it committed from DEFERRED; the removals are s_0, A, the line of A and the goal. □

The test `pgbfs-rounds` builds the instance with $x = 10, 100, 1000$ and 2 and 4 workers, and checks that PUHF removes at least $2x + 2$ states (the GBFS count plus $x - 1$) and that OBAT stays within the bound of Theorem

12.30 below. The next results are those of OBAT.

Theorem 12.27 (OBAT commits in an order of GBFS). Let $c_1, c_2, \ldots$ be the states committed by OBAT, in the order of their commits, with the a-states in expansion counted as committed at their removal. Then, for each i, c_i has the least h among the states generated by $c_1, \ldots, c_{i-1}$ (and s_0) and not among them; hence the sequence is the beginning of the expansions of GBFS with some breaking of the ties, and so is the sequence followed by the goal, when OBAT finds one.

Proof. Let O_i be the set of the states generated by $c_1, \ldots, c_{i-1}$ (with s_0) and not among them, i.e., the OPEN of the GBFS that has expanded $c_1, \ldots, c_{i-1}$ in this order (GBFS keeps the first parent of a state, as OBAT does). At the commit of c_i, each state v of O_i is (a) in OPEN, or (b) a node in expansion (removed, not yet finished, and a b-state, since an a-state in expansion is committed), or (c) in DEFERRED, or (d) a successor, not yet inserted, of an a-state in expansion, which is committed; this is because a state of O_i that has entered CLOSED is in OPEN until it is removed, then in expansion until it finishes, then committed or in DEFERRED, and a state of O_i that has not entered CLOSED is a successor of a committed state whose successors have not been inserted, i.e., of an a-state in expansion (a b-state inserts its successors at its commit, line 4). Now let c_i be committed from OPEN (line 13, an a-state): it is the first node of OPEN; it has $h(c_i) \leq \underline{h}$, which bounds (b) and, since an a-state has no successor of smaller h, also (d); and the least h of DEFERRED, if any, exceeds that of OPEN (otherwise line 3 would have applied), which bounds (c). Let instead c_i be committed from DEFERRED (line 4): it has the least h of DEFERRED, which is at most that of OPEN (line 3) and at most $\underline{h}$ (line 4), which bounds (a), (b) and (d). In both cases $h(c_i)$ is the least h of O_i, and $c_i \in O_i$. The goal is removed from OPEN under the same conditions as an a-state, and hence it has the least h of O_i as well. □

Lemma 12.28 (DEFERRED by values of h). Without SGE, at each instant, for each value v, the states of $h = v$ that are in DEFERRED or in expansion are at most k, if at least one of them is in DEFERRED.

Proof. Let D be the states of $h = v$ in DEFERRED at an instant τ, $D \neq \emptyset$, and let $\tau_0 \leq \tau$ be the last instant before τ at which a state of $h = v$ entered DEFERRED with no state of $h = v$ there, so that from τ_0 to τ DEFERRED always holds a state of $h = v$ and its least h is at most v. Then no state of $h \geq v$ is removed from OPEN in $(\tau_0, \tau]$, since line 3 applies whenever the first node of OPEN has $h \geq v$. Hence each state of $h = v$ that is in DEFERRED or in expansion at τ was removed from OPEN before τ_0 and was in expansion just before τ_0 (it enters DEFERRED only from expansion); without SGE each worker has at most one state in expansion, and the states in expansion just before τ_0 are at most k. □

Lemma 12.29 (DEFERRED by the path). Let OBAT find the path P. Then each state left in DEFERRED at the end has the h of a state of P.

Proof. Let d be left in DEFERRED, with $h(d) = v$; it entered DEFERRED at τ, after its removal from OPEN at $\tau_0 < \tau$, at which $h(d) \leq \underline{h}$ and $h(d)$ was the least h of OPEN. From τ to the end, the least h of DEFERRED is at most v, and, as in the proof of Lemma 12.28, no state of $h \geq v$ leaves OPEN after τ, and no state of $h > v$ leaves DEFERRED (it would need the least h of DEFERRED). Let $P = (s_0 = w_0, \ldots, w_m)$, each w_{j+1} inserted by the commit of w_j, and let j be the largest index such that w_j has been inserted in OPEN by τ_0; $j < m$, since the goal leaves OPEN only at the end, after τ_0, unless w_m is inserted by τ_0, a case treated below. At τ_0, w_j is in OPEN, or in expansion, or in DEFERRED (it is not committed with its successors inserted, by the choice of j). If w_j is in OPEN, then $v \leq h(w_j)$, and w_j leaves OPEN after τ_0: not after τ, where states of $h \geq v$ do not leave OPEN, so in (τ_0, τ), while d is in expansion, i.e., with $h(w_j) \leq \underline{h} \leq v$; hence $h(w_j) = v$. If w_j is in expansion, then $v \leq \underline{h} \leq h(w_j)$; if w_j is an a-state, w_{j+1} has $h \geq h(w_j) \geq v$ and is inserted after τ_0, and it leaves OPEN, which it must do for P to be found, only in (τ_0, τ), with $h(w_{j+1}) \leq v$, so that $h(w_{j+1}) = v$; if w_j is a b-state, it enters DEFERRED and must leave it for w_{j+1} to be inserted, either in (τ_0, τ), with $h(w_j) \leq \underline{h} \leq v$, or after τ, as the least of DEFERRED, with $h(w_j) \leq v$; in both cases $h(w_j) = v$. If w_j is in DEFERRED at τ_0, the removal of d from OPEN at τ_0 needs the least h of DEFERRED above v, so that $h(w_j) > v$, and w_j leaves DEFERRED later, which as above needs $h(w_j) \leq v$; this case does not occur. Finally, if the goal w_m is in OPEN at τ_0, then $v \leq h(w_m)$, and it leaves OPEN in (τ_0, τ) with $h(w_m) \leq v$, or after τ, which needs $h(w_m) < v$; the first case gives $h(w_m) = v$, and the second contradicts $v \leq h(w_m)$. □

Theorem 12.30 (OBAT is bounded). Without SGE, let OBAT with k workers find a path P of $|P|$ states, and let N be the expansions of the GBFS of Theorem 12.27 up to the goal. Then at most $k|P|$ states are left in DEFERRED, and OBAT removes at most $N + k|P| + k - 1$ states from OPEN.

Proof. By Lemma 12.29 the values of h in DEFERRED at the end are at most $|P|$, and by Lemma 12.28 each is held by at most k states. The states removed from OPEN are the committed ones and the goal, which are N by Theorem 12.27, the states left in DEFERRED, and the b-states still in expansion at the end, at most $k - 1$ (the worker of the goal holds none). □

Shimoda and Fukunaga state the bound as $N + k|P|$ (theorems 3 and 4 of [110]); the term $k - 1$ above comes from the counter of the library, which counts each removal, and is reached only by the b-states that the other workers hold when the goal is removed. The test `pgbfs-rounds` replays the commits of OBAT (the field `committed` of the trace) as a GBFS, checking at each step that the committed state has the least h of the simulated OPEN (Theorem 12.27), and counts the states of each value of h left in DEFERRED (Lemma 12.28 and Theorem 12.30), on the random graphs of 40 states drawn from 60 seeds that have a reachable goal (at least 20 of them) with 2, 3 and 5 workers, with and without SGE, and checks the order of the commits on an instance of the 8-puzzle; it checks Proposition 12.25 on the same instances. We remark that Lemma 12.28, and hence the bound $k|P|$, rests on at most k states in expansion: with SGE a worker that has generated the successors of a node becomes idle while the node waits for its evaluations, and it may remove another node, so that more than k nodes may be in expansion at once. The proof then gives the bound with k replaced by the largest number of nodes in expansion at once, and we have not proved the bound k with SGE, although the tests observe it on their instances.

The separate generation and evaluation keeps the other guarantees: the proofs of Proposition 12.25 and Theorem 12.27 compare only the values h of the nodes in OPEN, in DEFERRED and in expansion, and under SGE a node is in expansion until all its successors are evaluated and inserted.

HDGBFS

`HDGBFS< D , Dist , Executor >` (`ParallelGreedy.h:978`) is HDA* with the order of GBFS: each worker runs GBFS on its own OPEN and CLOSED lists, ordered by h and then by the order of arrival (or the opposite, with `TieBreak::lifo`), the states being assigned to the workers by the distribution, and the search ends when a worker selects a goal (Kuroiwa and Fukunaga [107]). The hash distribution keeps the duplicate detection exact, but not the global order: a worker expands the best node it owns, however bad, and HDGBFS is pathological in the sense above; a state reached again through a cheaper path is not reopened, and the states with $h = \infty$ are dropped. The nodes travel in the batches of the mailbox, and the end, when no goal is found, is its quiescence (§9.3); hence HDGBFS returns a path if a goal is reachable, and `unsolvable` otherwise, on a finite graph. It reads `threads`, `batch`, `flush_period`, `tie` and `node_limit` of its `ParallelGreedyOptions`.

The engineering

OPEN is an ordered map from the values of h to buckets, each with two queues, of the certain nodes and of the others, in the order of their arrival; the first node of a bucket is the earliest (FIFO) or the latest (LIFO) of the two fronts, or the certain one with `certain_first`. DEFERRED is a map from h to queues of the deferred expansions, each with its successors, and the values of h in expansion are a map with counts, so that $\underline{h}$ is its first key. All of them, with the successors waiting for their evaluation under SGE, are under one mutex, as in the papers, while CLOSED is a table of the states generated split in 64 shards, each with a mutex, an arena and a table (`ParallelGreedy.h:418`), each state being evaluated once (the table is also the cache of the evaluations, shared with the speculative search of SPUHF, which has a lock of its own). The goal test is at the removal, and the path is rebuilt through the parents (through the speculative parents for a goal of the speculation), its cost recomputed from the cheapest edge between consecutive states. A worker on an executor loops on ACQUIRE, WORK and the commit of its work, with the backoff of §8.4 when it finds nothing to do (`ParallelGreedy.h:898-914`); in rounds the loop is that of the description above. The test `pgbfs-thread` checks valid paths with each rule on the executors, Proposition 12.25 and Theorem 12.27 under real threads on the 8-puzzle, and HDGBFS.

12.9 Two baselines

Two sequential searches complete the parallel ones as references: the breadth-first search, which ignores both the costs and the heuristic, and the cost-bounded depth-first search, a single iteration of IDA*.

`BreadthFirst` (`BreadthFirst.h:85`), and the function `breadth_first( d , o )`, search the `Domain` breadth first with duplicate detection, the goal test at the generation; the layers are two vectors of nodes, and CLOSED is a table of nodes (sized for `expected_nodes`, if given, and otherwise for 1024), which `node_limit` bounds. The pseudocode of the library is the following.

Algorithm 12.9 (Breadth-first search).

```
1  procedure BFS(s_0)
2      if s_0 is a goal then return the path (s_0)
```

```
 3     CLOSED ← {s0};  L ← (s0)
 4     while L ≠ ∅ do
 5        L' ← ()
 6        for each u of L, in order, do
 7           for each successor v ∉ CLOSED of u do
 8              add v to CLOSED, with parent(v) ← u
 9              if v is a goal then return the path to v
10              append v to L'
11        L ← L'
12     return unsolvable
```

Proposition 12.31 (Breadth-first search). On a finite graph, `breadth_first` returns a path with the least number of edges to a goal, or `unsolvable` if no goal is reachable; with unit costs the path is optimal, and otherwise its cost, that of its edges, need not be the least.

Proof. By induction on d, the layer L of the iteration d holds exactly the states at distance d (in edges) from s_0 that are not goals at a smaller distance, since a state enters CLOSED the first time it is generated, from a state of the smallest distance. A goal at distance d is generated when its predecessor at distance $d-1$ is expanded, and no goal at a smaller distance exists, or it would have been returned; the path through the parents has d edges. If no goal is reachable, each reachable state enters a layer once, and the loop ends. □

The test `bfs` checks the distances of the 8-puzzle on 21 states, the path of fewest edges on the map of Romania (Arad, Sibiu, Fagaras, Bucharest, of cost 450 against the optimum 418), and the status `unsolvable` on a grid split by a wall.

`cost_bounded_dfs( d , β , o )` (IDAStar.h:340) is `IDAStar< D >( d , o ).solve( β )`, i.e., the first call of the depth-first search of IDA* with the bound β and no other (§4.4): it returns a path whose nodes all have $f \leq \beta$, if the depth-first search finds one, with the cost of that path, which need not be the least, or the status `unsolvable` if there is none. With an admissible h and $\beta = C^*$ the path found has cost at most β, i.e., C^*, and with $\beta < C^*$ the status is `unsolvable`; the test `bfs` checks both on two instances of Korf, the bound 0 on the 8-puzzle, and the bound 450 on the map of Romania, where the path found costs between 418 and 450. The search is the reference of the bounded iterations of TDS and PSDD with a given bound, and of each unit of the parallel IDA*.

12.10 The searches compared

The following table gathers, for each search of the chapter, what its workers share, where the duplicates are detected, how the end is detected, and in which regime it may pay off; as in §8.10, the last column states what the scheme suggests, and only measurements on a given machine and domain can decide among the searches.

Search	Shared	Duplicates	End	Regime
PSDD, BFPSDD	abstract graph, layer	in the scope	an empty layer	tight bound; large layers
IDPSDD	as PSDD	as PSDD	no node pruned	unknown bound
KBFS	nothing (master)	at the master	at the master	slow expansions, determinism
Multi-A*	one flag	in each worker	the first worker	expansions sensitive to ties
PRA*, APRA*	messages	at the owner	deficit counting	(a baseline of HDA*)
lending	published f	at the owner	deficit counting	unbalanced owners
TDS	table, by owners	at the owner	deficit counting	memory for the table
PIDA*	units, thresholds	near the frontier	barrier, or no unit below z	little memory, few cycles
PIDA*, MPI	reductions	near the frontier	reduction	many machines
parallel GBFS	OPEN, CLOSED	in CLOSED	the first goal	satisficing search
HDGBFS	messages	at the owner	the first goal	satisficing, fast expansions

In the table, "near the frontier" means that the parallel IDA* detects the duplicates in the table of the frontier (when `table_depth` is positive) and the moves back to the parent below it, and no other; the end of the synchronous scheme is the barrier at which some goal has been found, and that of the asynchronous one an empty queue of units with no search running.

12.11 Benchmark

We compare the searches of the chapter that find optimal paths with the sequential A* and with HDA* on the 15-puzzle, with the Manhattan distance, on the instances of Korf of campaign B of §8.11, run by `tools/bench/campaign.sh` through `AStarBench` (the algorithms `psdd`, `bfpsdd`, `idpsdd`, `dynpsdd`, `kbfs`, `multi`, `pra`, `tds` and `pida`), each run twice in separate processes, pinned to the cores of one NUMA node with its memory on that node, one worker per core. PSDD runs with the optimal cost as its bound, as in the experiments of Burns et al., and the nblocks are those of §11.1.

Table 12.1. The searches by layers, KBFS, Multi-A*, PRA*, TDS and the parallel IDA* on the 15-puzzle. Each cell gives the seconds summed over the instances (each the median of its two runs) and, in parentheses, the speedup over the sequential A* with the bucket queue on 1 core; an empty cell is a search that did not solve each instance. The machine and the compiler are given in the note of the table.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

Table 12.2. The parallel greedy searches on the 15-puzzle: for each rule (KPGBFS, PUHF, PUHF2, PUHF3, PUHF4, SPUHF and OBAT, with and without SGE) and for HDGBFS, the seconds summed over the instances and, in parentheses, the states removed relative to those of the sequential GBFS with the same breaking of the ties, for each number of cores.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The clock is the timer of the `Solution`, which includes the building of the frontier of the parallel IDA*, the creation of the nblocks of PSDD, the start of the threads of the executor (`AStarBench` lends none) and the rebuilding of the path, and does not include the reading of the instances, the construction of the `Domain` and the printing of the results. The tables are placeholders until the campaign is run on an idle machine. The counts of the parallel greedy searches in rounds do not depend on the machine, and the tests give some of them (Proposition 12.26).

12.12 Notes and references

Structured duplicate detection and PSDD are due to Zhou and Hansen ([139] and [128]), who built PSDD on their breadth-first heuristic search [129]; BFPSDD, IDPSDD and the dynamic bound by weighted A* are the variants of Burns et al. (section 2.4.1 of [3]), who found that the synchronization of the layers limits them and introduced PBNF (Chapter 11) for that reason. Our PSDD differs from the original in its end: it tests the goal at the generation but goes on until no list holds a node below the incumbent, which makes it optimal within its bound with any costs (Theorem 12.4). KBFS is due to Felner, Kraus and Korf [130]; we write it as the synchronous rounds of the farm, with K the workers times the batch. The portfolio of A* searches that differ only in their ties is, as far as we know, a baseline of the library rather than a search of the literature, in the spirit of the portfolios of planners; we give it only as a reference.

PRA* is due to Evett, Hendler, Mahanti and Nau [127], and APRA* and the comparison of the synchronous and asynchronous halves of its communication to Burns et al. [3]; the library has no retraction. The lending of

nodes is an engineering choice of the library, meant against the load imbalance of the distributions of Chapter 9 (the band effect of Jinnai and Fukunaga [5], section 3.1.1); we are not aware of a published A* with the same rule. TDS is due to Romein, Plaat, Bal and Schaeffer [131], and the bounds of IDA*-CR to Sarkar, Chakrabarti, Ghose and De Sarkar [132]; our histogram by binary exponents of the excess and the doubling of the last step when few nodes are pruned are ours. The parallel IDA* below a frontier follows the AIDA* of Reinefeld and Schnecke [55] and the A*+IDA* of Bu and Korf [56]; the frontier by depth is that of Sturtevant and of Siag et al. [57], the splitting of the stacks that of Rao and Kumar [58], and the assignment of the units over MPI the longest processing time first of Graham [66]. The unit bounds (Lemma 12.17) and the deterministic path are ours.

The analysis of the states that GBFS expands is due to Heusner, Keller and Helmert [112]; KPGBFS to Vidal, Bordeaux and Hamadi [106]; the pathology of the parallel GBFS, PUHF and SPUHF to Kuroiwa and Fukunaga [108], and that of HDGBFS to the same authors [107]; PUHF2 to PUHF4 to Shimoda and Fukunaga [109], who also introduced OBAT, with the bound of Theorem 12.30 [110], and the separate generation and evaluation [111]. The proofs of Lemma 12.24, Theorem 12.27 and Lemmas 12.28 and 12.29 are written here for the code of the library; for the first criterion of PUHF and for lemma 1 of [109] we rely on the papers. The breadth-first search and the cost-bounded depth-first search are classical (cf. Korf [7] for the latter); we refer to the survey of Fukunaga et al. [1] for the place of all these schemes among the parallel best-first searches.

12.13 Exercises

Exercise 12.1 (`ctest -R PBeFS.psdd-thread`). Follow Algorithm 12.1 on the vacuum world of Figure 12.1, with two workers, and show that at most $\lceil n/3 \rceil$ nblocks of a layer can be in use at once. Then show that PSDD with the bound $C^* - 1$ returns `unsolvable` and that the least f it prunes is C^* when h is consistent and the costs are 1; the test checks this on two instances of Korf with 1, 2, 3 and 8 workers.

Exercise 12.2 (`ctest -R PBeFS.psdd-thread`). Give a graph with costs other than 1 on which the scheme of Zhou and Hansen, which stops at the first goal generated, returns a path that is not optimal within the bound, while PSDD (Theorem 12.4) returns the optimum. The test checks PSDD, BFPSDD and DynamicBoundedPSDD ($w = 1.5$) on grids with costs from 1 to 10, and IDPSDD with and without IDA*-CR on a grid with costs from 1 to 3.

Exercise 12.3 (`ctest -R PBeFS.psdd-thread`). The test runs BFPSDD with `multiplier` and `min_expansions` at 0. Which values does the constructor take, and which step of the proof of Proposition 12.8 would fail if a worker could leave its nblock of BFPSDD before its first expansion?

Exercise 12.4 (`ctest -R PBeFS.farm-thread`). Prove part (iii) of Proposition 12.9 in detail, and show that with $p > 1$ or tasks of more than one node KBFS may expand a node that A* does not expand. Why does the test check only that two runs of KBFS expand the same, and not their expansions against A*?

Exercise 12.5 (`ctest -R PBeFS.multi-thread`). Show that the result of `MultiAStar` may depend on the scheduling of the workers (which worker wins), although its cost does not, and that with a node limit the status `node_limit` of one worker may hide the solution of another. The test checks one worker against `astar()` and the costs with 2, 3 and 8 workers.

Exercise 12.6 (`ctest -R PBeFS.hda-thread`). In Algorithm 12.4, the borrower does not insert the node u it has received. Show that this is right, i.e., that no improvement of $g(u)$ is lost, and that a lent node may still be expanded again by its owner. Then explain why an idle worker, which publishes ∞, receives a node only when no busy worker is far enough behind. The test runs PRA*, APRA* and the lending.

Exercise 12.7 (`ctest -R PBeFS.tds-thread`). Show that line 18 of Algorithm 12.5 can prune the only path to the goal with $f \leq \beta$ if the table compared the g of the entries of another iteration, and why the library marks each entry with the iteration that pushed it. Then show that TDS with one worker expands the same at each run, as the test checks.

Exercise 12.8 (`ctest -R PBeFS.tds-thread`). With `growth` 2 and `optimal` false, give an instance on which the first path of TDS is not optimal, and compute the bound of Theorem 12.14 on it. The test checks that the first path is valid and not below the optimum, and that with `optimal` it is optimal.

Exercise 12.9 (`ctest -R PBeFS.pida-thread`). Show that the parallel IDA* with one unit and one worker makes the expansions of IDA*, and that the unit bounds (Lemma 12.17) never increase the expansions of an iteration. Which of the variants of the test (the rules, the schedules, the schemes, the splitting and the table) can change the expansions with the same number of workers?

Exercise 12.10 (`ctest -R PBeFS.pida-thread`). Show that the deterministic option needs the synchronous scheme and no splitting, by an interleaving of two workers that, without it, returns different paths of the same cost. Then show that the frontier by depth cannot have a table.

Exercise 12.11 (`ctest -R PBeFS.bfs`). Prove Proposition 12.31, and show that with the goal test at the selection instead of the generation the breadth-first search would expand a whole layer more in the worst case. The test checks the path of fewest edges on the map of Romania.

Exercise 12.12 (`ctest -R PBeFS.pgbfs-rounds`). Compute the BTS of the graph of Proposition 12.23, and follow PUHF, PUHF2 and OBAT on it with two workers in rounds. Then show that PUHF4 with one worker need not be GBFS with FIFO, which is why the test leaves it out.

Exercise 12.13 (`ctest -R PBeFS.pgbfs-rounds`). Build an instance on which OBAT with SGE and two workers has three states of the same h in expansion at once (cf. the remark after Theorem 12.30), and discuss whether three of them can be left in DEFERRED at the end. The test checks the bound k per value of h on its random graphs, with and without SGE.

Exercise 12.14 (`ctest -R PBeFS.pgbfs-thread`). Show that the guarantees of Proposition 12.25 and Theorem 12.27 hold for each interleaving of the workers, and hence under real threads, and that the counts of a search in rounds are the same at each run. The test checks the BTS and the order of the commits under the three executors.

13. Branch-and-bound and its parallel schemes

[Source: `include/PBeFS/bnb/BranchAndBound.h` (the concept, the options, the solution, the queue and the sequential search), `include/PBeFS/bnb/ParallelBranchAndBound.h` (the shared incumbent, the shared and the distributed queues), `MultiQueueBranchAndBound.h`, `FarmBranchAndBound.h`, `DeterministicBranchAndBound.h`, `StealingBranchAndBound.h`, `SelfSplitBranchAndBound.h`, `MPIBranchAndBound.h`, `BranchAndBoundScheme.h`, `SkeletonBranchAndBound.h`, `RacingBranchAndBound.h`, `BBCheckpoint.h`, `MPIHubBranchAndBound.h`, `IntervalBranchAndBound.h`, `Knapsack.h`, `MaxClique.h`, `MaxCommonSubgraph.h`, `SubgraphIsomorphism.h`, `QAP.h`, `Flowshop.h` and `UTS.h` of the same directory; `tools/BranchAndBoundBench.cpp`; `test/test.cpp`, `test/bb.cpp` and `test/bnb.cpp`]

We are concerned in this chapter with the minimization of a function v over a finite (but possibly "huge") set X, i.e., with finding

$$z^* = \min\{\, v(x) : x \in X \,\},$$

when X can be partitioned recursively into smaller sets, each of which comes with a lower bound on the values that it contains; a maximization is the minimization of $-v$, which is how the library treats it internally. The method is branch-and-bound (B&B), which Land and Doig [146] introduced for discrete programming. A* is a special case of it [17]: the subsets are the nodes of a tree, the bound of a node plays the part of $f = g + h$, and the best value found so far (the *incumbent*) prunes the nodes that cannot improve on it. The layer of the library that implements B&B shares with the graph searches of the previous chapters the executors of the workers (§1.6), their placement on the cores and the detection of the termination. It does not share the node table and the OPEN lists of A*, which are specialized for integer costs and for the duplicates of a graph.

The structure of the chapter is as follows. §13.1 gives the concept `BranchAndBound` and the types around it, and §13.2 the notions of node, bound, estimate and pruning, on the 0-1 knapsack that serves as the running example of the chapter. §13.3 gives the sequential search, §13.4 its correctness and the bound it reports when it stops early, and §13.5 the rules by which it selects the next node, depth-first B&B among them, together with the queue that implements them and the cost of the whole search. §13.6 shows A* as a special case. §13.7 states what a parallel B&B has to guarantee, with the shared incumbent, the dive and the limits common to all the schemes. §13.8 and §13.9 then give the eight parallel schemes of the library, each with its pseudocode, the invariant that makes its result optimal, the proof that its detection of the end is sound and that the end comes, and its engineering: first those with one pool of open nodes (the shared queue, the MultiQueue, the farm and the deterministic rounds), then those with one pool per worker (the distributed queues, their version over MPI, the work stealing and the SelfSplit). §13.10 compares the schemes and shows how a program chooses one by its name. §13.11 treats the reoptimization from the leaves of a previous search, and §13.12 the interface that a B&B solver built on a modeling system would use. The next sections extend the core: §13.13 gives the children built one at a time and the modes that decide or enumerate instead of optimizing, §13.14 the depth-first search skeletons, §13.15 the racing ramp-up, §13.16 the checkpoints, §13.17 the master, hubs and workers over MPI, §13.18 the work stealing of intervals of leaves, and §13.19 the example problems that exercise them (the maximum clique, the maximum common subgraph, the subgraph isomorphism, the quadratic assignment, the permutation flowshop and the unbalanced tree search); §13.20 to §13.22 give the benchmark, the notes and the exercises of the whole chapter. The notation is that of Chapter 1, and the conventions of the pseudocode are those of §1.4. In the pseudocode, as in the headers, $\beta(n)$ is the bound of the node n, z the incumbent value, Q a queue of open nodes and p the number of workers.

13.1 The problem and its interface

A problem is given to the B&B layer as a type `P` that satisfies the concept `BranchAndBound` (BranchAndBound.h:417), i.e., that provides

- the copyable type `node_type` of its nodes;
- the constant `sense`, `Sense::minimize` or `Sense::maximize`;
- the method `root()`, which gives the node of the whole set X;
- optionally, the type `value_type` of its values, `double` if it gives none;
- either the methods `evaluate( n )` and `branch( n , sink )` of a `BBEvaluator` (BranchAndBound.h:369), which have to be safe to call concurrently in a parallel search, or the method `make_worker( i )`, which returns the `BBEvaluator` of the worker i.

In the header the concept reads as follows (the concepts `BBValue` and `BBEvaluator` are described below):

```
template< class P >
concept BranchAndBound = requires( const P & p , unsigned i ) {
 typename P::node_type;
 requires std::copyable< typename P::node_type >;
 requires BBValue< bb_value_t< P > >;
 { P::sense } -> std::convertible_to< Sense >;
 { p.root() } -> std::convertible_to< typename P::node_type >;
 } && ( BBEvaluator< const P , typename P::node_type , bb_value_t< P > > ||
        requires( const P & p , unsigned i ) {
         { p.make_worker( i ) }
          -> BBEvaluator< typename P::node_type , bb_value_t< P > >;
         } );
```

The method `evaluate( n )` computes a bound of the node, possibly storing in `n` what the branching needs (say, the fractional solution of a relaxation), and returns a `BasicEvaluation< V >` made of the bound, a flag for an infeasible node and an optional feasible value, of which `n` is the solution. In turn, `branch( n , sink )` calls `sink( c , e )` for each child `c` of the evaluated node `n`. Here `e` is a bound of `c` that is "cheap" to compute, typically the bound of `n` (or the infinity of the sense of `P`, which means the same thing); the search uses it to order the child and to prune it before its evaluation. An evaluator given by `make_worker( i )` exists for the problems whose evaluation keeps a state (e.g., a model and a solver of the relaxation that are modified at each node): each worker then has a copy of that state, and `P` is only read. In place of (or besides) `branch( n , sink )`, an evaluator may give the children one at a time, and the child k of a node without its evaluation (§13.13). Such an evaluator may also give `cutoff( z )`, which the searches call before each `evaluate` and `branch` with the incumbent value z (in the sense of `P`, infinite if there is none). With it, e.g., the relaxation of a node can stop as soon as it is proven not to improve z, the node being then returned as infeasible. Note that a worker should take z from here rather than from a value it shares with the others: the deterministic search of §13.8 passes the incumbent of the start of its round, and an evaluation against a value that the other workers change meanwhile would make its outcome depend on their timing.

Three refinements of the concept are tested by the searches that use them. A `HashedBranchAndBound` gives `hash( n )`, by which the distributed searches send a node to its owner. An `EstimatingBranchAndBound` gives `estimate( n )`, an estimate of the best feasible value under `n`, by which the best-estimate selection orders the nodes. Finally, the search over MPI needs an `MPIBranchAndBound` (MPIBranchAndBound.h:103), i.e., trivially copyable values and, unless its nodes are trivially copyable too, the methods `serialize( n , out )` and `deserialize( in )`, which write a node to bytes and read it back.

The values (the bounds, the feasible values, the incumbent and the absolute tolerance) are of the type `V = bb_value_t< P >`, which has to satisfy the concept `BBValue` (BranchAndBound.h:288), i.e., to be a copyable, totally ordered type, constructible from an `int`, with negation and subtraction. In addition, the traits class `BBValueTraits< V >` (BranchAndBound.h:231) gives the infinity of the type (that of a floating point type, the largest value of an integer one) and what the value of a solution with no feasible value is (a quiet NaN, or the infinity). It also gives the size of the tolerance of a gap, and decides whether a bound is "close enough" to the least open bound to dive into it. Its primary template serves the signed arithmetic types, and a caller with another type, e.g., the rationals of an exact arithmetic, specializes it (the test `bb-values` does so with a type `Rational`). An integer type pays off whenever the values of the solutions are integer, since the bound of a node can then be rounded, as §13.2 explains.

Options are given as a `BasicBBOptions< V >` (`BranchAndBound.h:473`), whose fields and defaults are

```
template< class V = double >
struct BasicBBOptions {

 Selection selection = Selection::best_first;   ///< the order of the nodes
 BBMode mode = BBMode::optimize;   ///< what the search looks for
 std::optional< V > target;     ///< the value sought by the mode decide
 V abs_gap = V( 0 );           ///< the absolute tolerance on the gap
 double rel_gap = 0;           ///< the relative tolerance on the gap
 double plunge_gap = 0.5;      ///< how far from the best bound a dive goes
 std::optional< V > incumbent;  ///< a known feasible value, if any
 /// a value that the search reads at each extraction and prunes by when it
 /// is better than its incumbent, e.g., the incumbent of other searches
 /// running at the same time; the Solution has only the feasible values
 /// that the search itself finds (or incumbent), so that it is infeasible
 /// when none of them is better than the values read, and its bound is
 /// still that of the subtrees searched
 std::function< V( void ) > shared_incumbent;
 std::size_t node_limit = 0;   ///< evaluated nodes (0 = no limit)
 double time_limit = 0;        ///< seconds (0 = no limit)
 bool keep_frontier = false;   ///< keep the leaves of the tree explored
 bool keep_open_only = false;  ///< ... only those left open by a limit
 bool prune_level = false;     ///< the children come by their estimates
 /// called with every new incumbent value (in a parallel search, by the
 /// worker that finds it, under a lock)
 std::function< void( V ) > on_incumbent;

 };  // end( struct( BasicBBOptions ) )
```

i.e.,

- `selection`, the order of extraction of the open nodes (§13.5), best-first by default;
- `abs_gap` and `rel_gap`, the absolute and relative tolerances τ_a and τ_r on the gap (both 0 by default);
- `plunge_gap`, the fraction γ of the dive of the plunging selection (0.5 by default);
- `incumbent`, a known feasible value, if any;
- `shared_incumbent`, a function that the search reads at each extraction and prunes by when it is better than the incumbent of the search (e.g., the incumbent of other searches running at the same time);
- `node_limit` (the nodes evaluated, by all the workers together in a parallel search, §13.7; 0 for none) and `time_limit` (seconds, 0 for none);
- `keep_frontier`, which requests the leaves of the tree explored (§13.11), and `keep_open_only`, which keeps only those left open by a limit (§13.16);
- `mode` and `target`, what the search looks for: an optimal value (the default), a value within the target, or the count of the feasible nodes (§13.13);
- `prune_level`, which states that the children come by their estimates, the best first (§13.13);
- `on_incumbent`, a function called with each new incumbent value.

The parallel searches take a `BasicParallelBBOptions< V >` (`ParallelBranchAndBound.h:140`). It adds the parameters of the parallel searches of the library, i.e., the number of workers `threads`, the nodes `batch` of a message, the evaluations `flush_period` between two flushes of the outboxes and the `seed` of the random owners; by default the batches have 16 nodes and are flushed at each evaluation, since an evaluation is usually "expensive". It also adds the parameters `round`, `split`, `split_dive` and `split_shared` of two of the schemes (§13.8 and §13.9), those of the skeletons (§13.14) and those of the racing ramp-up (§13.15).

Each search returns a `BBSolution< Node , V >` (`BranchAndBound.h:584`). It holds the status (`optimal` within the tolerance, `infeasible`, `node_limit` or `time_limit`), the incumbent node, its value, a bound on the optimum and the gap between the two, which is valid whatever the status (Theorem 13.6). It also holds the counters of `BBStats`: the nodes evaluated, branched, generated, found infeasible and pruned at each of the three tests of §13.3, the improvements of the incumbent, the nodes and messages sent, the largest queue, the time and the evaluations of each worker, and the feasible values found (`solutions`) and the evaluations made only to rebuild a node (`recomputed`), which §13.13 and §13.14 describe. It holds, finally, the least bound of the leaves that are not open (`closed_bound`, §13.16). With `keep_frontier`, finally, it holds the leaves of the tree explored as `BBLeaf` records, each with its node, its bound, its depth and the reason why it is a leaf (`open`, `pruned`, `feasible` or

`infeasible`). A maximization problem is solved by negating its values on the way in and on the way out (`detail::bb_signed()`), so that the code, the pseudocode and the proofs below only minimize.

13.2 Nodes, bounds and the incumbent

A *node* n stands for a subset $X(n) \subseteq X$, the root for the whole X, and *branching* n gives children $n_1, \ldots, n_k$ whose subsets cover $X(n)$ (usually they partition it). The *evaluation* of n gives a bound $\beta'(n)$ with

$$\beta'(n) \leq \min\{\, v(x) : x \in X(n) \,\},$$

e.g., the optimal value of a relaxation of the problem restricted to $X(n)$, together with two optional facts: that $X(n)$ is empty (the node is *infeasible*), or a feasible value $v(x)$ for some $x \in X(n)$ (e.g., when the solution of the relaxation is feasible, or by a heuristic). We call *incumbent* the best feasible value z found so far ($+\infty$ if none is known and none is given at the start), and *incumbent node*, if any, a node of value z.

Since a child is a subset of its parent, the bound of the parent is also a bound of the child. For each open node the library keeps the bound

$$\beta(m) = \max\{\, \beta(n),\ e(m) \,\}$$

for the child m of n, where $e(m)$ is the estimate given by `branch()` (e.g., the bound obtained by fixing one variable in the solution of the relaxation of the parent). After the evaluation of m it replaces $\beta(m)$ by $\max\{\beta(m), \beta'(m)\}$. Thus, the evaluation, which is usually "expensive", is postponed until the node is extracted, and the bound never decreases along a path of the tree; the second property is the rule that the literature of A* calls *pathmax* (cf. §13.6).

A node is *pruned*, i.e., discarded together with its subtree, when it is infeasible or when its bound cannot improve the incumbent by more than the tolerance, i.e., when

$$\beta(n) \geq z - \max\{\tau_a,\ \tau_r|z|\},$$

with an absolute tolerance $\tau_a \geq 0$ and a relative one $\tau_r \geq 0$ (`abs_gap` and `rel_gap`); when $z = +\infty$, the node is pruned only if $\beta(n) = +\infty$. We write $\theta(z) = z - \max\{\tau_a, \tau_r|z|\}$ for the threshold, and PRUNABLE(β) for the test $\beta \geq \theta(z)$. With $\tau_a = \tau_r = 0$ the test is $\beta(n) \geq z$, since a subtree whose bound equals z cannot hold a better value. In the library the test is the function `detail::bb_prunable()`, i.e., lines 30 to 32 of Algorithm 13.3 below, where the tolerance $\max\{\tau_a, \tau_r|z|\}$ is computed by `BBValueTraits::tolerance()` and is rounded down to $\max\{\tau_a, \lfloor \tau_r|z| \rfloor\}$ for an integer type of values.

When the values of the solutions are integer, $\lceil \beta(n) \rceil$ is a bound too, and the test applied to it prunes each node with $\beta(n) > z - 1$, i.e., those that could improve z by a fraction only (for an integer z and zero tolerances, $\lceil \beta \rceil \geq z$ if and only if $\beta > z - 1$). With doubles the same nodes are branched, unless one gives an absolute tolerance just below 1. In the library the rounding comes from the type of the values. A problem whose `value_type` is an integer type returns integer bounds, rounded towards the inside (down for a maximization, up for a minimization), and the test above then prunes a node whose rounded bound equals the incumbent; the test `bb-values` checks that, summed over its knapsacks, the integer bounds evaluate no more nodes than the double ones.

The running example: the 0-1 knapsack

The problem of the tests and of the benchmark is the 0-1 knapsack `BasicKnapsack< V >` of `Knapsack.h`, a maximization: given n items of profit $p_j > 0$ and weight $w_j > 0$ and a capacity C, find $x \in \{0,1\}^n$ maximizing $\sum_j p_j x_j$ subject to $\sum_j w_j x_j \leq C$. Its items are sorted by decreasing ratio p_j/w_j, and a node fixes the first k of them (the field `k`, with the weight and the profit of those taken). Its evaluation is the bound of the fractional relaxation of Dantzig [147]. In it, the free items are taken greedily in the order of the ratio while they fit, and the first one that does not fit (the *critical* item) is taken in the fraction that fills the capacity. As feasible value, the node gives that of the greedy integer completion, which skips the items that do not fit and goes on with the next ones. In pseudocode, with u a node that fixes the items $0, \ldots, k-1$, $W(u)$ and $P(u)$ the weight and the profit of the items that it takes, and the items numbered from 0, the evaluation is the following (`evaluate()` and `greedy()` in `Knapsack.h`).

Algorithm 13.1 (The evaluation of a node of the knapsack).

```
 1  function EVALUATE(u)                ▷ u fixes the items 0, ..., k − 1
 2     if W(u) > C then return infeasible
 3     r ← C − W(u);  b ← P(u);  j ← k
 4     while j < n and w_j ≤ r do r ← r − w_j;  b ← b + p_j;  j ← j + 1
 5     if j < n then b ← b + p_j · r / w_j        ▷ j: the critical item
 6     if the values are integer then b ← floor(b)
 7     return the bound b and the feasible value GREEDY(u)
 8  function GREEDY(u)                  ▷ the greedy integer completion
 9     r ← C − W(u);  v ← P(u)
10     for j = k, ..., n − 1 do
11        if w_j ≤ r then r ← r − w_j;  v ← v + p_j
12     return v
```

Both functions take $O(n-k)$ time, i.e., $O(n)$ per evaluation.

Lemma 13.1 (The bound of Dantzig). Let a node fix the items $0,\dots,k-1$, with residual capacity $r \geq 0$ and profit P of the items taken. Then the value b computed by Algorithm 13.1 is at least the profit of each completion of the node that fits, and the greedy value is the profit of one of them; hence, $v \leq z^*(n) \leq b$, where $z^*(n)$ is the best profit in $X(n)$.

Proof. We first remark that the best completion is at most P plus the optimum of the Linear Program $\max\{\sum_{j\geq k} p_j y_j : \sum_{j\geq k} w_j y_j \leq r,\ 0 \leq y \leq 1\}$. Let y be an optimal solution of the Linear Program, and suppose that $y_i < 1$ and $y_j > 0$ for some $i < j$; take i the least such index and j the largest one. Move the weight $\delta = \min\{(1-y_i)w_i,\ y_j w_j\} > 0$ from j to i, i.e., raise y_i by δ/w_i and lower y_j by δ/w_j: the solution stays feasible, and its value changes by $\delta(p_i/w_i - p_j/w_j) \geq 0$, since $p_i/w_i \geq p_j/w_j$, so that it stays optimal. Each exchange either fills y_i, which raises the least index, or empties y_j, which lowers the largest one; hence, at most $n-k$ exchanges give an optimal solution in which the items before some index c are taken whole, the item c in a fraction and the items after it not at all. Its weight is r unless all the free items are taken whole, since otherwise raising y_c would increase the value ($p_c > 0$). These two properties define the greedy fractional solution, whose value is $b - P$ (line 4 of Algorithm 13.1 takes the items $k, k+1, \dots$ while they fit, and the critical item j in the fraction that fills the residual capacity left by them). Finally, the greedy integer completion fits by construction, so its profit v is the value of a point of $X(n)$. □

A node whose weight exceeds the capacity is infeasible. The branching fixes the item k: the child that leaves it out is given first and the child that takes it last, the latter only if it fits. With ties, a depth-first search therefore goes on with the item taken, as the greedy order does. Both children get as estimate the infinity of the maximization, i.e., the bound of their parent. With `explicit_children` they get their bound instead, and the item is taken also when it does not fit, which gives children that are found infeasible at their evaluation (the tests use this to produce infeasible leaves). With an integer `V` the bound is rounded down, which is still a bound since the profits are integer. Each node also has a hash (of `k`, the weight and the profit, mixed by SplitMix64), an estimate (its greedy value), and the two methods of `MPIBranchAndBound`. Finally, `optimum( )` computes the optimal value by the dynamic program on the capacity, in $O(nC)$ time, and the tests check each search against it. The function `random()` draws an instance of one of the classes of Pisinger [148], i.e., uncorrelated (p_j uniform in $[1,R]$), weakly correlated (p_j uniform in $[w_j - R/10, w_j + R/10]$, at least 1) and strongly correlated ($p_j = w_j + R/10$), with w_j uniform in $[1,R]$ and the capacity half of the total weight. The branching is the following (`branch()` in `Knapsack.h`), where `sink` is the function of §13.1 and ∞ is the infinity of the maximization, i.e., an estimate that leaves the child with the bound of its parent (§13.2).

Algorithm 13.2 (The branching of a node of the knapsack).

```
1  procedure BRANCH(u, sink)
2     if k = n then return                          ▷ every item is fixed
3     c ← u with the item k left out, and k + 1 items fixed
4     sink(c, the bound of c if explicit_children, else ∞)
5     if explicit_children or W(u) + w_k ≤ C then
6        t ← c with the item k taken:  W(t) ← W(u) + w_k;  P(t) ← P(u) + p_k
7        sink(t, the bound of t if explicit_children and t fits, else ∞)
```

The example of the chapter has the four items of Figure 13.1 and the capacity $C = 9$. Its optimum is 15 (the items 1 and 3, numbered from 1 in the figures), while the greedy completion of the root takes the items 1 and 2 and gives 12. The root bound is $5 + 7 + 10 \cdot 3/7 = 16.29$ (to two decimals, as in all the figures). A node is named

by the decisions it fixes, e.g., 10 for “item 1 taken, item 2 left out”.

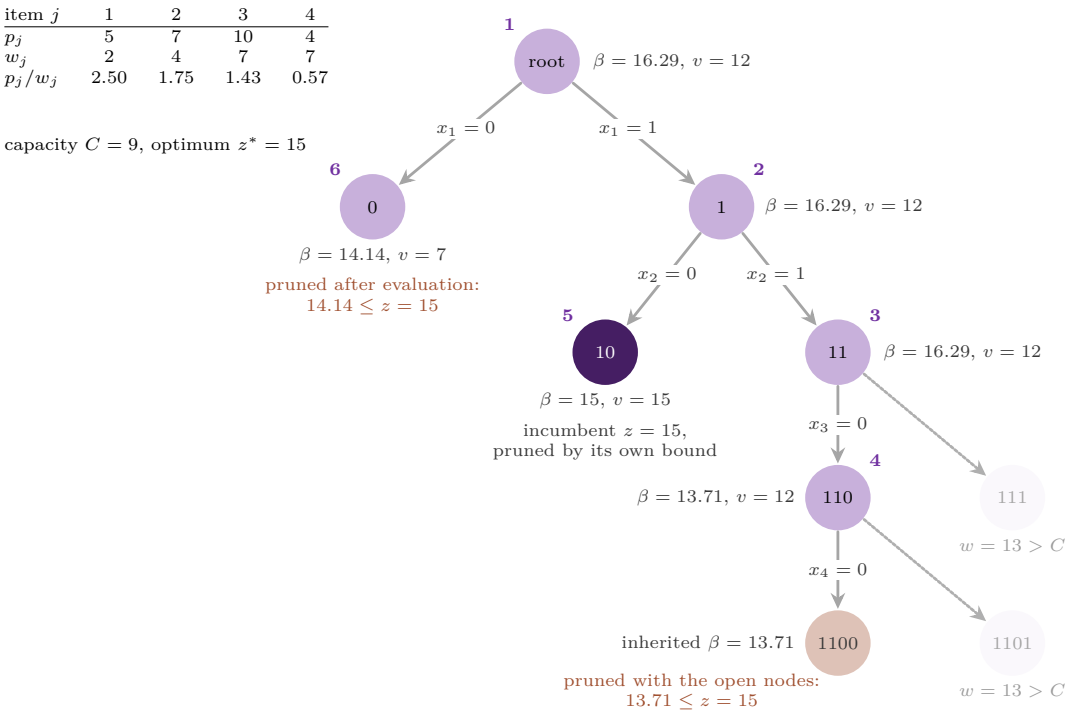


Figure 13.1. The tree that the best-first B&B explores on the example (a maximization): under each node its bound β (its inherited bound for 1100, which is never evaluated) and the value v of its greedy completion, and above it the order of the evaluations. Violet nodes are evaluated; the dark violet node is the node of the optimum, which its bound prunes, the rust node is pruned without evaluation, and the faint nodes are not generated, since their item does not fit.

13.3 The sequential search

The sequential search `BranchAndBoundSearch` (BranchAndBound.h:1206, or the function `branch_and_bound( p , options )`, whose declarations, without their bodies, are

```
[[nodiscard]] BBSolution< typename P::node_type , bb_value_t< P > >
branch_and_bound( const P & problem ,
                  const BasicBBOptions< bb_value_t< P > > & o = {} );
[[nodiscard]] BBSolution< typename P::node_type , bb_value_t< P > >
branch_and_bound( const P & problem ,
                  std::vector<
                    BBLeaf< typename P::node_type , bb_value_t< P > > > seeds ,
                  const BasicBBOptions< bb_value_t< P > > & o = {} );
```

for a model `P` of `BranchAndBound`) keeps the open nodes in a queue Q, ordered by the key of the selection (§13.5), and, with the plunging selection, a node d of the dive, which is extracted next. Its pseudocode, which is that of the Doxygen comment of BranchAndBound.h, is the following; R is the set of the nodes it starts from (the root, unless the caller gives others, §13.11).

Algorithm 13.3 (Sequential branch-and-bound).

```
1  procedure B&B(R)          ▷ the nodes to start from, {root} by default
2     z ← the given incumbent, or ∞;  Q ← R;  d ← nil  ▷ d: the dive
3     while Q ≠ ∅ or d ≠ nil do
4        βmin ← the least bound of Q and of d
5        if a limit is reached and not PRUNABLE(βmin) then break
6        if PRUNABLE(βmin) then the nodes of Q and d are leaves;  break
7        n ← d if d ≠ nil, else the first node of Q, removed;  d ← nil
8        K ← PROCESS(n)
```

```
 9       if the selection is plunging and K ≠ ∅ then
10          k ← the node of K of least bound
11          βmin ← min(the least bound of Q, β(k))
12          if z = ∞ or β(k) ≤ βmin + γ (z − βmin) then d ← k, from K
13       push the nodes of K in Q
14    the nodes left in Q and d are leaves            ▷ open by a limit
15    return z, its node, and the bound min(z, least bound of the leaves)
16 procedure PROCESS(n)
17    if PRUNABLE(β(n)) then n is a leaf (pruned);  return ∅
18    cutoff(z);  (β', infeasible, v) ← evaluate(n)
19    if infeasible then n is a leaf (infeasible);  return ∅
20    β(n) ← max(β(n), β')
21    if v is given and v < z then z ← v;  n is the node of z
22    if PRUNABLE(β(n)) then n is a leaf (pruned or feasible);  return ∅
23    K ← ∅;  cutoff(z)
24    for each child m of n, with its estimate e, given by branch(n) do
25       β(m) ← max(β(n), e)
26       if PRUNABLE(β(m)) then m is a leaf (pruned)
27       else K ← K ∪ {m}
28    if n has no child then n is a leaf
29    return K
30 procedure PRUNABLE(β)
31    if z = ∞ then return β = ∞
32    return β ≥ z − max(τ_a, τ_r |z|)
```

A node is thus tested three times (Figure 13.2): (i) when it is extracted (line 17), against an incumbent that may have improved since its generation, (ii) after its evaluation (line 22), against the bound it has just obtained, and (iii) when it is generated (line 26), against its inherited bound. A node that survives the second test is branched, also when it has a feasible value, since its subtree may hold better values than the one found. Line 6 is the termination test: as soon as the least bound of the open nodes cannot improve z, neither can any of them, and all of them would be pruned at their extraction. All the searches of the library but the MultiQueue, which does not know the least bound of its queues (§13.8), use this test, and they also end when Q is empty or a limit is reached; as in the pseudocode, the emptiness of line 3 and the termination test come before the limits of line 5, so that a search whose last evaluation reaches the node limit with no open node left, or with none that can improve z, ends as optimal. With best-first the test at line 6 often (but not always) ends the search while Q still holds many nodes, i.e., those generated before the incumbent was found and never extracted since.

The procedure PROCESS is `detail::bb_process()` (`BranchAndBound.h:1157`). All the searches of the library share it, each with the incumbent it sees (a function `inc()` that returns z in the minimization) and with the function `improve` by which it offers a feasible value; the children that survive line 26 go to a function `push` of the caller, which puts them in its queue or in its messages. The leaves go to a record of the worker, `detail::BB Frontier` (`BranchAndBound.h:884`), which keeps the least bound `low` of the leaves that lower the bound of the search and, only with `keep_frontier`, the leaves themselves.

Algorithm 13.4 (The record of the leaves of a worker).

```
1  procedure LEAF(n, β, kind, lowers)          ▷ a leaf n of bound β
2     if lowers then low ← min(low, β)
3     if keep_frontier then append (n, β, depth(n), kind) to the leaves
4  procedure LEAVES(Q, kind)                   ▷ all the nodes of a queue
5     low ← min(low, the least bound of Q)
6     if keep_frontier then append each node of Q with its bound and depth
7     Q ← ∅
8  procedure MERGE(L')                         ▷ the record of another worker
9     low ← min(low, low of L');  append the leaves of L'
```

In PROCESS, an infeasible leaf (line 19) and a node with no children (line 28) are recorded with `lowers` false: the former has no element, and the latter is solved by Assumption (A3) of §13.4, its value having been offered to z at line 21. The leaves of the three tests are of the kind `pruned`, or `feasible` at line 22 when the node gave a feasible value, and those left by a limit (line 14) of the kind `open`. A child pruned at line 26 is copied into the record only with `keep_frontier`; otherwise only its bound enters `low`.

Figure 13.3 traces Algorithm 13.3 on the example. The greedy completion of the root gives $z = 12$ at once. Then

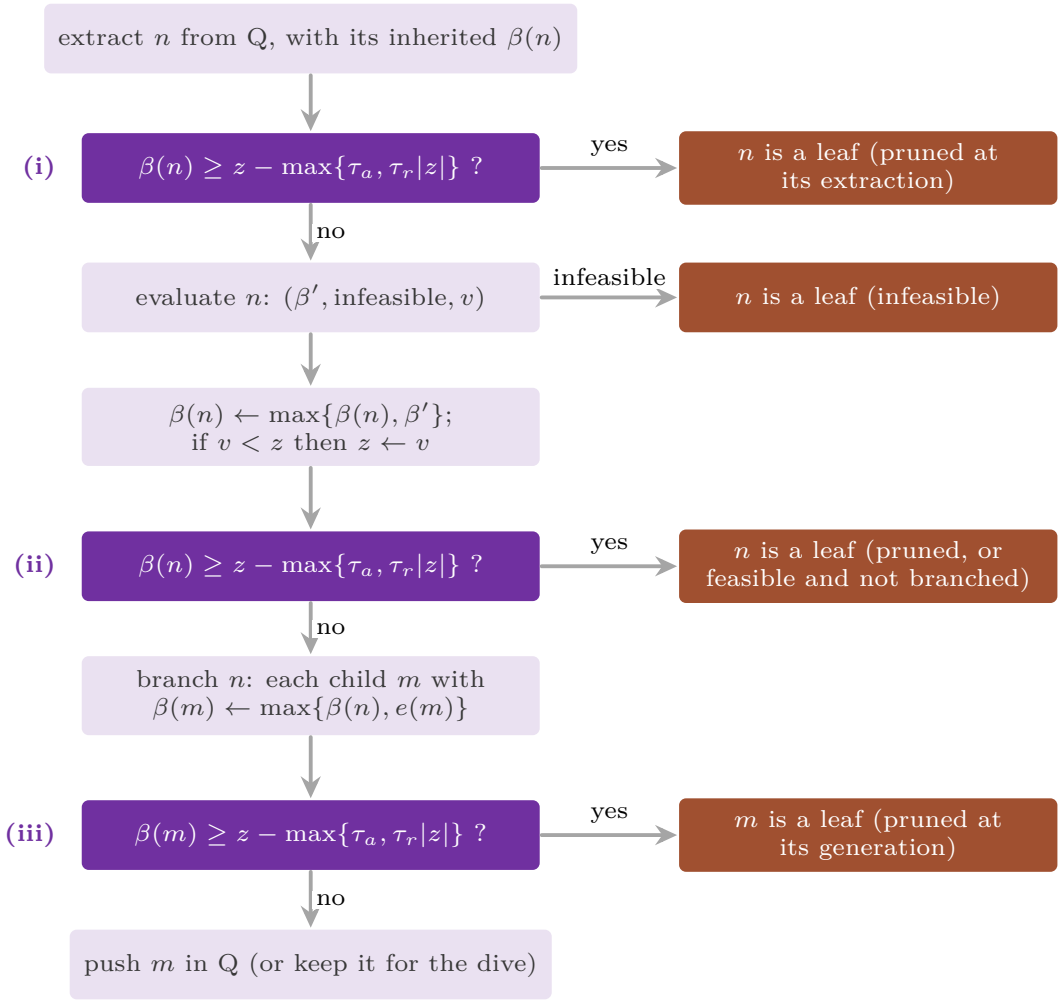


Figure 13.2. The procedure PROCESS of Algorithm 13.3: the three tests of pruning (i), (ii) and (iii), at lines 17, 22 and 26, and the leaves that each one makes. With $z = +\infty$ a test prunes only a bound $+\infty$.

the search goes down the right branch, whose nodes inherit the bound 16.29 (the ties among equal keys are broken in favor of the last node pushed, cf. §13.5); the node 110 has the bound 13.71, which is still above $z = 12$, and is branched. The node 10 gives the optimum 15 and is pruned by its bound, and the node 0 is pruned after its evaluation. Finally, the open node 1100, whose inherited bound is $13.71 \leq 15$, ends the search at line 6 without being evaluated; six nodes are evaluated in all.

step	extracted	evaluation and branching	Q after the step (first on the left)	z
0		(start)	root : $+\infty$	none
1	root	$\beta = 16.29$, $v = 12$: new z; children 0 and 1	1 : 16.29, 0 : 16.29	12
2	1	$\beta = 16.29$, $v = 12$; children 10 and 11	11 : 16.29, 10 : 16.29, 0 : 16.29	12
3	11	$\beta = 16.29$, $v = 12$; child 110 (111 does not fit)	110 : 16.29, 10 : 16.29, 0 : 16.29	12
4	110	$\beta = 13.71$, $v = 12$; child 1100 (1101 does not fit)	10 : 16.29, 0 : 16.29, 1100 : 13.71	12
5	10	$\beta = 15$, $v = 15$: new z; $\beta \leq z$: a leaf, not branched	0 : 16.29, 1100 : 13.71	15
6	0	$\beta = 14.14 \leq z$: a leaf, not branched	1100 : 13.71	15
7		the least open bound $13.71 \leq z$: the open nodes are leaves, the search ends with $z = 15$ and the bound $\max\{15, 14.14, 13.71\} = 15$		15

Figure 13.3. The steps of Algorithm 13.3 on the example, one row per evaluation: the node extracted, the outcome of PROCESS, the queue after the step with the nodes pushed by it in rust (each with its bound; the first node of the queue is on the left), and the incumbent. In this maximization, the test of pruning reads $\beta \leq z$.

13.4 Correctness and the reported bound

We now prove that Algorithm 13.3 returns a value within the tolerance of the optimum when no limit stops it, and that the bound it reports is a bound on the optimum whatever the status. We need four assumptions on the problem, which the concept cannot check:

- (A1) the children of a node cover its subset: $X(n) \subseteq X(n_1) \cup \cdots \cup X(n_k)$;
- (A2) the bound of an evaluation, the estimate of a child and the bound given with each node of R are lower bounds: $\beta'(n) \leq v(x)$ for every $x \in X(n)$, $e(m) \leq v(x)$ for every $x \in X(m)$, and likewise for the nodes of R (the root starts with $-\infty$);
- (A3) a node has no child only if it is solved, i.e., if its subset is empty or its evaluation gives a feasible value v with $v \leq v(x)$ for every $x \in X(n)$;
- (A4) every chain of branchings is finite, e.g., because the subsets shrink strictly and X is finite.

The knapsack satisfies them, as we now show; in its case $X(u)$ is the set of the $x \in \{0,1\}^n$ that agree with the node u on the items $0, \ldots, k-1$ and fit in the capacity.

Lemma 13.2 (The knapsack satisfies the assumptions). With or without `explicit_children`, and with values of a floating point or of an integer type, the knapsack of Algorithms 13.1 and 13.2 satisfies (A1) to (A4), the bounds being read in the minimization of the negated profits.

Proof. (A1) Let $x \in X(u)$ with $k < n$. If $x_k = 0$, then $x \in X(c)$. If $x_k = 1$, then $W(u) + w_k \leq \sum_j w_j x_j \leq C$, so that the child t is generated at line 5 of Algorithm 13.2, and $x \in X(t)$. (A2) If $W(u) > C$, then $X(u) = \emptyset$ and the node is reported infeasible; otherwise the bound of Algorithm 13.1 is at least the profit of each $x \in X(u)$ by Lemma 13.1, and so is its floor for an integer type, since the profits are integer. The estimate ∞ of a child is a trivial bound, and with `explicit_children` the estimate is the bound of the child, or ∞ if it does not fit. (A3) BRANCH gives no child only when $k = n$; then $X(u)$ holds at most the vector of the node, and if it fits its profit is $P(u)$, the greedy value of line 7 of Algorithm 13.1, while if it does not fit the node is infeasible. (A4) Each child fixes one item more than its parent, and therefore every chain of branchings has at most n steps. □

We also need the threshold to behave well when z improves.

Lemma 13.3 (The threshold is monotone). If $\tau_r \le 1$, the threshold $\theta(z) = z - \max\{\tau_a, \tau_r|z|\}$ is nondecreasing in z, and so is its version for an integer type, $z - \max\{\tau_a, \lfloor \tau_r|z| \rfloor\}$.

Proof. One has $\theta(z) = \min\{z - \tau_a,\ z - \tau_r|z|\}$, the minimum of two functions of z. The first is increasing, and the second has slope $1 - \tau_r \ge 0$ for $z > 0$ and $1 + \tau_r$ for $z < 0$; hence, both are nondecreasing, and so is their minimum. For the integer version, a unit increase of z changes $\lfloor \tau_r|z| \rfloor$ by at most $\lceil \tau_r \rceil \le 1$, so that $z - \lfloor \tau_r|z| \rfloor$ does not decrease, and the same argument applies. □

The condition $\tau_r \le 1$ is not checked by the library. With a larger relative tolerance the threshold of a positive incumbent falls when the incumbent rises, which one hardly wants (a relative gap of more than 100% accepts any value of the same sign). We call *open*, at a given moment of Algorithm 13.3, the nodes of Q and the node d of the dive.

Lemma 13.4 (The invariant). Under (A1) to (A3) and with $\tau_r \le 1$, at the start of each iteration of the loop at line 3 every $x \in X(R)$ with $v(x) < \theta(z)$ belongs to $X(n)$ for some open node n, where $X(R)$ is the union of the subsets of the nodes of R.

Proof. We proceed by induction on the iterations. At the start the open nodes are those of R, and the claim holds. Assume it at the start of an iteration, which either breaks (and then there is no next iteration) or removes a node n from the open ones at line 7 and processes it. First, z can only decrease (line 21), and then so does $\theta(z)$ by Lemma 13.3; hence, the set of the x to be covered can only shrink. Second, the elements of $X(n)$ below the threshold are covered again after PROCESS, in each of its four outcomes. If n is pruned at line 17 or 22, then $\beta(n) \ge \theta(z)$; since $\beta(n)$ is the maximum of $\beta'(n)$ and of bounds of supersets of $X(n)$, it is a bound of $X(n)$ by (A2), and no element of $X(n)$ is below the threshold. If n is infeasible (line 19), $X(n)$ is empty. If n has no child, each $x \in X(n)$ has $v(x) \ge v \ge z \ge \theta(z)$ by (A3), where v is its feasible value, offered at line 21. Otherwise, by (A1), each $x \in X(n)$ is in the subset of a child m, which is either pruned at line 26 (and then $v(x) \ge \beta(m) \ge \theta(z)$ by (A2)) or put in Q or in d at lines 12 and 13. □

Theorem 13.5 (Correctness and finiteness). Under (A1) to (A4) and with $\tau_r \le 1$, Algorithm 13.3 performs a finite number of iterations. If no limit stops it, then either $z < +\infty$ and

$$z - z^*_R \le \max\{\tau_a,\ \tau_r|z|\},$$

where $z^*_R = \min\{v(x) : x \in X(R)\}$, or $z = +\infty$ and $X(R)$ is empty (when no initial incumbent is given).

Proof. We note that each iteration that does not break removes one node from the open ones and adds only children of it; hence, each node of the trees below the nodes of R is processed at most once for each node of R above it. The set R is finite, and by (A4) and by König's lemma so are these trees, since each node has finitely many children (`branch()` returns); hence, so is the number of iterations. If no limit stops the search, it ends at line 3 with no open node, or at line 6 with each open node of bound at least $\theta(z)$. In both cases Lemma 13.4 shows that no $x \in X(R)$ has $v(x) < \theta(z)$, i.e., $z^*_R \ge \theta(z)$, which is the claim. If $z = +\infty$, then $\theta(z) = +\infty$ in the sense of the test at line 31, and no x exists. □

Note that with an initial incumbent value below the optimum, the search ends with that value and no node (the test `bb` checks this case). The value z of Theorem 13.5 is in any case a value found or given, and the inequality bounds it from above. When a limit stops the search, or when the tolerances are positive, the leaves give a bound on the optimum, which is the bound of `BBSolution`. The *leaves* of a search are the nodes it does not branch: those pruned at the three tests, the infeasible ones, those with no child, and those left open by a limit (lines 6 and 14). Algorithm 13.4 records the least bound `low` of those that lower the bound, i.e., all but the infeasible ones and those with no child, and it keeps the leaves themselves only with `keep_frontier`.

Theorem 13.6 (The reported bound). Under (A1) to (A3), whatever the status of the search,

$$z^*_R \ge L = \min\{\, z,\ \min\{\beta(l) : l \text{ a leaf that lowers the bound}\}\,\},$$

and therefore the gap $z - L$ reported by `BBSolution` bounds $z - z^*_R$.

Proof. If $X(R)$ is empty there is nothing to prove. Otherwise we let x^* be optimal over $X(R)$. By (A1), the subsets of the leaves cover $X(R)$; the argument is that of Lemma 13.4 with no threshold, since each node of the tree is either branched (and then covered by its children) or a leaf, the nodes still open when the search ends being leaves of kind `open`. Let l be a leaf with $x^* \in X(l)$. It is not infeasible, since $X(l) \ne \emptyset$. If it has no

child, then $z_R^* = v(x^*) \geq v \geq z \geq L$ by (A3), where v is its feasible value, offered to the incumbent. Otherwise it lowers the bound, and $\beta(l) \leq v(x^*)$ by (A2), so that again $z_R^* \geq L$. □

The leaf that has no child is excluded from L because its subtree is the node, whose value is offered to z; the infeasible leaf is excluded because its bound is $+\infty$. Note that the bound $\beta(l)$ of a leaf pruned at its generation or at its extraction is an inherited one, which is usually weaker than the bound its evaluation would give. The reported bound is therefore valid but not necessarily the best one available; it is exact (equal to z) when the search ends with zero tolerances and no limit, since then each leaf has $\beta(l) \geq z$. We remark that the theorem is stated for Algorithm 13.3, which prunes by its own z. With `shared_incumbent` the search prunes by the least c of z and of the values read, which are not offered to z, and the same proof gives only $z_R^* \geq \min\{L, c\}$: a leaf with no child whose value is not better than c is in neither z nor L (this is why the bound of Exercise 13.2 is at most the optimum). On the example the bound is $\max\{15, 14.14, 13.71, 15\} = 15$ in the maximization, and the gap is 0.

13.5 The selection rules

Lemma 13.4 holds whatever the order in which the open nodes are extracted. Therefore the order is a policy, the `Selection` of the options (`BranchAndBound.h:137`), which decides how many nodes are evaluated and how much memory the queue takes, while the value returned is within the tolerance anyway. In the queue we order the nodes by a key: the bound with best-first and plunging, minus the depth with depth-first, the depth with breadth-first, and the estimate of the problem with best-estimate (the bound, if the problem gives no estimate). Among equal keys the last node pushed comes first, but with breadth-first, where the first comes first. Figure 13.4 shows the five rules on the example.

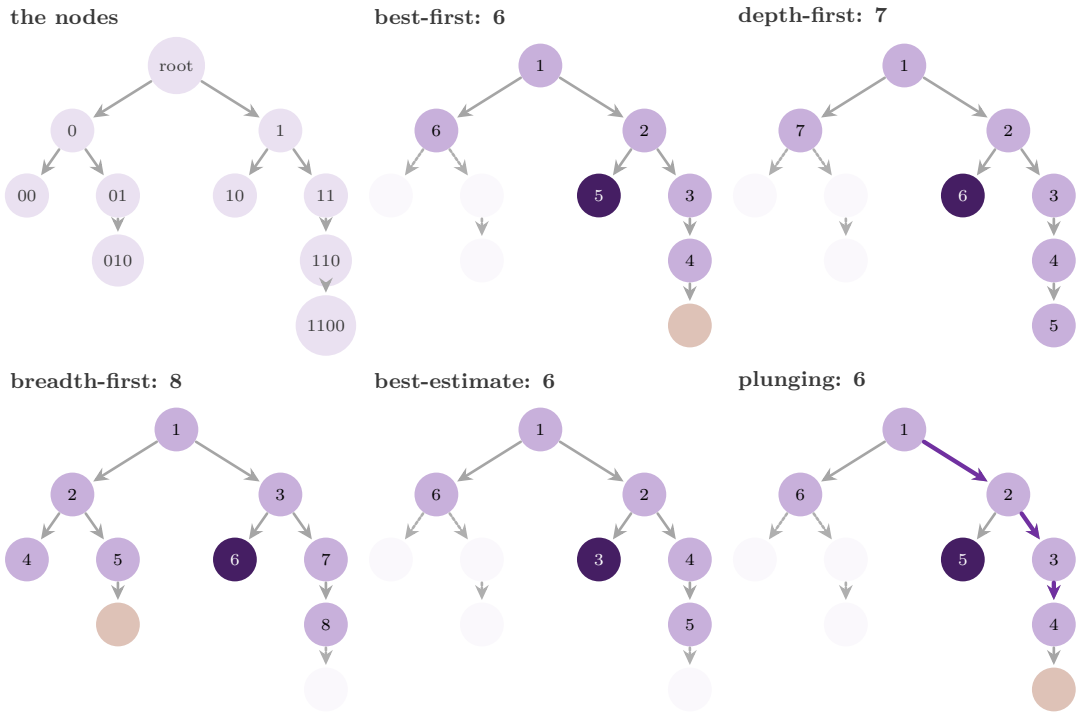


Figure 13.4. The five selections on the example: in each tree the number is the order of evaluation, the dark violet node is the node of the optimum, a rust node is pruned without evaluation and a faint one is not generated. In the plunging tree the thick violet edges are the dive from the root to 110; its child 1100 is not dived into, since its bound 13.71 is farther than $\gamma = 0.5$ of the gap from the least open bound.

Best-first

With best-first the node of least bound is extracted first. When the bound of a node does not depend on the order of the search, best-first evaluates the fewest nodes, in the following sense.

Proposition 13.7 (Best-first evaluates the necessary nodes, and "few" others). Assume (A1) to (A4), zero tolerances, no limit, no initial incumbent below z^*, and that the evaluation of a node depends only on the node

(no `cutoff`). Call a node *critical* if its inherited bound and the evaluated bounds of all its ancestors (taken as $+\infty$ for an infeasible one) are less than z^*. Then (i) each selection evaluates each critical node, and (ii) best-first evaluates only nodes whose inherited bound is at most z^*.

Proof. (i) We argue by induction along the path from the root. At the start the root is open. A critical node m that is open is never pruned at line 6, 17 or 26, since its bound is less than $z^* \leq z$; since the search ends with no open node of bound below z (Theorem 13.5), m is extracted and evaluated. If m has a critical child, its evaluated bound is less than z^* by the definition, and it is not pruned at line 22; it is therefore branched, and its critical children, whose inherited bounds are less than $z^* \leq z$, survive line 26 and are open. (ii) While $z > z^*$, some open node m holds an optimal x^* (Lemma 13.4, since $v(x^*) = z^* < z$ and the tolerance is 0), and $\beta(m) \leq z^*$ by (A2); best-first extracts a node of least bound, which is therefore at most z^*. Once $z = z^*$, a node extracted with $\beta \geq z^*$ is pruned at line 17 before its evaluation. □

Hence, the nodes evaluated by best-first are the critical ones plus some of those whose bound equals z^*, and how many of the latter depends on the ties (the test `bb` checks the optimum of every selection, not the number of the evaluations). This result is the counterpart of the optimality of A* among the algorithms that use the same heuristic (Chapter 2), and it fails in two ways that are common in practice: with `cutoff`, the evaluation of a node depends on the incumbent, and with a relaxation solved by a warm start from the previous node it depends on the order. Its price is the memory: the queue may grow to a "large" part of the tree before an incumbent is found, since each open node whose bound is below the incumbent of the moment stays in it until it is extracted.

Depth-first branch-and-bound

With depth-first the deepest node is extracted first, which typically finds feasible values early and keeps the queue "small". This is the depth-first B&B (DFBnB) of the literature of heuristic search, i.e., a depth-first search that prunes each node whose bound is not less than the best value found so far. It is an *anytime* algorithm, since an incumbent is available (through `on_incumbent`) long before the search ends. Since the children of a node have the same depth and the ties are broken in favor of the last pushed, the child that a problem gives last to the sink is the first one extracted. A problem that wants its children tried by increasing bound, as DFBnB with sorted successors does, gives them by decreasing bound (the knapsack gives the child that takes the item last, which is the child of the greedy order). The version of DFBnB on the graph of a `Domain`, without a queue, is in Chapter 4. Here, instead, the search works on any `BranchAndBound` problem, e.g., on the tree of the partial tours of the traveling salesperson problem that the benchmark uses (§13.20); there the first complete tour is reached after one evaluation per level of the tree, and from then on it prunes the partial tours whose bound is not less than its cost.

Proposition 13.8 (The memory of depth-first). With the depth-first selection, started from the root only, Q never holds more than $(b-1)D+1$ nodes, where b is the largest number of children of a node and D the depth of the tree.

Proof. We show that, at the start of each iteration, the nodes of Q at each depth $\delta \geq 1$ are children of one node, and that Q holds at most $b-1$ of them at each depth but its deepest one, where it holds at most b. This holds at the start, when only the root is in Q. An iteration extracts a node n of the deepest level δ of Q, which leaves at most $b-1$ nodes at depth δ, all siblings of n. It pushes at most b children of n at depth $\delta+1$, where Q had no node. The levels above δ are unchanged, and each of them lost one node (the ancestor of n at that depth) when it was the deepest; hence, each holds at most $b-1$ nodes. Summing over the depths $1, \ldots, D$ gives $(b-1)(D-1)+b = (b-1)D+1$. □

Depth-first pays in the evaluations: it may evaluate "many" nodes that best-first would prune, i.e., those whose bound lies between z^* and the incumbent of the moment (the depth-first tree of Figure 13.4 evaluates 1100, which best-first prunes unevaluated).

Breadth-first, best-estimate and plunging

Breadth-first extracts the shallowest node, the oldest among those of the same depth. It is rarely the right choice for a B&B whose feasible values come only from the leaves (on the example it is the worst of the five, with eight evaluations), but it gives the nodes of a level together, which some uses may want. *Best-estimate* extracts the node of least estimate of the best feasible value in its subtree, which the problem gives by `estimate( n )` (for the knapsack, its greedy value). It aims at finding a "good" incumbent early, while the bound still drives the pruning and the termination test.

Plunging is the usual compromise: the best child of the node just branched is extracted next (a *dive*) as long as its bound is within the fraction γ (`plunge_gap`) of the gap from the least open bound $\underline{\beta}$, i.e., while

$$\beta(k) \leq \underline{\beta} + \gamma\,(z - \underline{\beta}),$$

and always while there is no incumbent; when the dive stops, the search goes back to best-first (lines 9 to 12 of Algorithm 13.3). With $\gamma = 0$ a dive continues only into a child whose bound equals the least open bound. With $\gamma = 1$ it continues into each child that is not pruned, which makes the search depth-first in the order of the bound of the children and best-first among the nodes left behind (the test `bb` runs both extremes). One can expect a dive to be "cheap" when the evaluator keeps the state of the last node, e.g., a relaxation solved again from the basis of its parent. In the code (`BranchAndBound.h:1336`) the child k of line 10 is the last of the children of least bound, and the test of line 12 is `BBValueTraits::within()`, which computes the right-hand side in the type of the values for a floating point type and in `long double` for an integer one, which cannot overflow. The test is false when $\underline{\beta}$ is not finite (e.g., $-\infty$ for a node given back with an unknown bound, §13.11), and then no dive is taken while an incumbent exists.

The queue

The open nodes are kept in a `BBQueue` (`BranchAndBound.h:617`). Its nodes are in an array of slots, each holding a node with its bound and its depth, with a free list, since a node is destroyed when it is branched (in A* a node stays in CLOSED, while here no structure refers to a node after its processing). Its order is a pair of indexed 4-ary heaps on the indices of the slots, taken from the default data structures of the library (Chapter 3): one by the key of the selection, with the number of the push to break the ties, and one by the bound. With the second heap one has the least open bound in $O(1)$, which the termination test at line 6 and the reported bound need when the key is not the bound. Each change, the pop included, is made on both heaps at the places of the index, and the heaps grow by doubling with the slots. Figure 13.5 shows a pop with the depth-first selection.

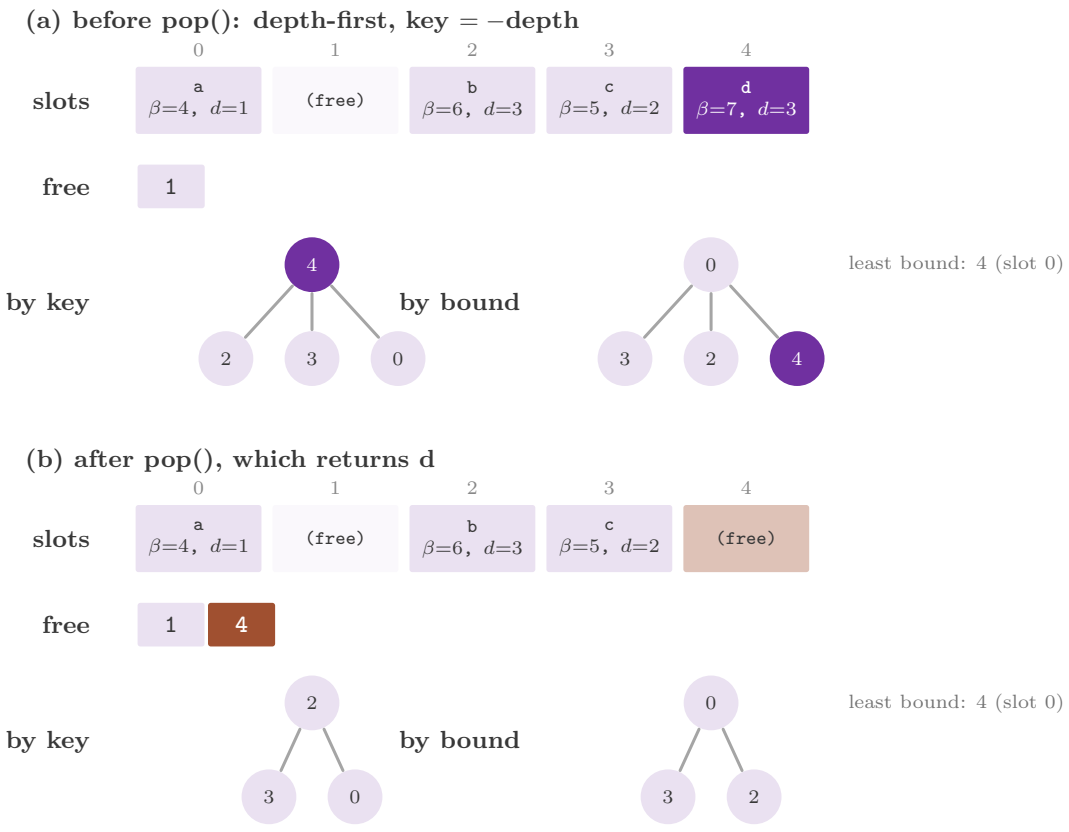


Figure 13.5. The queue of the open nodes before and after a pop with the depth-first selection (the key is minus the depth, the ties broken by the last push). Here the pop returns the node d of the slot 4, the top of the heap by key, removes the index 4 from both heaps at its places, and puts the slot on the free list, where the next push will take it; the least bound, at the top of the other heap, is still 4. The 4-ary heaps hold here at most four indices, i.e., a root and its children.

A push takes a free slot (or a new one), stores the node and pushes its index in both heaps, the key paired with the number of the push, complemented unless the order is first in, first out, and a pop takes the index of least key out of both heaps and frees its slot. In pseudocode, with H_K the heap of the keys and H_B that of the bounds, both on the indices of the slots, and s the number of the pushes so far:

Algorithm 13.5 (The queue of the open nodes).

```
 1  procedure PUSH(n, β, δ, key)          ▷ a node, its bound, depth, key
 2     i ← a slot of the free list, removed, or a new slot
 3     if i is beyond the capacity c of the heaps then
 4        rebuild both heaps with capacity 2i + 16, inserting their indices
 5     s ← s + 1;  slot[i] ← (n, β, δ)
 6     insert i in H_K with the key (key, s) if breadth-first, else (key, −s)
 7     insert i in H_B with the key β
 8  function POP()
 9     i ← the top of H_K;  remove i from H_K and from H_B
10     put i on the free list;  return the content of slot[i], emptied
11  function BESTBOUND()
12     return the key of the top of H_B, or ∞ if the queue is empty
```

Here $-s$ stands for the complement of s in 64 bits, which orders the pushes in the opposite way. The costs are the following.

Proposition 13.9 (The queue). Let N be the number of the open nodes and M the largest number of open nodes at any time. Then (i) the pairs (key, $\pm s$) are distinct, so that POP returns a well-defined node, the least key first and, among equal keys, the last pushed (the first pushed with breadth-first); (ii) BESTBOUND and the first node take $O(1)$ time, POP takes $O(\log N)$ time, and PUSH takes $O(\log N)$ time but for the rebuilding of line 4, whose total cost over a sequence of pushes is $O(M \log M)$, i.e., $O(\log M)$ amortized per push when at least M pushes are made; (iii) the queue uses $O(M)$ memory, besides the nodes themselves.

Proof. (i) Each push takes a new value of s, and the order on the pairs is lexicographic. (ii) In a 4-ary heap of N elements, an insertion moves an element up at most $\lceil \log_4 N \rceil$ levels with one comparison per level, and the removal of an element whose position is known (the heaps are indexed by the slots) moves the last element up or down along one path of at most $\lceil \log_4 N \rceil$ levels with at most 4 comparisons per level; a push makes one insertion in each heap and a pop one removal in each, and the free list is a stack. The top of a heap is its first element. The rebuilding of line 4 happens when the index i reaches the capacity c, and sets it to $2i + 16 > 2c$; the capacities are therefore at least doubled, and since $i < M + 1$ (a slot is new only when all the slots hold nodes), the rebuildings insert at most M indices each, with capacities $c_1 < c_2 < \cdots$ at least doubling up to $O(M)$, i.e., $O(\sum_t c_t \log M) = O(M \log M)$ in all. (iii) The slots are at most $M + 1$, and the heaps have capacity $O(M)$. □

The move of one node per push and per pop is apart, and so is the destruction of the node of a slot. These costs are usually negligible when the evaluation solves a relaxation, which is the case the layer is meant for. They matter only for problems whose bound costs about as much as a "few" comparisons; the knapsack of the tests is one, and it is therefore a test of the overheads of the schemes rather than of their intended use. One could also keep the bounds in a heap with lazy deletion, in which the indices popped by key stay until they reach its top: this saves the second walk of a pop, but the least bound must then skip the stale entries, while the indexed pair keeps the bound of each index exact.

We can now bound the cost of a whole sequential search in terms of the work of the problem. In the following, an *evaluation* costs at most T_e time and a branching, i.e., a call of `branch()`, at most T_b time besides the calls of `sink`.

Proposition 13.10 (The cost of the sequential search). Let Algorithm 13.3 process E nodes (line 8), generate G children, and let M be the largest number of its open nodes. Then it makes at most E evaluations and E branchings, and takes $O(E(T_e + T_b + \log M) + G \log M)$ time besides the leaves recorded at its end (lines 6 and 14, $O(M)$ time) and the calls of `shared_incumbent` and of the clock (one per iteration). It uses $O(M)$ memory for the open nodes, plus the leaves kept with `keep_frontier`.

Proof. We note that each iteration of the loop at line 3 that does not break extracts one node (line 7), by POP in $O(\log M)$ time (Proposition 13.9) or from the dive, and processes it. PROCESS makes at most one evaluation (line 18) and one branching (line 24), and for each child it makes $O(1)$ work and either a record in $O(1)$ (Algorithm 13.4, amortized for the append) or a push, in $O(\log M)$ amortized time; with plunging, lines 9 to 12 scan the children once. The least bound of line 5 is BESTBOUND, in $O(1)$. The memory holds the queue

(Proposition 13.9), the node of the dive, the children of one node and the record of the leaves. □

Note that $E \leq |R| + G$, since each node processed was pushed once, and that $G \leq bE$ when no node has more than b children; hence the overhead of the search is $O(b \log M)$ per evaluation, which is negligible when T_e is the time of a relaxation.

13.6 A* as a special case

Nau, Kumar and Kanal [17] formulated a general B&B of which A* (and AO*) are special cases. In our terms, A* on a tree is the B&B in which a node is a path from s_0 to a state u, and its subset is the set of the paths to a goal that extend it. Its bound is $f(u) = g(u) + h(u)$, which is a lower bound on the cost of these paths when h is admissible, and a goal gives the feasible value $g(u)$. In the benchmark of the chapter we use this construction (`TourTree` in `tools/BranchAndBoundBench.cpp`) on the traveling salesperson problem of Chapter 14, whose search space is a tree: `evaluate()` returns $g + h$ as the bound and g as the feasible value of a goal, and `branch()` gives each successor with its $g + h$ as estimate.

Proposition 13.11 (A* as best-first B&B). On the tree of the paths from s_0, with an admissible heuristic, $h = 0$ on the goals, nonnegative edge costs, zero tolerances and the best-first selection, the first goal that Algorithm 13.3 evaluates has the optimal cost C^*, and the search ends at the next test of line 6 with no further evaluation.

Proof. We first observe that, while no goal of cost C^* has been evaluated, $z > C^*$ and some open node m lies on an optimal path (Lemma 13.4); its bound is the largest f along that path up to m (by the pathmax of §13.2), which is at most C^* by admissibility. Let u be the first goal evaluated; when it was extracted it had the least bound among the open nodes, so that $\beta(u) \leq \beta(m) \leq C^*$, while $\beta(u) \geq f(u) = g(u) \geq C^*$ since $h(u) = 0$. Hence $g(u) = C^*$, the evaluation sets $z = C^*$, and the least bound of the open nodes is at least $\beta(u) = C^*$ (best-first extracted u as a node of least bound, and the children pushed since then have bounds of at least that of their parent). The test at line 6 then prunes all of them. □

We remark that the proof shows the role of pathmax. With an inconsistent heuristic f may decrease along a path, but the inherited bound does not, and best-first on the inherited bound extracts the nodes in nondecreasing order of their key. Still, the two searches differ in two respects. First, a graph has *duplicates*, i.e., several paths to the same state, of which only the cheapest matters. A* detects them with CLOSED and keeps one node per state, while a B&B tree has none and needs no closed list; on a graph with many duplicates the B&B view enumerates the paths, and A* is the right tool. Second, A* stops at the first goal extracted, while a B&B goes on until no open node can improve the incumbent. In a general B&B the first feasible value is often not optimal, and a node with a feasible value may hold better values below, as the root and the node 1 of Figure 13.1 do (value 12, and the optimum 15 below). This is why the two layers of the library share the parallel schemes and the executors, but not the node table and the OPEN lists of A*. For the same reason SafePBNF and PA*SE (Chapter 11 and Chapter 8) have no counterpart here: they exist to handle the duplicates and the re-expansions, which a tree does not have.

13.7 What a parallel search has to guarantee

We now turn to the parallel B&B searches of the library, all of which build one tree with p workers and differ in how they share the open nodes and the incumbent. In the classification of Gendron and Crainic [16] they are of the second type, i.e., the parallelism is in the exploration of the tree. Any parallelism inside the evaluation of a node (their first type) is left to the problem, and the search of several trees at once (their third type) is provided only at the start, by the racing ramp-up of §13.15. Within the second type the searches are asynchronous, but for the deterministic rounds. They keep either one pool of open nodes (the shared queue, the MultiQueue, the farm and the rounds, §13.8) or one pool per worker (the distributed queues, the processes of MPI, the work stealing and the SelfSplit, §13.9, and the skeletons, the hubs and the intervals of leaves of §13.14, §13.17 and §13.18).

Note that the invariant of the sequential search holds for a parallel one if the open nodes are taken in a wider sense: the nodes in any queue, those held by a worker (being processed, or the next node of a dive), and those traveling in a message, a task, an offer of a steal or a slot of a round. Since the incumbent is shared, each worker prunes by the value it sees, which may be older than the current one; the following lemma shows that this is safe. The incumbent is an object of the class `BBIncumbent` (`ParallelBranchAndBound.h:227`), whose two operations are lines 1 to 8 of the following algorithm; lines 9 to 17 give the dive and the test of the limits, which all the parallel searches share and which are described at the end of this section.

Algorithm 13.6 (The shared incumbent, the dive and the limits of the parallel searches).

```
 1  function VALUE()                        ▷ read by the workers to prune
 2     return z, an atomic read with relaxed order if the values have a
          lock-free atomic type, else a read under the lock M
 3  function IMPROVE(v, n)                  ▷ the feasible value v of n
 4     if the values are atomic and v ≥ VALUE() then return false
 5     lock M
 6     if v ≥ z then unlock M;  return false
 7     z ← v;  the node of z ← n;  call on_incumbent(v);  unlock M
 8     return true
 9  function DIVE(K, bl, z)                 ▷ bl: the least open bound known
10     if K = ∅ then return nil
11     if depth-first then return the last node of K
12     if not plunging then return nil
13     k ← the last node of K of least bound;  bl ← min(bl, β(k))
14     if z = ∞ or (bl is finite and β(k) ≤ bl + γ (z − bl)) then return k
15     return nil
16  function LIMIT(c)                       ▷ c: the nodes counted so far
17     return (node_limit > 0 and c ≥ node_limit) or the time limit has
          passed, setting the flags of the limit (and of the time) if so
```

Lemma 13.12 (The shared incumbent). In `BBIncumbent` (`ParallelBranchAndBound.h:227`), (i) the value never increases, (ii) each value it takes is the initial one or a feasible value offered by a worker, and (iii) the stored node is a node of the stored value whenever the lock is free, and the callback `on_incumbent` is called with strictly decreasing values. Hence, a worker that prunes by a value it read earlier prunes a subset of what it would prune by the current one.

Proof. We note that the value changes only at line 7 of Algorithm 13.6, under the lock M, after the test of line 6, which reads the value under the same lock. That value is the last one stored, since all the stores are made under M, which orders them; the store of the node and the call of the callback follow under M. The test of line 4 only returns early, when v is not below a value that z had at some moment, which is not below the current one, since each store of line 7 lowers z (line 6). This gives (i), (ii) and (iii). A worker that read the value $z' \geq z$ earlier (the read without the lock may return an older value) prunes $\beta \geq \theta(z')$, which implies $\beta \geq \theta(z)$ by Lemma 13.3. □

In the library, when the type of the values has a lock-free `std::atomic`, the value is an atomic variable on a line of cache of its own, which the workers read without a lock and with relaxed order when they prune. The first test of `improve()` discards without the lock the values that are not better than the one seen, and only an improvement, which is usually "rare", takes the lock. For another type (e.g., the rationals of an exact arithmetic) the value is read under the lock too.

Theorem 13.13 (Correctness of a parallel search). Under (A1) to (A4) and with $\tau_r \leq 1$, let a parallel search process each node by PROCESS with an incumbent value read from `BBIncumbent` (or, over MPI, received from it, §13.9), and let it move nodes only between the places of the wider open set. Then at any moment every $x \in X(R)$ with $v(x) < \theta(z)$, z the current incumbent, belongs to $X(n)$ for a node n of the wider open set. Hence, if the search ends only when the wider open set is empty, or when each node in it has a bound of at least $\theta(z)$, its value satisfies Theorem 13.5, and the bound computed from the leaves of all the workers satisfies Theorem 13.6.

Proof. We note that the proof of Lemma 13.4 applies to each step of each worker, the steps of different workers being interleaved. A step that moves a node keeps it in the wider open set. A step of PROCESS replaces the node by its children or makes it a leaf, and a leaf pruned by a value z' read earlier has no element below $\theta(z') \geq \theta(z)$ (Lemma 13.12). An improvement of z shrinks the set to be covered. Finally, the leaves collected by the workers cover $X(R)$ together with the wider open set, which gives the bound as in Theorem 13.6 once the nodes left open are added as leaves (each search does so at its end). □

All the schemes below satisfy the hypotheses, but for the SelfSplit, whose workers prune by incumbents of their own and which Theorem 13.23 treats apart. What remains to prove for each is that it ends only in one of the two situations of Theorem 13.13, and that it does end. Of the two, the end is the delicate part: a worker that finds no node cannot conclude that none exists, since another worker may be about to generate some (cf. the general discussion of the termination of parallel searches in Chapter 8 and Chapter 9). Also, one can stop each scheme at a node or time limit (LIMIT, lines 16 and 17 of Algorithm 13.6, `detail::bb_limit()`, `ParallelBranchAndBound.h:360`), and the node that a worker holds when it sees the limit becomes an open leaf. In every search the count c is that of the evaluations, i.e., of the calls of PROCESS that reach line 18 (a node pruned at line 17 is

not counted); in the shared queue, the MultiQueue, the farm, the distributed queues and the work stealing it is a counter shared by the workers, raised after each such call. The limit is then exceeded by a bounded amount.

Proposition 13.14 (The node limit of the asynchronous schemes). Let a node limit $L > 0$ be given, and no time limit. In the shared queue, the MultiQueue, the distributed queues and the work stealing, the evaluations are at most $L + p - 1$; in the farm, with batches of k nodes, at most $L + pk$.

Proof. In the first four schemes a worker calls LIMIT on the counter c before it processes each node, the node of a dive included, and raises c after an evaluation. Consider the last evaluation that starts: at its test $c \leq L - 1$, i.e., at most $L - 1$ evaluations had ended, and at most $p - 1$ others were under way, one per other worker; hence the evaluations are at most $(L - 1) + (p - 1) + 1$. In the farm the master tests the limit when a task comes back and dispatches no task after the test has succeeded, while a worker tests it only after each evaluation, to stop the dive of the node, and completes the at most k nodes of its task. The evaluations that end after c has reached L belong to the tasks out at that moment or sent by the dispatch under way, which are at most p together, since the master never has more than p tasks out; each of them evaluates at most its k nodes, since no dive goes on once $c \geq L$. □

The bound counts the evaluations; the nodes pruned when extracted, which cost no evaluation, are not counted. The rounds, instead, never exceed the limit (Theorem 13.18), and over MPI each process stops at its share (§13.9).

Anomalies

Of course, the evaluations of a parallel search are not those of the sequential one, and they may be more or fewer (often, but not always, more). On the one hand, a worker may evaluate a node that the sequential search prunes by an incumbent which, in the parallel search, another worker finds only later (a *detrimental* anomaly, which is the search overhead of a B&B). On the other hand, a worker may find an incumbent earlier than the sequential search does and prune nodes that the latter evaluates (an *acceleration* anomaly, which gives a speedup larger than p). Lai and Sahni [149] showed that both occur in parallel best-first B&B, and that adding workers may even "slow" a search down; Gendron and Crainic [16] discuss them with the literature that followed. With the best-first selection the detrimental part is somewhat limited, since the nodes extracted by the workers are among the best open ones. It grows with the distance between the order of extraction and the sequential one, and this distance distinguishes the schemes below.

The dive in a parallel search

With depth-first and plunging, a worker of a parallel search goes on with one child of the node it processed without putting it in a queue (DIVE, lines 9 to 15 of Algorithm 13.6, `detail::bb_dive()`, `ParallelBranchAndBound.h:335`). With depth-first it is the last child, which the queue would extract next anyway. With plunging it is the last child of least bound, if that bound is "close enough" to the least open bound bl that the worker knows: the least bound of the shared queue (read under its lock), that of the queue of the worker in the distributed queues, over MPI and in the work stealing, the least open bound of the moment at which the task was dispatched in the farm, and the bound of the node processed in the MultiQueue. We write DIVE(K) for that child, nil if there is none; it stays in the wider open set as a node held by the worker.

13.8 One pool of open nodes

The shared queue

In the shared queue (`SharedBranchAndBound`, `ParallelBranchAndBound.h:389`) the workers extract the nodes from one queue under a lock L, evaluate and branch them outside of it, and push the children back under it, as the workers of SPA* do with the states of A* (Chapter 8); the child of a dive stays with the worker. Its order of extraction is close to the sequential one, from which it departs by the p nodes processed at once and by the times at which the incumbents are found. Also, the lock is rather "cheap" when the evaluation dominates, which is the usual case of a B&B. The pseudocode, with a counter `busy` of the workers that hold a node, is the following.

Algorithm 13.7 (The shared queue).

```
1  procedure SHARED(R)        ▷ one queue Q, under the lock L, for all
2     Q ← R;  busy ← 0;  done ← false
3     in parallel, on each worker i = 0, ..., p − 1 do
```

```
 4        d ← nil
 5        loop
 6           if d ≠ nil then n ← d;  d ← nil
 7           else n ← NEXT();  if n = nil then return
 8           if a limit is reached then
 9              n is a leaf (open);  busy ← busy − 1;  done ← true;  return
10           K ← PROCESS(n)
11           lock L;  d ← DIVE(K);  push the nodes of K but d in Q
12           if d = nil then busy ← busy − 1
13           unlock L
14  procedure NEXT()
15     loop
16        if done then return nil
17        if Q = ∅ then                                  ▷ read without the lock
18           if busy = 0 then
19              lock L;  if Q = ∅ and busy = 0 then done ← true;  unlock L
20           else wait a little
21           continue
22        lock L
23        if PRUNABLE(the least bound of Q) then
24           the nodes of Q are leaves (pruned);  Q ← ∅;  unlock L
25           continue
26        busy ← busy + 1;  n ← the first node of Q, removed;  unlock L
27        return n
```

Figure 13.6 shows two steps: a worker extracts a node, raising `busy`, while another pushes the children of the node it processed, lowering it.

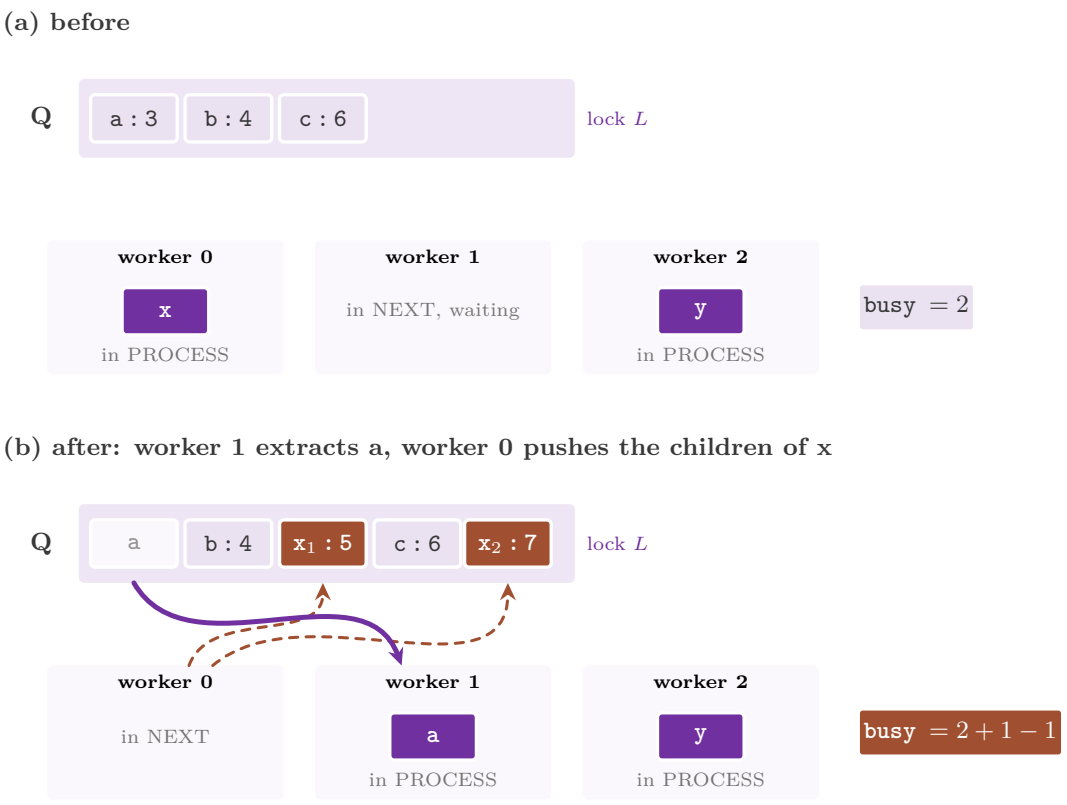


Figure 13.6. The shared queue before and after two steps of Algorithm 13.7 (minimization, each node with its bound): worker 1 extracts the first node a under the lock (line 26, busy +1), and worker 0, having processed x, pushes its children x_1 and x_2 under the lock and, having no dive, leaves the count (lines 11 and 12, busy −1).

Proposition 13.15 (The end of the shared queue). In Algorithm 13.7, (i) whenever L is free, `busy` is the number of the workers that hold a node (extracted and not yet pushed back, or kept for a dive); (ii) if a worker finds $Q = \emptyset$ and `busy` $= 0$ under L (line 19), no node is open in the wider sense; (iii) if the evaluations terminate,

each worker returns.

Proof. (i) We observe that a worker raises `busy` under L in the same critical section in which it takes a node from Q (line 26). It keeps it raised while it processes the node and the nodes of its dive (lines 6 and 10), and lowers it under L only in the critical section in which it has pushed the children and kept no dive (lines 11 to 13), or when it stops at a limit (line 9). (ii) The wider open set of this scheme is Q and the nodes held by the workers (no node travels), and both are empty. (iii) A worker loops in NEXT only while Q is empty and `busy` > 0, or while Q is not empty. In the first case some worker holds a node, which it processes in finite time, and whose dive ends at a leaf of the finite tree; that worker then pushes the children or lowers `busy`. In the second case the worker takes a node or empties Q at line 24. Since each node is processed once and the tree is finite (A4), the loop ends by line 16 or 19. □

The test of line 17 reads without the lock an atomic copy of the size of Q. Hence, a worker that finds no node waits without taking the lock, and with a growing pause, while the decision to stop is taken under the lock, which makes the stale reads harmless. The counters `busy`, the size of Q, the evaluations and the flag `done` sit on separate lines of cache. Also, when the least bound of Q cannot improve the incumbent, the whole queue becomes a set of leaves at once (line 24). The push of the children (lines 11 to 13) holds the lock for the time of the pushes of one node, i.e., $O(|K| \log N)$ by Proposition 13.9, and the extraction (lines 22 to 27) for one POP; each node processed thus costs two critical sections, besides those of line 19, and the lock of the incumbent only when it improves it.

The MultiQueue

The MultiQueue B&B (`MultiQueueBranchAndBound`, `MultiQueueBranchAndBound.h:129`) is the shared queue with the relaxed queue of MQA* (Chapter 8). The open nodes are in a MultiQueue of Rihani, Sanders and Dementiev [37], i.e., $2p$ sequential queues, each under a separate lock. A pop reads the least keys of two queues drawn at random, without locks, and takes the least node of the better one; no lock is common to all the workers. The price is a relaxed order of extraction: indeed, a pop may return a node that is not the least of all. In the sequential model of the MultiQueue (the nodes spread at random, then removed by the better of two queues drawn at random) the expected number of open nodes with a smaller key, the *rank error*, converges, as the pops go on, to a value below $\frac{5}{6} \cdot 2p$, i.e., it is $O(p)$ (cf. [37] and [38]). This is a result of the sequential model with two new queues at each pop (the library draws two new queues at each pop, i.e., with a stickiness of 1), which we do not prove here; it does not cover the buffers of the queues nor the push of all the children of a node into one queue, and we claim no bound on the rank error of the library. Its termination test cannot use a counter of busy workers, since there is no lock under which Q and the counter are read together. It uses instead a counter m of the nodes that are in the queue or held by a worker.

Algorithm 13.8 (The MultiQueue).

```
 1  procedure MULTIQUEUE(R)
 2     Q ← R;  m ← |R|;  done ← false
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        d ← nil
 5        while not done do
 6           if d ≠ nil then n ← d;  d ← nil
 7           else
 8              n ← a pop of Q, possibly nothing
 9              if no node was popped then
10                 if m = 0 then done ← true else wait a little
11                 continue
12              if PRUNABLE(the bound of n) then
13                 n is a leaf (pruned);  m ← m − 1;  continue
14           if a limit is reached then n is a leaf (open); done ← true; return
15           K ← PROCESS(n);  d ← DIVE(K)
16           m ← m + |K| − 1;  push the nodes of K but d in one queue of Q
```

Proposition 13.16 (The end of the MultiQueue). In Algorithm 13.8, m is at any moment at least the number of the nodes in Q plus those held by the workers; hence, $m = 0$ at line 10 means that no node is open, and the search then ends. If the evaluations terminate, each worker returns.

Proof. We first note that at the start $m = |R|$, and the claim holds with equality. A pop moves a node from Q to a worker and leaves m unchanged, and a node pruned at line 13 leaves both and is subtracted. At line 16 the

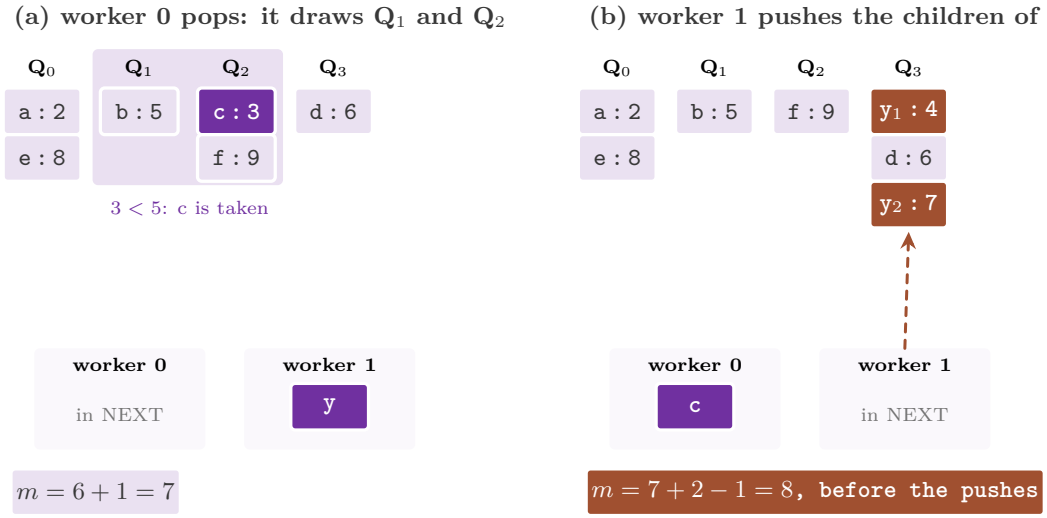


Figure 13.7. The MultiQueue with $p = 2$ workers and $2p = 4$ queues (minimization). (a) Worker 0 pops: it reads the least keys of Q_1 and Q_2, drawn at random, and takes c, of bound 3, from the better one, although a, of bound 2, is in Q_0; the counter m counts the six queued nodes and the node y held by worker 1. (b) Worker 1 has processed y: it adds its two children and drops y in one atomic step, $m \leftarrow m + 2 - 1$, before it pushes both children into one queue under one lock.

node processed leaves and $|K|$ children appear, of which DIVE(K) is held by the worker and the others enter Q. Since m is changed by $|K| - 1$ in one atomic step *before* the pushes, between this step and the end of the pushes m counts children not yet in Q, which is an overcount and never an undercount. A worker that stops at a limit sets `done` for all. For the second claim, note that a pop finds no node only after a scan of all the queues found them empty. Hence, a worker waits at line 10 only while $m > 0$ and each counted node is held by a worker or being pushed; those nodes are processed in finite time, and the tree is finite. □

The least bound of Q is not known, since each queue knows only its least key. Hence, a node is pruned when popped (line 12) rather than with all the others, and DIVE compares the children with the bound of the node processed. The children of a node go into one queue under one lock (the operation `pushAll` of the MultiQueue), which relaxes the rank somewhat, as a stickiness of their number would, and saves a lock per child. Furthermore, the values must be atomic with no lock (the concept `MultiQueueValue`, `MultiQueueBranchAndBound.h:105`), since the MultiQueue keeps the least key of each queue in an atomic word that the pops read without the lock. Similarly, the counter of the evaluations, which each worker would update, is touched only when a node limit is set. In the code m is the counter `held`, and line 16 is one atomic addition of $|K| - 1$ (a subtraction of 1 when $K = \emptyset$, in the modular arithmetic of the unsigned counter), made before the operation `pushAll` that moves the nodes of K but d into one queue under its lock; the case of one child, which replaces the node processed, needs no atomic operation. We expect the MultiQueue to pay off only when an evaluation is so "fast" that the lock of the shared queue is the bottleneck; when the evaluation dominates, its relaxed order perhaps costs more evaluations than the lock costs time.

The master-worker farm

The farm (`FarmBranchAndBound`, `FarmBranchAndBound.h:104`) is the master-worker form of the shared queue, written as a farm with feedback, the pattern of FastFlow [12], on the executor of the search (the tests run it on the three executors: the threads of C++, OpenMP and FastFlow). The master holds Q, extracts the nodes by the selection and sends them in tasks of `batch` nodes to the free workers. A worker evaluates and branches each node of its task (going on with the child of a dive in the same task), and sends the task back with the children that survive the pruning, which the master pushes in Q.

Algorithm 13.9 (The master-worker farm).

```
1  procedure MASTER(R, p)
2     Q ← R;  out ← 0
3     loop
4        DISPATCH()
5        if out = 0 and (a limit is reached or Q = ∅) then return
```

```
 6       receive a task T back from a worker;  out ← out − 1
 7       push the children of T in Q
 8  procedure DISPATCH()
 9     while no limit is reached, out < p and Q ≠ ∅ do
10        if PRUNABLE(the least bound of Q) then
11           the nodes of Q are leaves (pruned);  Q ← ∅;  return
12        T ← the first batch nodes of Q, removed
13        send T to a free worker;  out ← out + 1
14  procedure WORKER(T)                          ▷ on each of the p workers
15     for each n ∈ T do
16        while n ≠ nil do                     ▷ the node, then its dive
17           K ← PROCESS(n);  n ← DIVE(K), unless the node limit is reached
18           add the nodes of K but n to the children of T
19     send T back to the master
```

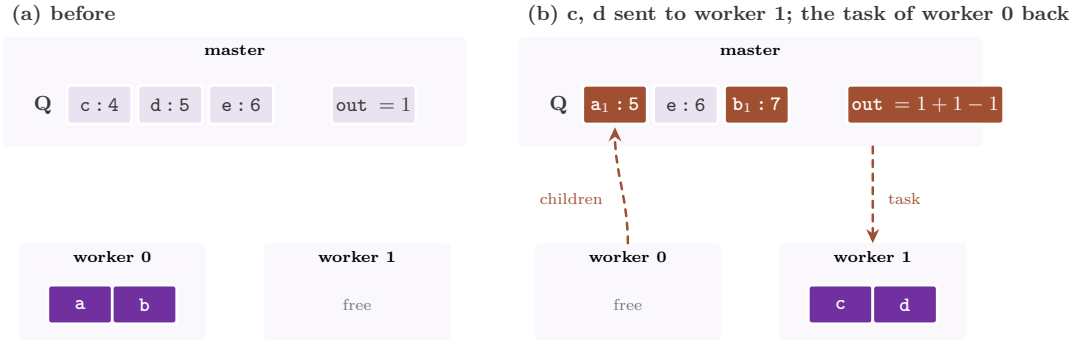


Figure 13.8. The farm with batches of two nodes. (a) The master holds Q and one task, with the nodes a and b, is out at worker 0. (b) The master has sent the task of c and d to the free worker 1, and worker 0 has sent its task back with the children a_1 and b_1, which the master has pushed in Q; out is $1 + 1 - 1 = 1$.

Proposition 13.17 (The end of the farm). In Algorithm 13.9 each open node is either in Q or in a task that is out at a worker, and `out` is the number of such tasks. Hence, the test at line 5 is exact: the master returns only when each task is back and either no node is open, or a limit is reached, or (through line 11) no open node can improve the incumbent.

Proof. We observe that only the master changes Q and `out`. It moves nodes from Q to a task and raises `out` (lines 12 and 13), and it lowers `out` when a task comes back, pushing its children (lines 6 and 7). A worker changes only the nodes of its task, replacing each node by the children that survive PROCESS and keeping the dive within the task. When `out` $= 0$ no task is out, and Q holds all the open nodes. □

The master also never waits for nothing: if the evaluations terminate, the farm ends. Indeed, a task out comes back in finite time, since its worker processes at most `batch` nodes and their dives, each of which ends at a leaf by (A4); and after each DISPATCH either `out` > 0, or Q is empty, or a limit is reached, since DISPATCH sends a task whenever `out` $< p$, Q is not empty and no limit is reached, unless it empties Q at line 11. Hence the receive of line 6 always has a task to wait for, each node is processed once, and the tree is finite.

In the farm the order of extraction is that of the selection, and the master only moves nodes; hence, the farm suits an evaluation that is "expensive" w.r.t. the transfer of a node. The batch of a task trades the messages for evaluations. The nodes of a task are extracted together, and the children of the first of them cannot overtake the others; a batch of 1 is therefore the one for an expensive evaluation, while the default of the parallel options, 16 nodes, amortizes the messages when the evaluation is "cheap". Also, the tasks are recycled through a free list of the master, and the farm runs $p + 1$ threads, the master on the first CPU of the affinity mask when the workers are pinned. The dispatch is lines 8 to 13 of Algorithm 13.9, in which each task also carries the least open bound of the moment of its dispatch, the bound bl of the DIVE of its worker.

The deterministic rounds

The rounds (`DeterministicBranchAndBound`, `DeterministicBranchAndBound.h:125`) make the outcome independent of the timing of the workers, as the deterministic mode of FiberSCIP [18] does, so that a run can be repeated exactly (e.g., for the experiments of a paper, or to find a bug). At each round the r best open nodes

are extracted from one queue ($r =$ `round`, or p if 0), and the worker i processes the nodes $i, i+p, i+2p, \ldots$ of the round against the incumbent z_0 of the start of the round. When all of them are done, one worker merges the outcomes in the order of the nodes of the round and extracts the next round, while the others wait at a barrier.

Algorithm 13.10 (The deterministic rounds).

```
 1  procedure DETERMINISTIC(R)
 2     Q ← R;  z ← the given incumbent, or ∞;  N ← ()
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        loop
 5           barrier, at which one worker runs NEXTROUND()
 6           if done then return
 7           for j = i, i + p, i + 2p, ... while j < |N| do
 8              (K_j, v_j) ← PROCESS(N_j) against z₀, v_j being the best
                   feasible value of N_j below z₀, if any
 9  procedure NEXTROUND()
10     for j = 0, ..., |N| − 1 do                    ▷ in the order of N
11        if v_j < z then z ← v_j;  N_j is the node of z
12        for each m ∈ K_j do
13           if PRUNABLE(β(m)) then m is a leaf (pruned) else push m in Q
14     if a limit is reached, Q = ∅ or PRUNABLE(the least bound of Q) then
15        done ← true;  return
16     N ← the first min(r, evaluations left) nodes of Q, removed
17     z₀ ← z
```

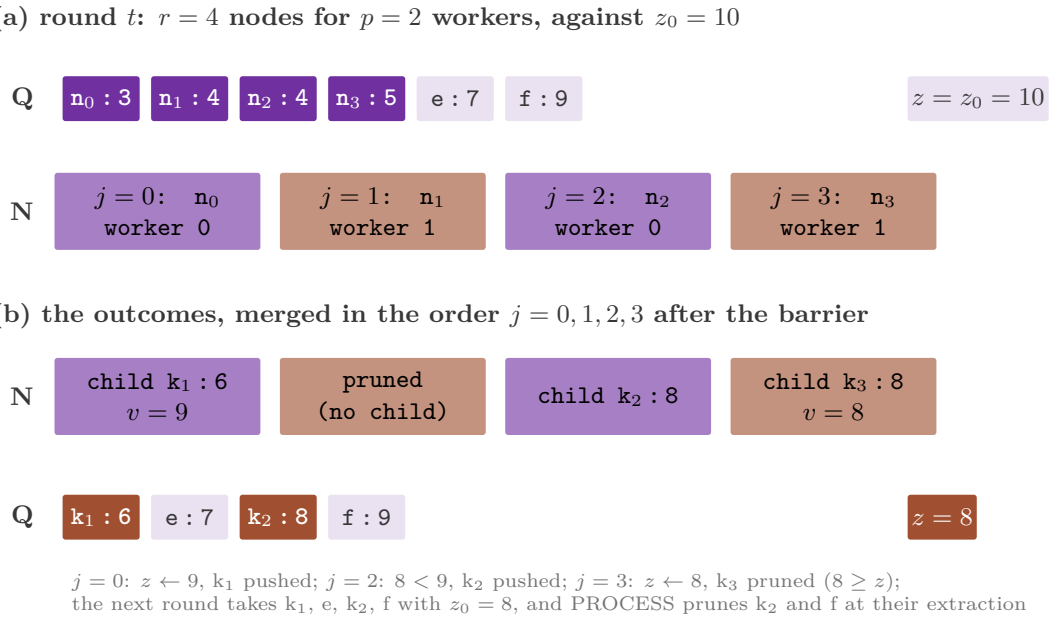


Figure 13.9. A round with $r = 4$ nodes and $p = 2$ workers (minimization). (a) The four best nodes of Q form the round, the node j going to worker j mod 2, and each node is processed against $z_0 = 10$. (b) After the barrier the outcomes are merged in the order of j: the feasible value 9 of N_0 becomes the incumbent and its child k_1 is pushed; the child k_2 of N_2 is pushed, since $8 < 9$; the value 8 of N_3 becomes the incumbent and prunes the child k_3, of bound 8. The order of the merge, not the order in which the workers finished, decides all of this.

Theorem 13.18 (Determinism of the rounds). Assume that no time limit is set, and that the evaluator of each worker returns, for a node and an incumbent given by `cutoff`, an outcome that depends only on them. Then the nodes evaluated, the incumbent value, its node, the bound and the counters other than the evaluations of each worker and the time are functions of the problem, of R, of the options and of r, and depend neither on the timing nor on the number p of the workers. If the outcome of the evaluator of worker i also depends on the nodes that i evaluated before, the same holds with p among the arguments. The node limit is exact.

Proof. We show by induction on the rounds that the state at the barrier (the content of Q with the numbers of their pushes, z and its node, the leaves and the counters) is such a function. At the start it is fixed by R and

the options. Given the state, NEXTROUND extracts N by the order of the queue, which depends only on the keys and the numbers of the pushes, and sets z_0. Each outcome (K_j, v_j) is computed by PROCESS on N_j with the incumbent z_0: the function `inc()` that the workers pass returns the snapshot of the round, and `improve` records the best value of the node without touching z. Hence, the outcome is a function of N_j and z_0 under the first assumption. Under the second one it is a function of N_j, z_0, p and the earlier rounds, since the nodes of worker i are those of index $i \bmod p$. The merge then runs on one thread in the order of j. Finally, the last round is shortened to the evaluations left (line 16), and a round never stops in the middle; thus the node limit is met exactly. □

We remark that the history of a worker depends on p, as the proof shows; hence, the independence of p needs an evaluation that depends on the node (and on z_0) only, as that of the knapsack does, and a history-dependent evaluator gives only the weaker property. Indeed, the test `bb-det-thread` checks that the status, the value, the node, the bound and the counters of the evaluated, generated and pruned nodes are the same with 1, 2, 3 and 8 workers and at each repetition, also when a node limit stops the search. The test `bb-cutoff` checks that the value, the bound and the numbers of the evaluated and of the infeasible nodes are the same with workers that fence the nodes by the incumbent given by `cutoff`. The time limit, checked between the rounds, makes the end depend on the timing.

The price is the wait of each round for its slowest node, which is usually "small" when the evaluation dominates and its time varies little among the nodes, and an incumbent that may be one round old (which may cost somewhat more evaluations). Also, no dive is taken, and the plunging selection extracts the nodes by their bound. In the code the barrier is a `std::barrier` whose completion function is NEXTROUND; the merge thus runs on one of the workers (the standard does not say which; in practice the last to arrive) while the others are blocked. Furthermore, the slots of a round, each with its node, its children, its best value, its counters and its leaves, are reused from round to round. An exception thrown by an evaluation is kept, and thrown again by the search after the workers have stopped. The work of a worker in a round is line 8 of Algorithm 13.10, with the function `inc()` of PROCESS returning z_0 and the function `improve` keeping in the slot the best value below z_0; the merge is lines 10 to 13, in the order of the slots, where the counters of each slot are added to those of the search, so that they do not depend on which worker processed the slot. It remains to show that the rounds are correct and end.

Proposition 13.19 (The end of the rounds). Under (A1) to (A4) and with $\tau_r \leq 1$, (i) at each barrier, after NEXTROUND, every $x \in X(R)$ with $v(x) < \theta(z)$ is in the subset of a node of Q or of the new round N; (ii) when NEXTROUND sets `done` with no limit reached, the value satisfies Theorem 13.5 and the bound Theorem 13.6; (iii) if the evaluations terminate, the search ends after finitely many rounds.

Proof. (i) We argue by induction on the rounds. The slots of a round are the nodes removed from Q at line 16, and each of them is processed exactly once, by the worker $j \bmod p$. PROCESS prunes by z_0, a past value of z, which is safe by Lemma 13.12, and the value found by a slot is offered to z at line 11, before the end test, so that a node with no child is covered as in Lemma 13.4. After the barrier each slot has been processed, and line 13 puts each child in Q or makes it a leaf by the current z; hence, at line 14, the open nodes are those of Q, and the claim follows from the proof of Lemma 13.4 applied to each slot. (ii) At line 14 no node is held by a worker and none is in a slot, since the end is decided by one thread while the others wait at the barrier; hence `done` is set only when Q is empty or no node of Q can improve z, which are the situations of Theorem 13.13. (iii) A round processes at least one node, since $r \geq 1$ at line 16 (with a node limit, NEXTROUND stops at line 14 as soon as the evaluations reach it), and each node is processed once in a finite tree; a round ends when its at most $\lceil r/p \rceil$ processings per worker end. □

13.9 One pool per worker

The distributed queues

In the distributed queues (`DistributedBranchAndBound`, `ParallelBranchAndBound.h:638`) each worker has a queue Q_i and sends each child, in batches, to the worker that owns it, as HDA* does with the states of A* (Chapter 9). Its owner is $\text{hash}(m) \bmod p$ if the problem gives a hash of its nodes, and a random worker otherwise. Since a tree has no duplicates, the owner only balances the load, and a uniform hash is perhaps the most suitable distribution; the distributions of HDA* that keep related states on the same worker (abstraction, abstract Zobrist hashing) have little to gain here. In the Doxygen comment of `ParallelBranchAndBound.h` the pseudocode follows that of the shared queue, as its lines 28 to 45; here its lines are numbered from 1.

Algorithm 13.11 (The distributed queues).

```
 1  procedure DISTRIBUTED(R)  ▷ a queue Q_i per worker, nodes to the owners
 2     put each node of R in the queue of its owner
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        d ← nil
 5        while the search is not over do
 6           push the nodes received in Q_i
 7           if d ≠ nil then n ← d;  d ← nil
 8           else if Q_i ≠ ∅ and not PRUNABLE(the least bound of Q_i) then
 9              n ← the first node of Q_i, removed
10           else
11              the nodes of Q_i are leaves (pruned);  Q_i ← ∅
12              send all the outboxes;  wait for mail or the end; continue
13           if a limit is reached then n is a leaf (open); stop all; return
14           K ← PROCESS(n);  d ← DIVE(K)
15           for each m ∈ K but d do
16              j ← hash(m) mod p, or a random worker without hash(m)
17              if j = i then push m in Q_i else send m to j
18           every k evaluations, send the outboxes not yet full
```

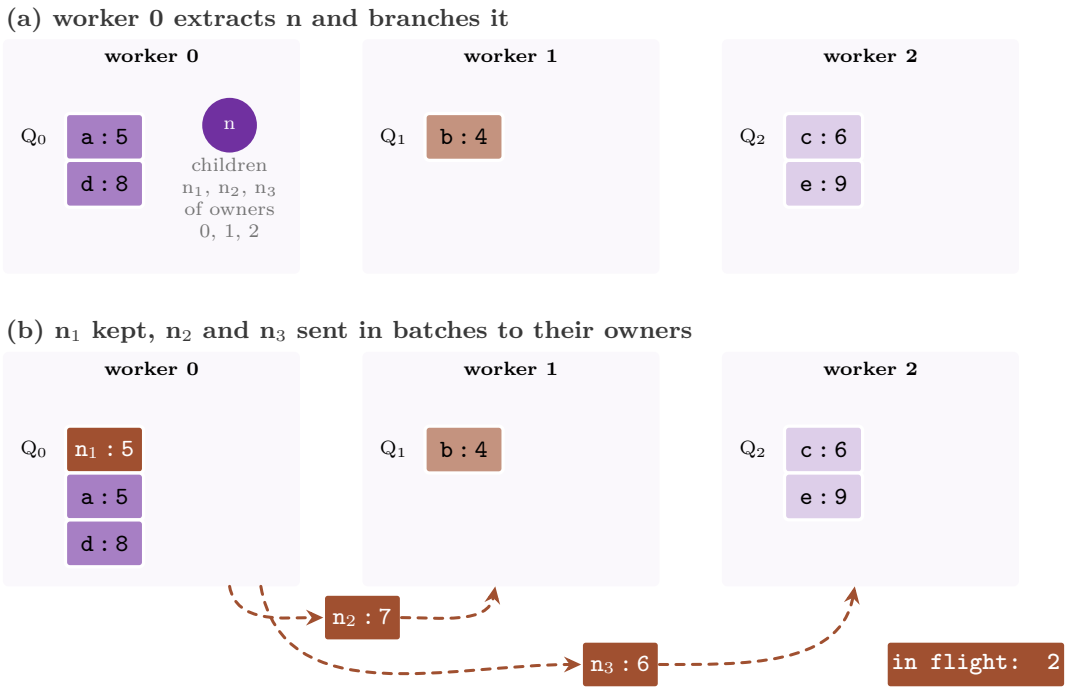


Figure 13.10. The distributed queues with three workers, the color of a node being that of its owner. (a) Worker 0 extracts n and branches it; its children are owned by the workers 0, 1 and 2. (b) Worker 0 keeps n_1 in its queue (first, since the ties go to the last node pushed) and sends n_2 and n_3, which are in flight and counted as such until their owners receive them.

Proposition 13.20 (The end of the distributed queues). Algorithm 13.11 ends only when no worker holds a node and no node is in flight, with each queue empty (a queue whose least bound cannot improve the incumbent having been emptied into leaves at line 11); if the evaluations terminate, it ends.

Proof. As we said, the wider open set is made of the queues, the dives and the nodes in the outboxes and in flight. A worker reaches the test of the end (line 12) only with no dive and an empty queue, after sending all its outboxes. The test is that of the mailbox (`quiescent()`), the deficit counting of Definition 9.6, whose hypotheses hold here: a worker sends only while it has work (at line 17, or when the mailbox flushes its outboxes before it becomes passive) and becomes active again only by draining a message (line 6). By Lemma 9.7 the mailbox reports the end only when no worker is active and no message exists, i.e., when each worker is at line 12 with no dive and an empty queue and each node sent has been received; at that moment the wider open set is empty. Conversely, while a node exists in the wider set, some worker holds it or will receive it, and processes it

in finite time; since each node is processed once and the tree is finite, the end comes, and by part (iii) of the lemma all the workers then read it. After a stop at a limit the mailbox reports the end to all the workers at once, and the nodes left in the queues, in the dives, in the outboxes and in the channels are collected as open leaves once the workers have returned (`leftovers()`), so that the bound covers them. □

The batches are of 16 nodes by default, and the outboxes are flushed at each evaluation (`flush_period` = 1), since an evaluation is usually "expensive" and a node that waits in an outbox delays its owner. The nodes a search starts from go to their owners, or to the worker i mod p for the i-th of them when the problem gives no hash. Without a hash, the owners come from a generator per worker, seeded by `seed` plus the index of the worker. Note that, since a worker extracts the best node of its queue only, it may evaluate nodes "far" worse than the best open one before an incumbent prunes them, and how many depends on when the incumbent is found, i.e., on the timing. Plunging, whose dives reach a leaf early, tends to keep them close to those of the sequential search. The choice of the owner is lines 15 to 17 of Algorithm 13.11, the random owner being drawn by a generator of 64 bits of each worker.

The distributed queues over MPI

Over MPI (`MPIBranchAndBoundSearch`, `MPIBranchAndBound.h:168`) the same scheme runs with one process per worker, and the nodes travel as bytes, written by `serialize()` and read by `deserialize()` (a trivially copyable node needs neither). Each process has its queue and a copy of the incumbent value. A process that improves it sends the value to all the others, while the node of the incumbent stays with the process that found it until the end, when the process of the least value sends it to all.

Algorithm 13.12 (The distributed queues over MPI).

```
 1  procedure MPI-B&B(R_i) on the process i       ▷ R_i: its own nodes
 2     Q_i ← R_i;  d ← nil
 3     loop
 4        for each message m received do
 5           if m is a batch of nodes then push them in Q_i
 6           else if m is an incumbent z' then z ← min(z, z')
 7           else if m is a stop then stop evaluating
 8        if not stopped and (d ≠ nil or not PRUNABLE(βmin(Q_i))) then
 9           n ← d, or the first node of Q_i, removed;  d ← nil
10           K ← PROCESS(n), a value v < z sending v to all the others
11           d ← DIVE(K)
12           for each m ∈ K but d do
13              j ← hash(m) mod p, or a random process without hash(m)
14              if j = i then push m in Q_i else send m to j
15           every k evaluations, send the outboxes not yet full
16           if the share of the node limit or the time limit is reached
                then stop all the processes
17        else send all the outboxes
18        if the mailbox detects the end then break
19     the nodes still travelling, after a stop, are put in Q_i
20     the process of the least z sends its node to all
```

Proposition 13.21 (Correctness and end over MPI). The value and the bound returned by Algorithm 13.12 satisfy Theorems 13.5 and 13.6 when no limit stops it, and the bound satisfies Theorem 13.6 in any case; the search ends when no process has a node that can improve its incumbent and no message is in flight.

Proof. We first note that the value z of a process is at any moment at least the least value found by all, which plays the part of the shared incumbent of Theorem 13.13. Indeed, it is the initial value or a value received, each value sent is feasible, and a process that prunes by a (possibly larger) value prunes less, as in Lemma 13.12. The end is detected by the four-counter method of Mattern [11] in the mailbox of MPI (proved sound in Theorem 9.18), which a process joins (line 18) only when it has no node to evaluate and has sent all its outboxes. The method counts each message, the incumbents included, and reports the end only when all the processes are passive and each message sent has been received. At that moment each process has turned its queue into leaves by its incumbent (line 8), which is at least the least value found, and no node travels; the wider open set is then empty, and Theorem 13.13 applies. After a stop at a limit, the nodes still in flight are received as open leaves (line 19), and the bound covers them. □

The node limit L is split among the processes, each stopping at $\lceil L/p \rceil$ evaluations, and a process that reaches its

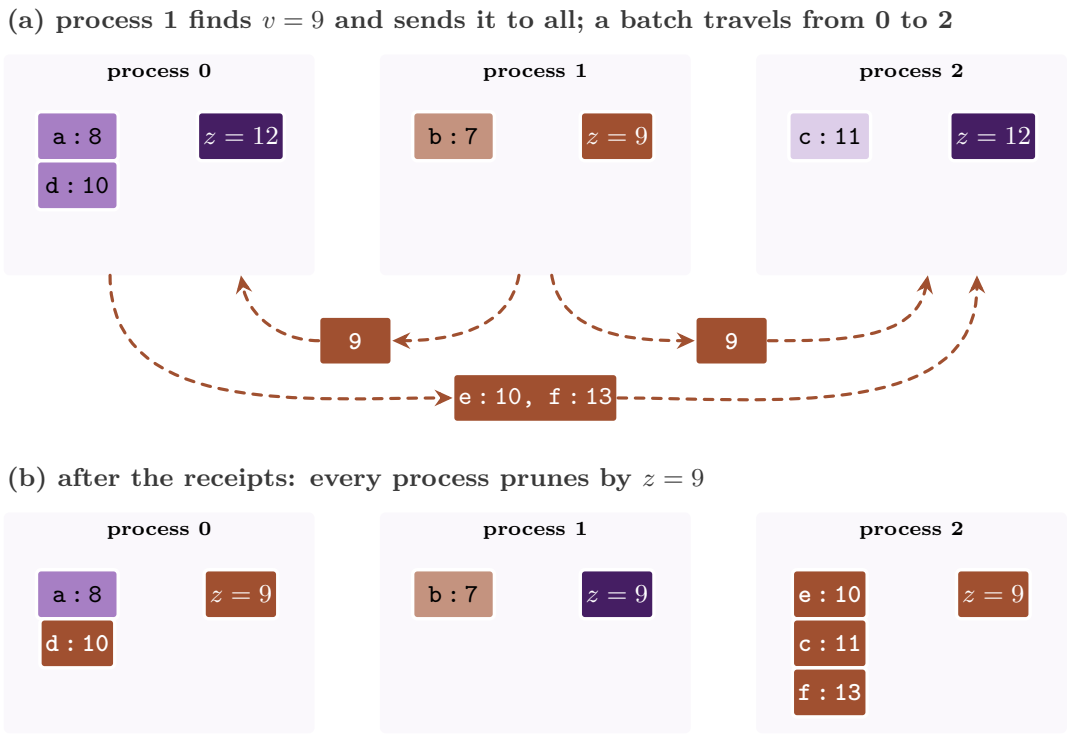


Figure 13.11. The distributed queues over MPI with three processes (minimization, the color of a node being that of its owner). (a) Process 1 finds the value 9 and sends it to the other two, while a batch of nodes travels from process 0 to process 2; the processes 0 and 2 still prune by $z = 12$, which is safe. (b) After the receipts each process prunes by $z = 9$, and the nodes whose bound is not less than 9 are pruned when they reach the top of their queue, or with their whole queue.

share, or the time limit, stops the evaluations of all. Each process returns the same value, node and bound (an all-gather of the values picks the process of the least value, the least rank on ties, which broadcasts its node). Each keeps, however, the leaves it owns, and the reoptimization of §13.11 then starts each process from them, with no transfer. All sends are non-blocking. MPI must be initialized by the caller, with at least `MPI_THREAD_FUNNELED` if the problem uses threads. The broadcast of an improvement is line 10 of Algorithm 13.12: a value v below the local z becomes the local incumbent, with its node, and is sent, as its bytes, to each other process with a tag of its own.

The work stealing

In the work stealing (`StealingBranchAndBound`, `StealingBranchAndBound.h:117`), after Blumofe and Leiserson [20], each worker pushes the children it generates in its queue and extracts the best node of that queue. A worker whose queue is empty steals about half of the queue of another one, drawn at random, the half that leaves the victim its best node (cf. the steal-half queues of Hendler and Shavit [36]). A worker then typically stays in its part of the tree. This pays off when moving its evaluator to a distant node costs (e.g., when the evaluator applies and undoes the branchings along the path from the root), while the nodes of the distributed queues land on any worker.

Algorithm 13.13 (The work stealing).

```
 1  procedure STEALING(R)
 2     the k-th node of R in Q_(k mod p)
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        d ← nil
 5        loop
 6           if d ≠ nil then n ← d;  d ← nil
 7           else n ← NEXT(i);  if n = nil then return
 8           if a limit is reached then
 9              n is a leaf (open);  stop all the workers;  return
10           K ← PROCESS(n)
11           d ← DIVE(K);  push the nodes of K but d in Q_i
12  procedure NEXT(i)
13     if Q_i ≠ ∅ and PRUNABLE(the least bound of Q_i) then
14        the nodes of Q_i are leaves (pruned);  Q_i ← ∅
15     if Q_i ≠ ∅ then return the first node of Q_i, removed
16     loop
17        if stopped, or no worker holds a node or works on one, return nil
18        v ← a worker ≠ i at random
19        if Q_v holds two nodes at least then
20           move about half of them, but the first, to Q_i
21           if some were moved, return the first node of Q_i, removed
22        wait a little
```

The queues are the stealable priority queues of the default data structures of the library, whose owner extracts its nodes in the exact order of the selection and with no lock. Each queue keeps its nodes in two 4-ary heaps, H_0 and H_1, which no other thread reads. A push goes into the smaller of the two, and hence the nodes pushed one after the other (e.g., the children of a node) are dealt to the two heaps in turn; a pop takes the better of their two least nodes. A thief that finds no node to claim raises a flag of the queue of the victim. The owner reads it at its next operation, and moves the heap whose least node is not the least of the queue into the *offer* of the queue, an atomic pointer, which a thief claims by an exchange with null (Figure 13.12). Note that the owner keeps its least node. Until a thief claims the offer, its nodes still belong to the owner: a pop whose least local key is not less than the least key of the offer takes the offer back by the same exchange, and whoever of the two exchanges first gets it all. Hence, the common case of the owner, a push or a pop with no thief waiting, costs no read-modify-write and no fence. A steal costs the thief two exchanges at most, and the owner the store of a pointer (or the time to deal its nodes again to two heaps of equal size, when they differ in size by more than a factor of 4). The price is that a thief waits for the next operation of the victim, i.e., at most the processing of one node.

Proposition 13.22 (The end of the work stealing). Let C be the number of the workers that hold a node or work on one, plus the thieves that are trying to claim an offer. A worker leaves C when its pop finds its queue and its offer empty, and a thief enters C before the exchange that claims an offer and leaves it if the exchange fails. Then $C = 0$ only when no node is open, and the test at line 17, which returns nil when $C = 0$, is exact; if

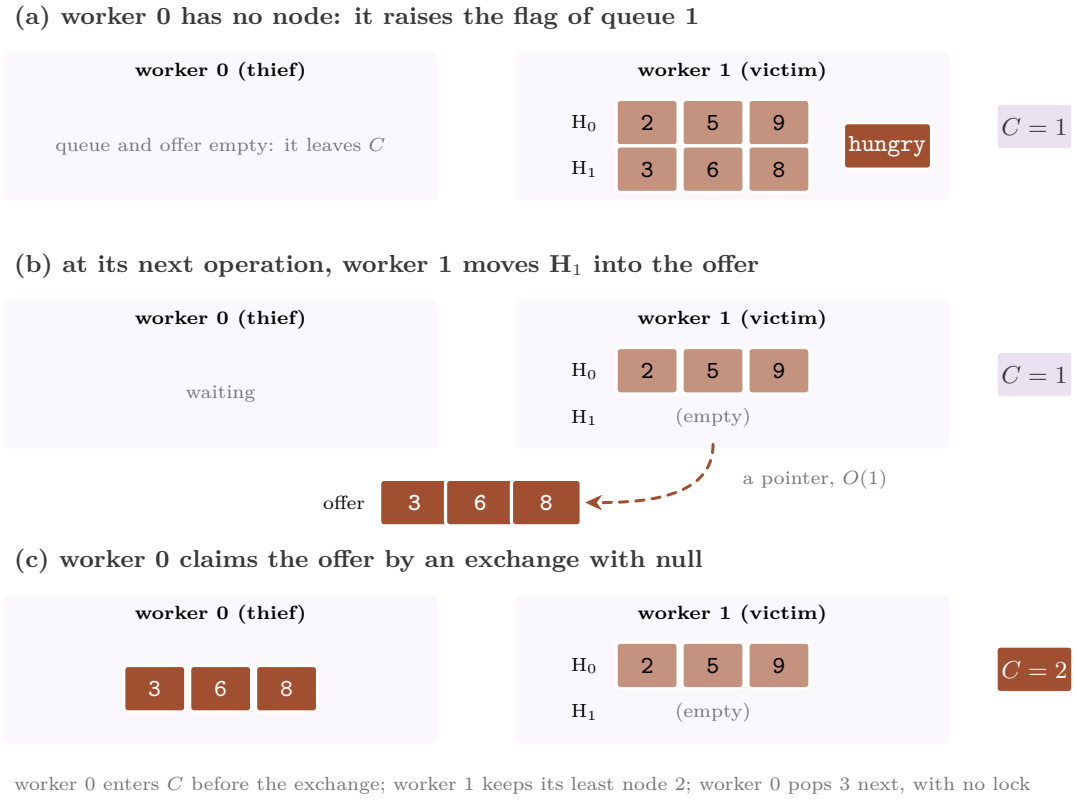


Figure 13.12. A steal, with the keys of the nodes in the two heaps of the victim. (a) Worker 0 has no node, leaves the counter C of the workers that hold or work on a node, and raises the flag of the queue of worker 1. (b) At its next operation worker 1 moves H_1, whose least key 3 is not the least of its queue, into the offer, by a pointer. (c) Worker 0 enters C and claims the offer by an exchange; worker 1 keeps its best node 2, and each worker pops its nodes with no lock.

the evaluations terminate, each worker returns.

Proof. We claim that a node is at any moment in a heap of its owner, in an offer (which belongs to the owner until claimed), in the hands of a thief between its exchange and its first pop, or held by a worker. In the first two cases the owner is counted in C, since it left C only after finding its heaps and its offer empty, and it re-enters only by a steal. In the third case the thief entered C before its exchange, and in the fourth the worker is counted. Hence a node exists only if $C > 0$. A worker loops at lines 16 to 22 only while $C > 0$, i.e., while some worker holds or works on a node. That worker processes it in finite time, and then pushes children (which a thief can claim after the next operation of the owner) or drops it; the tree is finite. □

The price of the scheme is a selection that is best-first within each queue only, and hence an order of extraction that may be farther from the sequential one than that of the shared queue. Also, the nodes have to be copyable with moves that do not throw, since a steal moves them to the thief, and the counter of the nodes sent in `BBStats` holds the steals. The extraction of a worker (lines 12 to 15 of Algorithm 13.13) drops its queue at once when its least bound, which counts the offer not yet claimed, cannot improve the incumbent.

The SelfSplit

The SelfSplit of Fischetti, Monaci and Salvagnin [19] (`SelfSplitBranchAndBound`, `SelfSplitBranchAndBound.h:130`) gives a B&B whose workers communicate only at the end. The tree is first explored sequentially (the *sampling*), by the evaluator of the worker 0 and without plunging, until the open nodes are σp (σ being `split`, 4 by default) or the tree is solved. Then, with `split_dive` (the default), the same evaluator dives from a copy of the first open node to a leaf, taking at each level the child of least estimate (of least bound, if the problem gives no estimate). Finally the k-th open node, in the order of the selection, goes to the worker k mod p. Each worker searches its subtrees sequentially (by `BranchAndBoundSearch`) with its evaluator, an incumbent starting from that of the sampling, and its share of the node limit and of the time left; the incumbents, the bounds and the leaves of the workers are merged in the order of the workers.

Algorithm 13.14 (The SelfSplit).

```
 1  procedure SELFSPLIT(R)
 2     Q ← R;  z ← the given incumbent, or ∞      ▷ the sampling, by worker 0
 3     while p > 1 and Q ≠ ∅ and |Q| < σ p do
 4        if a limit is reached then break
 5        if PRUNABLE(the least bound of Q) then
 6           the nodes of Q are leaves (pruned);  Q ← ∅;  break
 7        n ← the first node of Q, removed;  Q ← Q ∪ PROCESS(n)
 8     if p > 1, split_dive and Q ≠ ∅ then         ▷ the dive, by worker 0
 9        n ← a copy of the first node of Q
10        while n ≠ nil and no limit is reached do
11           n ← the child of PROCESS(n) of least estimate, or nil
12     for k = 0, 1, ..., while Q ≠ ∅ do           ▷ the split
13        add the first node of Q, removed, to R_(k mod p)
14     in parallel, on each worker i = 0, ..., p − 1 do
15        S_i ← B&B(R_i) on the evaluator of i, from the incumbent z, with
              its share of the node limit and the time left, pruning also
              by the incumbent of all the workers if split_shared
16     z ← min(z, the values of the S_i), the node of the least i on ties
17     return z, its node, the leaves of the sampling and of the S_i, and
          the least of their bounds
```

Theorem 13.23 (The SelfSplit). Under (A1) to (A4) and with $\tau_r \leq 1$, (i) when no limit stops it, the value returned by Algorithm 13.14 satisfies Theorem 13.5, and its bound satisfies Theorem 13.6 in any case. (ii) Without `split_shared` and without a time limit, its outcome depends only on the problem, R, the options, σ and p, provided that the evaluation of a node by a worker depends only on the node, the incumbent given by `cutoff` and the nodes evaluated by that worker before. (iii) If the evaluations terminate, the search ends, its end being the return of the last worker.

Proof. (i) We observe that the sampling is Algorithm 13.3 stopped early. Hence, by Lemma 13.4, each element of $X(R)$ below the threshold of its incumbent z_s is in an open node at its end, and its leaves bound the rest. As for the dive, it works on a copy and records no leaf; it may only lower z_s, by a value found. Then the split partitions the open nodes among the R_i, and each worker is a sequential B&B from R_i with the initial value z_s. Let $z = \min_i z_i$; the value $z_i \leq z_s$ of the worker i satisfies $\min\{v(x) : x \in X(R_i)\} \geq \theta(z_i) \geq \theta(z)$ by Theorem

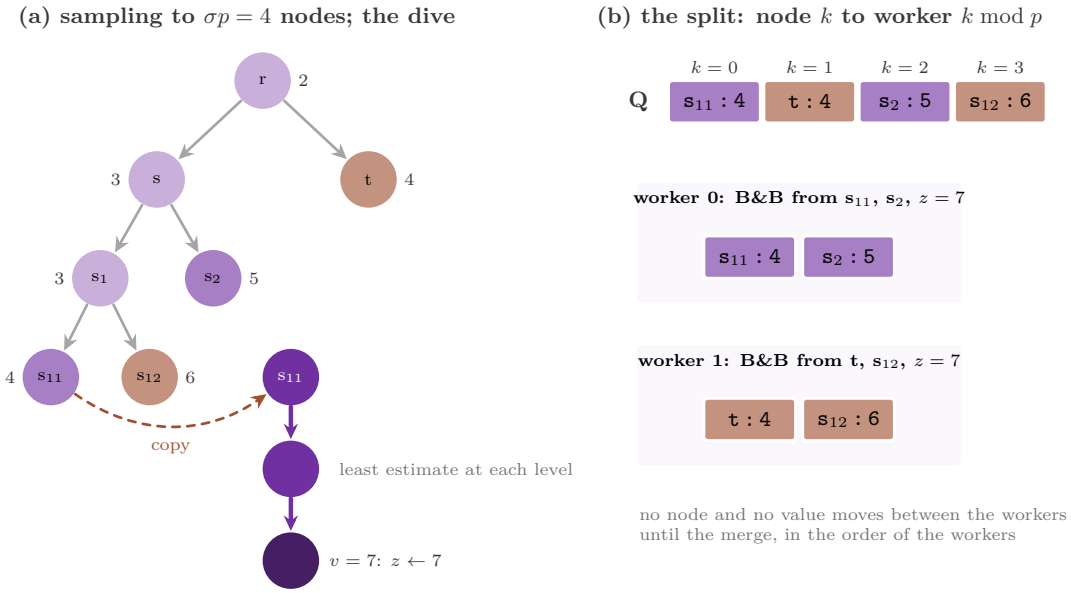


Figure 13.13. The SelfSplit with $p = 2$ and $\sigma = 2$ (minimization, the numbers being bounds). (a) The sampling by worker 0 evaluates r, s and s_1 and stops at four open nodes; the dive goes from a copy of the first of them, s_{11}, to a leaf of value 7. (b) The open nodes, in the order of the selection, go to the workers k mod 2 (the colors), and each worker runs a sequential B&B on its nodes from the incumbent $z = 7$, with no exchange until the merge.

13.5 and Lemma 13.3. With `split_shared` the worker i prunes by the least c_i of z_i and of the values found by the others, and the same argument gives the bound $\theta(c_i) \geq \theta(z)$, since $c_i \geq z$. The elements that the sampling left below $\theta(z_s)$ are in some $X(R_i)$, and those of its leaves are not below $\theta(z_s) \geq \theta(z)$; hence $z^*_R \geq \theta(z)$. The bound is the least of the bounds of the sampling and of the workers, each valid by Theorem 13.6 for the region it covers, and these regions cover $X(R)$. (ii) The sampling and the dive are sequential, and the split depends only on the order of the queue and on p. Worker i receives R_i, z_s and its share of the limit, all fixed, and runs with no exchange, with its evaluator; the merge goes in the order of the workers. (iii) The sampling stops when its queue holds σp nodes, or earlier, and processes each node once in a finite tree; the dive ends at a leaf by (A4); each worker runs Algorithm 13.3, which ends by Theorem 13.5; and the search waits for all the workers to return before the merge, so that no node is open anywhere when it merges. □

The proof also shows why the dive matters. A best-first sampling seldom reaches a leaf, and without an incumbent a worker whose subtrees do not hold the optimum evaluates each node of them whose bound is better than the best value in them, since no incumbent prunes them. Yet, the dive gives each worker the incumbent of one leaf, at the cost of the "few" nodes of the dive, which the worker whose subtree holds them evaluates again (both evaluations are counted). With `split_shared` the workers also prune by the best incumbent of all of them, which each reads at each extraction through `shared_incumbent`. Even so, the value is still the optimum, since the shared value is a value found (cf. Lemma 13.12), while the nodes evaluated, and the node of the optimum among equal values, depend on the timing. With one worker there is nothing to split: the root goes to it with neither the sampling nor the dive, and the nodes evaluated are those of the sequential search. The test `bb-split-thread` checks this, and that the outcome is the same at each repetition, with a split of 1 and 4 nodes per worker and with and without the dive.

In the library the sampling is done once, by the worker 0, rather than by each worker as in the original scheme, since the workers share the memory and can read its open nodes. Over MPI, where a node would otherwise travel as bytes, each process would repeat it, as Fischetti, Monaci and Salvagnin do. Its price is a load that is balanced only as far as the subtrees of the split are, and (without `split_shared`) an incumbent found by a worker that does not prune the nodes of the others. The split is lines 12 and 13 of Algorithm 13.14: with L' the evaluations left by the sampling, the worker i receives the share $\lfloor L'/p \rfloor + 1$ if $i < L' \bmod p$ and $\lfloor L'/p \rfloor$ otherwise, so that the shares sum to L', and a node whose worker has a share of 0 becomes an open leaf. The dive is lines 8 to 11, taking the first of the children of least estimate; its leaves go to a record of Algorithm 13.4 that keeps nothing, since the open nodes of the sampling still cover them.

13.10 The schemes compared and chosen by name

Table 13.1 gathers, for each scheme, where the open nodes are, how the end is detected and on what the outcome depends. Here "the timing" means that two runs with the same data may evaluate different nodes, and return a different node among those of equal value. The value is in any case within the tolerance of the optimum in each run that no limit stops.

Table 13.1. The B&B searches of the library: where the open nodes are, how the end is detected (the condition of the proposition of each scheme), and on what the outcome depends besides the problem and the options; for the rounds, p matters only with an evaluator that depends on the history of its worker (Theorem 13.18), and the SelfSplit depends on the timing with `split_shared`.

Scheme	Open nodes	End	Depends on
sequential	one queue	Q empty, or line 6	nothing
shared	one queue, a lock	Q empty, busy $= 0$	the timing
multiqueue	$2p$ queues, a lock each	$m = 0$	the timing
farm	the queue of the master	out $= 0$	the timing
deterministic	one queue, by rounds	between rounds	r (and p)
distributed	a queue per worker	message counts	the timing
MPI	a queue per process	Mattern's counters	the timing
stealing	a queue per worker	$C = 0$	the timing
selfsplit	a queue per worker	at each worker	σ, p
skeletons	stacks, a pool per worker	$\pi = 0$	the timing
ordered	stacks, one list of tasks	the list is taken	the timing
hubs (MPI)	a queue per process	Mattern's counters	the timing
intervals	an interval per worker	`active` $= 0$	the timing

We expect the choice among the schemes to depend on the cost of an evaluation w.r.t. that of moving a node, and on the variance of that cost among the nodes, which only a measurement on the instances of interest can settle. The shared queue and the farm keep the order of the selection, at the price of a lock or of a master, and the MultiQueue removes the lock at the price of the order. The distributed queues and the work stealing remove the common pool, the latter keeping each worker in its part of the tree. Finally, the rounds and the SelfSplit perhaps give up some speed for an outcome that does not depend on the timing. All the schemes but the farm and the search over MPI, which build their workers and their channels themselves, can be chosen at run time by a value of the enumeration `BBScheme`, or by its name, read, e.g., from a file of parameters. The enumeration is

```
enum class BBScheme {
 sequential ,      ///< one worker [see BranchAndBoundSearch]
 shared ,          ///< one shared queue [see SharedBranchAndBound]
 distributed ,     ///< a queue per worker [see DistributedBranchAndBound]
 deterministic ,   ///< synchronous rounds [see DeterministicBranchAndBound]
 selfsplit ,       ///< a split of a sampling [see SelfSplitBranchAndBound]
 stealing ,        ///< work stealing [see StealingBranchAndBound]
 multiqueue ,      ///< a MultiQueue [see MultiQueueBranchAndBound]
 depth_bounded ,   ///< depth-first, spawning above a depth [see BBRule]
 budget ,          ///< depth-first, spawning after a budget [see BBRule]
 stack_stealing ,  ///< depth-first, stealing from the stacks [see BBRule]
 ordered           ///< depth-first, ordered tasks [see BBRule]
 };
```

and the functions of `BranchAndBoundScheme.h` are declared as follows (without their bodies; the template head of `parallel_branch_and_bound()` has the problem type `P` and, as its second parameter, the executor):

```
[[nodiscard]] inline std::string_view bb_scheme_name( BBScheme s );
[[nodiscard]] inline BBScheme bb_scheme( std::string_view name );
[[nodiscard]] BBSolution< typename P::node_type , bb_value_t< P > >
parallel_branch_and_bound( const P & problem , BBScheme s ,
                           const BasicParallelBBOptions< bb_value_t< P > > &
                             o = {} ,
```

```
                        std::optional< std::vector< BBLeaf<
                          typename P::node_type , bb_value_t< P > > > >
                          seeds = std::nullopt );
```

Thus, a program built on the library gets each new scheme with no change of its code. The following example, from the case `manual` of the tester, which the build compiles and runs (`ctest -R PBeFS.manual`), solves a knapsack `kp` of 50 items by the sequential search `s`, with the options `o`, and by the scheme `sch` read from its name, with the parallel options `po` and 4 workers, and shows that an unknown name is refused; the case then checks both solutions against the dynamic program.

```
// a random 0-1 knapsack of 50 items, of class 0
const Knapsack kp = Knapsack::random( 0 , 50 , 1000 , 1 );
BBOptions o;
o.selection = Selection::best_first;     // the least bound first
const auto s = branch_and_bound( kp , o );
// s.status, s.value and the incumbent node s.node
ParallelBBOptions po;
po.threads = 4;
const BBScheme sch = bb_scheme( "stealing" );   // a name, e.g., from a file
const auto r = parallel_branch_and_bound( kp , sch , po );
bool refused = false;
try {
 ( void ) bb_scheme( "nonesuch" );
 }
catch( const std::invalid_argument & ) {     // not the name of a scheme
 refused = true;
 }
```

The names are `sequential`, `shared`, `distributed`, `deterministic`, `selfsplit`, `stealing`, `multiqueue`, and the four rules of the skeletons of §13.14, `depthbounded`, `budget`, `stackstealing` and `ordered` (an unknown name throws `std::invalid_argument`, and so does `multiqueue` with values that are not atomic with no lock). When the options list the settings of a race (§13.15), `parallel_branch_and_bound()` runs the race first and goes on by the scheme of the call from the leaves of the winner. The farm, the search over MPI, the hubs of §13.17 and the intervals of §13.18 are not among the named schemes, and they are used by their classes, `FarmBranchAndBound< P , Executor >( problem , options )` and `MPIBranchAndBoundSearch< P >( problem , options , comm )`, `MPIHubBranchAndBoundSearch< P >( problem , options , hub , comm )` and `IntervalBranchAndBound< P , Executor >( problem , options )`, whose `solve()` returns the solution. The executor, i.e., the pool of C++ threads, OpenMP or FastFlow, is the second template parameter of each parallel search on threads. Also, `set_executor()` gives a search an executor that outlives it (without one, each search builds a pool).

13.11 Reoptimization

When the data of the problem change in a way that leaves the branching valid (e.g., a restriction of the feasible region, or a change of the objective that the nodes do not store), the tree explored by a previous search need not be built again. Its leaves cover the subtrees of the nodes that the search started from, and a new search can start from them. With `keep_frontier`, the `BBSolution` holds these leaves, each with its bound (in the sense of the problem), its depth and its kind. The overload `solve( seeds )` of each search (and `branch_and_bound( p , seeds , options )`) starts from any set of such nodes instead of the root, each of them being an open node that is evaluated only when it is extracted. Figure 13.14 shows the simplest use: a search stopped by a node limit is resumed from its open leaves, together with its incumbent.

Proposition 13.24 (Reoptimization). Let L be the leaves of a search from R with `keep_frontier`. (i) Every element of $X(R)$ belongs to $X(l)$ for some $l \in L$. (ii) If the problem changes to X' and v', with (A2) to (A4) holding for it, with the children generated by the previous search still covering the new subset of their parent (A1), with the subsets of the nodes $X'(n) \subseteq X(n)$, and with $v' = v$, then a search from the non-infeasible leaves of L, with their bounds, solves the new problem over $X'(R)$ in the sense of Theorems 13.5 and 13.6. (iii) If instead $X'(n) \supseteq X(n)$, or $v' \neq v$, the same holds when each leaf, the infeasible ones included, is given back with an unknown bound ($-\infty$ in the minimization), or with a bound that the caller knows to be valid for the new problem.

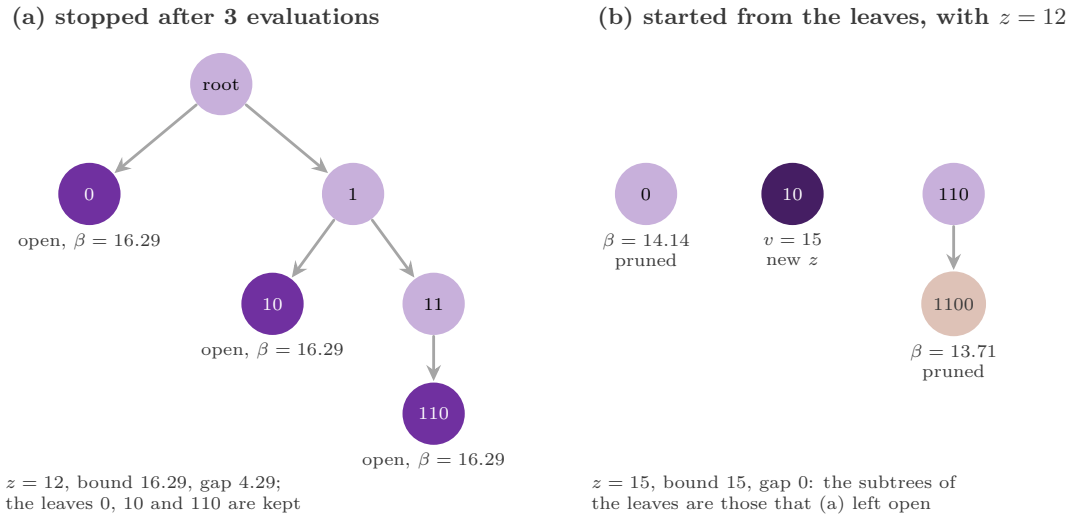


Figure 13.14. Resuming a search from its leaves. (a) The best-first search of the example stopped by a node limit of 3: the open leaves 0, 10 and 110, with their inherited bound 16.29, give the bound 16.29 and the gap 4.29. (b) A search started from these leaves with the incumbent 12 evaluates 110, 10, where it finds 15, and 0, then prunes the child 1100 of 110 with the open nodes, and ends with the gap 0, as the search of Figure 13.1 did.

Proof. Part (i) is the covering shown in the proof of Theorem 13.6. (ii) The bound of a leaf was a bound of $X(l)$ for v, and hence of $X'(l) \subseteq X(l)$; an infeasible leaf has $X'(l) \subseteq X(l) = \emptyset$. Hence, the seeds cover $X'(R)$ with valid bounds, and the proofs of Lemma 13.4 and of the theorems apply with the seeds in place of R; the covering of $X'(R)$ by the seeds follows, as in (i), from (A1) applied to the nodes that the previous search branched. (iii) An unknown bound is valid for any problem, and the seeds cover $X'(R)$ by the same argument, since (A1) for the new subsets is again assumed of the children that the previous search generated (all the leaves, the infeasible ones included, being given back). □

Which leaves are given back, and with which bounds, is left to the caller, since only the caller knows what has changed. After a restriction of the region the bounds of the pruned leaves remain valid, and the infeasible leaves need not be explored again. After an enlargement the infeasible leaves have to be given back with an unknown bound. After a change of the objective all of them have to be given back with an unknown bound, unless the old ones can be corrected (say, by a bound on the change of the objective over the node). A search stopped by a limit is resumed by giving back its open leaves together with its incumbent, as in Figure 13.14. The bound of the new `BBSolution` is that of the subtrees of the seeds, and the caller combines it with the bounds of the leaves that it leaves out. The test `bb` and the parallel tests check three uses on the knapsack. First, the leaves of a search carried out to the end bound the optimum and solve the knapsack of three quarters of the capacity. Second, with `explicit_children`, the leaves of the smaller knapsack, the infeasible ones among them, solve the original one again (without `explicit_children` the child that takes an item that did not fit is never generated, and (A1) fails for the larger capacity). Third, the open leaves of a search stopped by a node limit end it.

In the parallel searches each worker collects its leaves, which are merged at the end, and the seeds go to their owners as the children do. Over MPI each process keeps the leaves that it owns and starts from its seeds, which avoids sending them across the processes. We remark that in a tree where every inner node has two children the leaves are one more than the inner nodes, and therefore keeping them may take much more memory than the open nodes; hence, `keep_frontier` is off by default.

13.12 The interface for a branch-and-bound solver

A B&B solver built on a modeling system (e.g., for Mixed-Integer Linear Programs) would provide a problem whose `node_type` is the list of the branching decisions from the root, e.g., the bounds changed on the variables, which is "small" and hence "cheap" to copy and to send. Its `make_worker( i )` would return an object holding a copy of the model and of the solver of the relaxation. The `evaluate` of a worker applies the decisions of the node to its model (possibly undoing only those that differ from the previous node, which makes a dive "cheap") and solves the relaxation. It returns the value of the relaxation as the bound, its infeasibility, and the same value as a feasible one if the solution is integral. Its `branch` chooses a fractional variable in the solution stored in the

node, and gives the two children with the bound of the parent as their estimate. Its `cutoff( z )` should keep the incumbent value (and pass it to the solver of the relaxation as an objective limit), and its `evaluate` should return as feasible only the values found at the node, keeping the solution in the node; then the outcome is that of the scheme it runs, whichever it is. In particular, the deterministic rounds and the SelfSplit then give a solver whose runs can be repeated exactly, under the conditions of Theorem 13.18 and Theorem 13.23; a warm start from the previous node of the worker satisfies them only in their weaker form, in which the outcome depends on p. Such a solver calls `parallel_branch_and_bound()` with the name of a scheme read from its parameters, as in §13.10, and reads the incumbent, its node, the bound and the gap from the `BBSolution`. In the options it gives the selection, the tolerances, the initial incumbent and the callback, and for the parallel searches the number of workers, the batch, the nodes of a round and those of the split. The test `bb-cutoff` checks this interface with workers that fence the nodes by the incumbent they receive.

13.13 Lazy children, indexed children and the modes

The sections from here to the end of the chapter treat the parts of the layer that extend the core of the previous sections: the children given one at a time and the modes of the search (this section), the depth-first skeletons (§13.14), the racing ramp-up (§13.15), the checkpoints (§13.16), the master, the hubs and the workers over MPI (§13.17), the work stealing of intervals of leaves (§13.18) and the example problems that exercise them (§13.19); their benchmark, notes and exercises are in §13.20, §13.21 and §13.22 with those of the core.

The children one at a time

The method `branch( n , sink )` of §13.1 gives all the children of a node at once, which suits the searches that keep a queue, since each child that survives the test of line 26 of Algorithm 13.3 is pushed anyway. A depth-first search, instead, visits the children of a node one at a time and often abandons the rest of them, e.g., when the children come by increasing estimate and the first one pruned makes the others prunable too; building the children it never visits is then wasted work, which in some problems (the cliques of §13.19 among them) is most of the work of a branching. Hence, the evaluator may give the method `children( n )` in place of, or besides, `branch( n , sink )`: it returns a *cursor*, a model of the concept `BBCursor`, whose `next()` gives the next child of the evaluated node `n` with its estimate, as a `BBChild`, or nothing when there is no more. The child and the cursor are the following (`BranchAndBound.h:208` and `BranchAndBound.h:332`):

```
template< class Node , class V = double >
struct BBChild {
 Node node;          ///< the child
 V estimate;         ///< a bound of it that is cheap to compute
 };
```

```
template< class C , class N , class V = double >
concept BBCursor = requires( C & c ) {
 { c.next() } -> std::convertible_to< std::optional< BBChild< N , V > > >;
 };
```

The concept `BBEvaluator` of §13.1 accepts an evaluator that gives either method, and two refinements of it name the evaluators that give the optional ones (`BranchAndBound.h:383` and `BranchAndBound.h:392`):

```
template< class W , class N , class V = double >
concept LazyBBEvaluator = BBEvaluator< W , N , V > &&
 requires( W & w , const N & cn ) {
  { w.children( cn ) } -> BBCursor< N , V >;
  };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// an evaluator that gives the child k of a node [see BBEvaluator]

template< class W , class N , class V = double >
concept IndexedBBEvaluator = BBEvaluator< W , N , V > &&
 requires( W & w , const N & cn , std::size_t k ) {
  { w.child( cn , k ) }
```

```
  -> std::convertible_to< std::optional< BBChild< N , V > > >;
 };
```

A `LazyBBEvaluator` gives `children( n )`, and an `IndexedBBEvaluator` gives `child( n , k )`, the child k (counted from 0) of `n` built without the evaluation of `n`, or nothing if `n` has fewer children. The latter is used by the searches that rebuild a node from the path of the choices that lead to it (§13.14) or from its interval of leaves (§13.18); it requires that the children of a node come in an order that depends only on the node, and that `child( n , k )` give the k-th of them in that order. When the problem has no `make_worker( i )`, the problem itself is the evaluator of each worker, and it gives each of these methods when it declares it.

Every problem runs on every search, whichever methods its evaluator gives. The searches that take all the children at once call `branch( n , sink )` when the evaluator has it, and otherwise read the cursor of `childre n( n )` to its end (`detail::bb_branch()`); conversely, the skeletons of §13.14, which take the children one at a time, use the cursor of `children( n )` when the evaluator gives it, and otherwise a cursor on a vector filled by `branch( n , sink )` (`detail::bb_children()`). Only the skeletons draw a benefit from the lazy children, since they are the only searches that stop reading a cursor before its end; in the other searches a lazy evaluator builds all the children as an eager one does. The cursor of a frame of a skeleton may refer to its node, which does not move while the frame exists, as the cursor of `MaxClique` does.

The option `prune_level` of `BasicBBOptions` states that the children of every node come by their estimates, the best first, i.e., by nondecreasing estimate in the minimization. Under this assumption, a child pruned at its generation (line 26 of Algorithm 13.3) ends its siblings: the search marks the rest of them as pruned without testing them, and a skeleton never asks its cursor for them, so that they are never built (unless `keep_frontier` asks for them as leaves, in which case they are built, counted as generated and pruned, and recorded).

Proposition 13.25 (The children pruned by level). Let the estimates $e_1, e_2, \ldots$ of the children of each node be nondecreasing in the order of the children, in the minimization, and let $\tau_r \leq 1$. If the child m_j is pruned at its generation, then every later child m_k, $k > j$, would be pruned at its generation too, in each of the three modes defined below. Hence, with `prune_level` every search of the library keeps the guarantees of its scheme, a sequential search evaluates the same nodes and returns the same value and bound as without it, and the counter `generated` of a skeleton is lowered by the children it does not build.

Proof. Let z_j be the incumbent at the generation of m_j and z_k that at the generation of m_k; since the incumbent never increases (Lemma 13.12), $z_k \leq z_j$. The bound of m_k is $\beta(m_k) = \max\{\beta(n), e_k\} \geq \max\{\beta(n), e_j\} = \beta(m_j)$. In the mode `optimize`, $\beta(m_j) \geq \theta(z_j) \geq \theta(z_k)$ by Lemma 13.3, and therefore $\beta(m_k) \geq \theta(z_k)$. In the mode `enum erate`, $\beta(m_j) = +\infty$, and then so is $\beta(m_k)$. In the mode `decide`, either the target was met by z_j, and then it is met by $z_k \leq z_j$, or $\beta(m_j)$ exceeds the target or is infinite, and then so does $\beta(m_k)$. In each case the test PRUNABLE of the mode is true for $\beta(m_k)$. A child pruned at its generation lowers only the bound `low` of the record of the leaves (Algorithm 13.4) by its bound, which is at least the bound of the child that started the cut; hence, `low` is unchanged as well. □

The library does not check the assumption, and a problem whose children do not come by their estimates gets a search that may miss its optimum with `prune_level`. In the work stealing of intervals (§13.18) each child is built by `child( n , k )` on its own, and the option has no effect there.

The modes

The *mode* of the options (`BBMode`, `BranchAndBound.h:183`) says what the search looks for:

```
enum class BBMode {
 optimize ,     ///< an optimal value
 decide ,       ///< a value within the target, stopping at the first one
 enumerate      ///< every node, counting the feasible ones
 };
```

In the mode `optimize`, the default, the search seeks an optimal value, as in the previous sections. In the mode `d ecide` it seeks a feasible value at least as good as the `target` of the options (any feasible value if no target is given), and it stops as soon as one is found; this is the decision problem of, e.g., the subgraph isomorphism of §13.19. In the mode `enumerate` it visits the whole tree, pruning only the infeasible nodes, and the counter `solu tions` of `BBStats` holds the number of the feasible values found, i.e., of the feasible nodes of the tree, which is how the nodes of a tree of UTS or the embeddings of a pattern graph are counted. (The counter `solutions`

is raised at each evaluation that gives a feasible value, in every mode.) Both modes are implemented only in the test PRUNABLE (`detail::bb_prunable()`), which in the minimization, with t the target turned to the minimization ($t = +\infty$ when none is given) and z the incumbent, reads

$$\text{PRUNABLE}(\beta) = \begin{cases} \beta \geq \theta(z) \text{ (or } \beta = +\infty \text{ if } z = +\infty) & \text{in the mode optimize,} \\ \beta = +\infty & \text{in the mode enumerate,} \\ z \leq t \text{ or } \beta = +\infty \text{ or } \beta > t & \text{in the mode decide,} \end{cases}$$

where $z \leq t$ with z finite is the test `detail::bb_met()` that the target is met. In the mode `decide` every node is prunable once the target is met, and therefore every scheme stops there without a signal of its own: the termination test of line 6 of Algorithm 13.3 (or its version in each scheme) finds the least open bound prunable, and the queue is dropped as when no node can improve the incumbent. The status of a search that ends with no limit is then `optimal` if the target is met and `infeasible` otherwise (`detail::bb_status()`), the value being that of the incumbent. Note that an incumbent given in the options counts: with no target, a given incumbent meets the target at once, and the search ends before its first evaluation with that value and no node.

Proposition 13.26 (The modes decide and enumerate). Under (A1) to (A4), let a search of the library run with no limit. (i) In the mode `decide`, its status is `optimal` if and only if it found (or was given) a feasible value $z \leq t$, and if its status is `infeasible` then no $x \in X(R)$ has $v(x) \leq t$. (ii) In the mode `enumerate`, it evaluates exactly once each node of the tree $T(R)$ of the nodes reached from R by branching every node whose evaluation finds it neither infeasible nor of bound $+\infty$, the nodes whose inherited bound is $+\infty$ being left out; hence, `solutions` is the number of the nodes of $T(R)$ that give a feasible value, and the incumbent is the least of their values.

Proof. (i) The first claim is the definition of the status. For the second, we repeat the proof of Lemma 13.4 (in its parallel form, Theorem 13.13) with the set of the $x \in X(R)$ with $v(x) \leq t$ in place of those below the threshold, as long as the target is not met. A node pruned while the target is not met has $\beta = +\infty$ or $\beta > t$, and by (A2) holds no x with $v(x) \leq t$; an infeasible node holds none; a node with no child is solved by (A3), and its value, if not above t, would have met the target. Hence, while the target is not met, every such x lies in the subset of an open node. A search that ends with no limit and the target not met has no open node (the termination test, with the target not met, fails on any finite bound not above t), and therefore no such x exists. (ii) In the mode `enumerate` a node is pruned only if its bound is $+\infty$, and the termination test of line 6 succeeds only when every open bound is $+\infty$; such nodes have empty subsets by (A2), and they are not in $T(R)$. Every node of $T(R)$ is therefore processed, by (A4) the tree is finite, and each node is processed once, since every scheme moves a node between the places of the wider open set without copying it (Theorem 13.13). The evaluations made only to rebuild a node (§13.14, §13.18) are made outside PROCESS and are not counted among `solutions`. The incumbent is offered each feasible value at line 21, and it prunes nothing in this mode. □

Two schemes treat the modes apart. In the SelfSplit (§13.9) the workers share their incumbent in the mode `decide` even without `split_shared`, so that a worker that meets the target stops the others, and the dive that ends the sampling is skipped in the mode `enumerate`, since its nodes would be counted twice; the racing ramp-up of §13.15 is skipped in the mode `enumerate`, where there is nothing to race. Finally, the searches call `cutoff( z )` of an evaluator with $+\infty$ in the modes `decide` and `enumerate` (`detail::bb_cutoff()`): an evaluator that fenced its node by the incumbent would report as infeasible the nodes of the tree that do not improve it, which in the mode `enumerate` would lose them from the count, while in the mode `decide` the search applies the target itself. A richer enumeration than the count (e.g., the number of the nodes of each depth, or any other monoid) is left to the evaluator: the evaluators given by `make_worker( i )` may accumulate into storage of the problem indexed by i, which the caller combines at the end. The test `bb-modes` checks both modes with each of the eleven named schemes and with 1 and 3 workers, on small knapsacks: `decide` has to find a value within a target 50 below the optimum, none beyond it, and some value without a target, and `enumerate` has to count the feasible nodes of the tree as a recursion on `evaluate` and `branch` does.

13.14 The search skeletons

The parallel searches of §13.8 and §13.9 share a queue, or keep one per worker, and order it by the selection. The search skeletons of Archibald, Maier, Stewart and Trinder [67] [68] are depth-first instead: a worker searches a *task*, i.e., a node and its subtree, with a stack of frames, each holding an evaluated node and the cursor of its children, and the work is shared by one of four rules, the enumeration `BBRule` (`SkeletonBranchAndBound.h:133`):

```
enum class BBRule {
 depth_bounded ,   ///< the children of the nodes above a depth are tasks
 budget ,          ///< after a budget of backtracks, the rest are tasks
 stack_stealing ,  ///< an idle worker takes children from a busy one
 ordered           ///< the nodes of a depth, by a fixed priority
 };
```

The class `SkeletonBranchAndBound< P , Executor >` (SkeletonBranchAndBound.h:163) implements the four of them on the executors of the library, each worker keeping the tasks it makes in a pool by depth, from which its owner takes the deepest task and a thief the shallowest (each depth being first in, first out). Its constructor and its methods are declared as follows (the bodies are left out):

```
explicit SkeletonBranchAndBound( const P & problem ,
                                 BBRule rule = BBRule::stack_stealing ,
                                 const Options & options = {} );
void set_executor( Executor & ex ) noexcept;
[[nodiscard]] SolutionT solve( void ) const;
[[nodiscard]] SolutionT solve( std::vector< Leaf > seeds ) const;
```

where `Options` is `BasicParallelBBOptions< bb_value_t< P > >`, whose fields for the skeletons are

```
/// the depth above which the nodes spawn their children as tasks, with
/// the depth-bounded skeleton, and the depth of the ordered tasks [see
/// SkeletonBranchAndBound]
unsigned spawn_depth = 2;

/// the backtracks of a worker of the budget skeleton after which the
/// children left on its stack become tasks
std::uint64_t budget = 10000;

/// true if a steal from a stack takes all the children of the lowest
/// level, rather than the next one
bool steal_all = false;

/// true if the ordered tasks go by discrepancy, false if left to right
bool discrepancy = true;

/// true if a task of the skeletons is the path of the choices that lead
/// to its node, which the worker that takes it rebuilds
bool indexed = false;
```

i.e.,

- with `depth_bounded`, the children of the nodes of depth less than `spawn_depth` d become tasks, and those below are searched in place; the tasks are as many as the nodes of depth d that survive, whatever the shape of the tree;
- with `budget`, a worker that has backtracked `budget` b times turns into tasks the children left in all the frames of its stack but the top, and starts counting again, so that the tasks adapt to the tree: a worker in a small subtree finishes before its budget, and one in a large subtree shares it;
- with `stack_stealing`, an idle worker asks a busy one, which answers at its next step (i.e., after at most an evaluation) with the next child of the lowest frame of its stack that has one, the top excepted, or with all the children of that frame (`steal_all`), which are the largest pieces of work it holds;
- with `ordered`, the nodes of depth d are the tasks, made at the start by the worker 0, which evaluates and branches the nodes above them, and ordered by their *discrepancy*, i.e., the sum of the indices of the children on their path from the root, ties broken left to right (or only left to right, with `discrepancy` false); all the workers take them in this order.

The selection of the options is not used, the order being depth-first in the order of the children; the incumbent is the shared one of Lemma 13.12. The pseudocode, which is that of the header, is the following; EVALUATE(n) is lines 17 to 22 of PROCESS in Algorithm 13.3, d is the spawn depth, b the budget and P_i the pool of the tasks of the worker i.

Algorithm 13.15 (The search skeletons).

```
 1  procedure SKELETON(R)
 2     with ordered: T ← the nodes of depth d under R, by discrepancy;
          the nodes above them evaluated and branched
       else the k-th node of R in P_(k mod p)
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        while t ← NEXT(i) ≠ nil do DFS(t)
 5  procedure DFS(t)
 6     S ← ();  VISIT(t);  c ← 0                ▷ c: the backtracks
 7     while S ≠ () do
 8        if a limit is reached then the rest of S are leaves (open); stop
 9        with stack stealing, if a thief j asks, move to P_j the next
             child (all of them with steal_all) of the lowest frame of S,
             the top excepted, that has one
10        with budget, if c = b then move the children left in the
             frames of S, the top excepted, to P_i;  c ← 0
11        (m, e) ← the next child of the top frame of S, if any
12        if there is none then pop S;  c ← c + 1;  continue
13        β(m) ← max(β(top), e);  if PRUNABLE(β(m)) then m is a leaf
             (pruned), and so are the next children with prune_level
14        else VISIT(m)
15  procedure VISIT(n)
16     if EVALUATE(n) makes n a leaf then return
17     if depth-bounded and depth(n) < d then move all the children of n
          that survive to P_i
18     else push n and the cursor of its children on S
19  procedure NEXT(i)
20     loop
21        if ordered then return the next task of T, or nil
22        if P_i ≠ ∅ then return its deepest task, removed
23        if a pool P_v ≠ ∅ then return its shallowest task, removed
24        with stack stealing, ask a busy worker v for its work
25        if no task is left anywhere then return nil
```

In the code the victim of lines 23 and 24 is one worker drawn at random (by the `seed` of the options) at each turn of the loop, which waits a little after a turn that found nothing. The tasks of line 9 go to the thief j through a slot of its own (an *answer*), which the thief moves into its pool when it sees it. A task of the ordered rule is never split further, since the ordered rule neither spawns nor steals: its subtree is searched by the worker that took it. Every task, whatever the rule, is counted when it is made (by line 2, 9, 10 or 17) and when its subtree is done (line 4), in a shared counter π of the tasks made and not finished; line 25 is the test $\pi = 0$.

Tasks as nodes and tasks as paths

A task is the node itself, or, with `indexed`, the path of the choices that lead to it, i.e., the index of the child taken at each level, from the node of R it comes from (four bytes per level). The worker that takes a task given as a path rebuilds its node before VISIT: from the node of R it follows the path by `child( n , k )` if the evaluator is an `IndexedBBEvaluator`, and otherwise by evaluating each node of the path and taking its child k from its cursor. These evaluations are counted apart, in the counter `recomputed` of `BBStats`, and not in `evaluated`; they call `evaluate` outside PROCESS, so that they neither prune nor offer a value to the incumbent. A task then takes a few bytes per level, whatever the node, as the messages of a search over many processes would need; within one process it saves the copies of large nodes, at the price of the rebuilds.

Lemma 13.27 (The rebuild of a path). Let the children of every node come in an order that depends only on the node (and `child( n , k )` give the k-th of them, if the evaluator has it), and let the evaluation of a node, if done, not change which children the node has nor their order. Then the node rebuilt from a task given as a path is the node that was spawned, and the bound carried by the task is a bound of it.

Proof. We note that the path of a task spawned from the frame at level j of a worker, while it searches a task of path ρ_0, is ρ_0 followed by the indices of the children taken by the frames $0, \ldots, j-1$ of its stack and by the index k of the spawned child in the frame j (`task_of()`): the frame l holds the child of index ρ_l of the node of the frame $l-1$, since an index is pushed on the path exactly when a child is visited and popped when its frame is (the frames of the nodes spawned or pruned push no index). By induction on the length of the path, the node

reached after l steps from the node of R is the node of the frame $l-1$ of the worker that spawned the task: the step takes the child of the same index of the same node, which is the same child by the assumption on the order. The bound of the task is the bound $\max\{\beta(\text{parent}), e\}$ computed at the spawn, which is a bound of the spawned node by (A2). □

The assumption fails for a branching that depends on the incumbent, such as the dynamic branching of the flowshop of §13.19, whose `child( n , k )` therefore throws `std::logic_error`; with tasks as nodes, instead, such a branching is valid, since (A1) holds whichever direction a node takes.

Correctness and end

The skeletons keep the open nodes in a wider sense still than the schemes of §13.7: besides the tasks in the pools, they are the children that the cursors of the frames of the stacks have not given yet, and the tasks of an answer that the thief has not taken yet. A steal of line 9 follows a protocol of three states on a request slot of the victim v: free, asked by the worker j, or claimed. The thief moves it from free to "asked by j" (with an exchange that fails if another thief is asking), the victim moves it from "asked by j" to claimed before it builds the answer, and back to free after it has raised π by the tasks of the answer and set the flag `answered` of j; a thief that waits sees $\pi = 0$ or a limit and withdraws its request only by moving it from "asked by j" to free. A worker that waits for an answer, or that has no stack (it is in NEXT), or that ends, answers the requests it gets with nothing.

Proposition 13.28 (Correctness and end of the skeletons). Under (A1) to (A4) and with $\tau_r \leq 1$, and with the assumptions of Lemma 13.27 when `indexed` is set, Algorithm 13.15 satisfies the hypotheses of Theorem 13.13; $\pi = 0$ holds only when no node is open, an answer is never lost, no two workers wait for each other, and, if the evaluations terminate, each worker returns.

Proof. A child leaves a cursor only by the worker that owns the stack, when it visits the child (line 14), when it prunes it (line 13), or when it turns it into a task (lines 9, 10 and 17); a task leaves a pool under its lock, and an answer only by its thief. Hence, no open node is lost or taken twice, the moves keep the nodes in the wider open set, and the nodes are processed by PROCESS (lines 13 and 16, the test at generation included); the first claim follows, the node of a task given as a path being the spawned node by Lemma 13.27. Every open node belongs to a task not finished: a child of a frame belongs to the task being searched, a task in a pool or in an answer is one. Since π is raised before a task is published (in a pool, or in an answer, whose flag is set after the raise) and lowered only after the stack of its search is empty, $\pi = 0$ implies that no task is made and not finished, and hence that no node is open; a worker that returns nil at line 25 therefore leaves no work behind. For the protocol, the slot of v is "asked by j" only while j waits; if the victim claims it, the thief cannot withdraw (its exchange from "asked by j" fails) and the victim sets the flag after the answer, so that the thief receives it; if the thief withdraws, the victim's exchange from "asked by j" fails and no answer is built. A worker waiting for an answer answers its own requests with nothing; hence, in a cycle of thieves each one answers the next, and no two wait for each other. A victim answers at its next turn of the loop of line 7 (at most one evaluation later), or with nothing when it has no stack; a victim that has returned did so with $\pi = 0$ or after a limit, either of which makes the thief withdraw. Finally, if $\pi > 0$ some task is under search by a worker, which ends it in finite time by (A4), or is in a pool or an answer, from which its owner (or the thief) takes it at its next call of NEXT; as π counts the tasks of a finite tree, it reaches 0. □

After a limit (line 8) the children left on the stacks and the tasks left in the pools, in the answers and, for the ordered rule, in the part of T not yet taken, are open leaves of the Solution, the tasks given as paths being rebuilt, so that a search can be resumed from them as from the leaves of any other (§13.11). The counter `sent` of the statistics holds the tasks that moved between the workers, and `messages` the successful steals.

The node limit is tested by each worker at each turn of the loop of line 7, i.e., before each evaluation, with the shared count of the evaluations, so that the bound $L + p - 1$ of Proposition 13.14 holds for the evaluations of the search of the tasks. The ordered rule, however, does not test the limits while the worker 0 makes the tasks at line 2, which may therefore exceed a node limit by the nodes above the depth d.

What the skeletons repeat

With an incumbent that changes during the search, the nodes evaluated by a skeleton depend on the timing, as in the other asynchronous schemes. With a fixed incumbent, instead, they do not depend even on the rule or on the number of the workers, which makes the skeletons convenient to measure the cost of the exploration (the experiments of [67] and of [71] often give the optimum as the initial incumbent for this reason).

Proposition 13.29 (The nodes evaluated with a fixed incumbent). Let the incumbent not change during the

search (e.g., the optimal value is given in the options, which no feasible value improves), let the outcome of `evaluate( n )` depend only on the node, and let no limit stop the search. Then each skeleton, with each rule, with tasks as nodes or as paths, and with any number of workers, evaluates the same set of nodes, i.e., the nodes of the tree of the children that survive their generation below nodes that survive their evaluation, from R; each of them once.

Proof. With z fixed, the test PRUNABLE is a function of the bound alone, and the bound of a node is the function of its path $\beta(m) = \max\{\beta(n), e(m)\}$ (line 13), where $\beta(n)$ is the bound of the parent after its evaluation, a function of the parent by the assumption on `evaluate`. Hence, whether a node is pruned at its generation, at its extraction (where its bound is the one of its generation, and the test, with the same z, gives the same outcome) or after its evaluation, and which of its children are generated, depend only on its path. The cut of `prune_level` is made in the order of the cursor of a frame, which all the takers of a child (lines 9 to 13 and 17) read in turn, so that it falls on the same child. By the first part of the proof of Proposition 13.28, each child that survives its generation is processed exactly once, by some worker, and each node that is evaluated is branched exactly when it survives its evaluation. The set is therefore defined by the tree. The evaluations of a rebuild are counted in `recomputed`, not in `evaluated`. □

The test `bb-ivm-thread` checks the analogous statement for the intervals of §13.18; for the skeletons it is not checked by a test. With an incumbent that improves, the ordered rule gives a weaker guarantee, which is the reason of its existence.

Proposition 13.30 (The order of the ordered tasks). With the ordered rule, the sequence T of the tasks depends only on the problem, on R, on the options and on the evaluator of the worker 0 (and not on p or on the timing), and a task is started only after every task before it in T has been started.

Proof. We note that T is made by the worker 0 only, before any other worker starts (line 2), by a depth-first search of the nodes above the depth d that prunes by the incumbent it improves itself; it is then sorted by a stable sort on the discrepancy, a function of the path. The workers take the tasks by an atomic fetch-and-add on the index of the next task (`next`), which gives the indices in increasing order of the increments; hence, the task k is taken after the tasks $0, \ldots, k-1$. □

The proposition says *started*, not *finished*: a task is pruned by the incumbent found by the tasks before it only if they have found it by the time it is started (or while it runs), which depends on the timing. Archibald et al. [67] build on this order the replicability of their Ordered skeleton, i.e., runtimes that are never much worse than those of one worker searching the tasks in the same order, and that repeated runs make close to each other; we have not proved those bounds for this implementation, whose worker 0 takes the tasks as the others do. The order by discrepancy puts first the tasks whose path deviates least from the order of the children, i.e., from the heuristic of the branching, where an incumbent is most often found early [67].

The test `bb-skel-thread` (and its versions on OpenMP and FastFlow) checks the four rules, with nodes and with paths as tasks, and with 1, 2, 3 and 8 workers. On six random knapsacks of 30 items, with `budget` 10 and `spawn_depth` 3, each has to give the optimum of the dynamic program and its node; a search stopped by a node limit of 15 has to report a bound not worse than the optimum, and a search from its open leaves with its incumbent has to end it. On a tree of UTS (§13.19), in the mode `enumerate`, the counter `solutions` and the counter `evaluated` have to equal the number of the nodes of the tree counted by a plain recursion. Finally, on the cliques of a random graph of 60 vertices, the stack stealing and the budget have to give the same clique with and without `prune_level`, the latter building fewer children with one worker.

13.15 The racing ramp-up

At the start of a search there are fewer open nodes than workers, and the schemes of §13.8, §13.9 and §13.14 can only wait for the tree to grow. The racing ramp-up of the Ubiquity Generator of Shinano et al. [69] uses this time to search the tree from the root with several settings at once, keeping the tree of the best one. The function `racing_branch_and_bound()` of `RacingBranchAndBound.h` does so before any scheme, and `parallel_branch_and_bound()` calls it when the options list settings in `race`; its declaration, without its body, is

```
template< BranchAndBound P , ADS::WorkerExecutor Executor = ADS::ThreadPool ,
          class Search >
[[nodiscard]] BBSolution< typename P::node_type , bb_value_t< P > >
racing_branch_and_bound( const P & problem ,
                         const BasicParallelBBOptions< bb_value_t< P > > & o ,
```

```
                        Search && search ,
                        std::optional< std::vector< BBLeaf<
                          typename P::node_type , bb_value_t< P > > > >
                          seeds = std::nullopt );
```

where `search( problem , options , starts )` runs the scheme from the given nodes (`parallel_branch_and_bound()` passes itself, with the scheme of the call and the options without the race), and the fields of the options are

```
/// the settings that race on the root (the selection, the dive and the
/// prune_level of each, the rest being those of the options), the racer
/// k using the evaluator of the worker k; none for no race [see
/// RacingBranchAndBound.h]
std::vector< BasicBBOptions< V > > race;

/// the nodes each racer evaluates at most, 0 for 100 times the workers
std::size_t race_nodes = 0;
```

The racer k runs the sequential search of Algorithm 13.3 with the evaluator of the worker k, the options `o` but for the selection, the `plunge_gap` and the `prune_level` of `race[ k ]`, and at most `race_nodes` N evaluations (100 times the workers if 0, and no more than the node limit of `o`). The settings that differ in the evaluator (e.g., in the branching rule, or in the cuts of a relaxation) are obtained by giving the worker k an evaluator by `make_worker( k )`, which the problem builds as it likes. The racers run at once on an executor of as many threads as there are settings, built for the purpose. The pseudocode is that of the header.

Algorithm 13.16 (The racing ramp-up).

```
1  procedure RACE(R)
2     in parallel, for each setting k do
3        S_k ← B&B(R) with the selection of s_k, the evaluator of the
             worker k, at most N evaluations, pruning by the best
             incumbent of all the racers, and keeping its open leaves
4     if some S_k ended without a limit then return the first of them
5     w ← the k of the best bound of S_k, the fewest open leaves on ties
6     z ← the best value of all the S_k
7     S ← the scheme from the open leaves of S_w, with the incumbent z
8     return S, its bound lowered by the closed leaves of S_w
```

The racers share their incumbent through a `BBIncumbent` that each offers its improvements to (by `on_incumbent`) and that each reads at every extraction (by `shared_incumbent`, §13.1). Each racer keeps its leaves with `keep_open_only` (§13.16), so that its Solution holds its open leaves and, in `closed_bound`, the least bound of the others; when the caller asks for the whole frontier (`keep_frontier` without `keep_open_only`), the racers keep all their leaves, and the closed leaves of the winner are added to the frontier of the result. The node limit left to the scheme is that of `o` less the evaluations of all the racers, and the time limit is reduced by the time of the race. When the winner ended without a limit, the scheme is not run, and the result is the incumbent of the race with its bound. The counters of the Solution add those of all the racers.

Proposition 13.31 (The race). Under (A1) to (A4) and with $\tau_r \le 1$, the value returned by Algorithm 13.16 with no limit satisfies Theorem 13.5 over $X(R)$, and its bound satisfies Theorem 13.6 whatever the status.

Proof. Let c be the value of the shared `BBIncumbent` at the end of the race; it is the least of the initial incumbent and of the values found by the racers (Lemma 13.12), and line 6 computes the same value $z = c$. The winner S_w is a sequential search from R that prunes by the least of its incumbent and of the shared value, which is never below c and is the value of a feasible solution or the initial one. By the remark after Theorem 13.6, its leaves cover $X(R)$, and each $x \in X(R)$ outside the subsets of its open leaves has $v(x) \ge \min\{c, \beta_c\}$, where β_c is the least bound of its closed leaves that lower the bound (its `closed_bound`); moreover, the argument of Lemma 13.4 with the threshold of the moment shows that every such x has $v(x) \ge \theta(c') \ge \theta(z)$ for the value $c' \ge z$ that pruned it, or lies in a solved leaf of value at least $c' \ge z$ (Lemma 13.3). Now the scheme of line 7 starts from the open leaves O with the incumbent z, and by Theorem 13.5 (or its parallel form) it ends with a value $z' \le z$ and $z' - z^*_O \le \max\{\tau_a, \tau_r|z'|\}$. If the optimum x^* over $X(R)$ lies in $X(O)$, the claim follows; otherwise $v(x^*) \ge \theta(z) \ge \theta(z')$, since θ is nondecreasing and $z' \le z$, which is again the claim. When the winner ended

without a limit, O is empty and the same argument applies with $z' = z$. For the bound, the leaves of the scheme cover $X(O)$, and the closed leaves of the winner, the solved ones and the subsets of O cover $X(R)$; the bound of line 8, $\min\{\beta_c, z', L_S\}$ with L_S the bound of the scheme, is therefore a bound of z^*_R by Theorem 13.6 applied to both stages. □

The racers that lose give only their incumbents, and their trees are lost; this is the price of the race, which pays when the settings differ much in how early they find a good incumbent or how fast they raise the bound, as is typical of the first nodes of a B&B. In the code a racer that ends without a limit replaces the winner chosen so far when that one was stopped by the node limit or by the time limit (`RacingBranchAndBound.h:191-193`), as line 4 of the pseudocode asks. The test `bb-race` checks a race of three settings (best-first, depth-first and plunging) with budgets of 5, 50 and 100000 nodes and 3 workers, before each of the eleven named schemes, on six knapsacks: the value, its node and the bound have to be the optimum.

13.16 Checkpoints

A search that may run longer than the time granted to it (say, by a batch system) needs a *checkpoint*, i.e., a file from which it can be resumed. Since every search of the library already starts from any set of nodes (§13.11), what is missing is the file, and the bound of the part of the tree that the search has closed. The latter is given by the option `keep_open_only` of `BasicBBOptions`: with `keep_frontier` too, the Solution keeps only the leaves left open by a limit, and its field `closed_bound` holds the least bound of the other leaves that lower the bound (those pruned by the tolerance), so that the bound of the region they cover is known without them; in the sense of the problem, it is $+\infty$ ($-\infty$ in a maximization) when no such leaf exists.

The header `BBCheckpoint.h` gives the state that a checkpoint holds and the concept of the problems whose nodes can be written as bytes:

```
template< class Node , class V = double >
struct BBCheckpoint {
 std::vector< BBLeaf< Node , V > > open;       ///< the open nodes
 std::optional< V > incumbent;                 ///< the incumbent value
 std::optional< Node > node;                   ///< ... and its node
 V closed = BBValueTraits< V >::infinity();    ///< the bound of the closed
 BBStats stats;                                ///< the counters so far
 };
```

```
template< class P >
concept SerializableBranchAndBound = BranchAndBound< P > &&
 std::is_trivially_copyable_v< bb_value_t< P > > &&
 ( std::is_trivially_copyable_v< typename P::node_type > ||
   requires( const P & p , const typename P::node_type & n ,
             std::vector< char > & out , const char * & in ) {
    p.serialize( n , out );
    { p.deserialize( in ) } -> std::convertible_to< typename P::node_type >;
    } );
```

A `SerializableBranchAndBound` is the concept that the search over MPI needs as well (`MPIBranchAndBound` is the same concept): the values are trivially copyable and written as their bytes, and the nodes either are trivially copyable too or are written by `serialize( n , out )`, which appends their bytes to a vector, and read back by `deserialize( in )`, which moves the pointer `in` past them. The functions, whose declarations without their bodies are

```
template< SerializableBranchAndBound P >
void bb_write_checkpoint( const P & p ,
                          const BBCheckpoint< typename P::node_type ,
                                              bb_value_t< P > > & c ,
                          const std::string & file );
template< SerializableBranchAndBound P >
std::optional< BBCheckpoint< typename P::node_type , bb_value_t< P > > >
bb_read_checkpoint( const P & p , const std::string & file );
template< SerializableBranchAndBound P , class Search >
```

```
[[nodiscard]] BBSolution< typename P::node_type , bb_value_t< P > >
checkpointed_branch_and_bound( const P & problem ,
                               const BasicParallelBBOptions<
                                 bb_value_t< P > > & o ,
                               Search && search , const std::string & file ,
                               std::size_t every );
```

write, read and use a checkpoint. The file written by `bb_write_checkpoint()` holds, in this order, eight bytes of a signature, the size of a value, the incumbent value and its node (each preceded by a byte that says whether it is there), the bound `closed` of the region closed so far, the twelve counters of `BBStats` (all but `max_open`, `seconds` and `evaluated_by`), the number of the open nodes, and each open node with its bound and its depth. It is written first to a temporary file (the name with `.tmp` appended) that then takes the name of the checkpoint, so that a run stopped while writing leaves the previous checkpoint whole; `std::runtime_error` is thrown if the file cannot be written. `bb_read_checkpoint()` returns nothing if the file does not exist, and throws `std::runtime_error` if it is not a checkpoint, if its values have another size, or if its length does not match its content. The bytes are those of the machine that writes them, and a checkpoint is therefore read back on a machine with the same representation of the values and of the nodes (the same holds for the messages over MPI).

The function `checkpointed_branch_and_bound()` runs any search, given as `search( problem , options , starts )` (e.g., a call of `parallel_branch_and_bound()` with a scheme), in *segments* of `every` evaluations, writing a checkpoint after each. A segment is a search from the open nodes of the previous one (from the root at first, or from those of the file if it holds a checkpoint) with the options `o` changed in four fields: the incumbent is the one found so far, the node limit is that of the segment (and what is left of the node limit of `o`, which counts the evaluations of the earlier runs too), the time limit is the rest of that of `o`, and `keep_frontier` and `keep_open_only` are set. After a segment the incumbent and its node are those of the segment if better, `closed` is lowered by its `closed_bound`, the open leaves of a segment stopped by a limit become the open nodes, and the checkpoint is written; the loop stops when no open node is left, when the node limit of `o` is reached, or when a segment ends by the time limit. With `every` equal to 0, a segment has no limit of its own. The file is left with the last checkpoint, which has no open node if the search ended, and the Solution has the incumbent, the bound, the counters of all the segments (those of the earlier runs included) and, as frontier, the open nodes of the last checkpoint. A run given the name of an existing checkpoint starts from it, with any scheme and any number of workers; note that the incumbent of the checkpoint is then used, and not that of `o`.

Proposition 13.32 (Segments and checkpoints). Under (A1) to (A4) and with $\tau_r \leq 1$, after each segment the subsets of the open nodes of the checkpoint, together with the subsets of the closed leaves of all the segments so far, cover $X(R)$, and every $x \in X(R)$ outside the former has $v(x) \geq \min\{\theta(z), \text{closed}\}$, z being the incumbent of the checkpoint. Hence, the bound of the Solution, the least of `closed`, of the bounds of the open nodes and of the incumbent, satisfies Theorem 13.6 at every checkpoint, and a run whose last segment ends with no open node returns a value that satisfies Theorem 13.5.

Proof. We proceed by induction on the segments. Before the first one the open node is the root (or the nodes of R), and the claim holds with nothing closed. A segment starts from the open nodes O of the previous checkpoint with its incumbent z, and by Theorem 13.6 (or Theorem 13.13) its leaves cover $X(O)$: the open ones become the new O', and every x in a closed leaf has $v(x)$ at least the `closed_bound` of the segment, or at least the incumbent $z'' \geq z'$ by which a solved leaf was offered, where $z' \leq z$ is the incumbent after the segment; the threshold of a pruned leaf only decreases with the incumbent (Lemma 13.3). The elements outside $X(O)$ satisfy the claim for z and the old `closed` by the induction hypothesis, and $\theta(z) \geq \theta(z')$, while `closed` only decreases; this gives the claim for O', z' and the new `closed`. The bound is then the minimum over a cover, which is Theorem 13.6, and when O' is empty with zero tolerances and no limit the argument of Theorem 13.5 applies. □

The price of a segment is the synchronization of a search that ends and of one that starts, which is small when a segment is long w.r.t. the time of an evaluation; a run stopped at any moment loses at most the work of a segment. The test `bb-checkpoint` checks, on six knapsacks, with the sequential search, the shared queue, the stack stealing and the deterministic rounds, and with 3 workers, that segments of 7 evaluations give the optimum, its node and the exact bound, and leave an empty checkpoint; and that a run stopped by a node limit of 20, in segments of 6, leaves a checkpoint with open nodes and a valid bound, from which a second run ends the search with the counters going on.

13.17 Master, hubs and workers over MPI

Over many processes, a search that sends every child to its owner, as the distributed queues over MPI of §13.9 do, moves much more than the balance of the load needs. In the master-hub-worker organization of ALPS, the framework of Xu, Ralphs, Ladányi and Saltzman [70], each process searches its own subtrees, i.e., the open nodes it holds, a unit of work at a time, and reports its workload to the hub of its cluster: its *quantity* (the number of its open nodes) and its *quality* (the mean bound of its best nodes, the *elite*). A hub pairs the workers of its cluster that are short of work with those that have more, and tells each donor to give some of its nodes to its receiver; a receiver with no donor in its cluster is passed to the master, which passes it to the hub with the largest quantity. The class `MPIHubBranchAndBoundSearch< P >` (`MPIHubBranchAndBound.h:137`), for a model `P` of `MPIBranchAndBound`, implements this organization; its constructor and its methods are declared as follows (the bodies are left out):

```
explicit MPIHubBranchAndBoundSearch( const P & problem ,
                                     const Options & options = {} ,
                                     const BBHubParams & hub = {} ,
                                     MPI_Comm comm = MPI_COMM_WORLD );
[[nodiscard]] SolutionT solve( void ) const;
[[nodiscard]] SolutionT solve( std::vector< Leaf > seeds ) const;
```

where the options are the parallel ones (`threads` is ignored, the workers being the processes of the communicator), and the parameters of the balance are a `BBHubParams` (`MPIHubBranchAndBound.h:116`):

```
struct BBHubParams {
 unsigned cluster = 4;             ///< the processes of a cluster
 std::size_t unit_work = 32;       ///< evaluations between two reports
 std::size_t need_work = 2;        ///< open nodes at which one is a receiver
 std::size_t elite = 4;            ///< the best nodes of the quality
 double donor_threshold = 0.1;     ///< the quality better than the mean by
 double receiver_threshold = 0.1;  ///< ... and worse, as fractions
 };
```

i.e., the processes of a cluster (`cluster`, at least 1), the evaluations of a unit of work between two reports (`unit_work`), the open nodes at which a process is a receiver (`need_work`), the size of the elite (`elite`), and the fractions by which the quality of a donor is better, and that of a receiver worse, than the mean of its cluster (`donor_threshold` and `receiver_threshold`). The processes are split in clusters of `cluster` consecutive ranks, the first of each being its hub, and the rank 0 is also the master; every process searches. MPI must be initialized by the caller, all the processes of the communicator call `solve()`, and the rank 0 starts from the root (with `solve( seeds )` each process starts from its own seeds). The pseudocode is that of the header; u is `unit_work`.

Algorithm 13.17 (Master, hubs and workers).

```
 1  procedure HUB-B&B() on the process i
 2     Q_i ← {root} on the rank 0, ∅ elsewhere
 3     loop
 4        for each message m received do
 5           nodes: push them in Q_i;  incumbent z': z ← min(z, z')
 6           donate k to j: send to j up to k nodes of Q_i, the 2nd,
                4th, ... best, keeping the best;  report to the hub
 7           (hub) report of a worker: BALANCE()
 8           (master) report of a hub, or a need: send to the hub of the
                largest quantity "donate to j"
 9        evaluate up to u nodes of Q_i by PROCESS
10        report the quantity and the quality of Q_i to the hub, after
             each unit and when Q_i becomes empty
11        if Q_i = ∅ and the mailbox detects the end then break
12  procedure BALANCE() on a hub
13     R ← the workers of quantity ≤ need_work
14     if R ≠ ∅ then D ← those of quantity > need_work   ▷ quantity
15     else (quality, once every |cluster| reports) D ← those whose quality
          is better than the mean by more than the fraction
          donor_threshold, R those worse than it by more than
```

```
          receiver_threshold
16     pair the k-th donor of D with the k-th receiver of R, and send
          to each donor "donate half of your nodes to its receiver"
17     the receivers left unpaired and with no node are sent to the
          master
```

A few details of the code complete the pseudocode. The unit of line 9 stops early when no node of Q_i can improve the incumbent, which then turns Q_i into leaves. In BALANCE, the receivers are taken in the order of their ranks, and the donors, in the balance of the quantity, by their quality, the best first; in the balance of the quality both are taken in the order of their ranks, a donor needs more than `need_work` nodes, and the deviation from the mean is measured relative to $\max\{|\mu|, 1\}$, μ being the mean quality. A receiver paired with a donor is not paired again until that donor reports, which it does after each order (line 6), and a receiver sent to the master is released when the master answers that no hub has work for it; the hub counts the order as half of the quantity of the donor until the donor reports again. The master chooses, among the hubs other than that of the receiver, the one of the largest quantity above `need_work`, as last reported by the hubs after each balance. A donation moves the 2nd, 4th, ... nodes in the order of their bounds, so that the donor keeps its best node and the two halves have about the same quality; the counter `sent` holds the nodes donated, and `messages` the donations. The incumbent, the node limit (split evenly among the processes), the end and the Solution are those of `MPIBranchAndBoundSearch` (§13.9): an improvement is sent to all the processes, the node of the incumbent is broadcast at the end by the process of the least value, and every process gets the same Solution but for the frontier, which holds the leaves it owns.

Proposition 13.33 (Correctness and end of the hubs). Under (A1) to (A4) and with $\tau_r \leq 1$, the value and the bound returned by Algorithm 13.17 satisfy Theorems 13.5 and 13.6 when no limit stops it, and the bound satisfies Theorem 13.6 in any case. If the evaluations terminate, the search ends.

Proof. We note that a node moves only in a message of nodes (line 6), which the mailbox counts, and the incumbent value of each process is at least the least value found, as in the proof of Proposition 13.21; hence, Theorem 13.13 applies with the queues and the messages in transit as the wider open set. The end is detected by the four counters of Mattern [11] in the mailbox (sound by Theorem 9.18), which a process joins only with an empty queue (line 11), every message being counted; when it reports the end, every queue is empty and no message is in transit, and the theorem gives the claims, the nodes received after a stop by a limit being open leaves. For the end, let all the queues be empty and the messages of nodes be delivered, which happens in finite time since the tree is finite (A4) and each process evaluates the nodes it holds. Then no process sends a report but the one that says it is empty (line 10), once; a hub answers a report by at most one order per receiver, one message to the master and its report to the master, but no worker has more than `need_work` nodes, so that no order is sent and each receiver with no donor goes to the master; the master answers each such need by one message, which releases the receiver and is not answered. A donor that receives an order late donates nothing (it has fewer than two nodes) and reports once. Hence, the messages caused by the last reports are finite, after them no process sends anything, and the wave of Mattern ends the search. □

The proposition says nothing on the balance, which is a heuristic. In particular, a cluster whose processes are all empty runs its balance only when one of them reports, and its receivers, released by the master when no hub had work for them, wait until another report of their cluster; the work that other clusters generate meanwhile does not reach them. The test `bb-hub-mpi`, run with the processes that `ctest` gives it, checks clusters of 1, 2 and 4 processes, with units of 3 evaluations and an elite of the size of the cluster, on six knapsacks: the value and its node have to be the optimum, and a search stopped by a node limit of 20 has to report a bound not worse than the optimum; and, in the mode `enumerate`, with clusters of 2, the count of a tree of UTS has to be that of a plain recursion.

13.18 Intervals of leaves

In a tree whose nodes of depth d all have the same number r_d of children, as the trees of the permutation problems ($r_d = n - d$ for the permutations of n elements), a leaf is the vector of the indices of the children on its path, i.e., an integer in a mixed radix (the factorial number system for the permutations), and the leaves of a node are an interval of such integers. The Integer-Vector-Matrix (IVM) of Mezmaz, Leroy, Melab and Tuyttens [71] and of Gmys et al. [72] makes such an interval the unit of work: a worker explores depth-first the leaves of its interval $[a, b)$, holding the path to a (the vector and the matrix of the IVM are the digits of a and the nodes of the path), and an idle worker takes the second half of the interval of another one, which moves two vectors of digits, whatever the node. The class `IntervalBranchAndBound< P , Executor >` (`IntervalBranchAndBound`

.h:137) implements it for the problems of the concept RadixBranchAndBound (IntervalBranchAndBound.h:115):

```
template< class P >
concept RadixBranchAndBound = BranchAndBound< P > &&
 requires( const P & p , unsigned d ) {
  { p.arity( d ) } -> std::convertible_to< std::size_t >;
  } &&
 IndexedBBEvaluator< decltype( detail::bb_worker( std::declval< const P & >() ,
                                                  0u ) ) ,
                     typename P::node_type , bb_value_t< P > >;
```

i.e., for the problems that give `arity( d )`, the number r_d of the children of a node of depth d, 0 at the depth D of the leaves, and an evaluator with `child( n , k )` (§13.13). Its constructor reads the arities $r_0, \dots, r_{D-1}$ once, and its declarations, without their bodies, are

```
explicit IntervalBranchAndBound( const P & problem ,
                                 const Options & options = {} );
void set_executor( Executor & ex ) noexcept;
[[nodiscard]] SolutionT solve( void ) const;
```

with the parallel options; the search starts from the root only, and the selection of the options is not used.

The numbering of the leaves

A leaf is the vector $x = (x_0, \dots, x_{D-1})$ with $0 \le x_d < r_d$, the child of index x_d being taken at the depth d, and its *number* is

$$N(x) = \sum_{d=0}^{D-1} x_d\, w_d, \qquad w_d = \prod_{e=d+1}^{D-1} r_e,$$

so that the leaves are the integers of $[0, W)$, $W = r_0 w_0 = \prod_d r_d$. The code keeps a number as its vector of digits, compares two of them lexicographically, and lets the digit 0 exceed $r_0 - 1$, so that the end W of the tree is the vector $(r_0, 0, \dots, 0)$.

Lemma 13.34 (The numbering of the leaves). The map N is a bijection between the leaves and $[0, W)$ that preserves the lexicographic order of the vectors, i.e., the left-to-right order of the leaves. The leaves of a node n of depth d with path $(y_0, \dots, y_{d-1})$ are the interval $[f(n), f(n) + w_{d-1})$ of the numbers whose first d digits are those of the path ($[0, W)$ for the root), where $f(n) = N(y_0, \dots, y_{d-1}, 0, \dots, 0)$ is its first leaf; and a number a is the first leaf of the node of depth d on its path if and only if $a_e = 0$ for every $e \ge d$.

Proof. We note that $w_d\, r_d = w_{d-1}$, and that $\sum_{e>d} x_e w_e \le \sum_{e>d} (r_e - 1) w_e = w_d - 1$ (a telescoping sum). Hence, the digits after d contribute less than w_d, and two vectors that first differ at the digit d have numbers in the order of that digit; this gives the preservation of the order, and the injectivity. The image has $\prod_d r_d = W$ elements in $[0, W)$, by the same bound, which gives the bijection. The leaves of n are the vectors with the prefix of its path and any suffix, whose numbers are $f(n)$ plus the numbers of the suffixes, i.e., the w_{d-1} consecutive integers from $f(n)$. The last claim is the definition of f. □

The arithmetic of the steals works on the digits. The difference $b - a$ and the sum $a + h$ are computed digit by digit from the last, with a borrow or a carry, the digit 0 being unbounded, and the half $\lfloor (b-a)/2 \rfloor$ by the school division from the first digit, the remainder of the digit d entering the digit $d+1$ multiplied by r_{d+1}. Each is exact on the numbers of $[0, W]$, by the same identity $w_d r_d = w_{d-1}$, and costs $O(D)$ operations.

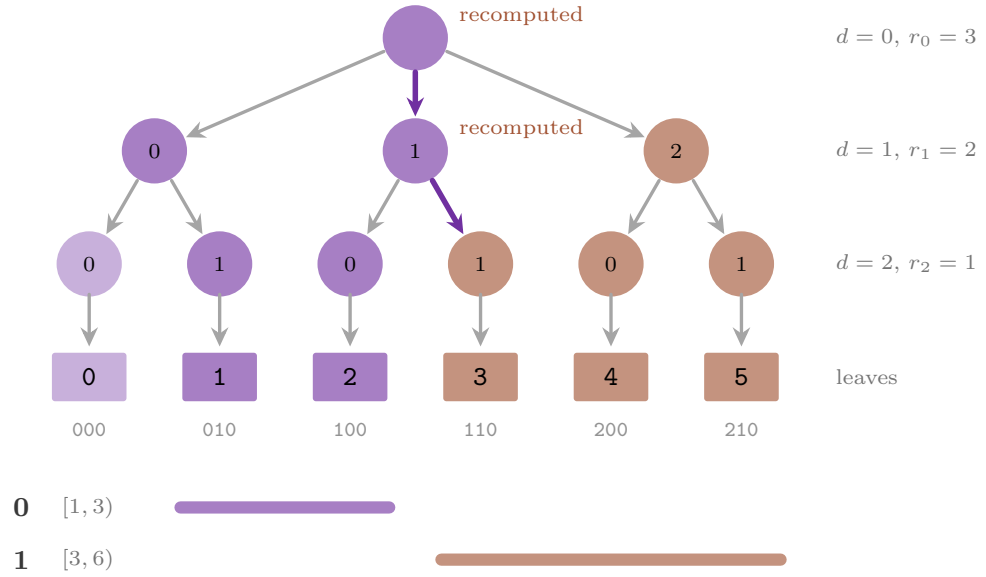
The search

The pseudocode is that of the header.

Algorithm 13.18 (The work stealing of intervals of leaves).

```
 1  procedure INTERVAL()
 2     the worker 0 holds [0, ∏ r_d), the others nothing
 3     in parallel, on each worker i = 0, ..., p − 1 do
 4        loop
 5           while a_i < b_i do
 6              n ← the node of the path to a_i at the current depth d
 7              if d = D or PRUNABLE(β(n)) after EVALUATE(n) then
 8                 a_i ← the first leaf after the subtree of n
 9              else go down to the child a_i[d] of n
10           if no worker holds leaves then return
11           v ← a random worker;  with the lock of v, if b_v − a_v ≥ 2
                then m ← a_v + ⌊(b_v − a_v)/2⌋;  e ← b_v;  b_v ← m;
                then, with the lock of i alone, [a_i, b_i) ← [m, e)
```

In the code a child is built by `child( n , k )` and tested at its generation by its estimate (lines 25 and 26 of Algorithm 13.3), and a child pruned there advances a_i past its subtree as at line 8; the path is kept as the nodes from the root to the current one with their bounds, and after line 8 only the nodes below the last digit changed are built again. A worker holds its interval under a lock of its own, which it takes to read a_i before each node, to compare it with b_i, and to move it; the evaluation itself is made without the lock. Line 10 is a shared counter of the workers that hold a non-empty interval (`active`), which the worker 0 starts at 1, a thief raises under the lock of its victim at line 11, and a worker lowers when its interval becomes empty. The node of line 6 is evaluated as such only if a_i is its first leaf; otherwise it is an ancestor of the leaf at which the worker took its interval, which the worker it came from evaluated or will evaluate as its own, and its evaluation, made only to obtain its bound, is counted in `recomputed`, neither in `evaluated` nor in `solutions`. Figure 13.15 shows a steal on the tree of the permutations of three elements.



worker 0 held $[1, 6)$ with $a = 010$; the thief 1 halves $b - a = 5$, takes $[m, 6)$ with $m = a + 2 = 110$, and builds the path to 110 by child(n, k), recomputing the root and the node 1, whose first leaves (0 and 2) are not in its interval

Figure 13.15. A steal of an interval on the tree of the permutations of three elements, radix $(3, 2, 1)$, whose six leaves are numbered 0 to 5, with their digits below; the color of a node is that of the worker whose interval holds its first leaf. Worker 0 has explored the leaf 0 and holds $[1, 6)$; the thief, worker 1, halves $b - a = 5$ and takes $[3, 6)$, leaving $[1, 3)$ to worker 0, then builds the path to the leaf 110 by `child( n , k )`, evaluating the root and the node 1 again for their bounds (they are counted as recomputed) and the node 11 as its own.

Proposition 13.35 (Correctness and end of the intervals). Under (A1) to (A4), with $\tau_r \leq 1$ and with the assumptions of Lemma 13.27 on `child( n , k )`, let the problem give the arities of its tree. Then (i) at any moment the intervals of the workers are disjoint, and every leaf of $[0, W)$ that is in no interval is a leaf of a subtree already explored or pruned, i.e., whose elements have values not below $\theta(z)$ or were offered to z; (ii) `act ive` is 0 only when every interval is empty; hence, Algorithm 13.18 satisfies the conclusions of Theorem 13.13,

and, if the evaluations terminate, each worker returns.

Proof. (i) At the start the only interval is $[0, W)$. A worker moves a_i forward only at line 8, past the subtree of a node that is a leaf of the tree, or infeasible, or pruned at its evaluation or at its generation, all of whose elements are therefore covered as in Lemma 13.4 (by Lemma 13.34 the leaves it skips are exactly those of the subtree, the remaining ones of the interval being after it). A steal takes the lock of the victim, cuts its interval at m with $a_v < m < b_v$, and releases the lock; the thief, whose interval is empty and which no other worker can take from (a steal needs $b - a \geq 2$), then sets $[m, e)$ under its own lock. Hence, the two halves are disjoint and their union is the old interval, the second half belonging only to the thief in between, and no worker holds two locks at once. (ii) The counter is raised by a thief under the lock of a victim whose interval is not empty, and which is therefore counted, since a worker lowers it only after it has seen its own interval empty under its lock, and an interval becomes non-empty again only by a steal of its holder, which raises the counter first; hence, the counter is never below the number of the non-empty intervals plus the thieves between lines 11 and 5, and 0 means that every interval is empty, i.e., by (i), that every leaf has been explored or pruned. The nodes of the paths are rebuilt by `child( n , k )`, which gives the node of the path by Lemma 13.27, with a bound computed along the path. Finally, each worker moves its a_i forward at each evaluation or pruning, and the leaves are finite; a worker that finds no victim with two leaves at least waits until `active` is 0, which happens when the others end their intervals. □

Proposition 13.36 (Each node once, and the nodes with a fixed incumbent). In Algorithm 13.18 each node of the tree is evaluated as its own by at most one worker, the one whose interval holds its first leaf when it reaches it, and at most once. If, moreover, the incumbent does not change during the search and the outcome of `evalua te( n )` depends only on the node, then the set of the nodes evaluated as their own is that of the sequential depth-first search (one worker), whatever the steals and the number of the workers.

Proof. A node n of depth d is evaluated as its own by the worker i only when $a_i = f(n)$ (by Lemma 13.34, a_i has zeros from the digit d), and a_i then moves past $f(n)$ before the worker returns to the depth d (it goes down, or it skips the subtree). Since the intervals are disjoint and their starts never move backwards, the leaf $f(n)$ is the start of an interval at most once, and in at most one worker. A thief that takes $[m, e)$ with $m = f(n)$ evaluates n as its own, and its victim, whose interval now ends at m, never reaches $f(n)$; the ancestors of m whose first leaf precedes m are evaluated by the thief only for their bound. For the second claim, with z fixed, whether a node is pruned at its generation or at its evaluation depends on its path alone, as in the proof of Proposition 13.29, since the bound of a node is the maximum of the estimates and of the bounds along its path, which the thief computes again by the same evaluations. Hence, a node is evaluated as its own if and only if none of its proper ancestors is pruned or is a leaf, which is the condition of the sequential depth-first search; the recomputations of the thieves do not change it. □

The test `bb-ivm-thread` (and its versions on OpenMP and FastFlow) checks both claims: on four random flowshops of 7 jobs on 3 or 4 machines, with both bounds, the forward and the alternate branchings and the children sorted by their bound, the search with 1, 2, 3 and 8 workers has to give the makespan found by enumerating all the permutations, and, with the optimum given as the incumbent, it has to evaluate the same number of nodes as with one worker. On three random quadratic assignments of 7 facilities (the third asymmetric) it has to give the cost of the enumeration with 1 and 4 workers.

After a limit, the leaves left in an interval $[a_i, b_i)$ are covered by the fewest nodes, each being the shallowest node whose leaves begin at the current a_i and end before b_i, after which a_i moves past its subtree; these nodes are open leaves of the Solution, built by `child( n , k )` from the root, with the largest estimate along their path as their bound.

Proposition 13.37 (The leaves after a limit). The nodes left by a worker after a limit have disjoint subtrees whose leaves are exactly those of $[a_i, b_i)$, and no set of fewer nodes has this property; their bounds are bounds of their subtrees by (A2).

Proof. We proceed by induction on the nodes chosen. The node chosen for a is the shallowest node whose interval $[f, f + w)$ has $f = a$ and $f + w \leq b$ (the leaf of a is one, at the depth D, so that one exists); after it a moves to $f + w$, and the rest of the interval is covered in the same way. The intervals of the chosen nodes are therefore disjoint and their union is $[a_i, b_i)$. Any cover of $[a_i, b_i)$ by subtrees needs a node whose interval begins at a_i and lies within the interval, whose interval is then contained in that of the chosen node, which is the largest such; by an exchange argument on the first node of a minimal cover that differs from the chosen ones, the greedy choice is minimal. The bounds are the maxima of estimates, each a bound by (A2). □

The test checks the restart: with a node limit of 30 and `keep_open_only`, a stack stealing from the open leaves, with the incumbent of the stopped search, has to find the optimum of the flowshop. Note that the dynamic

branching of the flowshop (§13.19), whose tree depends on the incumbent, makes `child( n , k )` throw, and is therefore refused by this search; also, the option `prune_level` has no effect here, since each child is built on its own.

13.19 The example problems

Besides the knapsack of §13.2, the layer comes with six problems that exercise its parts, each in a header of `include/PBeFS/bnb` and each a model of `BranchAndBound` that a program uses with any scheme. They are examples, i.e., they are not part of the engine of the library, and their bounds are classical ones, not those of the best solvers of each problem. Table 13.2 gathers what each one gives.

Table 13.2. The example problems of the B&B layer: the sense, the values, the methods of the evaluator beyond `evaluate`, and the parts of the layer it exercises.

Problem	Sense, values	Children	Exercises
`MaxClique`	max, `int`	`children`	lazy children, `prune_level`
`MaxCommonSubgraph`	max, `int`	`children`	lazy children
`SubgraphIsomorphism`	min, `int`	`children`	the modes
`QAP`	min, `long`	`branch`, `child`, `arity`	the intervals
`Flowshop`	min, `int`	`branch`, `child`, `arity`	the intervals, `cutoff`
`UTS`	min, `long`	`children`, `child`	the mode `enumerate`

The maximum clique

The header `MaxClique.h` gives two small structures on which the graph problems are built. A `BitSet` is a set of the integers below a given size, as words of 64 bits, with the operations of a set (`set`, `reset`, `test`, `count`, the least element `first()`, the intersection and the difference), and a `BitGraph` is an undirected graph without loops, as the bitsets of the neighbours of its vertices. `BitGraph::dimacs( file )` reads a graph of the format of the DIMACS challenges (the lines `p edge n m` and `e u v`, the vertices counted from 1) [174], and `by_degree ()` renumbers the vertices by decreasing degree, the lower number first on ties.

`MaxClique` is the maximum clique of a graph, the graph being renumbered by decreasing degree when the problem is built, with the bound of a greedy colouring computed on bitsets, as in the algorithms of Tomita and Kameda [74] and of San Segundo, Rodríguez-Losada and Jiménez [75]. A node is a clique C with the set P of the vertices adjacent to all those of C. Its evaluation colours P greedily, a colour class at a time, each class taking the vertices of P in their order that are not adjacent to those already in it; with k colours, the bound is $|C| + k$, and $|C|$ is a feasible value. The children are made in the reverse order of the colouring: the child of the vertex v of colour c adds v to C, restricts P to the neighbours of v that come before it in the colouring, and has the estimate $|C| + c$. The children come one at a time from a cursor that refers to the node (the evaluator gives no `branch`), and `vertices( n )` gives the clique numbered as in the file, while `check( n )` tests that it is a clique.

Lemma 13.38 (The colouring bound). The value $|C|+k$ bounds the size of every clique of the node, the estimate $|C| + c$ bounds the cliques of the child of a vertex of colour c, the estimates of the children are nonincreasing in the order of the cursor, and the children cover the cliques of the node. Hence, the assumptions (A1) to (A4) hold, and so does the assumption of Proposition 13.25, in the maximization.

Proof. A colour class is an independent set, and a clique has at most one vertex in each; hence, a clique $C \cup K$ with $K \subseteq P$ has $|K| \leq k$. The vertices that come before a vertex v of colour c in the colouring have colours at most c, and so do the vertices of the P of its child; hence, the same argument bounds the child by $|C|+1+(c-1)$. The cursor takes the vertices from the last coloured, whose colours do not increase. A clique of the node that contains some vertex of P contains a last one, v, in the reverse order of the colouring, and its other vertices of P come before v and are adjacent to it; hence, it is a clique of the child of v, and the clique C itself is the feasible value of the node. Each child adds a vertex, and P shrinks, which gives (A4); a node with no child has $P = \emptyset$ and is solved by its feasible value. □

In the test `bb-domains` the cliques of four random graphs of 50 to 80 vertices, by the shared queue, have to be those found by the algorithm of Bron and Kerbosch with a pivot [175], and that of the graph `brock200_1` of the DIMACS challenge, by the stack stealing with `prune_level` and 4 workers, has to have the size 21. The design note of the layer reports that on `brock200_1` the lazy children with `prune_level` build 693899 children against

7445660, with the same 524730 nodes evaluated (`docs/branch-and-bound.md`); we have not measured these counts again.

The maximum common subgraph

`MaxCommonSubgraph` (`MaxCommonSubgraph.h`) is the maximum common induced subgraph of two graphs G and H (the fields `g` and `h`, which are `BitGraph`s), with the partition of the vertices into label classes and the bound of the algorithm McSplit of McCreesh, Prosser and Trimble [76]. A node is a matching M of vertices of G to vertices of H, which induce isomorphic subgraphs, and a partition of the vertices left into classes (L_c, R_c), those of L_c in G and of R_c in H having the same adjacency to the matched vertices, so that only a vertex of L_c can be matched to one of R_c. Its bound is $|M| + \sum_c \min\{|L_c|, |R_c|\}$, and $|M|$ is a feasible value. The evaluation picks the class of the least $\max\{|L_c|, |R_c|\}$, and in it the vertex v of G of the largest degree; the children match v to each vertex w of R_c, by decreasing degree in H, splitting each class into the neighbours and the non-neighbours of v and of w (a class empty on either side is dropped), and the last child leaves v unmatched; the estimate of a child is its own bound.

Lemma 13.39 (The bound of McSplit). The bound of a node bounds the size of every common induced subgraph that extends its matching, and its children cover them.

Proof. Let $M' \supseteq M$ be the matching of a common induced subgraph. A pair $(x, y) \in M' \setminus M$ has the same adjacency to the matched vertices in G and in H, since the subgraphs are isomorphic under M'; hence, x and y are in the same class. The pairs of a class are a matching between L_c and R_c, of size at most $\min\{|L_c|, |R_c|\}$, which gives the bound. For the cover, either v is matched in M', to some $w \in R_c$ by the first argument, and M' extends the child of w, whose classes are those of the vertices with the same adjacency to the matched vertices and to the pair (v, w); or it is not, and M' extends the last child. □

The estimates of the children are not monotone, and `prune_level` is not valid for this problem. `check( n )` tests that a matching induces isomorphic subgraphs, and `modular_product()` gives the modular product of G and H, whose largest clique has the size of the largest common induced subgraph; the test `bb-domains` compares, on four pairs of random graphs of 7 or 8 and 8 vertices, the value of the sequential search, of the stack stealing and of the distributed queues with the clique of the modular product found by `MaxClique`, which is a different algorithm on a different graph. The vertices are stored in 16 bits, so that each graph has at most 65536 vertices.

The subgraph isomorphism

`SubgraphIsomorphism` (`SubgraphIsomorphism.h`) is the problem of an embedding of a pattern graph into a target graph (the fields `pattern` and `target`), i.e., of an injection of its vertices that maps its edges to edges, and, with `induced`, its non-edges to non-edges, in the modes `decide` (whether there is one) and `enumerate` (how many there are) of §13.13. Its values are all 0 (any embedding is a solution), and its bounds are the default 0 of `BasicEvaluation`. A node gives each vertex p of the pattern a domain D_p of target vertices, a bitset; at the root D_p holds the target vertices of degree at least that of p. The evaluation finds the node infeasible if a domain is empty, or if the union of the domains of the unassigned vertices (those with more than one value) is smaller than their number; it is feasible if all the domains are single values that make an embedding (and infeasible if they do not), and otherwise it picks the unassigned vertex of the smallest domain (of the largest degree on ties). Its children give that vertex each value t of its domain, by increasing number, removing t from the other domains and restricting the domains of its neighbours to the neighbours of t (and, with `induced`, of its non-neighbours to the non-neighbours of t), which is the forward checking of constraint programming. `lad( file )` reads a graph of the format of the solver LAD (the number of vertices, then, for each vertex, its degree and its neighbours, counted from 0), and `check( n )` tests an embedding.

Lemma 13.40 (The pruning of the subgraph isomorphism). No embedding is lost by the root, by the tests of infeasibility, or by the branching: every embedding f with $f(p) \in D_p$ for every p belongs to exactly one child of a node that is branched.

Proof. An embedding maps a vertex p to a vertex of degree at least that of p, since the neighbours of p go to distinct neighbours of $f(p)$; this gives the root. If a domain is empty no f exists. The values of the unassigned vertices are distinct and lie in their domains, hence in the union, which therefore has at least as many elements as they are. The child of the value $f(v)$ of the vertex v of the branching keeps f, since f is injective ($f(q) \neq f(v)$ for $q \neq v$) and maps the neighbours of v to neighbours of $f(v)$ (and, induced, the non-neighbours to non-neighbours); the children give distinct values to v, so that f belongs to one child only. A node whose domains are all single values is one candidate, which `check()` accepts exactly when it is an embedding. □

Hence, in the mode `enumerate` the counter `solutions` is the number of the embeddings, each being a distinct node of the tree (Proposition 13.26). The test `bb-domains` checks six pairs of random graphs (patterns of 4 or 5 vertices, targets of 9, two of them induced) by the sequential search, the shared queue, the depth-bounded skeleton and the stack stealing: the count has to be that of an enumeration of all the injections, and the mode `decide` has to find an embedding exactly when one exists.

The quadratic assignment

`QAP` (`QAP.h`) is the quadratic assignment problem of Koopmans and Beckmann [176]: given the flows $F = (f_{ij})$ between n facilities and the distances $D = (d_{kl})$ between n locations, it is

$$\min_{\pi} \sum_{i,j} f_{ij}\, d_{\pi(i)\pi(j)}$$

over the permutations π, with the bound of Gilmore [77] and Lawler [166]. `QAP::qaplib( file )` reads an instance of QAPLIB [78] (the size, then the two matrices), and `prepare()` orders the facilities by decreasing total flow (in and out, the lower number first on ties); a program that sets the matrices itself calls it. A node assigns the first d facilities of this order to locations, its children assign the next facility to each free location, in increasing order, with no estimate (the least `long`, so that a child gets the bound of its parent until it is evaluated), and `child( n , k )` gives the child k at once, without evaluating n, so that `arity( d )` $= n - d$ makes `QAP` a `RadixBranchAndBound`. The bound of a node, with U the facilities left and L the locations left, is the cost of the pairs of facilities assigned plus the value of the linear assignment of U to L whose cost of $i \to l$ is

$$c_{il} = f_{ii}d_{ll} + \sum_{j \text{ assigned}} \left(f_{ij}d_{l\pi(j)} + f_{ji}d_{\pi(j)l}\right) + \langle f_{i,U\setminus\{i\}}, d_{l,L\setminus\{l\}}\rangle_{-},$$

where $\langle\cdot,\cdot\rangle_{-}$ is the least scalar product of the two vectors, obtained by sorting the first up and the second down. The linear assignment is solved by `linear_assignment( c , m )`, the Hungarian method with potentials [177] in $O(m^3)$ for a square matrix of size m given by rows. A complete assignment is feasible, of its cost (`cost( n )`).

Lemma 13.41 (The Gilmore-Lawler bound). For every completion π of the assignment of a node, the cost of π is at least the bound of the node, for any (possibly asymmetric) matrices.

Proof. We split the sum over the ordered pairs (i, j) by the kind of i and j. The pairs of two assigned facilities give the fixed cost. A pair with $i \in U$ and j assigned, and the pair (j, i), give the middle terms of $c_{i\pi(i)}$; the pair (i, i) with $i \in U$ gives $f_{ii}d_{\pi(i)\pi(i)}$. The pairs (i, j) with $i \neq j$ both in U are counted once, by their first facility i, as $\sum_{j\in U\setminus\{i\}} f_{ij}d_{\pi(i)\pi(j)}$, the scalar product of the flows of i with a permutation of the distances from $\pi(i)$ to $L \setminus \{\pi(i)\}$; by the rearrangement inequality it is at least the least scalar product. Hence, the cost of π is at least the fixed cost plus $\sum_{i\in U} c_{i\pi(i)}$, and the latter is at least the value of the linear assignment. □

Each ordered pair being counted once, by its first facility, the bound holds for asymmetric matrices too, and the rearrangement inequality holds for any real entries; the code computes in `long` and does not check for an overflow. The locations are stored in 16 bits, so that $n \leq 32767$. The test `bb-domains` solves the instances `chr12a`, `had12` and `nug12` of QAPLIB by the stack stealing with 4 workers, with the published optimum plus 1 as the initial incumbent, and checks their published optima 9552, 1652 and 578.

The permutation flowshop

`Flowshop` (`Flowshop.h`) is the permutation flowshop of the least makespan, $F_m|prmu|C_{\max}$, with the one-machine and the two-machine bounds of Lageweg, Lenstra and Rinnooy Kan [79]. The n jobs are processed on the m machines in the same order, the job j taking p_{kj} on the machine k (the field `p`, by machines), and the makespan of a sequence is the time at which its last job leaves the last machine. A node fixes a head σ_1 and a tail σ_2 of the sequence, the unscheduled jobs U being in between. Let f_k be the completion time of σ_1 on the machine k (the least head $\min_j \sum_{l<k} p_{lj}$ if σ_1 is empty), b_k the time from the start of σ_2 on k to the end (the least tail if it is empty), and $r_k = \sum_{j\in U} p_{kj}$. The one-machine bound is $\max_k(t_k + b_k)$ with $t_0 = f_0 + r_0$ and $t_k = \max\{t_{k-1}, f_k + r_k\}$, and the two-machine bound is the largest, over the pairs $k < l$ of machines, of the makespan of the two-machine flowshop with the time lags $\sum_{k<i<l} p_{ij}$ on U, started at f_k and f_l and followed by b_k and b_l, in the order of Johnson [178] for the pair (computed once on all the jobs by `prepare()`). A complete sequence is feasible, of its makespan. The field `bound` chooses the bound (`Bound::one_machine` or `Bound::two_machine`, the default).

The children of a node append a job of U to σ_1 (forward), or prepend it to σ_2 (backward), by the field `branching`: the direction is fixed (`forward`, the default, or `backward`), alternates with the depth (`alternate`), or is the one with the fewer children whose bound is below the incumbent, the larger sum of their bounds on ties (`dynamic`), which needs the bounds of all the children of both directions. The bound of each child is computed when the node is branched, and is its estimate; its evaluation does not compute it again. The children come by increasing job, or, with `sort_children`, by increasing bound. `make_worker( i )` gives an evaluator that keeps the incumbent of `cutoff( z )` (§13.1), which the dynamic branching needs, and `child( n , k )` gives the child k of a node without evaluating it, but not with the dynamic branching, whose tree depends on the incumbent (§13.14); `arity( d )` is $n - d$. `Flowshop::taillard( id )` builds the instance `ta001` to `ta120` of Taillard [80] from its seed, by the generator of the paper, and `taillard_optimum( id )` gives the optimal makespans of `ta001` to `ta030` (the value -1 for the others).

Lemma 13.42 (The one-machine bound). For every completion of the head and of the tail of a node, the makespan is at least the one-machine bound.

Proof. In any completion the jobs of U are processed on the machine k after the head (from f_k at the earliest, the least head when the head is empty bounding the start of the first job on k), one at a time, and before the tail, which needs b_k after its start on k; hence, the machine k finishes U at $T_k \geq f_k + r_k$, and the makespan is at least $T_k + b_k$. Moreover, the last job of U on the machine k ends there after it ends on the machine $k-1$, so that $T_k \geq T_{k-1}$, and by induction $T_k \geq t_k$. □

For the two-machine bound we rely on [79]: it is the optimal makespan of a relaxation in which only the machines k and l are capacitated, the machines between them being delays, which the order of Johnson solves exactly; we have not proved here that the order computed once on all the jobs, and restricted to U, is optimal with the release times f_k and f_l. The test `bb-domains` checks four random flowshops of 7 jobs on 4 machines, with both bounds and the four branchings, against the enumeration of all the permutations, and the instance `ta001`, by the stack stealing with the dynamic branching, 4 workers and the optimum plus 1 as the initial incumbent, against its optimum 1278.

The unbalanced tree search

`UTS` (`UTS.h`) is the Unbalanced Tree Search benchmark of Olivier et al. [73], as a problem of the mode `enumerate` whose tree is implicit and whose count of nodes is the answer. A node holds the 20 bytes of the state of a random generator and its depth: the state of the child i is the SHA-1 digest [179] of the state of its parent followed by the four bytes of i (most significant first), the state of the root the digest of 16 zero bytes followed by the four bytes of the seed, and the last four bytes of a state, with the sign bit cleared, are the random number r of the node, read as the probability $u = r/2^{31}$. The digest is computed by the structure `SHA1` of the header, a plain implementation of the standard for messages of a few bytes. In a binomial tree the root has b_0 children, and any other node m children if $u < q$ and none otherwise; in a geometric tree the number of children of a node of depth d is $\lfloor \log(1-u)/\log(1-p) \rfloor$ with $p = 1/(1+b_d)$, i.e., geometric of mean b_d, where $b_d = b_0$ at the root and otherwise $b_0(1-d/g)$ (`linear`), $b_0 d^{-\log b_0/\log g}$ (`expdec`), $b_0^{\sin(2\pi d/g)}$ up to $d = 5g$ and 0 below (`cyclic`), or b_0 up to $d = g$ and 0 below (`fixed`), with g the depth parameter (`gen_mx`); a non-root node has at most 100 children, and the root of a binomial tree at most $\lceil b_0 \rceil$. The trees T1 (geometric, fixed, $b_0 = 4$, $g = 10$, seed 19), T2 (geometric, cyclic, $b_0 = 6$, $g = 16$, seed 502) and T3 (binomial, $b_0 = 2000$, $m = 8$, $q = 0.124875$, seed 42) of the benchmark are given by `UTS::t1()`, `t2()` and `t3()`, whose comments give 4130071, 4117769 and 4112897 nodes, and any other tree by `UTS::geometric()` and `UTS::binomial_tree()`.

Every node is feasible, of value 0, so that `solutions` counts the nodes of the tree (Proposition 13.26); the children come one at a time, and `child( n , k )` gives the child k of a node at once, so that the searches that send a node as the path that leads to it rebuild it without an evaluation. The number of the children depends on the node, not on its depth, and `UTS` is therefore not a `RadixBranchAndBound`. The nodes are trivially copyable, and travel over MPI as they are. The test `bb-domains` counts three smaller trees (a linear and a cyclic geometric one and a binomial one) by the sequential search, the work stealing, the budget and the ordered skeletons, with tasks as paths and 4 workers, against the plain recursion `count( n )` of the header; the counts of T1, T2 and T3 are not checked by a test.

13.20 Benchmark

The schemes are measured by `tools/BranchAndBoundBench.cpp`, run by the table F of `tools/bench/campaign.sh`, on two problems, the problems `knapsack` and `tsp-bb` of that table, each of which gives one part of Table

13.2. The first is made of random 0-1 knapsacks of the strongly correlated class of Pisinger with 90 items of weight at most 1000 (the seeds 3, 8 and 13), with values of type `double`; the feasible values are exact, since the profits are integer and their sums much smaller than 2^{53}, while the fractional bounds are rounded. Our second problem is the tree of the partial tours of random traveling salesperson problems with 24 cities of the unit square (the seeds 2, 3 and 4), whose B&B is the `TourTree` of §13.6, with the bound $g + h$ of the minimum spanning tree heuristic (Chapter 5) in double precision. In all the runs the selection is best-first. The sequential search runs on one core, and each parallel scheme on t cores, for the numbers of cores in the header of the table, with one worker per core, the cores of one NUMA node and the memory on that node. The shared queue, the distributed queues, the rounds (one node per worker and round), the SelfSplit (with and without the shared incumbent), the work stealing and the MultiQueue run on each of the three executors; the farm runs with $t - 1$ workers and its master on FastFlow, and the search over MPI with one process per core. Each run is a separate process, and each is run twice; the gap between the two is the A/A noise of the machine, which the note of the table gives.

The time is the wall-clock time measured by the `Stopwatch` of the search, which starts when `solve()` starts. Hence, it does not include the generation of the instance, the sorting of the items, the dynamic program that checks the optimum, nor, for MPI, the start of the processes and `MPI_Init`. It does include the start of the threads of the parallel searches, since the benchmark gives them no executor and each search builds a pool. As for the machine and the compiler, they are those stated in the note of the table.

Table 13.3. The B&B schemes on the two problems of table F described above, in two parts, `knapsack` and `tsp-bb`. In each part a row is a search and a column a number of cores; each cell is the time in seconds summed over the three instances of the problem, each the median of its two runs, and in parentheses the speedup over the row `B&B sequential`, i.e., the sequential search on one core; an empty cell is a search that did not solve every instance. The note under each part gives the machine and the A/A noise.

knapsack

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

tsp-bb

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The table is a placeholder until it is measured on an idle machine. Our benchmark also writes, for each run, the counters by which one can read the anomalies of §13.7. They are the nodes evaluated (column `expanded` of the CSV, in the format shared with the benchmark of A*), branched (`reexpanded`) and pruned (`duplicates`), the largest queue (`stored`), the load balance LB (the largest number of evaluations of a worker over their mean) and the communication overhead CO (the nodes sent to another worker over those generated). With `--slow` each evaluation first performs a fixed amount of busy work, as a bound computed by an "expensive" relaxation would; this moves the comparison towards the regime the layer is meant for.

The searches of §13.13 to §13.18 are measured by the same benchmark on the example problems of §13.19, whose options select them: `--domain` chooses the problem (`clique` and `qap` read their instance from `--file`, `flowshop` builds the instances of Taillard given by `--ids`, `uts` the trees given by `--size`), `--algo` the scheme (the four skeletons, `ivm` for the intervals, and, compiled for MPI, `hub` for the master, hubs and workers), `--indexed`, `--spawn`, `--budget`, `--stealall` and `--discrepancy` the options of the skeletons, `--race` and `--racenodes` the race, `--checkpoint` and `--every` the segments, `--cluster` and `--unit` the hubs, `--mode` the mode, and `--prune` the option `prune_level`. Table 13.4 has one part per problem: the clique of the graph `brock200_1` with `prune_level`, the quadratic assignment `nug12` and the flowshop `ta001` with the two-machine bound, each with its optimum plus 1 as the initial incumbent (so that, by Propositions 13.29 and 13.36, the nodes evaluated by the skeletons and by the intervals do not depend on the number of the workers, and the times measure the

exploration alone), and the tree T1 of UTS in the mode `enumerate`. The time is measured as for Table 13.3, and the counter `recomputed` of each run, i.e., the price of the tasks given as paths and of the steals of intervals, is in the column `reexpanded` of the CSV.

Table 13.4. The depth-first schemes on the example problems, in four parts, `clique` (`brock200_1`), `qap` (`nug12`), `flowshop` (`ta001`) and `uts` (T1). In each part a row is a search, i.e., the sequential search with the depth-first selection, the four skeletons with tasks as nodes and as paths, the intervals (for `qap` and `flowshop` only), the shared queue after a race of three selections, and the hubs over MPI, and a column a number of cores; each cell is the time in seconds, the median of two runs, and in parentheses the speedup over the row of the sequential search; an empty cell is a search that does not apply to the problem. The note under each part gives the machine and the A/A noise.

search	1	2	4	8	16	32	48	64	96

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The table is a placeholder until it is measured on an idle machine.

13.21 Notes and references

Branch-and-bound goes back to Land and Doig [146], who solved discrete programs by partitioning the feasible region and bounding each part by the Linear Programming relaxation. The bound of the knapsack used here is the fractional relaxation of Dantzig [147], and the classes of the random instances are among those that Pisinger [148] discusses in his study of which knapsacks are "hard". Nau, Kumar and Kanal [17] gave the general formulation of B&B of which A* and AO* are special cases, which is the view of §13.6; depth-first B&B on a graph is treated with the other memory-bounded searches in Chapter 4. The textbook of Jinnai [164] introduces B&B as a depth-first search pruned by the incumbent (its section 3.4) and gives the recursive DFBnB with the successors sorted by h (its section 6.2, Algorithms 8 and 9); in its Algorithm 9 a successor v of u is tested by $g + h(v) < U$, while its bound is $g + c(u, v) + h(v)$, which prunes more.

For the parallel schemes, the survey of Gendron and Crainic [16] gives the classification by type used in §13.7 and the literature on the anomalies, which Lai and Sahni [149] analyzed early on. The deterministic rounds follow the deterministic mode of FiberSCIP [18], and the SelfSplit is that of Fischetti, Monaci and Salvagnin [19]. The work stealing is that of Blumofe and Leiserson [20], with the steal-half of Hendler and Shavit [36] applied to priority queues. The MultiQueue is that of Rihani, Sanders and Dementiev [37], whose use as a priority scheduler Postnikova et al. [38] studied. Finally, the farm follows the farm with feedback of FastFlow [12], and the end of the search over MPI is detected by the four-counter method of Mattern [11].

With respect to the parallel B&B solvers of Mixed-Integer Programming, the layer is a framework of node-level schemes, and it is behind them in what concerns the problem: it has no cuts, no presolve, no branching rules and no primal heuristics. It leaves to the problem the parallelism inside an evaluation (type 1 of [16]), and racing, i.e., the search of several trees with different settings (type 3), is provided only as the ramp-up of §13.15; also, the SelfSplit does its sampling once instead of in each worker. On the other hand, each scheme runs on any problem that models the concept, and on the three executors, with no change of the program. The deterministic rounds and the SelfSplit give repeatable runs under the stated conditions, the bound reported is valid whatever the status, and the leaves of a search can seed the next one. These properties are checked by the tests of the library, while we have not measured their cost against a specific solver.

The lazy children, the modes and the skeletons of §13.13 and §13.14 follow the search skeletons of Archibald, Maier, Stewart and Trinder [67] [68], whose library separates the search (optimization, decision, enumeration) from the coordination (the depth-bounded, the budget, the stack-stealing and the ordered skeletons), and whose lazy node generators are the cursors of §13.13; the order of the ordered tasks by discrepancy, and the bounds on the runtimes that it gives, are from [67]. The racing ramp-up is that of the Ubiquity Generator framework, which Shinano et al. [69] used to solve open instances of MIPLIB with ParaSCIP on up to 80,000 cores, and the master, the hubs and the workers, with the balance of the quantity and of the quality of the work, are those of ALPS [70]. The intervals of leaves are the Integer-Vector-Matrix of Mezmaz, Leroy, Melab and Tuyttens [71], whose work stealing of private structures Gmys et al. [72] studied on multi-core processors, both on the

permutation flowshop. The example problems are classical: the colouring bound of the clique is that of Tomita and Kameda [74], computed on bitsets as San Segundo, Rodríguez-Losada and Jiménez [75] do; the bound of the common subgraph is that of McSplit [76]; the bound of the quadratic assignment is that of Gilmore [77] and Lawler [166], on the instances of QAPLIB [78]; the bounds of the flowshop are those of Lageweg, Lenstra and Rinnooy Kan [79], on the instances of Taillard [80]; and the unbalanced tree search is the benchmark of Olivier et al. [73]. The bounds of the examples are the classical ones, and we make no claim on their speed w.r.t. the solvers dedicated to these problems.

13.22 Exercises

Exercise 13.1 (`ctest -R PBeFS.bb`). Run Algorithm 13.3 by hand on the example of Figure 13.1 with the breadth-first and the best-estimate selections, and check the orders of Figure 13.4. Then prove that the value returned is the same for each selection, and explain why the numbers of evaluations differ; the test checks the optimum of each selection, in both senses, against the dynamic program.

Exercise 13.2 (`ctest -R PBeFS.bb`). With `shared_incumbent` returning the optimum, the sequential search returns the status `infeasible`, no node, a bound at most the optimum, and as many evaluations as with `incumbent` set to the optimum. Explain each of the four facts from Algorithm 13.3 and Theorem 13.6, and say what changes when the value returned by `shared_incumbent` is worse than the optimum.

Exercise 13.3 (`ctest -R PBeFS.bb-values`). Show that with an integer `value_type` and bounds rounded towards the inside the test PRUNABLE prunes, besides the nodes that the double values prune, those whose bound is within less than 1 of the incumbent. Then explain why this does not imply that each instance needs fewer evaluations with integer bounds (hint: the rounding creates ties among the keys of best-first), and why the test compares the evaluations summed over its instances.

Exercise 13.4 (`ctest -R PBeFS.bb-shared-thread`). Suppose that in Algorithm 13.7 a worker lowered `busy` right after PROCESS, outside the lock, and pushed the children afterwards. Give an interleaving of two workers that ends the search with a node still to be processed, and show which statement of Proposition 13.15 fails. The test checks the algorithm as written: with 1 to 8 workers and every selection it ends with the optimum of the dynamic program.

Exercise 13.5 (`ctest -R PBeFS.bb-mq-thread`). In Algorithm 13.8, suppose that m were raised by $|K|$ after the pushes and lowered by 1 for the node processed before them. Show that $m = 0$ could then be read while a node is in a queue, and that the search could end with a suboptimal value. The test checks the algorithm as written, which ends with the optimum with 1 to 8 workers and every selection.

Exercise 13.6 (`ctest -R PBeFS.bb-det-thread`). The test checks that, with rounds of 4 nodes, the outcome on the knapsack is the same with 1, 2, 3 and 8 workers and at each repetition; explain why by Theorem 13.18, whose first assumption the knapsack satisfies. Then give an evaluator whose outcome depends on the nodes its worker evaluated before (e.g., one that reuses the bound of its previous node when it is the parent), and show on a "small" tree that the rounds then evaluate different nodes with $p = 1$ and $p = 2$, while each p repeats exactly; this is the weaker form of the theorem, which the test does not cover.

Exercise 13.7 (`ctest -R PBeFS.bb-cutoff`). Explain why a worker of the deterministic rounds must take the incumbent from `cutoff( z )` rather than from a value it shares with the other workers, by an example in which the relaxation of a node stops early against a value that another worker has just found. The test checks, with workers that find infeasible each node whose bound cannot exceed the z they receive, that the rounds give the same value, bound and numbers of evaluated and infeasible nodes with 1 to 8 workers.

Exercise 13.8 (`ctest -R PBeFS.bb-split-thread`). Prove that the SelfSplit with one worker evaluates exactly the nodes of the sequential search, and that with p workers and without `split_shared` two runs evaluate the same nodes; the test checks both on the knapsack. Then show, on a tree in which the optimum is in the subtree of the last worker, how many nodes the other workers evaluate without the dive and with it (a question that the test does not cover).

Exercise 13.9 (`ctest -R PBeFS.bb-scheme`). Write a program that reads the name of a scheme from its command line and solves a knapsack by `parallel_branch_and_bound()`, first from the root and then, with three quarters of the capacity, from the leaves of a search carried out to the end on the original capacity; explain, by Proposition 13.24, which leaves it has to give back and with which bounds. The test does both with each named scheme, giving back every leaf with an unknown bound.

Exercise 13.10 (`ctest -R PBeFS.bb-steal-thread`). Show that the counter C of Proposition 13.22 must count a thief from before its exchange, by an interleaving in which a thief that entered C after a successful exchange lets another worker see $C = 0$ while the stolen nodes are open. The test checks the scheme as written, which ends with the optimum with 1 to 8 workers and every selection.

Exercise 13.11 (`ctest -R PBeFS.bb-modes`). Show that in the mode `decide` the status `infeasible` of a search stopped by a node limit says nothing on the existence of a value within the target, while the same status without a limit does (Proposition 13.26). Then explain why the mode `enumerate` counts each feasible node of the knapsack once with each of the eleven named schemes, and why the SelfSplit must skip its dive in that mode; the test checks the count against a recursion on `evaluate` and `branch`.

Exercise 13.12 (`ctest -R PBeFS.bb-cutoff`). The searches call `cutoff( z )` with $+\infty$ in the modes `decide` and `enumerate`. Give an evaluator that fences each node by the z it receives, as the one of the test does, and show that with the incumbent passed in the mode `enumerate` it would make the count of Proposition 13.26 depend on the order in which the feasible nodes are found. The test checks the fencing in the mode `optimize` only.

Exercise 13.13 (`ctest -R PBeFS.bb-skel-thread`). Give an interleaving of a thief and of its victim in which the thief withdraws its request by a plain store of "free", instead of the exchange from "asked by j", and the answer of the victim is lost with its tasks in it; show which claim of Proposition 13.28 fails, and that π never returns to 0. The test checks the protocol as written, with the four rules, with 1, 2, 3 and 8 workers and with `steal_all` for 3 workers.

Exercise 13.14 (`ctest -R PBeFS.bb-skel-thread`). With tasks as paths and an evaluator without `child( n , k )`, show that the rebuild of a task of depth l below the root adds l to `recomputed`, and that a skeleton with `depth_bounded` and spawn depth d adds at most $\sum_{l=1}^{d} l\,T_l$, T_l being the tasks of depth l. Then explain why on the tree of UTS, whose evaluator gives `child( n , k )`, `recomputed` stays 0 and `evaluated` equals the number of the nodes with paths too, as the test checks.

Exercise 13.15 (`ctest -R PBeFS.bb-skel-thread`). Prove that `prune_level` is valid for `MaxClique` (Lemma 13.38) and not for `MaxCommonSubgraph`, by a node of the latter whose second child has a larger bound than its first. The test checks, on a random graph of 60 vertices, that the clique is the same with and without the option and that the option builds fewer children.

Exercise 13.16 (`ctest -R PBeFS.bb-race`). Show that with zero tolerances the closed leaves of the winner of a race never lower the bound of line 8 of Algorithm 13.16 below the final incumbent, and give a case with a positive relative tolerance in which they do, so that the scheme from the open leaves alone would report a bound that is not valid. The test checks, with zero tolerances, that the bound is the optimum with budgets of 5, 50 and 100000 nodes before each named scheme.

Exercise 13.17 (`ctest -R PBeFS.bb-checkpoint`). Suppose that a run is stopped by the system while `bb_write_checkpoint()` writes its file. Show that the next run starts from the checkpoint of the previous segment, and that by Proposition 13.32 it loses at most the evaluations of one segment. Then explain why the counters of the resumed run are at least those of the stopped one, as the test checks with segments of 6 evaluations and a node limit of 20.

Exercise 13.18 (`ctest -R PBeFS.bb-hub-mpi`). Show that with `need_work` equal to 0 a donor with one node gets an order, donates nothing and reports, and that the hub may pair it again at once; then prove that the messages of Proposition 13.33 are still finite once every queue is empty. The test runs clusters of 1, 2 and 4 processes with the default `need_work` of 2.

Exercise 13.19 (`ctest -R PBeFS.bb-ivm-thread`). On the radix $(4, 3, 2, 1)$ of the permutations of four elements, compute by the arithmetic of Lemma 13.34 the half of the interval $[N(0, 2, 1, 0), N(4, 0, 0, 0))$ and the first leaf of the second half; list the ancestors that the thief recomputes, and check that each of them has its first leaf before the cut. The test checks that, with the optimum given, the nodes evaluated with 1 to 8 workers are as many as with one (Proposition 13.36).

Exercise 13.20 (`ctest -R PBeFS.bb-domains`). Prove that the two-machine bound of the flowshop is at least the one-machine bound restricted to the two machines k and l, and give a flowshop of 3 jobs on 3 machines in which it is larger. The test checks both bounds and the four branchings against the enumeration of all the permutations, and the optimum 1278 of `ta001`.

Exercise 13.21 (`ctest -R PBeFS.bb-domains`). Prove that the bound of Lemma 13.41 is the cost of the assignment when $d = n - 1$. Then decide whether the bound of a child of `QAP` can be smaller than that of its

parent, and explain why line 20 of Algorithm 13.3, which keeps the larger of the two, makes the answer irrelevant to the correctness. The test checks the published optima of `chr12a`, `had12` and `nug12`.

Exercise 13.22 (`ctest -R PBeFS.bb-values`). The checkpoints and the search over MPI write the values as their bytes. Show that the type `Rational` of the test satisfies the condition of `SerializableBranchAndBound` on the values, and say why its incumbent is read under the lock of `BBIncumbent` and why the MultiQueue refuses it, which the test checks.

Exercise 13.23 (`ctest -R PBeFS.bb-scheme`). The test reads back by `bb_scheme()` the names of the first seven schemes only. Write the check of the four names of the skeletons, and explain why a race before the scheme `ordered` makes the worker 0 order the tasks below the open leaves of the winner rather than below the root.

14. The example domains

[Source: include/PBeFS/astar/SlidingTile.h, SequenceScore.h, Grid.h (with src/Grid.cpp), ExplicitGraph.h, Romania.h, NQueens.h, VacuumWorld.h, WaterJugs.h, MissionariesCannibals.h, TravelingSalesman.h, SlowDomain.h, Pancake.h, TopSpin.h, RubiksCube.h, TowersOfHanoi.h, DimacsGraph.h, RobotNav2D.h; examples/*.cpp, tools/AStarBench.cpp, data/]

This chapter describes the problems that the library ships as models of the concept `Domain` of Chapter 1, on which the searches of the other chapters are run, tested and measured. For each one we give the state and its encoding (with the trade-offs we made in it), the operators and their costs as pseudocode, the heuristic with the proof of its admissibility or consistency (or a counterexample where it fails), the refinements of `Domain` it models with the proofs of the conditions that the searches put on them, its instances, and how the examples and the benchmark `AStarBench` run it. The sliding-tile puzzles (§14.2 and §14.3) and the grids (§14.4) are the usual benchmarks of the parallel searches in the literature; the puzzles are the only domain that models all the refinements, and the grids model all of them but the incremental features. The explicit graphs (§14.5), the problems of the textbook of Russell and Norvig (§14.6, cf. chapter 3 of [13]) and the traveling salesperson problem (§14.7) cover the other shapes of a state space that a search may (and usually does) meet (a tree with dead ends, an irreversible problem, real costs, a heuristic that is admissible but not consistent), and `SlowDomain` (§14.8) makes any of them slow to expand. The permutation puzzles, i.e., the pancakes (§14.9), TopSpin (§14.10) and Rubik's cube (§14.11), and the Towers of Hanoi with four pegs (§14.12) are the benchmarks of the pattern databases of Chapter 5 and of the parallel IDA*; the road graphs of DIMACS (§14.13) are large explicit graphs with a consistent heuristic of their coordinates; the navigation of a square robot (§14.14) is the domain of the searches by edges, whose evaluations are expensive; and the boards of Burns et al. and the other refinements of the grids are in §14.15. The problems that are not domains of the library, the runs of the domains by the benchmark, the notes and the exercises close the chapter (§14.16 to §14.19). We leave the knapsack of the branch-and-bound searches to Chapter 13, and the multiple sequence alignment to Chapter 15.

14.1 The domains and their concepts

Table 14.1 gives, for each domain, the refinements of `Domain` it models; they decide which searches and which work distributions of HDA* can be run on it (Figure 1.5, where each refinement is introduced). A domain built on another one inherits its refinements where the code forwards them (`SequenceScore` forwards none, `SlowDomain` all but `IncrementalFeatureDomain`), and the last column is the concept `DiskDomain` of the search with delayed duplicate detection of Chapter 4, i.e., integer costs and trivially copyable states (DDDAStar.h:136).

Table 14.1. The refinements of `Domain` modeled by each domain: F feature, IF incremental feature, AF abstract feature, AS abstract state, NB nblock, PW pairwise, OP operator, PH perfect hash, DK disk (x modeled, (a) only up to 20 cells, (b) only with 4 moves, (c) only with an integral `Cost`, (d) only up to 20 objects; the cost type of the grids is `int` with 4 moves and `double` with 8; TopSpin, Rubik's cube and the Towers of Hanoi also prune the redundant pairs of moves of the tree searches, and the road graphs of DIMACS are explicit graphs).

domain	cost	F	IF	AF	AS	NB	PW	OP	PH	DK
`SlidingTile< W , H >`	`int`	x	x	x	x	x	x	x	(a)	x
`SequenceScore< W , H >`	`int`									x
`Grid< 4 >`, `Grid< 8 >`	both	x		x	x	x	x	x	x	(b)
`ExplicitGraph< Cost >`	`Cost`	x							x	(c)
`Romania`	`double`	x							x	
`NQueens`	`int`	x								x
`VacuumWorld`	`int`	x		x	x	x	x		x	x

domain	cost	F	IF	AF	AS	NB	PW	OP	PH	DK
`WaterJugs`	`int`	x							x	x
`MissionariesCannibals`	`int`	x							x	x
`TravelingSalesman`	`double`	x							x	
`PancakePuzzle< N >`	`int`	x	x					x	(d)	x
`BurnedPancakePuzzle< N >`	`int`	x	x					x		x
`TopSpin< N , K >`	`int`	x	x					x	(d)	x
`RubiksCube`	`int`	x	x					x		x
`TowersOfHanoi< D >`	`int`	x	x					x	x	x
`RobotNav2D`	`double`						x			

Every domain packs its state into an integer, or into a small trivially copyable record (the board of `NQueens`, the arrays of bytes of the permutation puzzles). A state is thus compared and copied as a word, or as a few, and its hash is the mixing function of SplitMix64 applied to it (§1.5), which is injective on the words of 64 bits (Lemma 1.7), or, for the arrays of bytes, `bytes_hash` of §5.14; this is the representation that Burns et al. [4] recommend, since the state is then at the same time its key and the input of its hash. The declarations by which a program builds the instances are the constructors below (quoted by hand from the headers, without their bodies; `board_type` is an array of `W * H` bytes, `City` an enumeration of the 20 cities and `matrix_type` a vector of vectors of `double`).

```
explicit SlidingTile( const board_type & start ,
                      const board_type & goal = korf_goal() );
explicit SequenceScore( const board_type & start ,
                        const board_type & goal = Base::korf_goal() );
static GridMap GridMap::random( int width , int height , double p ,
                                std::uint64_t seed , int max_cost = 1 );
static GridMap GridMap::read_movingai( std::istream & in );
Grid( const GridMap & map , int sx , int sy , int gx , int gy ,
      int block = 100 , GridNBlocks nblocks = {} ,
      GridCost cost = GridCost::cells , bool cut_corners = false );
explicit Grid( const GridBoard & b , int block = 100 ,
               GridNBlocks nblocks = {} );
ExplicitGraph( std::size_t n , state_type s , state_type t );
Romania( City from = Arad , City to = Bucharest );
explicit NQueens( int n );
VacuumWorld( int n , int start , std::uint32_t dirty );
WaterJugs( std::vector< int > capacity , int goal );
MissionariesCannibals( int m , int c , int b );
explicit TravelingSalesman( matrix_type dist );
SlowDomain( const D & d , unsigned long work );
explicit PancakePuzzle( const state_type & start , unsigned ignore = 0 );
explicit BurnedPancakePuzzle( const state_type & start );
explicit TopSpin( const state_type & start );
explicit RubiksCube( const state_type & start = solved() );
TowersOfHanoi( state_type start , state_type goal );
template< class Cost = std::int64_t >
ExplicitGraph< Cost > read_dimacs_graph( std::istream & in ,
                                         std::uint32_t source = 1 ,
                                         std::uint32_t target = 1 );
RobotNav2D( const NavMap & map , int sx , int sy , int gx , int gy ,
            const NavOptions & o = {} );
```

The example below, compiled and run by the tester of the library (in the case `manual`, `ctest -R PBeFS.manual`), builds the 8-puzzle of figure 3.28 of [13] and a random grid with 4 and with 8 moves, solves them with the sequential A* of Chapter 2, and checks the costs: 26 moves for the puzzle with both OPEN lists, the cost found by the algorithm of Dijkstra (i.e., by A* with $h = 0$) for the grid with 4 moves, and, for the grid with 8 moves, a cost not larger than that with 4.

```
// the 8-puzzle towards the goal with the blank in the cell 0
const SlidingTile< 3 , 3 > p( { 7 , 2 , 4 , 5 , 0 , 6 , 8 , 3 , 1 } );
```

```
const auto sp = astar( p );                    // the buckets: int costs
const auto sh = astar( p , OpenKind::heap );
// a random grid of 300 x 200 cells, each blocked with probability 1/4,
// from the bottom-left to the bottom-right corner
const auto m = GridMap::random( 300 , 200 , 0.25 , 1 );
const Grid< 4 > g( m , 0 , 199 , 299 , 199 );    // 4 moves, int costs
const auto sg = astar( g );
const Grid< 8 > g8( m , 0 , 199 , 299 , 199 );   // 8 moves, double costs
const auto s8 = astar( g8 );
```

14.2 The sliding-tile puzzles

The sliding-tile puzzle on a board of $W \times H$ cells has $WH - 1$ tiles, numbered from 1, and one empty cell, the *blank*; a move slides a tile adjacent to the blank into it, at cost 1, and the goal is a given arrangement of the tiles. In the library, the class template `SlidingTile< W , H >` (`SlidingTile.h:200`) models it for $WH \le 25$ and $W, H \ge 2$, with the aliases `EightPuzzle`, `FifteenPuzzle` and `TwentyFourPuzzle` for the boards of 3×3, 4×4 and 5×5 cells; the goal is by default that of Korf [7] (`korf_goal()`), with the blank in the first cell and the tiles $1, 2, \ldots$ in row-major order.

The state and the moves

A state is packed into one integer, in which the field of β bits of the cell $i = rW + k$ holds the tile in row r and column k (0 for the blank), i.e., the integer $s = \sum_i \sigma(i)\, 2^{\beta i}$ for the board σ. With $\beta = 4$ the boards of up to 16 cells fit a word of 64 bits, and with $\beta = 5$ the larger ones fit a `Word128`, an unsigned integer of 128 bits written on two words of 64 in order to be the same with all compilers (`SlidingTile.h:77`). This integer is at the same time the state, its equality test and the key of its hash. A move exchanges the blank, in the cell b, with the tile t in a cell c adjacent to it: the field of c is cleared and t is written in the field of b, two operations on the word (Figure 14.1).

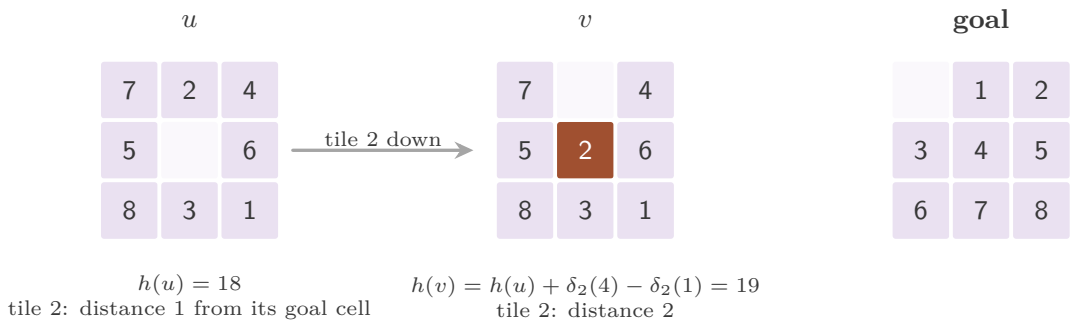


Figure 14.1. A move of the 8-puzzle, on the instance of figure 3.28 of [13] (u, whose optimal solution has 26 moves, towards the goal on the right). Sliding tile 2 down into the blank moves it from distance 1 to distance 2 from its goal cell, and the Manhattan distance of the successor follows from that of u by the change of that tile alone.

The cell of the blank is not stored: it is found from the word by a few bitwise operations (Algorithm 14.1, `SlidingTile.h:697`, and Figure 14.2), in which $L = \sum_i 2^{\beta i}$ is the mask of the lowest bit of each field, $\gg$ the right shift, |, & and $\sim$ the bitwise or, and and complement, and TZ the number of trailing zeros of a word.

Algorithm 14.1 (The cell of the blank).

```
1  procedure BLANK(s)
2     x ← s
3     for k ← 1 to β − 1 do
4        x ← x | (s >> k)        ▷ or the bits of each field into its lowest
5     x ← ~x & L                 ▷ one bit set: the lowest of the zero field
6     return TZ(x) / β           ▷ integer division
```

Lemma 14.1 (The cell of the blank). For every board σ with its blank in the cell b, BLANK(s) returns b, with $\beta - 1$ shifts and ors and one count of the trailing zeros.

Proof. The bit βi of x after line 4 is the or of the bits $\beta i, \beta i+1, \ldots, \beta i+\beta-1$ of s, all of which belong to the field of the cell i (the shifts bring into the higher bits of a field the bits of the next field, but never into its lowest bit, since $k \leq \beta-1$). Hence, the bit βi of x is 0 if and only if the field of i is 0, i.e., if and only if $i=b$, the tiles being nonzero and the blank unique. After line 5, $x=2^{\beta b}$, and its trailing zeros divided by β give b. □

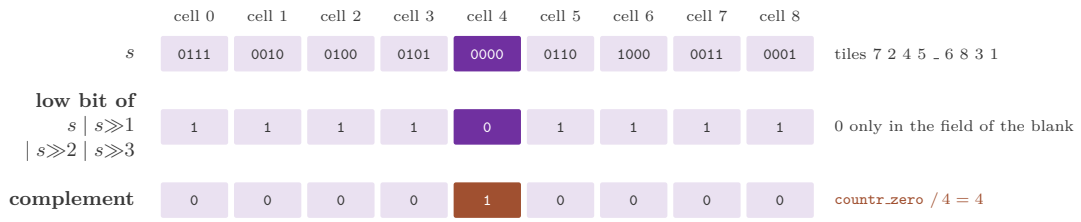


Figure 14.2. The cell of the blank of the instance of Figure 14.1, from its packed word (four bits per cell): the lowest bit of $s \mid s{\gg}1 \mid s{\gg}2 \mid s{\gg}3$ is 0 only in the field of the blank, and the trailing zeros of the complement (masked by the lowest bits) give the cell 4.

The cells adjacent to each cell are precomputed at construction, in the order up, left, right and down (`SlidingTile.h:255`), in a table $N(b)$ of at most 25 entries of 5 integers (hence usually in the cache), and so is the table $\delta_t(c)$ of the Manhattan distance of the cell c from the goal cell of the tile t (`SlidingTile.h:252`, with $\delta_0=0$ for the blank). Also, the domain is an `OperatorDomain` whose operator is the cell of the blank in the parent, and Algorithm 14.2 is its expansion with an operator (`SlidingTile.h:341`); the expansion without an operator is the same without line 5 (`SlidingTile.h:309`).

Algorithm 14.2 (The successors of a board).

```
1  procedure TILESUCCESSORS(s, h(s), o, f)   ▷ o: the cell the blank left
2     b ← BLANK(s)
3     for each cell c ∈ N(b), in the order up, left, right, down, do
4        if c = o then
5           continue                ▷ the move back to the parent
6        t ← the tile in the field of c of s
7        s' ← s with the field of c set to 0 and that of b set to t
8        f(s', 1, h(s) + δ_t(b) − δ_t(c), b)   ▷ b: the operator of s'
```

Line 7 is $s'=(s\& \sim(m \ll \beta c)) \mid (t \ll \beta b)$, with $m=2^\beta-1$ the mask of a field; since the field of b is 0 in s, the or writes t there, and s' is the packed word of the board with t and the blank exchanged. Hence an expansion costs one BLANK and, per successor, a constant number of operations on the word and two lookups of δ, with no loop over the cells; the value of line 8 is the Manhattan distance of s' by Proposition 14.5 below. The inverse of the move from s to s' is the move of the blank of s' (now in c) back to b, i.e., to the operator of s', at the same cost; hence the hypothesis of Lemma 1.8 holds, and line 5 skips only duplicates.

The reachable states

Only half of the arrangements of the tiles can be reached from a given one, and the constructor refuses a start and a goal that are not in the same half (`SlidingTile.h:242`), as well as a board that is not a permutation of $0, \ldots, WH-1$. We now state the invariant, after fixing the notation. For a board σ, let $N(\sigma)$ be the number of *inversions* of the tiles read in row-major order without the blank (the pairs of tiles $t>t'$ with t before t'), and $r(\sigma)$ the row of the blank. The function `parity` (`SlidingTile.h:900`) computes $N(\sigma)$ mod 2 if W is odd and $(N(\sigma)+r(\sigma))$ mod 2 if W is even, by a double loop over the cells, in $O((WH)^2)$ time.

Proposition 14.2 (Parity of the sliding-tile puzzles). If W is odd, $N(\sigma)$ mod 2 is invariant under the moves; if W is even, $(N(\sigma)+r(\sigma))$ mod 2 is. Hence, at most half of the $(WH)!$ boards are reachable from a given one.

Proof. A horizontal move does not change the order of the tiles read in row-major order without the blank, nor the row of the blank. A vertical move takes a tile t over the $W-1$ tiles that lie between its cell and that of the blank in row-major order, and changes the row of the blank by 1. Each of these $W-1$ tiles forms an inversion with t before the move but not after it, or conversely, and therefore N changes by $W-1-2j$ for some j, i.e., by an even number if W is odd and by an odd one if W is even. In the first case N mod 2 is preserved; in the second, N and r change parity together, and their sum keeps it. Finally, the boards on which the invariant takes the other value are half of the $(WH)!$ boards. In fact, exchanging two tiles changes N by an odd number and leaves r unchanged, and it is therefore a bijection between the two halves. □

The converse, that every board on which the invariant takes the value of the start is reachable, is the theorem of Johnson and Story [150] for the 15-puzzle, and it holds for all the boards of at least 2×2 cells; we use it without reproducing its proof. Hence, the 8-puzzle has $9!/2 = 181440$ reachable states, which the test `domains` checks by a search of the whole space, the 15-puzzle $16!/2 \approx 1.05 \cdot 10^{13}$ and the 24-puzzle $25!/2 \approx 7.8 \cdot 10^{24}$.

Lemma 14.3 (The blank is uniform). Assume the converse of Proposition 14.2. Then, for each cell b, exactly $(WH-1)!/2$ reachable boards have the blank in b.

Proof. With the blank fixed in b, the boards are the $(WH-1)!$ arrangements of the tiles on the other cells, and $r(\sigma)$ is the same for all of them; hence the invariant is the parity of N, possibly shifted by the constant r. Exchanging two tiles changes the parity of N, and is a bijection between the arrangements of the two parities, which are therefore $(WH-1)!/2$ each; by the converse, those of the right parity are exactly the reachable ones. □

On the puzzles, the branching factor of a state is the number of the neighbours of its blank, 2 in a corner, 3 on a side and 4 inside the board, and by the lemma its mean over the reachable states has a closed form (Figure 14.3).

Proposition 14.4 (Branching factor of the puzzles). Over the reachable states of the puzzle of $W \times H$ cells, the mean number of moves is $2(2WH - W - H)/(WH)$, i.e., 8/3 for the 8-puzzle, 3 for the 15-puzzle and 16/5 for the 24-puzzle; without the move back to the parent it is 1 less on every state but s_0.

Proof. By Lemma 14.3 the blank is uniformly distributed over the cells, and the mean number of moves is the mean degree of a cell in the grid graph of the board, $2|E|/(WH)$, with $|E| = H(W-1) + W(H-1) = 2WH - W - H$ edges. The three values follow with $|E| = 12, 24, 40$ (cf. Figure 14.3). Skipping the parent removes exactly one successor of each state but s_0, since the cell left by the blank is adjacent to its current cell. □

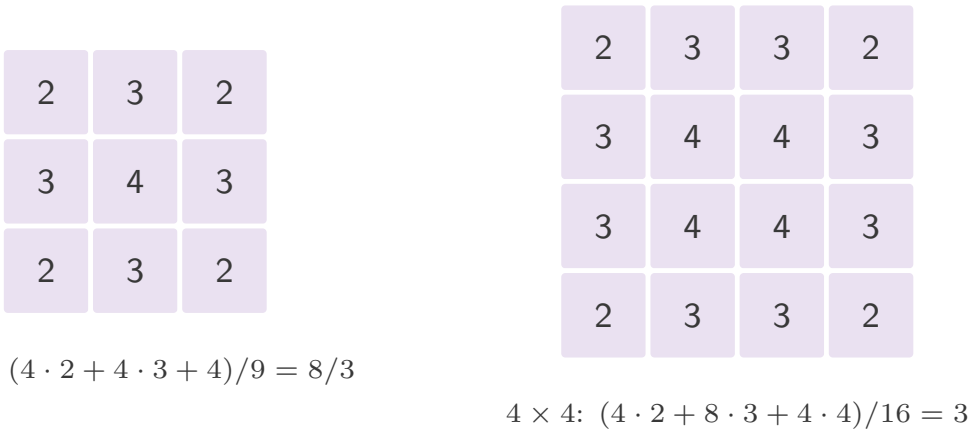


Figure 14.3. The number of moves of the blank in each cell of the boards of 3×3 and 4×4 cells. Since the blank is uniformly distributed over the cells of the reachable states, the mean branching factor is the mean of these numbers (Proposition 14.4).

The Manhattan distance

The heuristic is the Manhattan distance: the sum, over the tiles, of the distance between the cell of the tile and its goal cell in moves of the grid, $h(\sigma) = \sum_{t \neq 0} \delta_t(c_\sigma(t))$, with $c_\sigma(t)$ the cell of t in σ; computed from scratch, by `heuristic( s )`, it takes one lookup per cell.

Proposition 14.5 (The Manhattan distance is consistent). On the sliding-tile puzzle, for each move from σ to σ' of a tile t from the cell c to the cell b of the blank, $h(\sigma') = h(\sigma) + \delta_t(b) - \delta_t(c)$ and $|h(\sigma') - h(\sigma)| = 1$; hence, h is consistent, and therefore admissible.

Proof. Only the tile t changes its cell, which gives the first identity. The cells b and c are adjacent, so that one coordinate changes by 1 and the other not; hence their distances from the goal cell of t differ by exactly 1, since the distance changes by ± 1 in that coordinate. Therefore $h(\sigma) \leq 1 + h(\sigma') = c(\sigma, \sigma') + h(\sigma')$, and h is 0 at the goal; consistency implies admissibility (Lemma 2.4). □

The first identity is how Algorithm 14.2 computes h, two lookups instead of a sum over the tiles, which is the

incremental heuristic of [4]. The second identity has a consequence on the shape of the search.

Corollary 14.6 (The parity of f). On the puzzles with the Manhattan distance, $f = g + h$ changes by 0 or 2 along each edge, and hence the values of f of all the states generated by a search from s_0 have the parity of $h(s_0)$.

Proof. Along an edge, g grows by 1 and h changes by ± 1 (Proposition 14.5), so that f changes by 0 or 2. A generated state has $f = g + h$ for the cost g of a path from s_0, and by induction on the length of the path its parity is that of $f(s_0) = h(s_0)$. □

Hence, iterative deepening raises its bound by exactly 2 at each iteration on the puzzles, since the least f above the bound (Chapter 4) has the parity of the bound. Of course, the pattern databases of Chapter 5 replace this heuristic in `SlidingTilePDB`, which keeps the rest of the domain.

The refinements

The puzzles model all the refinements of `Domain` (Figure 14.4). Their feature $i = tWH + c$ is "tile t is in cell c", for $t \neq 0$ (`num_features()` is $(WH)^2$, of which those with $t = 0$ are never used). A move changes two features, and `for_each_feature_change` (Algorithm 14.3, `SlidingTile.h:420`) finds them without knowing the moved tile, nor the blank.

Algorithm 14.3 (The features changed by a move).

```
1  procedure FEATURECHANGE(s, s', f)
2     d ← s xor s'                ▷ bitwise exclusive or of the two words
3     if d = 0 then
4        return
5     i ← TZ(d) / β              ▷ the first cell where s and s' differ
6     k ← the field of i of d    ▷ the moved tile
7     j ← TZ(d with the field of i cleared) / β
8     f(k WH + i);  f(k WH + j)
```

Lemma 14.7 (The features changed by a move). If s' is a successor of s, Algorithm 14.3 calls `f` exactly on the two features of the symmetric difference of the features of s and s', in constant time.

Proof. Let the move slide the tile k from c to the cell b of the blank of s. The fields of s and s' coincide in every other cell; in b they are 0 and k, and in c they are k and 0. Hence d has the field $k \neq 0$ in the two cells b and c and 0 elsewhere, line 5 finds the lower of the two, line 6 reads k in it, and line 7 the higher one. The features that differ are "k in c" (in s only) and "k in b" (in s' only), the blank having no feature, and these are the two calls of line 8. □

Next, the abstract feature of AZH (`abstract_feature( i )`) keeps of the cell of a tile only whether its row is at least $\lfloor H/2 \rfloor$ (the lower half of the board) or not, which is the projection of figure 8(b) of Jinnai and Fukunaga [5] for the 15-puzzle; there are $2WH$ abstract features, two per tile.

Proposition 14.8 (The moves that change the abstract features). A horizontal move never changes the abstract features of a board; a vertical move changes them (and then two of them) if and only if it moves a tile between the rows $\lfloor H/2 \rfloor - 1$ and $\lfloor H/2 \rfloor$.

Proof. A move changes the cell of one tile only. A horizontal move keeps its row, and hence the half of its cell; a vertical move changes the row by 1, and the half changes if and only if one of the two rows is below $\lfloor H/2 \rfloor$ and the other is not, i.e., if and only if the rows are $\lfloor H/2 \rfloor - 1$ and $\lfloor H/2 \rfloor$; then the abstract feature of the tile in the old half is replaced by that in the new one. □

Hence, under HDA*[AZ], a successor can have an owner other than that of its parent only through the moves across the middle of the board (one pair of rows out of $H - 1$), and the others stay on the worker of their parent; how often such moves occur along a search depends on the instance and is only measured, not bounded, by the counters of Chapter 9. As in Burns et al. [3], the abstract state of AHDA* (`abstract_state( s )`) is the triple of the cells of tiles 1, 2 and 3, coded as an integer.

The nblock of PBNF (`nblock( s )`) is the triple (p_0, p_1, p_2) of the cells of the blank and of tiles 1 and 2, coded as $(p_0WH + p_1)WH + p_2$ among `num_nblocks()` $= (WH)^3$ codes, of which only those of three distinct cells are used. Its abstract successors (`for_each_nblock_successor`, `SlidingTile.h:548`) are the triples reached

by a move of the blank to a cell $c \in N(p_0)$, which moves tile 1 or 2 if it lies in c: they are (c, p_0, p_2) if $c = p_1$, (c, p_1, p_0) if $c = p_2$, and (c, p_1, p_2) otherwise. Safe PBNF requires the two conditions of Definition 11.1.

Proposition 14.9 (The nblocks of the puzzles). The nblocks of `SlidingTile` satisfy Definition 11.1: (i) for each move from u to v, the nblock of v is an abstract successor of that of u, and (ii) the relation of the abstract successors is symmetric and irreflexive.

Proof. (i) A move takes the blank from p_0 to an adjacent cell c and the tile of c to p_0; tile 1 moves if and only if $c = p_1$, and tile 2 if and only if $c = p_2$, which are the three cases of the abstract successors. (ii) The blank changes cell in each case, so that no nblock is its own successor. The successor of (p_0, p_1, p_2) through c has its blank in c, adjacent to p_0, and the move of that blank back to p_0 puts back in p_0 the tile it moved, if any; by the three cases this abstract successor is (p_0, p_1, p_2). □

Also, the pairwise heuristic of PA*SE (`pairwise_heuristic( s , t )`) is the sum over the tiles of the distance between their cells in the two boards, whose properties are proved with those of the grids and of the vacuum world in Proposition 14.18. Finally, the perfect hash (`perfect_hash( s )`) is the lexicographic rank of the permutation of the cells of the tiles $0, 1, \ldots, WH - 1$, i.e., the rank of the pattern databases of Chapter 5 with all the WH tiles; by Lemma 5.4 with $k = n = WH$, it is a bijection from the boards onto $[0, (WH)!)$, and it is given only up to 20 cells, since $20! < 2^{64} < 21!$. Burns et al. [3] used it as a work distribution modulo p, and, as Jinnai and Fukunaga [5] show, modulo p it then depends on very few tiles (Proposition 9.14); it is kept as the poor distribution to compare against.

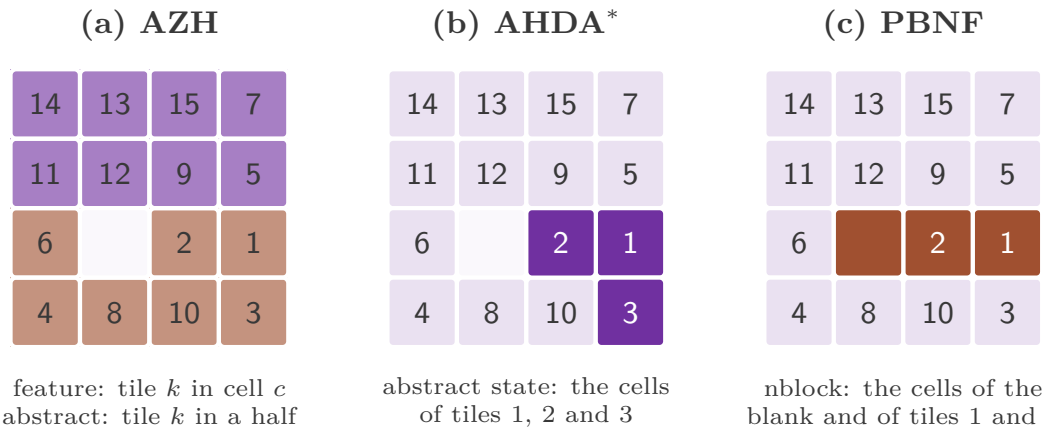


Figure 14.4. The abstractions of the 15-puzzle on the instance 1 of Korf: (a) the abstract feature of AZH, a tile in the upper or in the lower half of the board; (b) the abstract state of AHDA*, the cells of tiles 1, 2 and 3; (c) the nblock of PBNF, the cells of the blank and of tiles 1 and 2.

Instances

The file `data/15-puzzle/korf100.txt` holds the 100 instances of the 15-puzzle of Korf [7], each with its optimal cost, which the tests and the benchmarks use as the reference, and `data/24-puzzle/instances50.txt` the 50 instances of the 24-puzzle distributed with the code of HDA* of Jinnai and Fukunaga. In the examples, `8Puzzle` solves the instance of figure 3.28 of [13], or the first state found at a given distance from the goal, and `15Puzzle` an instance of Korf, sequentially or with HDA*.

14.3 The sequence score of Nilsson

`SequenceScore< W , H >` (`SequenceScore.h:54`) is the same puzzle with the heuristic of Nilsson [8], $h = M + WS$ (i.e., $M + 3S$ on the 8-puzzle), where M is the Manhattan distance and S the *sequence score*. Let next(c) be the cell after c in the *ring*, which is, on the 3×3 board, the cycle of the eight cells of the border in clockwise order (the centre having no next cell), and, on the other boards, the cells $0, 1, \ldots, WH - 1$ in row-major order (the last cell having no next cell). The *follower* fol(t) of a tile t is the tile that the goal puts in the next cell of the goal cell of t or, when that cell has no next cell, in the cell 0 on the 3×3 board and in the cell 1 on the others. The centre c_0 is the cell 4 on the 3×3 board and the cell 0 on the others. Then `heuristic ( s )` is Algorithm 14.4 (`SequenceScore.h:110`).

Algorithm 14.4 (The sequence score).

```
1  procedure SEQUENCESCORE(σ)
2     M ← 0;  S ← 0
3     if σ(c0) ≠ the goal tile of c0 then
4        S ← 1
5     for each cell c with a tile t = σ(c) ≠ 0 do
6        M ← M + δ_t(c)
7        if next(c) exists and σ(next(c)) ≠ fol(t) then
8           S ← S + 2
9     return M + W · S
```

It takes $O(WH)$ time per state, and the successors of `SequenceScore` are those of the puzzle with this value computed from scratch for each of them.

Proposition 14.10 (The sequence score is not admissible). On the 8-puzzle with the goal of Korf, the state one move away from the goal in which tile 1 has moved into the first cell has $h = 13$, while $h^* = 1$.

Proof. Only tile 1 is out of place, at distance 1, and $M = 1$. Since the centre holds tile 4, its goal tile, line 3 adds 0. Along the border (Figure 14.5), tile 1 in the first cell is followed by the blank instead of tile 2, and tile 3 in the cell 3 is followed (in the first cell) by tile 1 instead of the blank, which follows it in the goal; each other tile of the border is followed by its goal follower, and the blank adds nothing (line 5). Hence $S = 4$ and $h = 1 + 3 \cdot 4 = 13 > 1$. □

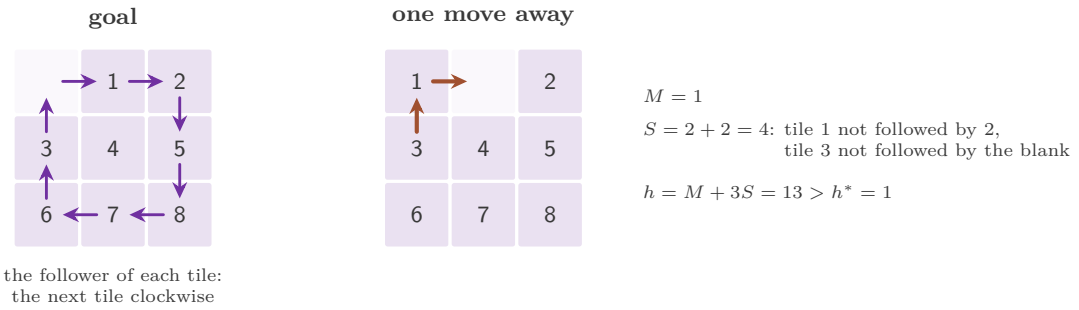


Figure 14.5. The sequence score on the 8-puzzle. Left: the goal, with the clockwise order of the border that defines the follower of each tile. Right: the state one move away of Proposition 14.10, where the two pairs in rust break the order.

Since the heuristic may overestimate, A* with it is not guaranteed (cf. Theorem 2.9) to return an optimal path, and it does not: on the instance of Figure 14.1 (optimal cost 26, with $M = 18$ and $h = 69$), the example `8Puzzle -h seq` returns a path of 28 moves after fewer than a hundred expansions (a run, not a bound; the test `domains` only checks the overestimate of Proposition 14.10). This domain is there to show this trade, which weighted A* makes with a bound (Chapter 6). It forwards no refinement of `SlidingTile`.

14.4 Pathfinding on grids

We call *grid* a rectangular map of $W \times H$ cells, each "blocked" or free with a cost $\kappa(u) \in \{1, \dots, 255\}$ (`GridMap`, `Grid.h:82`, where `cost( x , y )` is 0 for a blocked cell or one outside the map), on which the agent moves from cell to cell. It is the benchmark of the video games and of robotics [15], and many problems reduce to it (the multiple sequence alignment of Chapter 15 is a shortest path on a lattice of higher dimension). In the library, `Grid< Moves >` (`Grid.h:408`) models pathfinding with 4 moves (up, left, right, down) or 8 (the four diagonals too), from the cell (s_x, s_y) to the cell (g_x, g_y), and its constructor throws if either is blocked. A state is the index $yW + x$ of the cell (x, y), which is also the perfect hash. A move *leaves* a cell at its cost, times $\sqrt{2}$ for a diagonal, i.e., $c(u, v) = \kappa(u)$ or $\sqrt{2}\,\kappa(u)$ (written cost and sqrt2 in Algorithm 14.5), and a diagonal move is allowed only if both the cells it cuts are free (Figure 14.6); hence, the costs are integers with 4 moves and real numbers with 8. The successors are generated by a `GridAdjacency< Moves >` (`Grid.h:240`) with the std and Boost data structures, and by the grid graph of the default ones, which generates the same moves at the same costs, in the same order with 4 moves and by increasing index of the cell with 8 (an order that changes only the ties of OPEN); Algorithm 14.5 is the former.

Algorithm 14.5 (The moves on a grid).

```
1  procedure GRIDSUCCESSORS(x, y, f)
2     D ← (0,−1), (−1,0), (1,0), (0,1), (−1,−1), (1,−1), (−1,1), (1,1)
3     for each of the first Moves pairs (δx, δy) of D do
4        (x', y') ← (x + δx, y + δy)
5        if (x', y') is blocked or outside the map then
6           continue
7        if δx ≠ 0 and δy ≠ 0 and ((x', y) or (x, y') is blocked) then
8           continue                ▷ no corner cutting
9        c ← cost(x, y) if δx · δy = 0 else sqrt2 · cost(x, y)
10       f((x', y'), c, h(x', y'))
```

An expansion thus costs $O(\text{Moves})$, the heuristic of each successor being a formula of its coordinates (below).

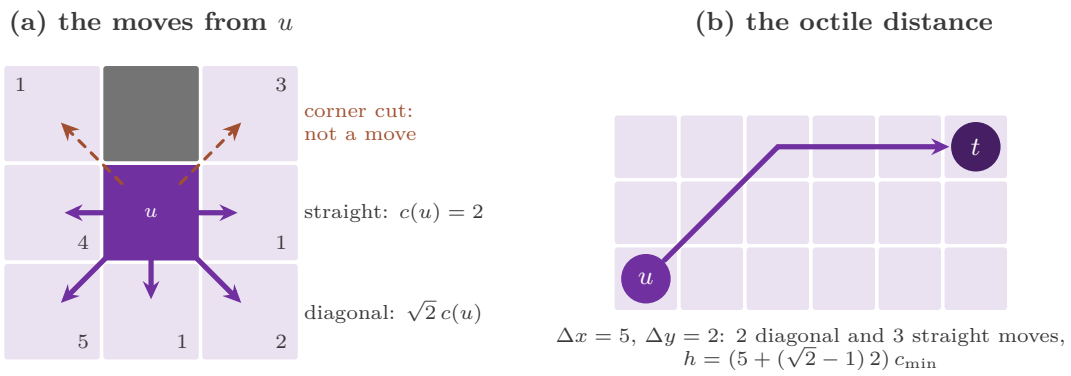


Figure 14.6. (a) The moves from a cell u of cost 2 with the cell above it blocked: the straight moves cost 2, the diagonal ones $2\sqrt{2}$, and the two diagonals that would cut the corner of the blocked cell are not moves. (b) The octile distance between two cells: as many diagonal moves as the smaller difference of the coordinates, then straight moves.

Note that the cost of an edge depends on the cell left: $c(u, v) = \kappa(u)$ and $c(v, u) = \kappa(v)$ differ on a map with several costs. The graph is therefore directed, and the problem is reversible (in the sense of §1.2) only on the maps of uniform cost. A grid is nonetheless an `OperatorDomain`, since the skip of the move back to the parent needs only nonnegative costs (§14.15).

The heuristics

The heuristic is the distance on the empty grid times the least cost $\kappa_{\min}$ of a free cell (`GridMap::min_cost()`, 1 if no cell is free): the Manhattan distance $d_4(u, t) = \Delta x + \Delta y$ with 4 moves, and the octile distance

$$d_8(u, t) = \max(\Delta x, \Delta y) + (\sqrt{2} - 1)\min(\Delta x, \Delta y)$$

with 8 moves, Δx and Δy being the absolute differences of the coordinates of u and of the goal t (`Grid.h:851-854`). Our proof of consistency rests on d_4 and d_8 being the costs of the shortest paths on the grid with no blocked cell and unit cell costs; for d_4 this is immediate (each move changes $\Delta x + \Delta y$ by at most 1, and the straight path attains it), and for d_8 it is the following lemma.

Lemma 14.11 (The octile distance). On the infinite grid with 8 moves, straight moves of cost 1 and diagonal moves of cost $\sqrt{2}$, the least cost of a path between two cells whose coordinates differ by Δx and Δy is d_8.

Proof. A path with k diagonal and l straight moves changes the sum of the coordinates by at most $2k + l$ and each coordinate by at most $k + l$; hence $2k + l \geq \Delta x + \Delta y$ and $k + l \geq \max(\Delta x, \Delta y)$. Its cost is $\sqrt{2}\,k + l = (\sqrt{2} - 1)(2k + l) + (2 - \sqrt{2})(k + l) \geq (\sqrt{2} - 1)(\Delta x + \Delta y) + (2 - \sqrt{2})\max(\Delta x, \Delta y)$, since both coefficients are positive, and this equals d_8 since $\Delta x + \Delta y = \max + \min$. The path of $\min(\Delta x, \Delta y)$ diagonal moves followed by $|\Delta x - \Delta y|$ straight ones attains the bound. □

Proposition 14.12 (The grid heuristics are consistent). On a grid with 4 moves, $h = \kappa_{\min}\,d_4$ is consistent; on a grid with 8 moves, $h = \kappa_{\min}\,d_8$ is consistent.

Proof. Both d_4 and d_8 are shortest-path distances on the empty grid (Lemma 14.11 for d_8), and therefore they satisfy the triangle inequality $d(u, t) \leq d(u, v) + d(v, t)$. For a move from u to a neighbour v, $d(u, v)$ is

1 for a straight move and $\sqrt{2}$ for a diagonal one, while $c(u,v)$ is $\kappa(u) \geq \kappa_{\min}$ times the same factor; hence $h(u) = \kappa_{\min} d(u,t) \leq \kappa_{\min} d(u,v) + \kappa_{\min} d(v,t) \leq c(u,v) + h(v)$, and $h(t) = 0$. Note that the blocked cells and the rule on the corners only remove edges, which keeps the inequality on the remaining ones. □

We remark that the Manhattan distance is *not* admissible with 8 moves, whether a diagonal move costs $\sqrt{2}$ or 1: from $(0,0)$ to $(1,1)$ it gives 2, while one diagonal move costs $\sqrt{2}$ (or 1). Also, with 8 moves the costs and the heuristic are computed in double precision, and Proposition 14.12 holds in exact arithmetic: two paths to the same cell whose exact costs are equal may differ by an ulp, and a search may then (rarely) reopen a cell, which costs time but not optimality (the search of Chapter 2 reopens whenever it finds a smaller g). As a check, the test `grid` compares A* with the algorithm of Dijkstra, written apart from the domain, on random maps of unit costs and of costs 1 to 5, with 4 and 8 moves, and the test `domains` checks the consistency on every edge of three random maps of costs 1 to 5.

Proposition 14.13 (The moves of an open grid). On a grid of $W \times H$ free cells, there are $2(2WH - W - H)$ moves with 4 moves, and $2(2WH - W - H) + 4(W-1)(H-1)$ with 8.

Proof. The straight moves are the two orientations of the $H(W-1)$ horizontal and of the $W(H-1)$ vertical pairs of adjacent cells. With no blocked cell no diagonal is cut, and each of the $(W-1)(H-1)$ squares of 2×2 cells has two diagonals, each in two orientations. □

On 5×5 cells this gives 80 and 144 moves, i.e., the mean branching factors 3.2 and 5.76, which the test `domains` checks.

The refinements and the maps

The features of Zobrist hashing are the column and the row of the cell ($W + H$ features), the abstract features of AZH the column and the row of the block of $B \times B$ cells containing it ($B =$ `block()`, 100 by default), and the abstract state of AHDA* that block, the two distributions coinciding on this domain as in section 4.1 of [5]. In turn, the nblocks of PBNF are the rectangles of `GridNBlocks` (§14.15), by default the blocks of $S \times S$ cells ($S =$ `nblock_side()`, 20), and the abstract successors of an nblock are its 4 (with 4 moves) or 8 (with 8 moves) neighbouring nblocks inside the map (Figure 14.7).

Proposition 14.14 (The nblocks of the grids). The nblocks of `Grid` satisfy Definition 11.1.

Proof. (i) A move changes each coordinate by at most 1, and hence the coordinates of the block, $\lfloor x/S \rfloor$ and $\lfloor y/S \rfloor$, each by at most 1; with 4 moves only one coordinate changes. Thus the nblock of the successor is that of the parent or one of its neighbours of the right kind, both inside the map. (ii) Being a neighbour is symmetric and irreflexive. □

A grid does not model `IncrementalFeatureDomain` (a move changes one or two of its features, but the domain does not enumerate them); its pairwise heuristic for PA*SE is the heuristic between any two cells (Proposition 14.18).

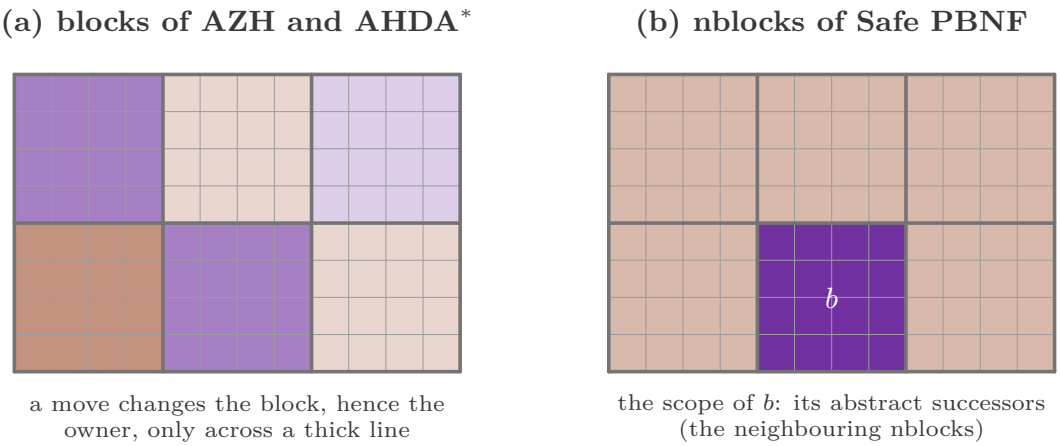


Figure 14.7. (a) The blocks of AZH and AHDA* on a grid, coloured by owner: a move changes the owner only across a thick line. (b) The nblocks of Safe PBNF: the duplicate detection scope of the nblock b is the set of its neighbouring nblocks, which no other worker may use while b is acquired (Chapter 11).

A map is read from a file of the Moving AI benchmarks of Sturtevant [15] by `GridMap::read_movingai` (a header `type octile`, `height`, `width`, `map`, then the rows, in which `.`, `G` and `S` are free cells of cost 1 and any other character is blocked). Otherwise, it is drawn at random by `GridMap::random( W , H , p , seed , c_max )` (`src/Grid.cpp:53`), which blocks each cell with probability p and gives each free cell the cost 1 (if $c_{\max} = 1$) or a cost drawn uniformly in $\{1, \dots, c_{\max}\}$, row by row. Then it makes the bottom-left and the bottom-right corners free with cost 1, since they are the start and the goal of the instances of Burns et al. [3]. The draws are made from the words of `std::mt19937_64` (Lemma 1.9), and hence a seed gives the same map on all platforms. In the examples, `FindPath` finds and draws a path on such a map.

14.5 Explicit graphs and the map of Romania

`ExplicitGraph< Cost >` (`ExplicitGraph.h:144`) is a weighted graph given by its arcs, with nodes $0, \dots, n-1$: `add_edge( u , v , c )` adds an edge (or an arc, with `directed`), refusing a negative cost, `set_heuristic( u , h )` gives h node by node (0 by default), `set_source( s )` changes the source, and the search goes from the source to the target. It serves the small graphs of the tests (e.g., the graph of Figure 1.3) and the map of Romania; since h is given, its admissibility and consistency are the responsibility of whoever gives it. Its arcs are kept in an `ExplicitAdjacency` of the chosen set of data structures (Chapter 3), which enumerates the arcs that leave a node in the order of their insertion with the std and the default sets, but not with that of Boost; the default set also drops a loop (u, u), which never shortens a path. Its single feature is the node, and its perfect hash the node itself.

`Romania` (`Romania.h:61`) is the road map of figure 3.2 of Russell and Norvig [13], with 20 cities, 23 roads whose cost is their length in km, and real costs. Towards Bucharest the heuristic is the straight-line distance of figure 3.22 of the book; towards any other city t it is the Euclidean distance $e(u, t)$ between the coordinates of the cities in the code that accompanies the book (aima-python), both written into the graph by the constructor (`Romania.h:83`).

Proposition 14.15 (The heuristics of Romania are consistent). On the map of Romania, the straight-line distance to Bucharest of the book and the Euclidean distance between the coordinates of aima-python, towards any city, are consistent.

Proof. For the Euclidean distance e, the triangle inequality gives $e(u, t) \le e(u, v) + e(v, t)$. Also, $e(u, v)$ is at most the length of the road (u, v) for each of the 23 roads (the largest ratio of $e(u, v)$ to the length of the road is about 0.996, on the road from Rimnicu Vilcea to Pitesti, on the data of `Romania.h`); hence $h(u) \le c(u, v) + h(v)$, and $h(t) = 0$. However, the table of the book is not derived from the coordinates, and its consistency is checked edge by edge: for each of the 23 roads (u, v), $|h(u) - h(v)| \le c(u, v)$, the least slack being 1 km, on the road from Pitesti to Bucharest ($100 - 0 \le 101$), and the next one 4 km, from Rimnicu Vilcea to Pitesti ($193 - 100 \le 97$). Since the roads are two-way, the inequality holds in both directions. □

Of course, the proof is a finite check, which the test `domains` performs on each edge towards every city, and the test `graph` checks what matters to the searches: A* finds the 418 km from Arad to Bucharest of the book, through Sibiu, Rimnicu Vilcea and Pitesti (Figure 14.8), and the distance to Bucharest from each city, against a table computed apart. Also, the example `RouteFinding` finds a road between any two cities (e.g., `RouteFinding Lugoj Neamt`).

14.6 Four problems of the textbook

The next four domains are problems of chapter 3 of Russell and Norvig [13], each generalized by a size, by which it can be made hard enough to measure a parallel search. They are small in code, and each shows one shape of state space that the searches must handle.

The n-queens problem

`NQueens` (`NQueens.h:67`) places $1 \le n \le 32$ queens on an $n \times n$ board in the incremental formulation of section 3.2.1 of the book. A state is the number k of queens placed and the row row(c) of the queen of each column $c < k$ (a record of 33 bytes), and a move, Algorithm 14.6 (`NQueens.h:124`), adds a queen in the column k, in a row that no queen already on the board attacks, at cost 1.

Algorithm 14.6 (The successors of a placement).

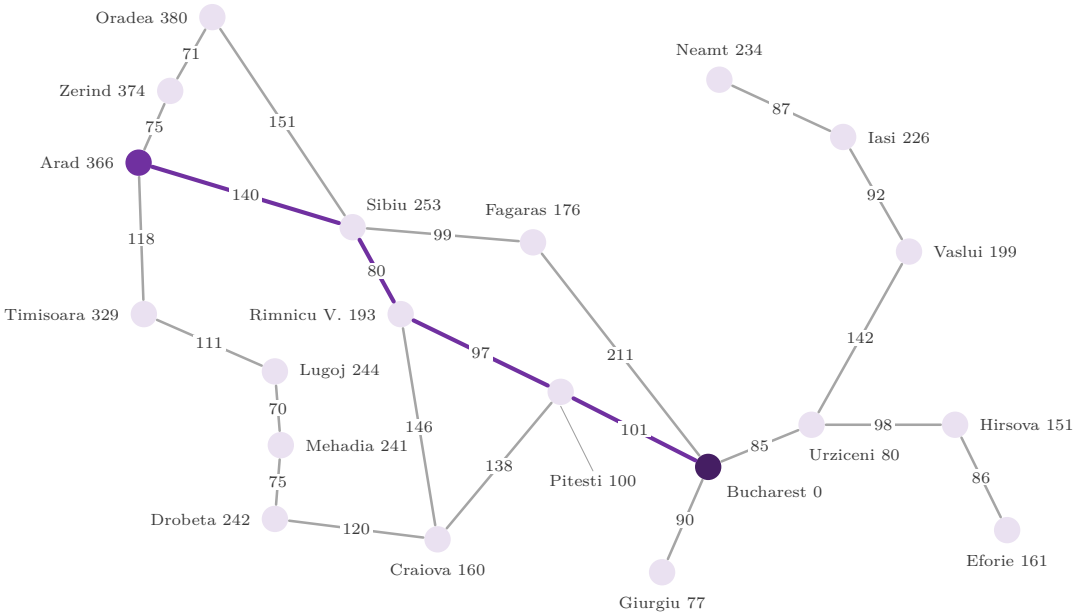


Figure 14.8. The map of Romania of [13], with the length of each road in km and, next to each city, its straight-line distance to Bucharest; the path returned by A* from Arad to Bucharest (violet) costs $140 + 80 + 97 + 101 = 418$.

```
1  procedure QUEENSUCCESSORS(k, row, h, f)
2     for r ← 0 to n − 1 do
3        if r ≠ row(c) and |r − row(c)| ≠ k − c for each column c < k then
4           f((k + 1, row with row(k) = r), 1, h − 1)
```

The test of line 3 excludes the rows and the two diagonals of the queens placed; hence an expansion takes $O(nk)$ time. The state space is a tree whose paths from the root have length at most n, since each state has a single parent (the placement without its last queen), and the goals are the states with $k = n$; a state of the tree may have no successor, a dead end (Figure 14.9). The features of Zobrist hashing are the pairs (column, row) of the queens placed, and the domain has no perfect hash.

Proposition 14.16 (The heuristic of the queens). The heuristic $h = n - k$, the number of the empty columns, is consistent, with $h(u) = c(u, v) + h(v)$ on each edge, and it equals h^* on the states from which a goal is reachable.

Proof. Each edge adds one queen at cost 1, so that h drops by exactly 1, and $h = 0$ at a goal. A state with k queens from which a goal is reachable needs exactly $n - k$ more moves to reach it, each move adding one queen. □

Hence $f = n$ on all the states. The ties of f are broken by the larger g (Chapter 3), and A* then expands the states in a depth-first order, backtracking at the dead ends; the counters of the example `NQueens` measure how many it meets. The 8-queens problem has 92 solutions, and its tree 2057 states (the root included), two counts that the test `domains` obtains by enumerating the tree; for $n = 2$ and $n = 3$ the problem has no solution, and A* reports it after exhausting the tree.

The vacuum world

`VacuumWorld` (`VacuumWorld.h:68`) is the vacuum world of section 3.2.1 of the book on a row of $1 \le n \le 32$ cells instead of 2: a robot moves `Left` or `Right`, or `Sucks` the dirt of its cell, each action costing 1, and the goal is a clean row. A state packs the set D of the dirty cells in the bits 0 to 31 of a word and the cell r of the robot in the bits 32 to 37, and the word is also the perfect hash; Algorithm 14.7 is its expansion (`VacuumWorld.h:128`).

Algorithm 14.7 (The successors in the vacuum world).

```
1  procedure VACUUMSUCCESSORS(r, D, h, f)
2     if r ∈ D then
3        f((r, D \ {r}), 1, h − 1)      ▷ Suck
```

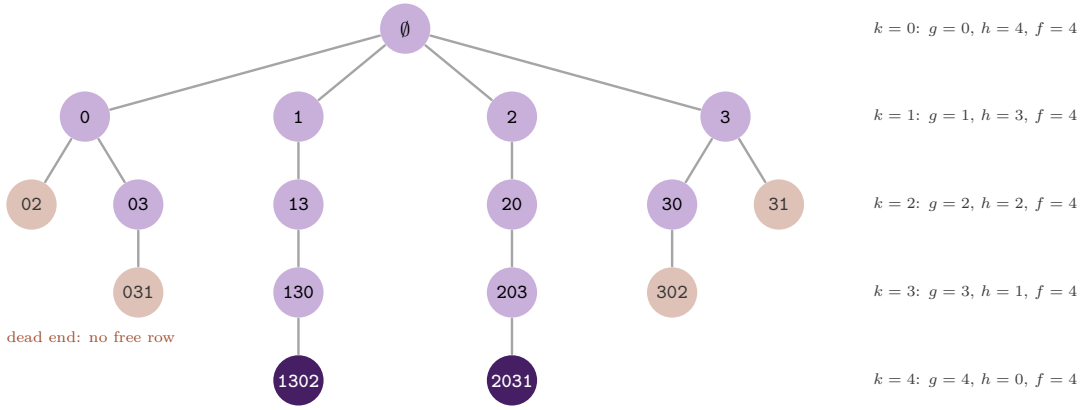


Figure 14.9. The tree of the 4-queens problem, each state labelled by the rows of the queens placed, column by column. The dead ends (rust) have no free row in the next column; the two goals are the two solutions, and all the states have $f = 4$.

```
4     if r > 0 then
5        f((r − 1, D), 1, h)            ▷ Left
6     if r + 1 < n then
7        f((r + 1, D), 1, h)            ▷ Right
```

Proposition 14.17 (The vacuum world). From a start with all the cells dirty, all the $n2^n$ states are reachable; the heuristic $h = |D|$ is consistent; and the nblocks of PBNF, i.e., the cells of the robot with the neighbouring cells as abstract successors, satisfy Definition 11.1.

Proof. A state (r, D) is reached from $(r_0, \{0, \ldots, n-1\})$ by walking along the row and sucking every cell not in D, and then by walking to r. A `Suck` lowers h by 1 at cost 1 and a move leaves it unchanged, so that $h(u) \le c(u, v) + h(v)$ on each edge, and $h = 0$ at a goal. Finally, an action keeps the cell of the robot or moves it to a neighbour, and being neighbours is symmetric and irreflexive. □

The heuristic is also "weak", since it ignores the moves needed to reach the dirt, which usually cost about as much as the cleaning. Furthermore, the domain models the other abstractions of the parallel searches, all of which keep the dirt and forget the robot. Thus, the features of Zobrist hashing are the dirty cells $i < n$ and the cell $n + r$ of the robot, the abstract features of AZH are the dirty cells (the n features of the robot being mapped to one abstract feature, which all the states have), and the abstract state of AHDA* is D; hence, a move keeps the owner of a state under these two distributions, and only a `Suck` may change it.

The pairwise heuristic of PA*SE between s and t, `pairwise_heuristic( s , t )`, is the number of the cells dirty in s and clean in t. We now prove the properties (H1) and (H3) that PA*SE requires (stated in Chapter 8, where PA*SE is given) for the pairwise heuristics of the three domains that have one, $h(t, s)$ being the pairwise heuristic and $d(t, s)$ the least cost of a path from t to s.

Proposition 14.18 (The pairwise heuristics). For the sliding-tile puzzles with the Manhattan distance, the grids with $\kappa_{\min} d_4$ or $\kappa_{\min} d_8$ and the vacuum world with $|D|$, the pairwise heuristic satisfies (H1) $h(t, s) \le d(t, s)$ and (H3) $h(t) \le h(t, s) + h(s)$, for all states t and s.

Proof. On the puzzles, $h(t, s) = \sum_{k \ne 0} \mathrm{dist}(c_t(k), c_s(k))$, with dist the Manhattan distance between cells; a move changes the cell of one tile by one step, and hence $h(t, \cdot)$ by at most 1, which gives (H1), and the triangle inequality of dist, tile by tile, gives (H3), since $h(t)$ is the same sum with the goal in place of s. On the grids, $h(t, s) = \kappa_{\min} d(t, s)$ for $d = d_4$ or d_8: (H1) is the argument of Proposition 14.12 with s in place of the goal, and (H3) is the triangle inequality of d. On the vacuum world, $h(t, s) = |D_t \setminus D_s|$: each cell of $D_t \setminus D_s$ needs a `Suck` on any path from t to s, which gives (H1), and $|D_t| \le |D_t \setminus D_s| + |D_s|$, which is (H3). □

The water jugs

`WaterJugs` (`WaterJugs.h:67`) is the problem of exercise 3.6(d) of the book (the `PourProblem` of aima-python). Up to 4 jugs of capacities $\gamma_j \le 65535$ units are empty at the start, a move fills a jug from the faucet, empties it on the ground, or pours it into another jug until either the first is empty or the second is full, and the goal is a jug holding a given quantity q (Figure 14.11). Every move costs 1, and the moves that do not change the

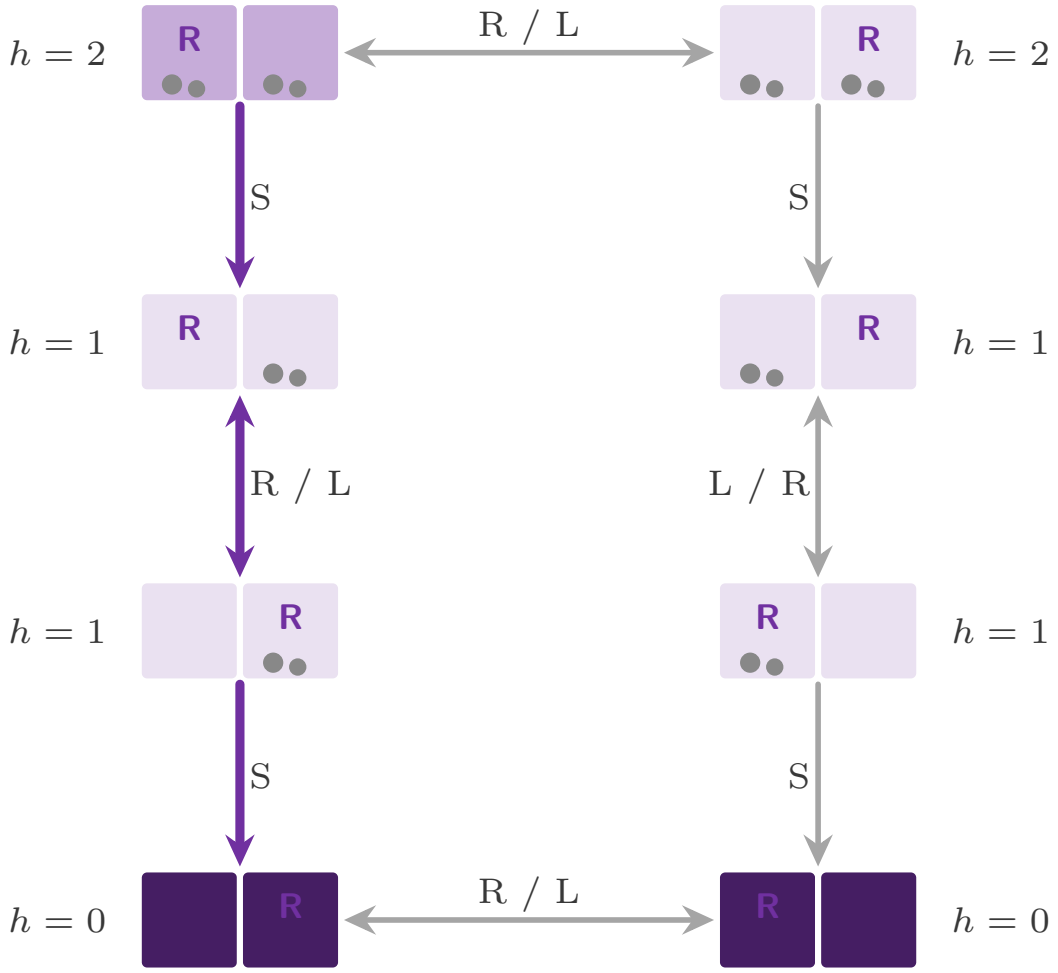


Figure 14.10. The state space of the vacuum world of two cells of the book: each state is the row with the robot (R) and the dirt (dots), the edges are the actions, and the goals (dark violet) are the clean rows. The path returned by A* (violet) is Suck, Right, Suck.

state are not generated (Algorithm 14.8, WaterJugs.h:133, where L is the vector of the levels). In the state, the level L_j of the jug j is packed in the bits $16j$ to $16j + 15$, and the word is also the perfect hash; the features of Zobrist hashing are the pairs (jug, level).

Algorithm 14.8 (The successors of the water jugs).

```
1  procedure JUGSUCCESSORS(L, f)
2     for each jug i do
3        MOVE(L with L_i = γ_i)                          ▷ fill i
4        MOVE(L with L_i = 0)                            ▷ empty i
5        for each jug j ≠ i do
6           δ ← min(L_i, γ_j − L_j)
7           MOVE(L with L_i = L_i − δ and L_j = L_j + δ)    ▷ pour i into j
8  procedure MOVE(L')
9     if L' ≠ L then
10       f(L', 1, h(L'))
```

An expansion generates at most $J^2 + J$ states with J jugs, each with the goal test of $O(J)$ that gives its heuristic, in $O(J^3)$ time.

Proposition 14.19 (The water jugs). The heuristic, 0 at a goal and 1 elsewhere, is consistent. A goal is reachable only if q is a multiple of the greatest common divisor γ of the capacities and $q \leq \max_j \gamma_j$.

Proof. Along an edge of cost 1 the heuristic changes by at most 1, and it is 0 at the goals. For the second claim, each level is a multiple of γ at the start, and lines 3, 4 and 7 keep it so: γ_i, 0, and δ, the minimum of two multiples of γ, are multiples of γ; also, no level exceeds the capacity of its jug. □

The heuristic is (perhaps) the best estimate that does not use the number theory of the capacities; the converse of the condition is not needed by the domain, which leaves an instance that violates it to be proved unsolvable by the search. Note that the problem is not reversible: emptying a jug cannot be undone in one move.

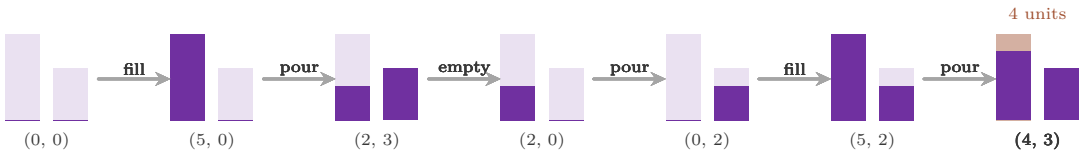


Figure 14.11. Measuring 4 units with jugs of 5 and 3: the six moves of an optimal solution, with the levels of the two jugs under each state.

The missionaries and cannibals

`MissionariesCannibals` (MissionariesCannibals.h:65) is the problem of exercise 3.9 of the book with $m \leq 65535$ missionaries, $c \leq 65535$ cannibals and a boat of $b \geq 1$ seats: everybody starts on the left bank with the boat, a crossing carries from 1 to b people, and on neither bank may the missionaries be outnumbered by the cannibals if any missionary is there. Every crossing costs 1. A state packs the numbers m_L and c_L of the missionaries and the cannibals on the left bank and the bank of the boat, and the word is the perfect hash; the features of Zobrist hashing are m_L, c_L and the bank. Algorithm 14.9 is the expansion (MissionariesCannibals.h:146), where (a_m, a_c) are the people on the bank of the boat and $safe(x, y)$ means $x = 0$ or $x \geq y$.

Algorithm 14.9 (The crossings).

```
1  procedure CROSSINGS(m_L, c_L, boat, f)
2     (a_m, a_c) ← the people on the bank of the boat
3     for d_m ← 0 to min(a_m, b) do
4        for d_c ← (0 if d_m > 0 else 1) to min(a_c, b − d_m) do
5           (m', c') ← the people left on the left bank after the crossing
6           if safe(m', c') and safe(m − m', c − c') then
7              f((m', c', the other bank), 1, h(m', c', the other bank))
```

The heuristic counts the crossings that would carry the $n = m_L + c_L$ people left on the left bank over the river with the boat always full, `heuristic( s )` (MissionariesCannibals.h:129):

$$h_L(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } 0 < n \le b, \\ 2\lceil (n-b)/(b-1) \rceil + 1 & \text{if } n > b > 1, \\ 2n - 1 & \text{if } n > b = 1, \end{cases}$$

with $h = h_L(n)$ if the boat is on the left bank and $h = h_L(n) + 1$ otherwise (for $n > 0$).

Proposition 14.20 (Admissible, not consistent). The heuristic of `MissionariesCannibals` is admissible. It is not consistent: with 3 missionaries, 3 cannibals and a boat of 2 seats, the first crossing of an optimal solution lowers it by 3.

Proof. With the boat on the left, a solution alternates crossings to the right, each carrying at most b people, and crossings back, each bringing at least 1. After k round trips at most $k(b-1)$ people have left, and the last crossing carries at most b; hence $n \le k(b-1)+b$, i.e., $k \ge \lceil (n-b)/(b-1) \rceil$ for $n > b$ and $b > 1$, and the solution has at least $2k+1$ crossings; for $0 < n \le b$ it has at least 1. For $b = 1$ no round trip makes progress, no solution exists when $n > 1$, and $h^* = +\infty$ bounds any value. With the boat on the right and $n > 0$, the first crossing brings back $j \ge 1$ people, and the rest needs at least $h_L(n+j) \ge h_L(n)$ crossings, h_L being nondecreasing; hence $h^* \ge 1 + h_L(n)$, the value of h. For the second claim, the start has $n = 6$ and $h = 2\lceil 4/1 \rceil + 1 = 9$; carrying a missionary and a cannibal (the first crossing of the solution of Figure 14.12) leaves $n = 4$ with the boat on the right, $h = 2\lceil 2/1 \rceil + 1 + 1 = 6$, and $9 > 1 + 6$. □

The proof shows where the slack may be: on the right bank the heuristic adds 1 to $h_L(n)$, while the argument gives $1 + h_L(n+1)$, which is larger by 2 for some $n \ge b$ (for each $n \ge b$ when $b = 2$). With 3, 3 and 2 the answer is 11 crossings, and with 4, 4 and 2 there is no solution, as the test `domains` checks by exhausting the state space. A* is optimal with an admissible heuristic, reopening the closed states when needed (Theorem 2.9); the searches that assume consistency are not. For instance, the search with delayed duplicate detection of Chapter 4 refuses a successor of a smaller f than its parent with an exception (`DDDAStar.h:363`), and it does so on this domain.

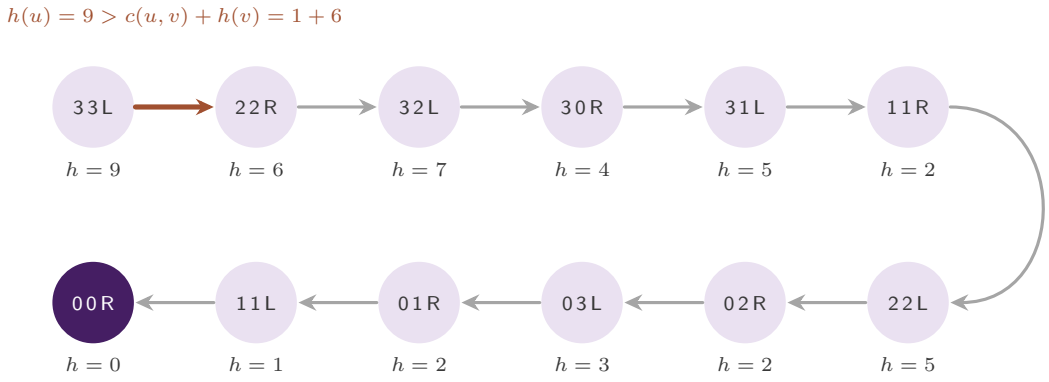


Figure 14.12. The 11 crossings of an optimal solution with 3 missionaries, 3 cannibals and a boat of 2 seats; a state is the numbers of missionaries and cannibals on the left bank and the bank of the boat. The heuristic drops by 3 on the first crossing (rust), which costs 1 (Proposition 14.20).

14.7 The traveling salesperson problem

`TravelingSalesman` (`TravelingSalesman.h:78`) finds a tour of least cost through $2 \le n \le 32$ cities, starting and ending at the city 0, given the matrix d of the distances (`random_cities( n , seed )` draws n points of the unit square and gives their Euclidean distances). A state is the set S of the visited cities and the current one c, packed as S in the bits 0 to 31, c in the bits 32 to 36 and, in the bit 37, whether the tour has gone back to the city 0; the word is also the perfect hash, and the features of Zobrist hashing are the visited cities, the current one and the closing of the tour. The start is $S = \{0\}$, $c = 0$, and Algorithm 14.10 is the expansion (`TravelingSalesman.h:227`).

Algorithm 14.10 (The successors of a partial tour).

```
1  procedure TOURSUCCESSORS(S, c, f)
2     if S = {0, ..., n − 1} then
3        f(the closed tour, d(c, 0), 0)     ▷ the goal
4        return
5     for each city k ∉ S, by increasing k, do
6        f((S ∪ {k}, k), d(c, k), MST((V \ (S ∪ {k})) ∪ {k, 0}))
```

The heuristic of a state u with current city c and set $U = V \setminus S$ of unvisited cities (V being the set of all the cities) is the cost $\mathrm{MST}(U \cup \{c, 0\})$ of a minimum spanning tree of the unvisited cities together with the current and the first one (exercise 3.30 of [13]), and 0 at the goal; it is computed by the algorithm of Prim on the complete graph of these m cities, Algorithm 14.11 (`TravelingSalesman.h:190`), in $O(m^2)$ time.

Algorithm 14.11 (The algorithm of Prim).

```
1  procedure MST(u₁, ..., u_m)
2     key(u₁) ← 0;  key(u_k) ← ∞ for k > 1;  T ← ∅;  w ← 0
3     for it ← 1 to m do
4        x ← the city not in T of least key
5        T ← T ∪ {x};  w ← w + key(x)
6        for each city y ∉ T do
7           key(y) ← min(key(y), d(x, y))
8     return w
```

Lemma 14.21 (The algorithm of Prim). If d is symmetric, Algorithm 14.11 returns the cost of a minimum spanning tree of the complete graph on $u_1, \ldots, u_m$ with the costs d.

Proof. Let the city x of line 4 be joined to the city of T for which its key was last lowered at line 7 (none for u_1); then w is the cost of the tree F so built on T, since $\mathrm{key}(x)$ is the least cost of an edge from x to T. We show by induction on $|T|$ that F is contained in a minimum spanning tree. Let F^* be a minimum spanning tree containing F before x joins, and let $e = (x, y)$, $y \in T$, be the edge added. If $e \notin F^*$, the path of F^* from x to y crosses the cut between T and the rest by an edge e', of cost at least $\mathrm{key}(x)$, since x has the least key among the cities outside T and the key of a city is its least edge to T; replacing e' by e gives a spanning tree of cost at most that of F^*, which contains $F \cup \{e\}$. After m steps F spans all the cities, and it is a minimum spanning tree. □

Its state space is small, as an exhaustive count shows.

Proposition 14.22 (The states of the tour). The states reachable from the start are $(n-1)\,2^{n-2} + 2$: the start, the closed tour, and the pairs of a nonempty set $S' \subseteq \{1, \ldots, n-1\}$ and of a current city $c \in S'$, the visited set being $S' \cup \{0\}$.

Proof. Line 6 adds to S the city it moves to, so that every state but the start and the closed tour has its current city in $S \setminus \{0\}$; conversely, each pair (S', c) is reached by visiting the cities of $S' \setminus \{c\}$ in any order and then c. There are $\sum_{j=1}^{n-1} \binom{n-1}{j} j = (n-1)\,2^{n-2}$ such pairs. □

One can compare this number with the $(n-1)!/2$ tours of a symmetric instance: this is the state space of the dynamic program of Held and Karp [14], and A* on it explores a part of that table, the part selected by the heuristic.

Proposition 14.23 (The MST heuristic is consistent). If the distances are symmetric and nonnegative, the heuristic of `TravelingSalesman` is consistent, and hence admissible.

Proof. Let u have current city c and unvisited set $U \neq \emptyset$, and let v be the successor that moves to $k \in U$; thus, v has current city k and unvisited set $U \setminus \{k\}$, and $h(v) = \mathrm{MST}(U \cup \{0\})$. A minimum spanning tree of $U \cup \{0\}$ together with the edge (c, k) is a spanning tree of $U \cup \{c, 0\}$, of cost $h(v) + d(c, k)$ (if $c = 0$ the edge is not needed); hence $h(u) \leq d(c, k) + h(v) = c(u, v) + h(v)$ (Figure 14.13). When $U = \emptyset$ and $c \neq 0$, the only successor is the goal, at cost $d(c, 0)$, and $h(u) = \mathrm{MST}(\{c, 0\}) = d(c, 0) = c(u, v) + 0$; at the goal $h = 0$. The values of h are those of a minimum spanning tree by Lemma 14.21. □

We remark that the proof needs the symmetry, since a spanning tree is a set of undirected edges; on an asymmetric matrix Algorithm 14.11 still returns a number (it reads d from the tree towards the other cities), but the proposition no longer applies. Admissibility is also seen directly: the rest of a tour from u is a path from c through all the cities of U to 0, i.e., a spanning tree of $U \cup \{c, 0\}$, and it costs at least the minimum one. An expansion costs $O(n^3)$, i.e., one MST of $O(n^2)$ for each of the at most $n-1$ successors; the test `domains`

checks the consistency on every edge of the state spaces of 5 to 9 random cities, and the number of states of Proposition 14.22.

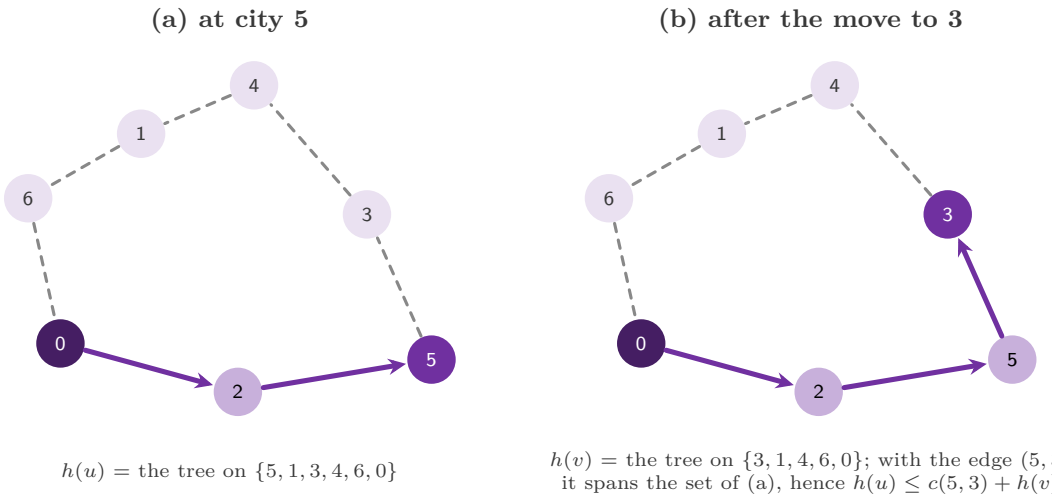


Figure 14.13. The MST heuristic on seven cities. (a) At city 5, after the cities 0 and 2 (violet path), h is the cost of the tree (dashed) on the unvisited cities and on 5 and 0. (b) After the move to 3, the tree on the cities left and on 3 and 0, together with the edge $(5, 3)$, spans the set of (a); hence h drops by at most the cost of the move (Proposition 14.23).

The header also gives the dynamic program of Held and Karp [14], as `held_karp()` (`TravelingSalesman.h:300`), by which the test `domains` and the example `TravelingSalesman` check the cost found by A* (the example up to 20 cities). With $C(S, j)$ the least cost of a path that starts at the city 0, visits exactly the cities of the set $S \ni 0$ and ends at $j \in S$, it rests on the recurrence $C(S \cup \{k\}, k) = \min_{j \in S}\{C(S, j) + d(j, k)\}$ for $k \notin S$, and the optimal tour costs $\min_{j \neq 0}\{C(V, j) + d(j, 0)\}$.

Algorithm 14.12 (The dynamic program of Held and Karp).

```
1  procedure HELDKARP(d, n)
2     C({0}, 0) ← 0;  C(S, j) ← ∞ for each other pair
3     for each set S ∋ 0, by increasing value of its bits, do
4        for each j ∈ S with C(S, j) < ∞ do
5           for each k ∉ S do
6              C(S ∪ {k}, k) ← min(C(S ∪ {k}, k), C(S, j) + d(j, k))
7     return min over j ≠ 0 of C(V, j) + d(j, 0)
```

Proposition 14.24 (Held and Karp). Algorithm 14.12 returns the cost of an optimal tour, in $O(2^n n^2)$ time with a table of $2^n n$ entries.

Proof. The recurrence holds since the last edge of an optimal path that ends at k is some (j, k), and the rest is a path that visits S and ends at j, which may be replaced by an optimal one. Line 3 visits each set after all its subsets, since adding a city raises the value of the bits; hence, when S is visited, every $C(S, j)$ has received at line 6 all the terms of its recurrence, and by induction on $|S|$ it is final. A tour is a path through V ending at some $j \neq 0$, followed by $(j, 0)$, which gives line 7. The table has 2^n sets times n cities, and lines 4 to 6 take $O(n^2)$ per set. □

The time and the memory are why the method serves to check the searches on small instances only.

14.8 Slow expansions

`SlowDomain< D >` (`SlowDomain.h:60`) wraps a domain `D` and adds to each expansion (both `for_each_successor`, the one with an operator included) a given number w of iterations of the computation $i \leftarrow 11i \bmod 9999943$, from $i = 7$ (`SlowDomain.h:151`), which keeps a core busy with no access to memory, as in the experiment of section 3.1.1 of Jinnai and Fukunaga [5] on the effect of slow expansions. It models the problems in which the expansion dominates the cost of a search (planning with an expensive heuristic, or motion planning with collision checks), the regime in which PA*SE of Chapter 8 is meant to pay. The result of the loop is added to a

thread-local variable, which is visible outside the function, and therefore the compiler may not remove the loop.

Proposition 14.25 (A slow domain is the same problem). `SlowDomain< D >` has the states, the initial state, the goals, the edges with their costs, the heuristic, the hash and the refinements of `D` but `IncrementalFeatureDomain`; hence every guarantee of a search on `D` holds on it, and each expansion costs $\Theta(w)$ more.

Proof. Each method of the wrapper calls the method of the same name of `D` with the same arguments, the expansions after the busy loop, which changes no state of the search. □

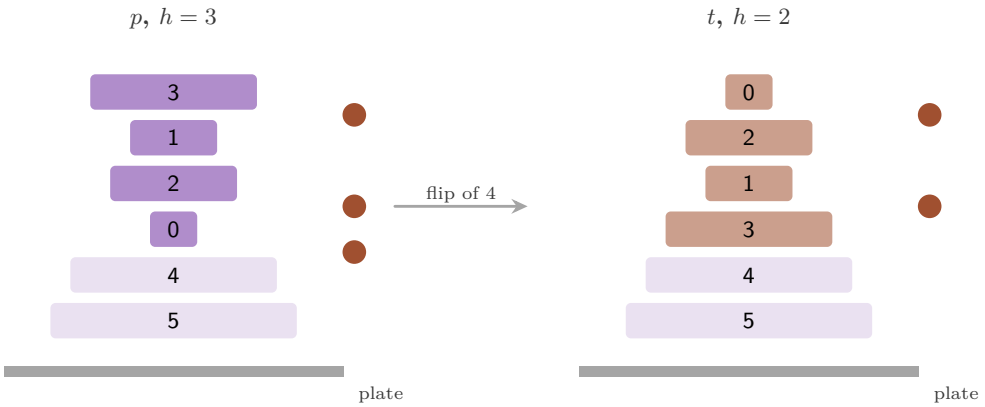
Since `IncrementalFeatureDomain` is not forwarded, the Zobrist keys of the successors of a slow domain are computed from scratch; with an expansion that is slow by construction, this cost is negligible. In the benchmark, the option `--slow` gives w.

14.9 The pancake problems

`PancakePuzzle< N >` (`Pancake.h`, $2 \le N \le 254$) is the pancake problem of Dweighter [63]: a stack of N pancakes of the sizes $0, \dots, N-1$ is an array of N bytes, the size of each pancake from the top, and the flip of k, $2 \le k \le N$, reverses the top k pancakes at cost 1; the goal is the identity, the largest pancake at the bottom. Its constructor takes the start and the x of the heuristic GAP-x (0 for the gap heuristic), and refuses a start that is not a permutation; the heuristic and its properties are those of §5.16 (Proposition 5.21), and the successors receive it incrementally, by the change of the one pair at the boundary of the flip (Figure 14.14). The successors are generated from the flip of N down to the flip of 2 (Algorithm 14.13, which transcribes the code; the header has no pseudocode), the operator of the flip of k is k, and since a flip is its own inverse, at the same cost, the expansion with an operator skips it, which Lemma 1.8 justifies.

Algorithm 14.13 (The flips of a stack).

```
1  procedure FLIPS(p, h, o, f)    ▷ o: the flip that made p (none at the root)
2     for k ← N down to 2 do
3        if k = o then continue
4        b ← p[k] if k < N else the plate
5        h' ← h − gap(p[k − 1], b) + gap(p[0], b)
6        f(p with p[0..k−1] reversed, 1, h', k)
```



dots: the gaps; only the pair at the boundary of the flip changes, from the gap $(0, 4)$ *to the pair* $(3, 4)$

Figure 14.14. The flip of 4 on the stack $p = (3, 1, 2, 0, 4, 5)$ of 6 pancakes, the top first: the block of the flip is reversed, its adjacent pairs stay the same unordered pairs, and only the pair at its boundary changes, from the gap $(0, 4)$ to $(3, 4)$, so that the gap heuristic drops from 3 to 2 (Proposition 5.21).

The features of Zobrist hashing are the N^2 pairs (position, size), and the features changed by a flip of k are found by comparing the two stacks position by position, i.e., in $O(N)$ time; the perfect hash is the lexicographic rank of the stack, by `LexRanking` of §5.14, for $N \le 20$. The domain is also a `PermutationDomain` of one group of N positions without orientations, whose moves are the flips, so that the tables of §5.15 apply to it. A stack is read from a text by `parse( text )`, its sizes from the top separated by blanks, and `random( seed )` draws a uniformly random stack, the same for a seed on every platform, by the shuffle of Fisher and Yates on the words of `std::mt19937_64` (Lemma 1.9).

In `BurnedPancakePuzzle< N >` ($1 \le N \le 127$) each pancake also has a side burned: a pancake is a byte, its size in the low 7 bits and the bit 7 (`turned`) set if the burned side is up, a flip of k, now for $1 \le k \le N$, also turns every pancake it moves (the flip of 1 only turns the top pancake), and the goal is the identity with no pancake turned. The heuristic counts the pairs that are not good (Proposition 5.22), incrementally as above; the features are the $2N^2$ triples (position, size, side), there is no perfect hash, and the domain is a `PermutationDomain` of one group of modulus 2, the flip of k turning the positions $0, \dots, k-1$. In a text, a pancake is its size plus one, negative if it is turned (e.g., `-3 1 -2 5 4`), and `random( seed )` also turns each pancake with probability 1/2.

Proposition 14.26 (The stacks are connected). Every stack of $N \ge 2$ pancakes reaches the goal in at most $2N-3$ flips, and every stack of N burned pancakes in at most $3N$; since the flips are involutions, the $N!$ stacks, and the $N!\,2^N$ burned ones, are all reachable from any of them.

Proof. For $j = N-1$ down to 2, if the pancake j is not at the position j, one flip brings it to the top (none if it is there), and the flip of $j+1$ puts it at the position j, leaving the positions below untouched; the two pancakes left are then sorted by at most one flip of 2, which gives $2(N-2)+1$ flips. With the burned pancakes, for $j = N-1$ down to 0, one flip brings the pancake j to the top, the flip of 1 turns it if it is not turned, and the flip of $j+1$ puts it at the position j with its burned side down, i.e., at most 3 flips for each pancake. A flip being its own inverse, the graph is undirected, and a stack is reachable from another through the goal. □

The bounds are those of the method of the proof, not the diameters of the graphs, which are much harder questions. The test `pancake` checks the $7! = 5040$ stacks of 7 pancakes and the $5!\,2^5 = 3840$ stacks of 5 burned pancakes, all reached from the goal, and the reading of a burned stack. In the benchmark, `AStarBench --dom ain pancake` takes N from `--size` (12, 16, 20, 32 or 64) and the x of GAP-x from `--gap`, `--domain burned` takes $N = 8$, 10, 12 or 16, a random stack for each seed of `--ids`, and `--pdb k` adds the tables of the first k objects, of the next k, and so on.

14.10 TopSpin

`TopSpin< N , K >` (`TopSpin.h`, $2 \le K \le N < 255$, $K = 4$ by default) is the TopSpin puzzle of Felner et al. [167]: N tokens on a ring, and a turnstile that reverses K consecutive tokens. A state is an array of N bytes, the token at each position; the move i, $0 \le i < N$, reverses the tokens at the positions $i, \dots, i+K-1$ modulo N at cost 1, and the goals are the N rotations of the identity, i.e., the tokens in increasing order around the ring wherever the token 0 is. The constructor refuses a start that is not a permutation; the heuristic, $\lceil b/2 \rceil$ for b breakpoints, is that of Proposition 5.23, computed anew for each successor in $O(N)$ time. The operator of the move i is i, its own inverse, which the expansion with an operator skips; for the tree searches, the domain is also a `TreeOperatorDomain` (`Domain.h`), whose `for_each_tree_successor` skips, after the move i, the moves $j < i$ whose turnstiles are disjoint from that of i, since two such moves commute. The test of disjointness is `commu te( i , j )`: with $d = (j-i) \bmod N$, the turnstiles are disjoint if and only if $d \ge K$ and $N - d \ge K$. The features are the pairs (position, token), the perfect hash the lexicographic rank of the ring for $N \le 20$, and the domain is a `PermutationDomain` of one group whose goals are the N rotations; `parse( text )` reads a ring from the position 0, and `random( seed , n )` is a uniform ring if $n = 0$ and the identity after n random moves otherwise. The pruning of the tree searches is an instance of a general rule, which serves also Rubik's cube and the Towers of Hanoi.

Lemma 14.27 (Redundant pairs of moves). Let a `TreeOperatorDomain` skip, after a move a, (i) moves b such that a followed by b has the effect of a single move m, or of none, at a cost not larger than $c(a) + c(b)$, and (ii) moves $b \prec a$ that commute with a, for a strict order $\prec$ of the moves fixed in advance. Then for every path from s to t there is a path from s to t, of a cost not larger, whose consecutive pairs of moves are never skipped; hence IDA* and the parallel IDA*, which enumerate the kept paths only, remain complete and optimal.

Proof. Rewrite the path as long as it has a skipped pair: a pair of type (i) is replaced by the single move m or removed, which shortens the path and does not raise its cost, and a pair of type (ii), a followed by $b \prec a$, is replaced by b followed by a, which reaches the same state at the same cost, has the same length and one inverted pair fewer under $\prec$ (the number of the pairs of positions $p < q$ whose moves satisfy $m_q \prec m_p$ decreases by exactly one by an exchange of adjacent moves). The pair (length, number of inverted pairs) decreases in the lexicographic order at each step, and the rewriting ends with a path that has no skipped pair, from s to t and of a cost not larger than the original one. The searches of the lemma try every kept move at each node, so that they enumerate every path without skipped pairs, among which a cheapest path to a goal. □

For TopSpin, (i) is the repetition of a move, whose effect is none, and (ii) the disjoint turnstiles with the order of their indices. We remark that the searches with a closed list do not use these prunings (`Domain.h`, the comment

of the concept), since the kept path to a state may go through a state closed with another operator; they use only the skip of the inverse, by Lemma 1.8. The test `topspin` checks, on the $8! = 40320$ rings of TopSpin with 8 tokens and a turnstile of 4, all reachable from the goals, the consistency of the breakpoints and of a table of 5 tokens, and IDA* and A* with and without the table against the exact distances. In the benchmark, `AStarBench --domain topspin` takes $N = 12$, 14, 16 or 17 with $K = 4$, the identity after 1000 random moves for each seed.

14.11 Rubik's cube

`RubiksCube` (`RubiksCube.h`) is Rubik's cube in the half-turn metric, in the cubie model of Kociemba [172]: a state is the cubie and the orientation at each of the 8 corner and 12 edge positions, in the order of Kociemba (the corners URF, UFL, ULB, UBR, DFR, DLF, DBL, DRB and the edges UR, UF, UL, UB, DR, DF, DL, DB, FR, FL, BL, BR), stored as an array of 20 bytes that holds $3c + o$ at the corner positions 0 to 7 and $2e + o$ at the edge positions 8 to 19, c and e being the cubies and o their orientations. A move turns a face (U, R, F, D, L or B) by a quarter turn clockwise, a half turn or a quarter turn counterclockwise, at cost 1, and its operator is $3f + t$ for the face f and the turn $t \in \{0, 1, 2\}$. The 18 moves are tables of 20 positions, the permutation from and the turns: the cubie that comes to the position i is the one at $\mathrm{from}[i]$, its orientation grown by $\mathrm{turn}[i]$ modulo 3 (corners) or 2 (edges). They are built once from the six quarter turns of the cubie model by composition (the move a followed by b has $\mathrm{from}[i] = \mathrm{from}_a[\mathrm{from}_b[i]]$ and $\mathrm{turn}[i] = \mathrm{turn}_a[\mathrm{from}_b[i]] + \mathrm{turn}_b[i]$), so that the cube is a `PermutationDomain` of two groups, the corners and the edges, whose orientations sum to 0. The heuristic of the domain is 0, the tables of Korf being given by `RubikPDBHeuristic` and `PDBDomain` of §5.16. The features of Zobrist hashing are the $20 \cdot 24$ pairs (position, value), with their changes, and the domain has no perfect hash. `parse_moves( text )` reads a sequence of moves in the notation of Singmaster (a face, followed by nothing, by 2, or by `-` or `'`), `scramble( text )` applies it to the solved cube (`solved()`), and `random_walk( n , seed )` makes n random moves, each of a face other than the last one.

The expansion with an operator skips the three turns of the face just turned. `for_each_tree_successor` also skips the turns of the face opposite to the one just turned when that face comes second in its pair of opposite faces in the order U, R, F, D, L, B (after D it skips U, after L it skips R, and after B it skips F), since two turns of opposite faces commute. These are the prunings of Korf [59], to which Lemma 14.27 applies: two turns of one face make a single turn of that face or none, and the faces are ordered as above.

Proposition 14.28 (The moves of the cube). (i) Each move has order 4 (a quarter turn) or 2 (a half turn), and the inverse of the move $3f + t$ is $3f + 2 - t$. (ii) The skip of the three turns of the face just turned changes neither the outcome nor the order of the expansions of a search that keeps CLOSED, never increases a g and keeps the expanded states. (iii) With the prunings of the tree searches, the number of the nodes at the depth $d \geq 1$ of the tree is $x_d + y_d$, with $x_1 = y_1 = 9$, $x_{d+1} = 6x_d + 6y_d$ and $y_{d+1} = 9x_d + 6y_d$, i.e., 18, 243, 3240, 43254 and 577368 nodes at the depths 1 to 5, and its asymptotic branching factor is $6 + 3\sqrt{6} \approx 13.34847$.

Proof. (i) The tables of the turns $t = 0, 1, 2$ of a face are the first three powers of its quarter turn, which has order 4 in the cubie model; hence the turn t followed by the turn $2 - t$ is the fourth power, the identity. (ii) Let u be generated from v by a turn of the face f; a second turn of f from u leads to v itself, or to a state w that is a single turn of f away from v, i.e., a sibling of u. In the first case Lemma 1.8 applies. In the second, w is generated by the expansion of v, which precedes that of u, with the cost $g(v) + 1$, and the cost of w known when u is expanded is at most that value, since a g never increases, while the skipped generation would give $g(u) + 1 \geq g(v) + 2$; the generation would thus be a duplicate, which changes nothing but the counters. (iii) Let x_d and y_d count the nodes at the depth d whose last face is the first of its pair (U, R, F) and the second (D, L, B). At the depth 1 there are 9 of each. After a first face, the 5 other faces are kept, 2 first and 3 second ones, with 3 turns each; after a second face, the 4 faces other than it and its opposite are kept, 2 of each kind; this gives the recurrence and the counts. The matrix $\begin{pmatrix} 6 & 6 \\ 9 & 6 \end{pmatrix}$ has the eigenvalues $6 \pm \sqrt{54}$, and the largest one, $6 + 3\sqrt{6}$, is the growth of $x_d + y_d$. □

The counts of (iii) are those of table 1 of Korf [59], who gives the branching factor 13.34847, and the test `rubik` checks them by enumerating the tree to the depth 5; it also counts the distinct states at the depths 1 to 4 by a breadth-first search, 18, 243, 3240 and 43239, i.e., 15 states of the depth 4 are reached by two kept paths. It further checks (i) on a scrambled cube for each move, reads the ten instances of Korf, scrambles of 100 moves in `data/rubik/korf10.txt` (from the code of HOG2 [165], under the MIT license), and checks smaller tables of 4 corners and of 4 and 4 edges by IDA* and the parallel IDA* on random walks of 5 moves. In the benchmark, `AStarBench --domain rubik` solves the scrambles of `--file` or the random walks of `--size` moves, with the

tables of Korf if `--pdb korf`, built or read from `--cache` (with `--pdbinfo 1`, the time to build or load them, their sizes, means and largest values, and the heuristic of each start, on stderr), and `tools/bench/rubik.sh` runs IDA* and the parallel IDA* on 1 to 32 workers on the ten instances of Korf.

14.12 The Towers of Hanoi with four pegs

`TowersOfHanoi< D >` (`TowersOfHanoi.h`, $1 \leq D \leq 32$) places D disks, 0 being the smallest, on four pegs: a state is a word of 64 bits that holds the peg of the disk d in its bits $2d$ and $2d+1$, the disks of a peg being stacked by size, and a move takes the top disk of a peg to another peg whose top disk is larger, or which is empty, at cost 1. Any start and any goal are given to the constructor, which keeps their $2D$ low bits; all the 4^D states are reachable from each other (Proposition 14.29), and the heuristic of the domain is 0, the additive tables of ranges of disks of §5.16 (`HanoiPDB` with `PDBDomain`) being its heuristic. The moves of a state are found from the top disks of the four pegs (`hanoi_tops`), and they are generated by the peg of the disk, then by the peg it goes to; the operator of a move of the disk d from the peg a is $4d + a$. The expansion with an operator skips the move of the disk just moved back to the peg it left, which is the inverse at the same cost (Lemma 1.8). In turn, `for_each_tree_successor` skips all the moves of the disk just moved: two moves of the same disk in a row have the effect of one move of it or of none, and Lemma 14.27 (i) applies. The merged move is legal, since the disk is on top of its first peg and the top disk of its last peg is the same before and after the first move. The features are the $4D$ pairs (disk, peg), and the features changed by a move are those of the one disk that differs, the lowest pair of bits in which the two words differ; the perfect hash is the word itself.

Proposition 14.29 (The moves of the Towers of Hanoi). A state has at most 6 moves, and every state reaches every other one.

Proof. Order the four pegs by their top disks, an empty peg after all the others. The top disk of the i-th peg in this order can only go to the pegs after it, i.e., to at most $3, 2, 1, 0$ pegs for $i = 1, \ldots, 4$, which gives at most 6 moves. For the second claim it suffices, the moves being reversible, to bring any state to any goal γ. By induction on the number of disks: if $D = 1$, one move suffices. With $D > 1$, move the $D - 1$ smaller disks, by the induction (they are a puzzle of their own, and the largest disk never prevents their moves), to a peg that is neither the peg of the largest disk nor its peg in γ (or any other peg if these coincide); then move the largest disk to its goal peg if needed, which is now possible since its peg and the goal peg hold no other disk, and finally move the smaller disks to their goal pegs by the induction again. □

A state is read by `read( in )` in the format of the instances of Siag et al. [57]: the four pegs, each written `(p)` and followed by its disks numbered from 1 (the smallest) to D; it throws `std::invalid_argument` on a bad peg, a bad or repeated disk, or a missing one, and `write( s )` writes a state in the same format, the disks of each peg from the bottom. The file `data/hanoi/toh20.txt` holds their 20 instances of 20 disks, each line the start and then the goal (from the code of their experiments, under the MIT license). The test `hanoi` reads and writes back the 20 instances, and checks the tables on 8 disks (§5.16) with A* and the parallel IDA* on 16 instances; in the benchmark, `AStarBench --domain hanoi` takes $D = 12$, 16 or 20 from `--size`, the ranges of the tables from `--pdb` (e.g., `4+16`, the heuristic of Siag et al.), and with `--reverse 1` searches from the goal to the start.

14.13 The road graphs of DIMACS

The road graphs of the 9th DIMACS Implementation Challenge on shortest paths, edited by Demetrescu, Goldberg and Johnson [64], are given by a file of arcs (`.gr`) and a file of coordinates (`.co`), which `DimacsGraph.h` reads into an `ExplicitGraph` of §14.5. A file `.gr` has the line `p sp n m` and m lines `a u v w`, each the arc (u, v) of the integer cost w, with the nodes numbered from 1. The lines `c` are comments, and both directions of a road are arcs of their own. `read_dimacs_graph< Cost >( in , source , target )` (`Cost` is `std::int64_t` by default) returns the graph, searched from `source` to `target` (numbered from 1, as in the file), with each arc added as a directed one. It throws `std::invalid_argument` if the file has no line `p sp` or one with no node, if the source or the target is out of range, on an arc out of range or of negative cost, and if it holds fewer arcs than its line `p` says. A file `.co` has the line `p aux sp co n` and n lines `v v x y`, the longitude and the latitude of the node in millionths of a degree, which `read_dimacs_coordinates( in , n )` reads into a `DimacsCoordinates`; it throws if the line `p` is missing or is not for n nodes, or on a node out of range, and a node that the file does not list keeps the coordinates $(0, 0)$, which the reader does not check.

```
struct DimacsCoordinates {
 std::vector< std::int32_t > lon;     ///< the longitude of each node
```

```
std::vector< std::int32_t > lat;     ///< the latitude of each node
};
```

The heuristic of the node u for the target t is $h(u) = \lfloor \kappa\, \delta(u,t) \rfloor$, where δ is the great-circle distance in meters, which `dimacs_distance( c , u , v )` computes by the formula of the haversine on the sphere of radius 6371008.8 meters, and

$$\kappa = \min_{(u,v):\delta(u,v)>0} \frac{w(u,v)}{\delta(u,v)}$$

over the arcs between distinct points. `dimacs_scale( g , c )` computes κ, lowered by the factor $1 - 10^{-9}$ against the rounding of the distances (0 if no arc joins distinct points). `set_dimacs_heuristic( g , c , kappa )` writes h into the graph for its current target (with κ computed if `kappa` is negative) and returns κ; after a change of the target by `set_target`, the heuristic has to be set again.

Proposition 14.30 (The heuristic of the coordinates). In exact arithmetic, with integer costs, the heuristic $\lfloor \kappa\, \delta(\cdot,t) \rfloor$ is consistent and 0 at the target, for every target t and every $\kappa \geq 0$ with $\kappa\, \delta(u,v) \leq w(u,v)$ on each arc; the κ above is the largest such constant, whatever the units of the costs.

Proof. The great-circle distance is a metric, so that for an arc (u,v), $\kappa\, \delta(u,t) \leq \kappa\, \delta(u,v) + \kappa\, \delta(v,t) \leq w(u,v) + \kappa\, \delta(v,t)$ (for an arc between two equal points the middle term is $0 \leq w(u,v)$). Since $w(u,v)$ is an integer, $\lfloor w(u,v) + z \rfloor = w(u,v) + \lfloor z \rfloor$, which gives $h(u) \leq w(u,v) + h(v)$, and $h(t) = 0$. A larger constant violates $\kappa\, \delta(u,v) \leq w(u,v)$ on the arc of the minimum. □

The costs are lengths in meters for the graphs USA-road-d of the challenge and travel times in tenths of a second for USA-road-t, and κ is then the least ratio of the length of a road to the distance of its ends for the former, and the inverse of the largest speed of a road for the latter; the proposition holds for both, as long as the data satisfy it. In double precision the computed distances satisfy the triangle inequality only up to the rounding, and the factor $1 - 10^{-9}$ is a margin for $\kappa\, \delta(u,v) \leq w(u,v)$, not a proof of consistency, which we do not claim in floating point; the test `dimacs` checks it on every arc of the sample graph of `data/dimacs`, for 5 random pairs of a source and a target, and checks A* against the algorithm of Dijkstra on them. The sample is a synthetic graph of 900 nodes and 3106 arcs, a perturbed grid of 30×30 points in the format of the challenge, written by `tools/bench/dimacs_sample.py`, while `tools/bench/fetch_dimacs.sh` downloads the graphs of the challenge, from New York (264346 nodes) to the whole United States (23947347 nodes). In the benchmark, `AStarBench --domain road --file base` reads `base.gr` and `base.co`, and solves the pair of `--points s,t`, or a random pair for each seed of `--ids`.

14.14 The navigation of a square robot

`RobotNav2D` (`RobotNav2D.h`) is the domain on which the edge-based parallel searches were compared (section "2D Grid World" of Mukherjee and Likhachev [122], and section "Evaluation" of Yang, Mukherjee and Likhachev [123], as the comment of the header states): a square robot moves on a map by long straight motions, each checked for collisions cell by cell, which makes the evaluation of an edge the expensive part of a search. The map is a `NavMap`, a rectangle of free and blocked cells (`NavMap( width , height )` is free), read from a file of the Moving AI benchmarks of Sturtevant [15] by `read_movingai( in , scale , trees_free )`, which makes each cell a square of $k \times k$ cells, $k =$ `scale`; the cells `.`, `G` and `S` are free, and `T` (the trees) too if `trees_free`, as in the experiments, unlike `GridMap::read_movingai` of §14.4, which blocks them. The parameters are those of `NavOptions`, by default those of the experiments.

```
struct NavOptions {

 int length = 25;               ///< the poses checked along a motion, which
                                ///< moves by length - 1 cells
 int half = 16;                 ///< the footprint covers the cells
                                ///< [ - half , half - 1 ] around the pose
 double radius = 25;            ///< a goal is closer than this to the goal
                                ///< point
 bool factors = false;          ///< the cost of a motion multiplied by a
                                ///< factor of its first cell
```

```
    std::uint64_t seed = 1;          ///< the seed of the factors
    unsigned repeat = 1;             ///< the check of a diagonal motion is done
                                     ///< this many times, so that it is the
                                     ///< expensive one if repeat > 1

};  // end( struct( NavOptions ) )
```

A state is the cell (x, y) of the centre of the robot, packed as x in the low 16 bits of a word of 32 and y in the high ones (`pack`, `x_of`, `y_of`), and the footprint of the robot is the square of the cells $[x-r, x+r-1] \times [y-r, y+r-1]$, $r =$ `half` (32 cells a side by default; the single cell (x, y) if $r = 0$). The 8 actions move by $L - 1$ cells, $L =$ `length`, along the 8 directions of the grid, and the edge of an action exists if the footprint is inside the map and on free cells at each of the L poses $(x, y) + i\,\vec{d}$, $0 \le i < L$, which `evaluate_edge` checks cell by cell, i.e., $L(2r)^2$ cells, 25600 with the defaults; its cost is the length of the motion, $L - 1$ or $(L - 1)\sqrt{2}$, times, if `factors`, a factor of the first cell drawn uniformly from $\{1, \ldots, 99\}$ by `seed`. With `repeat` > 1 the check of the 4 diagonal motions is done `repeat` times, which makes them the expensive edges of GePA*SE ([122]; `expensive_edge`), and `lazy_edge` checks the last pose only, with the cost of the motion. The goals are the states closer than $R =$ `radius` to the goal point γ, and the heuristic is $h(s) = \max\{0, \|s - \gamma\| - R\}$, the pairwise heuristic of PA*SE and of the edge-based searches being $h(s, t) = \|s - t\|$, both Euclidean. The domain models `EdgeDomain`, `HeterogeneousEdgeDomain`, `LazyEdgeDomain` and `PairwiseDomain` (`Domain.h`), and its `for_each_successor` evaluates the 8 edges, so that a search by states and a search by edges search the same graph; the constructor refuses a start outside the map, a map wider or higher than 65535 cells, a `length` below 1 and a negative `half`, but it does not check that the footprint fits at the start.

Proposition 14.31 (The heuristics of the navigation). On `RobotNav2D`, (i) h is consistent, and 0 at the goals; (ii) the pairwise heuristic satisfies (H1) $h(t, s) \le d(t, s)$ and (H3) $h(t) \le h(t, s) + h(s)$ for all states t and s; (iii) the lazy edges include the edges, with the same costs, and h is consistent with their costs.

Proof. Every motion from s to s' costs at least its Euclidean length $\|s - s'\|$, the factors being at least 1. (i) By the triangle inequality, $\|s - \gamma\| - R \le \|s - s'\| + \|s' - \gamma\| - R \le c(s, s') + \max\{0, \|s' - \gamma\| - R\}$, and also $0 \le c(s, s') + h(s')$, so that $h(s) \le c(s, s') + h(s')$; a goal has $\|s - \gamma\| < R$ and $h = 0$. (ii) A path from t to s costs at least the sum of the lengths of its motions, which is at least $\|t - s\|$, which gives (H1); and $\max\{0, \|t - \gamma\| - R\} \le \|t - s\| + \max\{0, \|s - \gamma\| - R\}$, which is (H3). (iii) An edge exists only if the footprint fits at its last pose, which is all that `lazy_edge` checks, and both give the successor and the cost of the motion; (i) holds for these costs, which are the same. □

The heuristic of the experiments was, according to the comment of the header, the distance to the goal point, 0 within the radius; the header calls it admissible but not consistent, and with the goal test of the library it is not even admissible: a state at distance exactly $R = 25$ from γ has the value 25, while a free straight motion of 24 cells towards γ reaches a goal at the cost 24. This is why the library uses the heuristic above. The maps of the experiments, hrt201n, den501d, den520d, ht_chantry and brc203d, scaled there by 5, 5, 5, 10 and 5, are in `data/robot-nav-2d` with 50 pairs of start and goal points each (*nav2d_starts.txt* and *nav2d_goals.txt*, which `read_points( starts , goals , scale )` reads, scaled and rounded to cells), from the code of those experiments, under the MIT license; the header reports their heuristic weight of 50, and their costs without or with the random factors. On a map with a wall and a gap, the test `edge` checks that the edges of the actions are the successors, checks the pairwise heuristic, and checks that A*, eA* and the lazy search find the same cost (with `repeat` 30) and prove the map with a closed wall unsolvable. On the first pair of den520d scaled by 5, with and without factors, it checks that PwA* and eA* find the cost of A* and, with the weight 50, a cost within 50 times it. In the benchmark, `AStarBench --domain nav2d --file map` reads the map and its pairs, with `--scale`, `--factors` (the seed, 0 for none) and `--diag` (the `repeat` of the diagonal motions).

14.15 The boards of Burns et al. and the refinements of the grids

The grids of §14.4 also read the boards of the experiments of Burns et al. [3], whose moves to a corner may cut a blocked cell and whose costs are either the costs of the cells or the *life* costs, in which leaving a cell of the row y costs y (times $\sqrt{2}$ to a corner), whatever the cell, which makes the rows of small y cheaper, the first one free, and the shortest path longer than the cheapest one. A board is a `GridBoard`, and the costs are given by the enumeration `GridCost` (`cells` or `life`).

```
enum class GridCost {
```

```
 cells ,                                  ///< the cost of the cell left
 life                                     ///< the row of the cell left
 };

/*------------------------------------------------------------------------*/
/*---------------------------- STRUCT GridBoard ---------------------------*/
/*------------------------------------------------------------------------*/
/// a pathfinding instance in the format of the grids of Burns et al.
/** The format of the instances of Burns et al. (JAIR 2010) and of their
 * generators: a line "H W" with the height and the width, a line
 * "Board:", H lines of W characters, '#' for a blocked cell and any other
 * one for a free cell, a line "Unit" or "Life" (the GridCost), a line
 * "Four-way" or "Eight-way" (the moves, a diagonal move being allowed even
 * if it cuts a blocked cell) and the four numbers \f$ x_s \f$,
 * \f$ y_s \f$, \f$ x_g \f$ and \f$ y_g \f$ of the start and the goal.
 * The files of the authors count \f$ y \f$ from the last line of the
 * board up, those of later generators from the first line down: read()
 * takes the convention, and stores the cells with \f$ y \f$ counted from
 * the first line, as GridMap does. */

struct GridBoard {

 GridMap map;                                ///< the free cells
 int sx = 0;                                 ///< the start column
 int sy = 0;                                 ///< the start row
 int gx = 0;                                 ///< the goal column
 int gy = 0;                                 ///< the goal row
 int moves = 4;                              ///< 4 or 8
 GridCost cost = GridCost::cells;            ///< unit or life costs

 /// reads a board, \p from_bottom telling whether \f$ y \f$ is counted
 /// from the last line up
 /** Throws std::invalid_argument if the board is malformed or the start
  * or the goal is outside it. A blocked start is made free, which changes
  * no path, since a shortest one never comes back to its start: some
  * instances of the authors have one. */

 [[nodiscard]] static GridBoard read( std::istream & in ,
                                      bool from_bottom = true );

 };  // end( struct( GridBoard ) )
```

The format is a line `H W` (the height and the width), a line `Board:`, H lines of the board (`#` for a blocked cell and any other character for a free one, a short line being free beyond its end), a line `Unit` or `Life`, a line `Four-way` or `Eight-way`, and the four numbers x_s, y_s, x_g and y_g. The files of the authors count y from the last line of the board up, those of later generators from the first line down; `GridBoard::read( in , from_bottom )` takes the convention, and stores the cells with y counted from the first line, as `GridMap` does. It throws `std::invalid_argument` on a malformed board or on a start or a goal outside it. It also frees a blocked start (which some files of the authors have), which changes no optimal cost, since a path of least cost from the start never comes back to it. The constructor `Grid( board , block , nblocks )` takes the costs of the board and lets the moves to a corner cut a blocked cell; it throws if the board has not the moves of the `Grid`. The general constructor is

```
Grid( const GridMap & map , int sx , int sy , int gx , int gy ,
      int block = 100 , GridNBlocks nblocks = {} ,
      GridCost cost = GridCost::cells , bool cut_corners = false );
```

in which `cut_corners` allows a move to a corner whatever the two cells it cuts. With the life costs, the heuristic of `Grid` is the cost of a cheapest path on the board without blocked cells, with the moves to a corner costing as the straight ones when there are 8 moves, which the header derives in closed form. Let $V(a, b)$ be the cost of going straight from the row a to the row b, i.e., $\sum_{k=b+1}^{a} k = (a(a+1) - b(b+1))/2$ upwards and $\sum_{k=a}^{b-1} k = (b(b-1) - a(a-1))/2$ downwards, and let the path go from (x, y) to (u, v), with $\Delta x = |x - u|$ and

$\Delta y = |y - v|$.

Proposition 14.32 (The life heuristic with 4 moves). With 4 moves and the life costs, the least cost of a path from (x, y) to (u, v) on the board without blocked cells is

$$\min\{V(y,0) + V(0,v),\ V(y,r) + r\,\Delta x + V(r,v)\}, \qquad r = \min\{y, v\},$$

and this heuristic is consistent.

Proof. Let a path have r' as its highest row, i.e., the least y it visits, $r' \le \min\{y, v\}$. It leaves upwards each row of $(r', y]$ at least once, at the cost of the row, and downwards each row of $[r', v)$ at least once, and it makes at least Δx horizontal moves, each leaving a cell of a row at least r'; these moves are distinct, so that the path costs at least $F(r') = V(y, r') + r'\,\Delta x + V(r', v)$, which the path up to the row r', along it and down to v attains. The least cost is thus $\min_{0 \le r' \le \min\{y,v\}} F(r')$; by the closed forms of V, $F(r') = C + r'\,\Delta x - r'^2$ for a constant C, a concave function, whose minimum over an interval is at one of its ends, $r' = 0$ or $r' = \min\{y, v\}$. The board without blocked cells is a relaxation of the grid, each move of the grid being a move of the board at the same cost, and the heuristic is consistent by Proposition 5.2. □

Proposition 14.33 (The life heuristic with 8 moves). With 8 moves, the life costs and the moves to a corner costing as the straight ones, the least cost of a path from (x, y) to (u, v) on the board without blocked cells is

$$S(n) = \sum_{k=0}^{n-1} \max\{y - k,\ v - n + k,\ 0\}, \qquad n = \max\{\Delta x, \Delta y\},$$

and this heuristic, with the real costs of the moves to a corner, is consistent.

Proof. Let a path have n' moves, and let ρ_k be the row of the cell left by its k-th move, $\rho_0 = y$. Since a move changes the row by at most one, $\rho_k \ge y - k$, $\rho_k \ge v - (n' - k)$ and $\rho_k \ge 0$, so that the path costs at least $S(n')$; also $n' \ge n$, since a move changes each coordinate by at most one. The sequence of the bounds for $n' + 1$ moves is that for n' moves with one term inserted at its least point: with $c = \lfloor (y - v + n')/2 \rfloor$, the terms of index $k \le c$ are $\max\{y - k, 0\}$ in both, and the term $k > c$ for n' is the term $k + 1$ for $n' + 1$, since $v - (n' + 1) + (k + 1) = v - n' + k$ and $y - k - 1 < y - k < v - n' + k$; the inserted term is not negative, and $S(n' + 1) \ge S(n')$. Hence every path costs at least $S(n)$. A path with n moves attains it: the rows $\rho_k = \max\{y - k, v - n + k, 0\}$ change by at most one from a move to the next, and they end at the row v; if $\Delta x = n$ each move also changes the column, and if $\Delta x < n$, then $n = \Delta y$ and the bounds are $y - k$ for every k (or $y + k$), strictly monotone, so that each move changes the row and the Δx changes of the column can be placed on any of them. The board without blocked cells, with the corners cut freely and the moves to a corner at the cost of the straight ones, is a relaxation of the grid, whose moves to a corner cost $\sqrt{2}$ times as much, and the heuristic is consistent by Proposition 5.2. □

The header computes $S(n)$ in $O(1)$ time as two arithmetic series, the first for the indices up to $c = \lfloor (y - v + n)/2 \rfloor$, where $y - k$ is the largest bound, and the second from $\max\{c + 1, n - v\}$ on, where $v - n + k$ is. The backward heuristic is the same estimate of the cost from the start, and it is consistent for the predecessors for the same reason, being the distance of the relaxation from the start. The test `board` checks both closed forms against the algorithm of Bellman and Ford on the relaxation, on boards of 9×7 cells with 9 goals and both moves. It checks A* against the algorithm of Dijkstra, and its consistency, on random boards of 60×40 cells of both moves and both costs, and the cut corner between two blocked cells, allowed by the boards and refused by default. Last, it reads boards with y from the bottom and from the top, refuses the malformed ones, and checks the costs of five boards of the data (`data/grid`) against the costs found by the code of their authors, e.g., 362873 on a board of 500×500 cells with the life costs and 4 moves.

The nblocks of PBNF (§14.4) are the rectangles of `GridNBlocks`, of `nblock_width()` times `nblock_height()` cells (squares of 20 cells a side by default, `nblock_side()` being then their side). `GridNBlocks::coarse( w , h , cols , rows )` gives the rectangles that cut a map of $w \times h$ cells into about `cols` times `rows` equal ones, as the coarse abstraction of the grids of [3]. The number `cols` is raised by one if it does not divide w, and the nblock is w/cols cells wide (at least 1), and likewise for the rows. Hence the number of the columns of nblocks is $\lceil w/\lfloor w/\text{cols} \rfloor \rceil$, which may exceed `cols` by more than one; the test `psdd-thread` checks, e.g., that 2001×7 cells cut into 50 by 50 give nblocks of 39×1 cells. Proposition 14.14 holds for these rectangles with the same proof, the column and the row of an nblock being $\lfloor x/W_n \rfloor$ and $\lfloor y/H_n \rfloor$ for its width W_n and height H_n.

A grid is also an `OperatorDomain`: the operator of a move by $(\delta x, \delta y)$ is $3(\delta y + 1) + \delta x + 1$, and its inverse 8 minus it, which the expansion with an operator skips. With the costs of the cells, the inverse of a move

does not cost as much as the move, and Lemma 1.8 is stated for an inverse of the same cost; its conclusion holds nonetheless, since its proof needs only $g(u) + c(u, v) = g(v) + c(v, u) + c(u, v) \geq g(v)$, which holds for any nonnegative costs (the comment of the concept `OperatorDomain` makes the same remark). Furthermore, a grid is a `PartialExpansionDomain` for EPEA* of Goldenberg et al. [114]: `for_each_successor_delta( s , h , op , delta , f )` generates, but for the move back, the successors whose $\Delta f = c + h(t) - h(s)$ is `delta` up to `cost_slack( delta )` (0 with the integer costs of 4 moves, $10^{-9} \max\{1, |\Delta|\}$ with the real ones of 8 moves, so that two values that differ by the rounding are taken as the same), and returns the least Δf above that, or ∞ if there is none; it runs over the at most 8 moves at each call, but it generates only those of the value asked. The test `classic` checks EPEA* against A* on random grids of 4 and 8 moves with cells of the costs 1 to 4.

The other refinements of the grids serve the bidirectional, greedy and incremental searches. `for_each_predecessor( s , f )` gives the free neighbours t of s with the cost of the move from t to s (the cost of t, or its row with the life costs, times $\sqrt{2}$ to a corner), the moves between two cells being allowed in both directions or in neither. With `goal_state()`, `backward_heuristic( s )` and the predecessors with the backward heuristic, the grid is a `BidirectionalDomain`. Last, `distance( s )` is the number of the moves to the goal without obstacles (the Manhattan distance with 4 moves, the larger difference of the coordinates with 8), `preferences( s )` prefers the moves that lower the heuristic, and `for_each_goal_feature` gives the column and the row of the goal. Finally, `GridMap::set_cost` may change the map after the construction: `refresh()` reads it again, and `cell_changed( x , y , old )` records the change of the cost of each move among the 3×3 cells around (x, y) (the only ones that the cell can change: its own moves, those into it, and the diagonals that cut it), as an `EdgeCostChanged` with ∞ for a move not allowed, the changes of one move being merged into one; `drain_events( f )` passes them to an incremental search and forgets them (`EventDomain`). The heuristic keeps the least cost of the map at the construction, so that the new costs of the free cells must not go below it for the heuristic to stay admissible, and both methods throw if the start or the goal becomes blocked; `min_edge_cost()` is the least cost of a cell, or 0 with the life costs (`MinCostDomain`).

14.16 Problems not in the library

> **Status — planned.** The problems of this section are not domains of the library; the section states how they fit the model, and what the searches would need.

A problem of the textbook of Jinnai (section 2.2 of [164]) and of the literature is not provided. Sokoban, PSPACE-complete, is the typical irreversible problem: the player moves at no cost and pushes a box at a positive cost; a domain would thus have edges of cost 0 (which the searches of the library accept, but which the delayed duplicate detection refuses) and many dead ends (a box pushed into a corner), which a heuristic must detect to be useful. Rubik's cube, the classical benchmark of the pattern databases, is the domain of §14.11. Finally, the multiple sequence alignment, reduced to a shortest path on a lattice of n dimensions with $2^n - 1$ moves, is the subject of Chapter 15.

14.17 Running the domains

Table 14.2 collects the cost of an expansion of each domain, which follows from the count of the loops of the algorithm cited (with J jugs, k queens placed, and the $O(1)$ of the operations on a word).

Table 14.2. The cost of an expansion, and of the heuristic of a successor, in each domain.

domain	algorithm	successors	expansion	h of a successor
`SlidingTile`	14.1, 14.2	at most 4	$O(\beta)$	$O(1)$, incremental
`SequenceScore`	14.2, 14.4	at most 4	$O(WH)$	$O(WH)$
`Grid< Moves >`	14.5	at most Moves	O(Moves)	$O(1)$
`ExplicitGraph`		out-degree	O(out-degree)	$O(1)$, a lookup
`NQueens`	14.6	at most n	$O(nk)$	$O(1)$
`VacuumWorld`	14.7	at most 3	$O(1)$	$O(1)$
`WaterJugs`	14.8	at most $J^2 + J$	$O(J^3)$	$O(J)$, the goal test
`MissionariesCannibals`	14.9	$O(b^2)$	$O(b^2)$	$O(1)$
`TravelingSalesman`	14.10, 14.11	at most $n - 1$	$O(n^3)$	$O(n^2)$
`PancakePuzzle`	14.13	at most $N - 1$	$O(N^2)$	$O(1)$, incremental

domain	algorithm	successors	expansion	h of a successor
`BurnedPancakePuzzle`	14.13	at most N	$O(N^2)$	$O(1)$, incremental
`TopSpin`		at most N	$O(N^2)$	$O(N)$
`RubiksCube`		at most 18	$O(1)$	the tables
`TowersOfHanoi`		at most 6	$O(1)$	the tables
`RobotNav2D`		at most 8	$O(Lr^2)$	$O(1)$

Each domain is run in two ways. First, the programs of `examples` solve one instance with A* and print the path and its cost, the counters and the time of the search. They are `8Puzzle` (the instance of figure 3.28 of the book, or a state at a given distance, with the misplaced tiles, the Manhattan distance or the sequence score; with `-t` it repeats the comparison of figure 3.29 of the book), `15Puzzle` (an instance of Korf, sequentially or with HDA*), `FindPath`, `RouteFinding`, `NQueens`, `VacuumWorld`, `WaterJugs`, `MissionariesCannibals` and `TravelingSalesman`; each run with a known answer is also a test of `ctest`, labelled `example`. Second, the benchmark `AStarBench` runs any search of the library, on any runtime and with any distribution, on the domain chosen by `--domain` (`tiles8`, `tiles15`, `tiles24`, `grid4`, `grid8`, `board`, `queens`, `vacuum`, `missionaries`, `jugs`, `tsp`, `romania`, `pancake`, `burned`, `topspin`, `rubik`, `hanoi`, `road` and `nav2d`, besides the domains of other chapters). The puzzles read their instances from `--file` (the instances chosen by `--ids`), the grids from a Moving AI map or from `--random WxH:p:seed[:c]`, and the other domains are sized by `--size` (e.g., `queens 28`, `missionaries 65535:65535:100`, `jugs 1:12:8:3`, `tsp 16` with the seeds of `--ids`). It prints one line of CSV per run, with the cost, the counters of §1.3, LB, CO and the seconds.

Table 14.3. The problems of the examples, solved by sequential A*.

example	arguments	cost	expanded	milliseconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

Table 14.3 is written by `manual/build/tables.py` from the table E of the campaign of `tools/bench/campaign.sh`. The campaign runs each example program 10 times with the arguments of the second column (no argument being the default instance of the program, e.g., the instance of figure 3.28 for `8Puzzle` and the instance 1 of Korf for `15Puzzle`; `FindPath -r 1000x1000:0.3:2` is a random map of 1000×1000 cells with 30% of them blocked and the others costing 1 to 5, with 4 moves and, with `-8`, with 8). Its columns give the cost of the path returned (in moves, in km or in the units of the domain, exact for the integer costs and in double precision for the real ones), the number of expansions, which does not depend on the machine, and the median of the 10 wall-clock times in milliseconds. Note that the time is the field `seconds` of the `Solution`, measured by the `Stopwatch` started at the beginning of `solve()` (`AStar.h:194`) and read after the path has been traced; it does not include the start of the process, the construction of the domain (the reading or the drawing of the map, the precomputed tables of the puzzle), nor the printing of the path. The note of the table gives the machine, the compiler and the flags. Since the problems of the examples are meant to be solved in milliseconds, we measure (in turn) the scaling of the parallel searches on larger instances of the same domains in Chapter 8 and Chapter 9.

14.18 Notes and references

The sliding-tile puzzles became the benchmark of heuristic search with the 100 instances of the 15-puzzle that Korf [7] solved by IDA* with the Manhattan distance; the parity argument of Proposition 14.2 and its converse are due to Johnson and Story [150]. In turn, the 24-puzzle needs pattern databases [22]. The instances of `data/24-puzzle` come with the code of HDA* of Jinnai and Fukunaga [5], whose abstractions of the puzzles and of the grids the domains of this chapter reproduce; the abstractions of AHDA* and of PBNF are those of Burns et al. [3], and the packed state, the incremental heuristic and the pruning of the parent follow Burns et al. [4]. Our grids follow the benchmarks of Sturtevant [15] and the random maps of [3]; the map of Romania, the n-queens, the vacuum world, the water jugs, the missionaries and cannibals and the MST heuristic of the traveling salesperson come from chapter 3 of Russell and Norvig [13], the sequence score from Nilsson [8], and the dynamic program of the traveling salesperson from Held and Karp [14]. The domains of section 2.2 of the textbook of Jinnai [164] (the grid, the puzzles and the traveling salesperson) are all here.

It is fair to remark that the literature of the parallel searches uses these domains, and therefore they allow the searches of the library to be compared with the published ones; however, they are written by hand for one problem each, with hand-made abstractions, and the choice of the abstraction from the problem itself is open (Chapter 10). Also, the representation of the puzzles in one or two words is that of the fastest published codes [4], while the other domains use a word for convenience more than for speed; their expansions are cheap (Table 14.2), and on them the measures of the parallel searches are dominated by the bookkeeping of the searches, which is the regime in which those searches are hardest to scale.

The pancakes go back to Dweighter [63], and their gap heuristic to Helmert [60]; TopSpin is the puzzle of the experiments of Felner et al. [167] and of Holte et al. [30]; Rubik's cube, its prunings of the redundant pairs of moves, its tree and its ten instances are those of Korf [59], in the cubie model of Kociemba; the Towers of Hanoi with four pegs and their instances are those of Siag et al. [57]; the road graphs are those of the 9th DIMACS Implementation Challenge [64], with a heuristic of their coordinates that we scale so as to make it consistent whatever the units of the costs; the navigation of a square robot is the domain of the edge-based searches [122], [123], with a heuristic that we changed to make it consistent (Proposition 14.31); and the boards with the life costs come from Burns et al. [3], for which we derived the exact heuristic of Proposition 14.32 and Proposition 14.33. Of course, the domains of the permutation puzzles are written for one puzzle each, as the others, and what they share is the description of their moves as permutations (§5.15), on which the tables are built.

14.19 Exercises

Exercise 14.1 (`ctest -R PBeFS.rubik`). Pack the state of `RubiksCube` (§14.11) into two words of 64 bits, and say how many bits it needs at least, given the $8! \cdot 3^7 \cdot 12! \cdot 2^{11}/2 \approx 4.3 \cdot 10^{19}$ reachable states; then explain why the tree searches skip the turns of the opposite face in one of the two orders only, and what would be lost by skipping both. The test checks the size of the tree of Korf to the depth 5.

Exercise 14.2 (`ctest -R PBeFS.astar`). Show that the 8-puzzle has $9!/2 = 181440$ reachable states and the 15-puzzle $16!/2$, using Proposition 14.2 and its converse; the test checks the first number by a breadth-first search of the whole space from the goal.

Exercise 14.3 (`ctest -R PBeFS.domains`). Using Proposition 14.13, compute the mean branching factor of a grid of 5×5 free cells with 4 moves ($80/25 = 3.2$) and with 8 moves ($144/25 = 5.76$); then measure it with the counters of `AStarBench`, as `generated / expanded` on a search that expands all the cells (e.g., towards a goal walled off by blocked cells), and explain the difference between the two numbers. The test checks the two counts of moves.

Exercise 14.4 (`ctest -R PBeFS.domains`). Prove Proposition 14.4 in detail for the 8-puzzle, i.e., that the mean branching factor over its reachable states is exactly $8/3$, and $5/3$ without the move back to the parent; the test checks the first value on the whole state space, as 3 times the number of the moves equal to 8 times the number of the states.

Exercise 14.5 (`ctest -R PBeFS.grid`). Show that on a grid with 8 moves the heuristic $\kappa_{\min}\, d_4$ (the Manhattan distance) is not admissible, and that $\kappa_{\min} \max(\Delta x, \Delta y)$ (the Chebyshev distance) is admissible but less informed than $\kappa_{\min} d_8$. The test checks A* with d_8 against Dijkstra on random maps.

Exercise 14.6 (`ctest -R PBeFS.domains`). Prove that the heuristic $1 + h_L(n+1)$, used on the right bank in place of $1 + h_L(n)$, is still admissible for the missionaries and cannibals, and decide whether it is consistent. The test checks that the heuristic of the library is admissible, by the distances to the goal of all the reachable states, but not consistent, on the instance with 3, 3 and 2.

Exercise 14.7 (`ctest -R PBeFS.domains`). Using Proposition 14.22 and Proposition 14.23, show that A* with the MST heuristic expands each state at most once in exact arithmetic, and hence never expands more states than there are entries in the table of Held and Karp. The test compares the cost of A* with `held_karp()` on random instances and checks the consistency of the heuristic on all the edges of their state spaces.

Exercise 14.8 (`ctest -R PBeFS.domains`). Show that A* on `NQueens`, with its default OPEN list (the bucket queue), expands the states in the order of a depth-first search that tries the rows in decreasing order (the successors are given by increasing row, and the ties of f and g are broken in favor of the last pushed), and count the states of the tree for $n = 8$; the test checks that the tree has 2057 states and 92 goals, and that the problem with 3 queens is unsolvable.

Exercise 14.9 (`ctest -R PBeFS.graph`). Check Proposition 14.15 on the data of `Romania.h`, and explain why the table of the book cannot serve as the heuristic towards a city other than Bucharest, while the coordinates

can. The test checks the distances to Bucharest from each city.

Exercise 14.10 (`ctest -R PBeFS.pancake`). Prove Proposition 14.26, and show that the flip of 1 is needed to reach the goal of the burned pancakes from some stacks, while it would be the identity for the pancakes without a burned side. How many successors does a stack of N pancakes have with and without the operator of its parent? The test checks that the 5040 stacks of 7 pancakes and the 3840 stacks of 5 burned ones are reached from the goal, and the reading of `-3 1 -2 5 4`.

Exercise 14.11 (`ctest -R PBeFS.topspin`). For TopSpin with 8 tokens and a turnstile of 4, list the moves j that commute with the move 5, and the successors of a state generated by the move 5 in a tree search. Then prove that the prunings of `for_each_tree_successor` satisfy the hypotheses of Lemma 14.27. The test checks IDA*, with and without a table, against the exact distances of 20 rings.

Exercise 14.12 (`ctest -R PBeFS.rubik`). Derive the recurrence of Proposition 14.28 (iii), compute the nodes of the tree at the depth 6, and show that the asymptotic branching factor is $6 + 3\sqrt{6}$. Why are there 43239 distinct states at the depth 4, while the tree has 43254 nodes there? The test checks both counts, the order of each move and its inverse, and the ten instances of Korf.

Exercise 14.13 (`ctest -R PBeFS.hanoi`). Prove Proposition 14.29, and give a state with exactly 6 moves and one with 3. Show that two moves of the same disk in a row have the effect of one move of it, or of none, and conclude by Lemma 14.27 that the tree searches lose no optimal path. The test reads and writes back the 20 instances of Siag et al., and solves 16 instances of 8 disks with A* and the parallel IDA*.

Exercise 14.14 (`ctest -R PBeFS.dimacs`). Prove Proposition 14.30, and show that the heuristic stays consistent with $\kappa\,\delta(u,t)$ rounded to the nearest integer, or up, in place of the floor; why is the floor the safer choice when the distances are computed in floating point? What is κ when the graph has an arc of cost 0 between two distinct points? The test checks the consistency on every arc of the sample graph for 5 pairs, A* against the algorithm of Dijkstra, and the refusal of an arc out of range.

Exercise 14.15 (`ctest -R PBeFS.board`). Compute by Proposition 14.32 the life heuristic with 4 moves from $(0,6)$ to $(8,6)$, and say whether the cheapest path goes through the row 0; then compute by Proposition 14.33 the heuristic with 8 moves from $(0,3)$ to $(2,3)$. The test checks both closed forms against the algorithm of Bellman and Ford on boards of 9×7 cells, and the costs of the boards of the data.

Exercise 14.16 (`ctest -R PBeFS.classic`). On a grid with 8 moves and the costs of the cells, list the values of Δf of the moves of a cell of cost 2 next to a cell of cost 1, with the octile heuristic, and show that `for_each_successor_delta` generates them in increasing order over the calls of EPEA*. Why does the skip of the parent stay correct although the inverse of a move may cost less than the move? The test checks EPEA* against A* on random grids with cells of cost 1 to 4.

Exercise 14.17 (`ctest -R PBeFS.edge`). Prove Proposition 14.31, and complete the example that shows that the heuristic of the experiments of the edge-based searches, the distance to the goal point and 0 within the radius, is not admissible with the goal test of `RobotNav2D`. How many cells does the evaluation of an edge check with the defaults, and how many does the lazy edge check? The test checks the searches by edges against A* on a map with a gap and on den520d, and the closed wall.

15. Multiple sequence alignment and the instances

[Source: include/PBeFS/astar/SequenceAlignment.h (SequenceAlignment, ScoreMatrix, pam250), include/PBeFS/astar/Distribution.h (HyperplaneDistribution), the concept LevelDomain of include/PBeFS/astar/Domain.h, the readers of the instance files of the domains, and the instance sets of data/]

> **Status — beta.** The code that this chapter describes is in `main` but not yet in a release, and its interface may still change.

We are concerned in this chapter with two subjects that the benchmarks of the parallel searches share. The first is the optimal alignment of several protein sequences (the Multiple Sequence Alignment, MSA), formulated as a shortest path on an n-dimensional lattice by Ikeda and Imai [41] and used, after Yoshizumi, Miura and Ishida [42], as the test of a search with a large branching factor ($2^n - 1$ successors of a state) and nonuniform costs; on it Kobayashi, Kishimoto and Watanabe [43] evaluated HDA* and proposed the hyperplane work distribution, which we describe here together with the concept of a levelled graph on which it rests. The second is the collection of the instance sets of `data/` (where each set comes from, in which format it is written, which reader of the library parses it, and under which license it is redistributed), together with the options of `AStarBench` that select them. The notation is that of Chapter 1; the heuristics and their properties (admissibility, consistency) are those of §2.4, the work distributions of HDA* those of §9.4, and the nblocks of Safe PBNF those of Definition 11.1.

The structure of the chapter is as follows. Section 15.1 states the problem and the interface of the domain, section 15.2 the costs of a column and their relation with the score of an alignment, and section 15.3 the pairwise heuristic, with its proof of admissibility and consistency, and the cost of an expansion. Section 15.4 gives the features and the abstractions of the domain for the parallel searches, and section 15.5 the levelled graphs and the hyperplane work distribution, with the bound on the number of workers that the successors of the states of a worker reach. Section 15.6 describes the instance sets, section 15.7 the readers of their files, and section 15.8 the options of `AStarBench`; sections 15.9 to 15.11 give the benchmark, the notes and the exercises.

15.1 The problem and the interface

We are given $n \geq 2$ sequences $s_0, \ldots, s_{n-1}$ over the alphabet of the amino acids, of lengths $L_0, \ldots, L_{n-1}$, and a cost $c(a, b) \geq 0$ for each pair of symbols, the gap $-$ included, with $c(-, -) = 0$. An alignment inserts gaps into the sequences so that they have a common length m, and it is read as m columns, the column k holding the k-th symbol of each padded sequence; no column is made of gaps only. The cost of a column $x = (x_0, \ldots, x_{n-1})$ is the *sum of pairs*

$$\mathrm{cost}(x) = \sum_{0 \leq i < j < n} c(x_i, x_j),$$

and the cost of an alignment is the sum of the costs of its columns. We want an alignment of least cost.

Definition 15.1 (The lattice of an alignment). The *lattice* of the sequences is the set of the points $p = (p_0, \ldots, p_{n-1})$ with $0 \leq p_i \leq L_i$, where p_i is the number of the residues of s_i already aligned. There is an edge from p to $p + e_M$ for each nonempty set $M \subseteq \{i : p_i < L_i\}$, where e_M is the vector with 1 on the coordinates of M and 0 elsewhere, which stands for the column whose symbol i is the residue $s_i[p_i]$ if $i \in M$ and the gap otherwise; its cost is the cost of that column. Its initial state is the origin and the goal is $(L_0, \ldots, L_{n-1})$.

The alignments are in one-to-one correspondence with the paths from the origin to the goal (the k-th edge of the path is the k-th column), and the cost of a path is the cost of its alignment; hence an optimal alignment is a shortest path of the lattice [41]. The lattice has $\prod_i (L_i + 1)$ points and, at a point where all the sequences are

still open, $2^n - 1$ successors; it is acyclic, since each edge raises $\sum_i p_i$. Figure 15.1 shows the lattice of two short sequences and an optimal path.

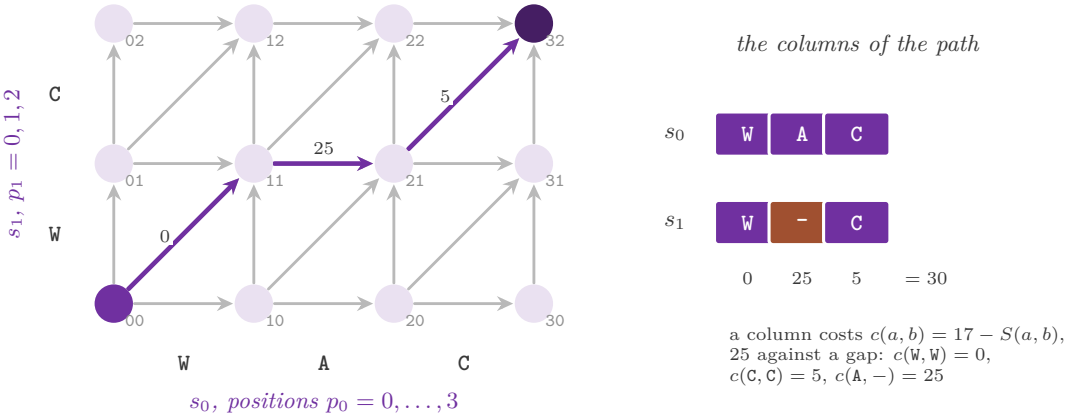


Figure 15.1. The lattice of the sequences $s_0 = \mathtt{WAC}$ and $s_1 = \mathtt{WC}$: the point $p_0 p_1$ has the successors that advance s_0 (right), s_1 (up) or both (diagonal). The path in the primary colour from the origin to the goal is an optimal alignment with the costs of PAM250: the column W/W costs 0, A against a gap 25 and C/C 5, in all 30 (right, the gap in the accent colour).

In the library, the domain is the class template `SequenceAlignment< K >` (SequenceAlignment.h:230), whose parameter is the largest number of sequences (10 by default, from 2 to 16 by a `static_assert`). A state is a `std::array< std::uint16_t , K >` of positions (those beyond the n-th are 0), which is why a sequence must have fewer than 65535 residues, and the cost type is `int`. Its symbols and matrices of scores are declared at namespace scope as follows:

```
/// the symbols of a substitution matrix: the 20 amino acids, the
/// ambiguity codes B, J, Z and X, and the gap '*', in this order

inline constexpr char amino_symbols[] = "ARNDCQEGHILKMFPSTWYVBJZX*";

/// the number of symbols, the gap included

inline constexpr int amino_count = 25;

/// the index of the gap among the symbols

inline constexpr int amino_gap = 24;

/// a substitution matrix: the score of each pair of symbols

using ScoreMatrix = std::array< std::array< int , amino_count > ,
                                amino_count >;
```

i.e., `amino_symbols` lists the 20 amino acids, the ambiguity codes `B`, `J`, `Z` and `X`, and the gap `*` (`amino_count` symbols, the gap at the index `amino_gap`), and a `ScoreMatrix` gives the score $S(a, b)$ of each pair of them. The constant `pam250` is the matrix at 250 PAMs of Dayhoff, Schwartz and Orcutt [44], with the scores of the ambiguity codes and the score -8 of a residue against a gap. The members of the class that a user calls are the following (quoted from the header, without their bodies):

```
explicit SequenceAlignment( const std::vector< std::string > & sequences ,
                            const ScoreMatrix & scores = pam250 ,
                            int block = 4 , int nblock_side = 16 ,
                            int penalty = 0 );
static std::vector< std::string > read_fasta( std::istream & in );
static ScoreMatrix read_scores( std::istream & in );
static std::vector< std::string > random_sequences( std::uint64_t seed ,
                                                    int n = 0 ,
                                                    int total = 5000 );
static int symbol( char a ) noexcept;
```

```
int size( void ) const noexcept;
int length( int i ) const noexcept;
cost_type pair_cost( int a , int b ) const noexcept;
std::vector< std::string > alignment(
                           const std::vector< state_type > & path ) const;
int block( void ) const noexcept;
int nblock_side( void ) const noexcept;
```

The constructor takes the sequences as strings of the symbols of `amino_symbols` but the gap, the matrix of scores, the side `block` of the boxes of Abstract Zobrist hashing and of AHDA* (section 15.4), the side `nblock_side` of the nblocks of Safe PBNF, and a linear gap penalty (section 15.2; 0 keeps the gap scores of the matrix). It throws `std::invalid_argument` if there are fewer than 2 or more than `K` sequences, if a sequence is empty or has 65535 residues or more, if it holds a symbol that is not in `amino_symbols` or the gap, if a side is not positive, if the penalty is negative, or if a cost of section 15.2 is negative (i.e., if a gap scores more than the best pair of residues). It then fills the tables of the heuristic (section 15.3), and hence its cost is that of the dynamic programs of all the pairs. `symbol( a )` is the index of the character `a` in `amino_symbols` (-1 if it is not there); `size()` and `length( i )` give n and L_i; `pair_cost( a , b )` gives $c(a, b)$ by the indices of the symbols; and `alignment( path )` turns a path of states, e.g., the `path` of a `Solution`, into the n rows of its alignment, `-` standing for a gap. Its three static members read the sequences of a FASTA file, read a matrix of scores, and draw random sequences; we describe them in section 15.7.

The domain models the concepts `IncrementalFeatureDomain`, `AbstractFeatureDomain`, `AbstractStateDomain`, `NBlockDomain`, `DiskDomain` and `LevelDomain` (§1.5), which the case `msa` of the tester checks by a `static_assert`; it is neither an `OperatorDomain` nor a `PairwiseDomain` nor a `PerfectHashDomain`. Hence it runs under A*, IDA*, the search with delayed duplicate detection of §4.6 (whose states must be trivially copyable), HDA* with the distributions Z, AZ, A, H and the hyperplanes, Safe PBNF, and the other searches that need no more than these concepts.

15.2 The costs

A substitution matrix gives a score, which is to be maximized, whereas a search minimizes a cost. Hence the library turns the scores into costs by a shift. Let $S_{\max}$ be the largest score $S(a, b)$ between two residues (a and b ranging over the 24 symbols that are not the gap), and let $P \geq 0$ be the penalty given to the constructor. The cost of a pair of symbols is

$$c(a, b) = S_{\max} - S(a, b), \qquad c(-, -) = 0 \qquad (P = 0),$$

$$c(a, b) = 2\,(S_{\max} - S(a, b)), \quad c(a, -) = c(-, a) = S_{\max} + 2P, \quad c(-, -) = 0 \qquad (P > 0),$$

a and b being residues (`SequenceAlignment.h:284-304`). With PAM250, $S_{\max} = S(\mathtt{W}, \mathtt{W}) = 17$, and with $P = 0$ we have $c(\mathtt{W}, \mathtt{W}) = 0$, $c(\mathtt{C}, \mathtt{C}) = 5$ and a residue against a gap costs $17 + 8 = 25$, the values that the case `msa` checks. The costs between two residues are nonnegative by the choice of $S_{\max}$; those against a gap are nonnegative unless, with $P = 0$, the matrix gives a gap a score larger than $S_{\max}$, which the constructor refuses.

The shift by $S_{\max}$ changes the problem only by a constant and a gap penalty, as follows.

Proposition 15.2 (Minimum cost and maximum score). Let an alignment induce on the pair $i < j$, once its columns of two gaps are dropped, m_{ij} columns of two residues, whose scores sum to Σ_{ij}, and g_{ij} columns of a residue and a gap. If $P = 0$ and G is the score of a residue against a gap (the same for all the residues, as in PAM250), the cost of the alignment is

$$\sum_{i<j} \Big(S_{\max} \frac{L_i + L_j}{2} - \big(\Sigma_{ij} - \big(\tfrac{S_{\max}}{2} - G \big) g_{ij} \big) \Big),$$

and if $P > 0$ it is

$$\sum_{i<j} \Big(S_{\max} (L_i + L_j) - 2\,(\Sigma_{ij} - P\, g_{ij}) \Big).$$

Hence an alignment of least cost is one of largest sum-of-pairs score with the linear gap penalty $S_{\max}/2 - G$ (16.5 for PAM250) if $P = 0$, and with the linear gap penalty P if $P > 0$.

Proof. Each residue of s_i and of s_j appears in exactly one of the columns kept, a column of two residues holding two of them and a column of a residue and a gap one, whence $2m_{ij} + g_{ij} = L_i + L_j$. Columns of two gaps cost 0. If $P = 0$, the cost of the pair is

$$\sum (S_{\max} - S) + (S_{\max} - G)\, g_{ij} = S_{\max}(m_{ij} + g_{ij}) - \Sigma_{ij} - G\, g_{ij},$$

the first sum running over the m_{ij} columns of two residues, and $m_{ij} + g_{ij} = (L_i + L_j)/2 + g_{ij}/2$ gives the first formula. If $P > 0$, the cost of the pair is $2S_{\max}m_{ij} - 2\Sigma_{ij} + (S_{\max} + 2P)\, g_{ij} = S_{\max}(2m_{ij} + g_{ij}) - 2(\Sigma_{ij} - P g_{ij})$, which is the second. In both cases the first term does not depend on the alignment, and the cost is a decreasing function of the score. □

The penalty $P = 8$ is that of the experiments of Jinnai and Fukunaga ([5], section 4.1.3, as the header reports it), and with $P > 0$ the costs stay integers, the doubling absorbing the half of $S_{\max}$. We check both statements, in the case `msa`, on the pair `HEAGAWGHEE`, `PAWHEAE`: the optimal cost with $P = 8$ equals $17(L_0 + L_1) - 2\sigma$, σ being the best score with the penalty 8 computed by the recurrence of Needleman and Wunsch, and the optimal cost with $P = 0$ equals $(17(L_0 + L_1) - 2\sigma')/2$, σ' being the best score with the penalty 16.5. A user who wants another matrix gives it to the constructor, e.g., as read by `read_scores` (section 15.7).

15.3 The pairwise heuristic

The heuristic of the library is the classical one of the optimal pairwise alignments [41]: for each pair $i < j$, let $D_{ij}(q_i, q_j)$ be the least cost of an alignment of the suffixes $s_i[q_i..]$ and $s_j[q_j..]$ with the costs c, and let

$$h(p) = \sum_{0 \le i < j < n} D_{ij}(p_i, p_j).$$

The tables D_{ij} are filled once by the constructor, backwards from the corner (L_i, L_j), by the dynamic program of the following algorithm, which transcribes the private member `pairwise_tables` (SequenceAlignment.h:642; the header has no pseudocode of it).

Algorithm 15.1 (The tables of the pairwise heuristic).

```
 1  procedure PAIRWISETABLES()
 2     for each pair i < j do
 3        for q_i ← L_i down to 0 do
 4           for q_j ← L_j down to 0 do
 5              D ← 0 if (q_i, q_j) = (L_i, L_j), ∞ otherwise
 6              if q_i < L_i then D ← min(D, D_ij(q_i + 1, q_j) + c(s_i[q_i], −))
 7              if q_j < L_j then D ← min(D, D_ij(q_i, q_j + 1) + c(−, s_j[q_j]))
 8              if q_i < L_i and q_j < L_j then
 9                 D ← min(D, D_ij(q_i + 1, q_j + 1) + c(s_i[q_i], s_j[q_j]))
10              D_ij(q_i, q_j) ← D
```

Then `heuristic( p )` (SequenceAlignment.h:442) sums $n(n-1)/2$ entries of the tables.

Lemma 15.3 (The tables). Algorithm 15.1 sets $D_{ij}(q_i, q_j)$ to the least cost of an alignment of $s_i[q_i..]$ and $s_j[q_j..]$, in time and memory $O((L_i + 1)(L_j + 1))$ for the pair; the tables of all the pairs take $O(\sum_{i<j}(L_i + 1)(L_j + 1))$ words.

Proof. The lattice of the two suffixes is the lattice of Definition 15.1 for $n = 2$, restricted to the points $\ge (q_i, q_j)$, and the first column of an alignment of the suffixes advances s_i, s_j or both; lines 5 to 9 are the Bellman equation of the shortest path to the corner on this acyclic graph, whose successors come earlier in the order of the loops of lines 3 and 4. By induction on $(L_i - q_i) + (L_j - q_j)$ each entry is the least cost of a path to the corner. Each entry costs $O(1)$. □

Theorem 15.4 (The pairwise heuristic is admissible and consistent). For each point p of the lattice, $h(p)$ is at most the least cost of a path from p to the goal, h is 0 at the goal, and $h(p) \le c(p, p') + h(p')$ for each edge (p, p').

Proof. A path from p to the goal is an alignment A of the suffixes $s_i[p_i..]$; its cost is $\sum_{i<j}\sum_{x \in A} c(x_i, x_j)$, and for the pair $i < j$ the columns of A, once those with two gaps on i and j are dropped (they cost $c(-,-) = 0$), form an alignment of $s_i[p_i..]$ and $s_j[p_j..]$, whose cost is at least $D_{ij}(p_i, p_j)$ by Lemma 15.3. Summing over the pairs gives $h(p) \le$ the cost of A. At the goal each table is read at its corner, where it is 0. Let now the edge

(p, p') be the column x. For the pair $i < j$, either $x_i = x_j = -$, and then $p'_i = p_i$, $p'_j = p_j$ and $c(x_i, x_j) = 0$, or x restricted to $\{i, j\}$ is an edge of the lattice of the pair from (p_i, p_j) to (p'_i, p'_j), of cost $c(x_i, x_j)$, and the Bellman equation of the proof of Lemma 15.3 gives $D_{ij}(p_i, p_j) \leq c(x_i, x_j) + D_{ij}(p'_i, p'_j)$. In both cases the inequality holds for the pair, and summing over the pairs gives $h(p) \leq \text{cost}(x) + h(p')$. □

Since h is consistent, A* never reopens a state of the lattice (Theorem 2.13), which the case `msa` checks on the instances of 4 sequences of `data/msa/easy` against the exact distances of all the points (`check_consistent_astar`), together with the cost of A* against a dynamic program over the whole lattice written from the definition of the sum of pairs (`alignment_dp` of `test/files.cpp`). It is worth noting that h is exact for $n = 2$ and that, with more sequences, it ignores the interaction among the pairs; the tighter heuristics of the literature (e.g., from the optimal alignments of triples) are not in the library.

Each expansion generates the successors by `for_each_successor` (`SequenceAlignment.h:457`), which collects the set O of the sequences not yet ended and the residues $s_i[p_i]$ of these, and then enumerates the nonempty subsets $M \subseteq O$ as bit masks in decreasing order (`m = open` first, then `m = ( m - 1 ) & open`); thus the first successor is the column that advances all the open sequences. For each one it computes the cost of the column and the heuristic of the successor.

Proposition 15.5 (The cost of an expansion). An expansion of a point whose k sequences are open generates $2^k - 1$ successors in time $O(2^k n^2)$, the cost of a column and the heuristic of a successor costing $O(n^2)$ each; the goal test and the hash of a state cost $O(K)$.

Proof. The loop over the masks visits each nonempty subset of O once, and for each one the two double loops over the pairs take $O(n^2)$, the table lookups costing $O(1)$. For the goal test, the code compares two arrays of K positions, and `hash` mixes the positions four at a time, 16 bits each, by SplitMix64. □

Hence the domain is the opposite of the puzzles of Chapter 14: the expansion is costly and the branching factor is large (127 successors for 7 open sequences), and with HDA* the communication is relatively cheap and the search overhead and the load balance weigh more.

15.4 The features, the abstractions and the nblocks

The parallel searches need the structure of the lattice in three forms, which the domain provides as follows.

The features of Zobrist hashing

The features of a point p are the pairs (i, p_i), one for each sequence, numbered from the offset of the sequence i in the list of all the pairs (`num_features()` is $\sum_i (L_i + 1)$, and `for_each_feature( p , f )` calls `f` for the n features of p). A move changes the features of the sequences it advances, and `for_each_feature_change( p , p' , f )` calls `f` for the old and the new feature of each of them, whence the incremental Zobrist key of §9.5 costs $O(|M|)$ per successor; the case `msa` checks that the keys of the successors computed from the key of the parent are those computed from scratch, for 1, 3 and 64 workers.

The boxes of Abstract Zobrist hashing and of AHDA*

With the side b (`block()`, 4 by default, as in section 4.1.3 of [5] as the header reports it), the abstract feature of (i, p_i) is $(i, \lfloor p_i/b \rfloor)$ (`abstract_feature( k )` maps the feature k to it), and `num_abstract_features()` is $\sum_i (\lfloor L_i/b \rfloor + 1)$. The abstract state of AHDA*, `abstract_state( p )`, is the box $(\lfloor p_0/b \rfloor, \ldots, \lfloor p_{n-1}/b \rfloor)$ of b^n points, written in mixed radix. Under both distributions the owner of a point is a function of its box, and hence each move inside a box keeps the owner of the state; the moves that cross the boundary of a box (a coordinate p_i reaching a multiple of b) may change it. The abstraction of AHDA* of [5] keeps the position in the longest sequence only (as the header reports), i.e., it cuts the lattice into slices instead of boxes; ours is the product abstraction. On the instance `test3`, with boxes of side 3 and 64 workers, the case `msa` checks that both AZH and AHDA* keep on the owner of the parent more than four times as many successors as Zobrist hashing.

The nblocks of Safe PBNF

The nblocks are the boxes of side ν (`nblock_side()`, 16 by default), $(\lfloor p_0/\nu \rfloor, \ldots, \lfloor p_{n-1}/\nu \rfloor)$, with $\lfloor L_i/\nu \rfloor + 1$ boxes along the coordinate i (`num_nblocks()` is their product, and `nblock( p )` the box of p in mixed radix),

and `for_each_nblock_successor( b , f )` (SequenceAlignment.h:573) calls `f` for each box that differs from b by $+1$ in a nonempty set of coordinates, or by -1 in a nonempty set of coordinates, within the lattice.

Proposition 15.6 (The boxes are an abstraction). The boxes of side ν with the relation of `for_each_nblock_successor` satisfy the conditions (i) and (ii) of Definition 11.1, and each box has at most $2(2^n - 1)$ abstract successors.

Proof. (i) An edge $p \to p + e_M$ raises each coordinate in M by one and leaves the others. Hence each coordinate of the box rises by 0 or 1, all upwards; if none rises the box is the same, and otherwise the box of $p + e_M$ differs from that of p by $+1$ in a nonempty set of coordinates, which the code lists (both boxes lie within the lattice). (ii) The code lists the boxes that differ by $+1$ on a set T and those that differ by -1 on a set T; b' is listed from b with $+1$ on T if and only if b is listed from b' with -1 on T, and b itself is never listed, since T is not empty. The bound counts the nonempty subsets in the two directions. □

The edges of the lattice all go upwards, and the graph of the boxes is in fact directed; the downward successors are listed only because `SafePBNF` takes the interference scope of an nblock from its successors alone and needs the relation symmetric (§11.1). In the case `msa`, we check both conditions on all the moves of the instance `test3` with boxes of side 4.

15.5 Levelled graphs and the hyperplane work distribution

The level of a point p is $\ell(p) = \sum_i p_i$ (`level( p )`), the number of the residues aligned, and `max_level()` is $\sum_i L_i$, the level of the goal. Each move raises the level by $|M| \in \{1, \dots, n\}$, and the lattice is therefore a *levelled graph*, the structure that the concept `LevelDomain` of Domain.h states: a nonnegative integer `level( s )` that every move raises, and the largest one, `max_level()`, that of the goals.

The hyperplane work distribution of Kobayashi, Kishimoto and Watanabe ([43], section 4.2, as the header reports it) gives the hyperplanes $\ell(p) = c$ of the lattice, or slabs of d consecutive ones, to the workers in turn. Its rationale is the following. In the lattice of MSA a point is reached by many paths of different costs, and with Zobrist hashing its copies are generated by workers spread over the whole machine (the successors of a point go to up to $\min\{p, 2^n - 1\}$ workers); the duplicates are then detected only after the messages have crossed, and a worker that receives a cheaper path late has to reopen a state that it had already expanded, which matters with the nonuniform costs of this domain. If, instead, the successors of the states of a worker go to a few workers only, the copies of a state meet more often on the same worker in the order of their f, and fewer states are reopened.

Definition 15.7 (The hyperplane distribution). Let ℓ be the level of a levelled graph, p the number of workers, Z a hash of the states, and the thickness d either a positive integer or the inverse $1/k$ of one. The plane of a state s is

$$\mathit{Plane}(s) = \lfloor \ell(s)/d \rfloor \quad (d \geq 1), \qquad \mathit{Plane}(s) = k\,\ell(s) + (Z(s) \bmod k) \quad (d = 1/k),$$

and the owner of s is $\mathit{Plane}(s) \bmod p$.

With $d \geq 1$ the slab of d consecutive levels goes to one worker, and with $d = 1/k$ each level is split by the hash among k consecutive planes, i.e., a level is shared by k workers; consecutive planes go to consecutive workers. In the library, the distribution is the class template `HyperplaneDistribution< D >` (Distribution.h:531-586), for any `D` that models `LevelDomain`; its members are the following (quoted from the header, without their bodies):

```
explicit HyperplaneDistribution( const D & domain , std::uint64_t seed = 1 ,
                                 unsigned planes = 1 , unsigned split = 1 );
unsigned owner( const state_t< D > & s , unsigned p ) const;
static std::pair< unsigned , unsigned > thickness(
               const D & domain , unsigned p , double lambda = 0.003 );
static std::string name( void );
```

The thickness is \$d = \$ `planes` if `split` is 1 and $d = 1/$`split` if `planes` is 1; the constructor throws `std::invalid_argument` if either is 0 or both exceed 1. `owner( s , p )` (Distribution.h:550) computes the plane of Definition 15.7, with $Z(s)$ the hash of the `Domain` mixed with the seed by SplitMix64 (Kobayashi et al. take the Zobrist hash, of which only the residue modulo k is used), and reduces it modulo p; unlike the other distributions of §9.4, it does not use the multiply-high reduction, since the consecutive planes must go to consecutive workers. `name()` is `"Hyperplane"`. This distribution has no incremental key, and HDA* calls `owner` on each successor, at the cost of `level` (a sum of n positions) and, for $d < 1$, of the hash of the state.

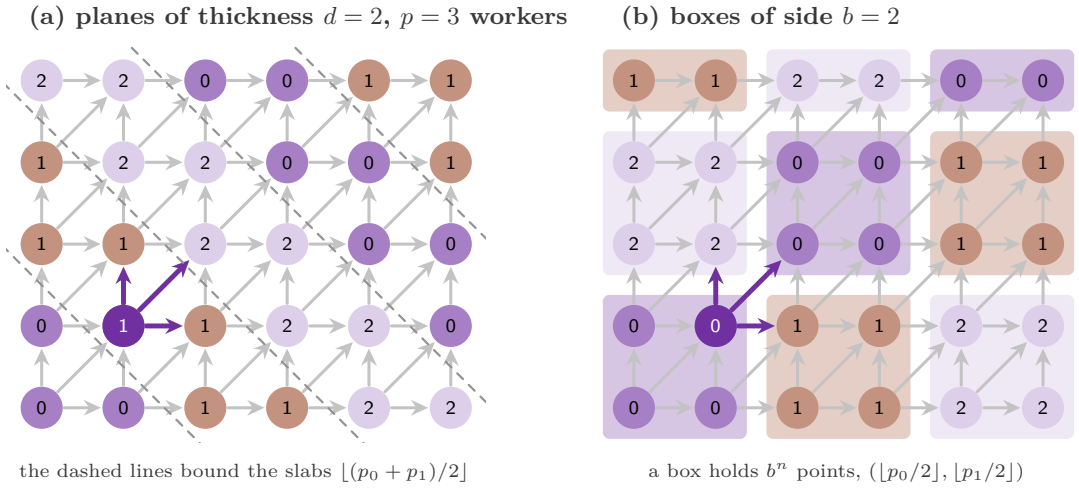


Figure 15.2. (a) The owners of the points of a lattice of two sequences under the hyperplane distribution of thickness $d = 2$ with $p = 3$ workers: the dashed lines bound the slabs of two levels, and the successors of the point in the primary colour (level 2) go to the workers 1 and 2. (b) The boxes of side $b = 2$ of AZH and of AHDA*, each owned by one worker (the owners drawn for the figure): each move inside a box keeps the owner of its state, while the three successors of the point in the primary colour lie in three boxes.

The following theorem bounds the number of workers that the successors of the states of a worker reach, over all the states of that worker; Figure 15.2 shows it for $d = 2$.

Theorem 15.8 (The workers reached from a worker). Let every move raise the level by at least 1 and at most Δ ($\Delta = n$ for the alignment of n sequences), and let w be a worker. Under the hyperplane distribution, the successors of all the states owned by w are owned by at most

$$\min\{p, \lceil \Delta/d \rceil + 1\} \quad (d \geq 2), \qquad \min\{p, \Delta\} \quad (d = 1), \qquad \min\{p, (\Delta + 1)k - 1\} \quad (d = 1/k)$$

workers, namely the workers $w + j \bmod p$ for j in $\{0, \ldots, \lceil \Delta/d \rceil\}$, in $\{1, \ldots, \Delta\}$ and in $\{1, \ldots, (\Delta + 1)k - 1\}$ respectively.

Proof. Let s be a state of w and t a successor of s, so that $\ell(s) + 1 \leq \ell(t) \leq \ell(s) + \Delta$. Let first $d \geq 1$ be an integer and $P = \lfloor \ell(s)/d \rfloor$, so that $Pd \leq \ell(s) \leq Pd + d - 1$. Then $\mathit{Plane}(t) \geq P$, and

$$\mathit{Plane}(t) \leq \left\lfloor \frac{Pd + d - 1 + \Delta}{d} \right\rfloor = P + \left\lfloor \frac{\Delta + d - 1}{d} \right\rfloor = P + \lceil \Delta/d \rceil.$$

Since $P \equiv w \pmod p$, the owner of t is $w + j \bmod p$ with $0 \leq j \leq \lceil \Delta/d \rceil$, a set that does not depend on s. If $d = 1$, $\mathit{Plane}(t) = \ell(t) > \ell(s) = P$, so that $j \geq 1$, which leaves Δ values. Let now $d = 1/k$, with $\mathit{Plane}(s) = k\ell(s) + r$ and $\mathit{Plane}(t) = k\ell(t) + r'$, $0 \leq r, r' < k$. Then

$$\mathit{Plane}(t) - \mathit{Plane}(s) \geq k(\ell(s) + 1) - k\ell(s) - r = k - r \geq 1,$$

$$\mathit{Plane}(t) - \mathit{Plane}(s) \leq k(\ell(s) + \Delta) + k - 1 - k\ell(s) - r \leq (\Delta + 1)k - 1,$$

and the owner of t is $w + j \bmod p$ with $1 \leq j \leq (\Delta + 1)k - 1$. In each case the number of the values of j, and the number p of the workers, bound the number of the owners. □

Remark 15.9 (The bound of Kobayashi et al.). The header of `HyperplaneDistribution` reports that Remark 1 of [43] writes the bound $n/d + 1$ for $d \geq 1$, which fails when d does not divide n: with $n = 4$ and $d = 5$, a state of level 4 (plane 0) has a successor of level 5 (plane 1), and the successors of a worker reach two workers, more than $4/5 + 1$; Theorem 15.8 gives $\lceil n/d \rceil + 1$ instead. For $d = 1/k$, the case `msa` checks the bound $(n + 1)/d = (n + 1)k$, which Theorem 15.8 lowers by one. We have not checked the paper itself. In the case `msa`, we check the bounds $\lceil n/d \rceil + 1$ and $(n + 1)k$ with 64 workers on the instance `test3` of 4 sequences, for $d \in \{1, 2, 5, 1/2, 1/4\}$, against the owners of all the moves of the lattice.

Under Zobrist hashing, in contrast, a worker owns states scattered over the whole lattice, and the successors of its states may reach all the p workers; under the boxes, the successors of a single state lie in at most $2^n - 1$ boxes other than its own, but the successors of all the states of a worker are bounded by nothing smaller than p.

Of course, the hyperplanes cost some load balance: the levels are expanded roughly in the order of f, and if d is large, the few slabs that hold the nodes of the current f belong to a few workers. The thickness trades the two effects, and `thickness( domain , p , lambda )` gives the rule of section 4.3 of [43] (as the header reports it): with

$$x = \frac{\lambda\, L}{\log_2 p}, \qquad L = \texttt{max_level()},$$

the thickness is round(x) if $x \geq 1$ and $1/\text{round}(1/x)$ otherwise, with $\lambda = 0.003$ as in that paper, returned as the pair (`planes`, `split`) of the constructor, and $(1, 1)$ for $p < 2$. That paper does not give the base of the logarithm (the header remarks), and the library takes 2. For instance, on the instance `test3`, of total length $L = 73$, with $p = 64$, $x = 0.003 \cdot 73/6 = 0.0365$ and the rule gives $d = 1/27$, the value that the case `msa` checks.

15.6 The instance sets

The directory `data/` holds the instances on which the tests and the benchmarks run, each set with a `NOTICE` (or a `LICENSE`, or comments at the head of the file) that says where it comes from and under which license it is redistributed. Table 15.1 summarizes them; the paragraphs that follow give the details, and section 15.7 the readers.

Table 15.1. The instance sets of `data/`: the directory, what it holds, the reader that parses it (section 15.7, or the chapter that describes the domain), the source and the license of the files.

directory	contents	reader	source, license
`15-puzzle`	`korf100.txt` (100 boards), `korf_1`, `korf_12`, `korf_55`	`SlidingTile::parse`, `SlidingTile::read_positions`	Korf [7]; the code of PBNF, MIT
`24-puzzle`	`instances50.txt`, `instances100.txt`, `fromkorf8.txt`	`SlidingTile::parse`	the code of HDA* and the benchmark set of Jinnai, MIT
`grid`	5 boards of 10×8 to 500×500 cells	`GridBoard::read`	the code of PBNF and the benchmark set of Jinnai, MIT
`msa`	36 files `.tfa` in `balibase`, `easy`, `papers`	`SequenceAlignment::read_fasta`	BAliBASE 3.0 [45]; the sets of Jinnai, MIT
`tsp`	`tsp_40_0` to `tsp_40_9`, 40 cities each	`TravelingSalesman::read_cities`	the benchmark set of Jinnai, MIT
`rubik`	`korf10.txt`, 10 scrambles of 100 moves	`RubiksCube::parse_moves`	Korf [59]; HOG2 [165], MIT
`hanoi`	`toh20.txt`, 20 instances of 4 pegs and 20 disks	`TowersOfHanoi::read`	Siag et al. [57], MIT
`dimacs`	`sample.gr`, `sample.co` (900 nodes), `brock200_1.clq`	`read_dimacs_graph`, `read_dimacs_coordinates`	written by `tools/bench/dimacs_sample.py`; the clique graph from YewPar, MIT
`qaplib`	15 instances `.dat` of 12 to 16 facilities	the QAP of Chapter 13	QAPLIB [78]; P3D-DFS, BSD 3-clause
`planning`	28 tasks `output.sas` and `optimal.txt`	`SASTask::read_file`	IPC, translated by `tools/pddl2sas.py` (Chapter 16)
`robot-nav-2d`	5 Moving AI maps, 50 pairs of points each	`NavMap::read_movingai`, `RobotNav2D::read_points`	the code of ePA*SE [121], MIT

The sliding-tile puzzles

`data/15-puzzle/korf100.txt` holds the 100 random instances of the 15-puzzle of Korf [7], one per line, the 16 tiles cell by cell (0 being the blank and the goal $0, 1, \ldots, 15$) followed by the optimal cost, as described in §14.2. The files `korf_1`, `korf_12` and `korf_55` are the instances 1, 12 and 55 in the format of Burns et al. [3], which gives the cell of each tile (section 15.7), taken from the directory src/tiles of the code of PBNF under the MIT

license of its authors (NOTICE). In data/24-puzzle, instances50.txt holds the 50 instances of the 24-puzzle distributed with the code of HDA* of Jinnai and Fukunaga [5], instances100.txt the 100 instances of the benchmark set of Jinnai (of which the instances 1 to 7 and 11 are those of fromkorf8.txt, and the instances 82, 83 and 84 repeat the instances 8, 9 and 10), and fromkorf8.txt the 8 instances of that set taken from Korf, which its list instance_numbers_korf numbers 1, 2, 7, 8, 10, 12, 15 and 18; each file gives, in its comments, its origin and the MIT license of the code it comes from, and holds one board per line, the 25 tiles cell by cell. In the case files, we check that the three files korf_* are the corresponding instances of korf100.txt, and that the instances of instances100.txt and fromkorf8.txt are boards from which the goal is reachable (a board of the wrong parity is refused by SlidingTile::parse, §14.2).

The grids

data/grid holds five boards in the format of Burns et al. [3] (section 15.7): ethan_unit_four-way_1 (a board of 10×8 cells), jordan_unit_four-way_100x100_1, jordan_unit_four-way_500x500_1 and jordan_life_four-way_500x500_1, from the directory src/grid of the code of PBNF, whose row 0 is the last line of the board; and grid_500-500-0.7_0, a board of 500×500 cells each free with probability 0.7, from the benchmark set of Jinnai, whose row 0 is the first line. The larger boards of that set (1000×1000 to 5000×5000 cells, and the boards of 500×500 cells with the probabilities 0.45 to 0.6) are not redistributed; they are written by its generator grid-gen/grid_gen.cc, which draws each cell free with the given probability and the start and the goal in the first and in the last third of each side, and the reader of the library reads them as they are. Both repositories are under the MIT license (NOTICE). In the case board, we check the optimal costs of the five boards (23, 143, 763, 362873 and 755) against Dijkstra's algorithm on the board and against the costs that the code of their authors finds.

Multiple sequence alignment

data/msa holds 36 files in the FASTA format (.tfa) of BAliBASE 3.0 of Thompson et al. [45], in three directories:

- balibase: the 15 files of the directory msa/BAliBASE of the benchmark set of Jinnai (4 to 6 sequences of 51 to 974 residues), sequences of BAliBASE 3.0 as selected there;
- easy: the 6 files test0 to test5 of BAliBASE/easy of the code of HDA* of Jinnai and Fukunaga, short sequences cut from the same benchmark (3 to 7 sequences of 3 to 46 residues);
- papers: the 15 files of the directories RV11 and RV12 of BAliBASE 3.0 used in the experiments of Kobayashi et al. [43] (BB12003, BB12022, BB12023, BB12032, BB12036, BBS11026, BBS11037, BBS12005, BBS12010, BBS12023, BBS12032) and of Jinnai and Fukunaga ([5], section 4.1.3: BB12021, BB12022, BB12036, BBS11010, BBS11026, BBS11035, BBS11037, BBS12016, BBS12023, BBS12032), taken from the archive of BAliBASE (4 to 9 sequences of 58 to 738 residues); BBS11035 is also in balibase.

The files of balibase and easy come from repositories under the MIT license; the archive of BAliBASE states no license, and its README names J. D. Thompson as the contact (NOTICE). In the case msa, we check the optimal costs of five of the instances of easy against the dynamic program on the whole lattice (those of 4 sequences) and against the costs that the code of HDA* of Jinnai and Fukunaga finds; Table 15.2 gives them, with the size of the lattice. It also reads the 15 files of balibase and computes their heuristic at the origin, solves test2 (7 sequences) with A*, and solves test0 with IDA* and with the search with delayed duplicate detection.

Table 15.2. The instances of data/msa/easy with the costs of PAM250 and no gap penalty ($P = 0$): the lengths of the sequences, the number of the points of the lattice, and the optimal cost, as checked by the case msa of test/files.cpp.

instance	lengths	points of the lattice	optimal cost
test0	9, 5, 16, 15	16 320	1741
test1	31, 12, 6, 11	34 944	2665
test2	17, 13, 13, 9, 16, 3, 5	14 394 240	6055
test3	19, 19, 20, 15	134 400	2022
test4	46, 38, 39	73 320	2312
test5	17, 4, 7, 3, 6, 3, 5	483 840	3990

Random instances, as those of section 4.1.3 of [5], are drawn by random_sequences (section 15.7) and are not stored.

The traveling salesperson

`data/tsp` holds the first 10 files of `tsp/40_cities` of the benchmark set of Jinnai, `tsp_40_0` to `tsp_40_9`, each the number of the cities (40) followed by the coordinates x and y of each city in the unit square, under the MIT license (`NOTICE`). The domain of §14.7 takes the first n cities of a file. In the case `files`, we check the optimal tours of the first 6, 8, 10 and 12 cities of `tsp_40_0` with the Manhattan distances (28975, 31254, 32734 and 33861, those that the code of HDA* of its authors finds). It also checks the tours of 6, 9 and 12 cities with both metrics against the dynamic program of Held and Karp [14].

Rubik's cube and the Towers of Hanoi

`data/rubik/korf10.txt` holds the ten instances of Korf [59], each a scramble of 100 moves in the notation of Singmaster (`-` for a quarter turn counterclockwise, `2` for a half turn), one per line, from `apps/rubik` of HOG2 [165] under the MIT license. `data/hanoi/toh20.txt` holds the 20 instances of the Towers of Hanoi with four pegs and 20 disks of Siag et al. [57], each line giving the four pegs of the start and then the four pegs of the goal, a peg being its number in parentheses followed by its disks from the bottom, from the code of that paper under the MIT license. The domains and the readers of both (`RubiksCube::parse_moves`, `TowersOfHanoi::read`) are described with the permutation puzzles and their pattern databases.

The DIMACS graphs

`data/dimacs` holds `sample.gr` and `sample.co`, a synthetic graph of 900 nodes and 3106 arcs (a perturbed grid of 30×30 nodes, the lengths in meters) in the format of the road graphs of the 9th DIMACS Implementation Challenge [64], written by `tools/bench/dimacs_sample.py` and small enough for the tests; the graphs of the challenge (from NY, of 264346 nodes, to USA, of 23947347) are downloaded by `tools/bench/fetch_dimacs.sh`. It also holds `brock200_1.clq`, a graph of 200 vertices and 14834 edges of the second DIMACS Implementation Challenge (by M. Brockington) [174], copied from the repository YewPar of B. Archibald under the MIT license (`LICENSE`), on which the case `bb-domains` runs the maximum clique of Chapter 13.

The other sets

`data/qaplib` holds 15 instances of QAPLIB [78] (`chr12a` to `tai12a`), copied from the repository P3D-DFS of G. Helbecque under the BSD 3-clause license (`LICENSE`), for the quadratic assignment of Chapter 13. `data/planning` holds the 28 planning tasks of Chapter 16, whose list `optimal.txt` states their origin and the absence of a license of the tasks of the International Planning Competitions. `data/robot-nav-2d` holds five maps of the Moving AI benchmarks [15] (`brc203d`, `den501d`, `den520d`, `hrt201n` and `ht_chantry`), each in a directory with the files *nav2d_starts.txt* and *nav2d_goals.txt* of its 50 pairs of points, the instances of the experiments of the edge-based parallel searches [121], [122]; the directory has the MIT license of the code of those experiments (`LICENSE`) and no `NOTICE`.

15.7 The readers of the instance files

Each reader is a static member or a free function that takes a `std::istream` (a test may thus read a file or a string alike) and throws `std::invalid_argument` on a malformed input. The readers of the Moving AI maps (`GridMap::read_movingai`) and of the boards of the puzzles (`SlidingTile::parse`) are described in §14.4 and §14.2.

The sequences and the matrices

`SequenceAlignment::read_fasta( in )` (`SequenceAlignment.h:336`) reads the FASTA format: each line that starts with `>` (the name of a sequence) opens a new sequence, which may span several lines; the lines before the first name are skipped, the spaces and the characters `-` and `.` of an aligned file are dropped, and the symbols are turned into upper case. It throws if no sequence is read. It does not check the symbols, which the constructor of the domain does: in particular, a file whose sequences end with `*` (as some FASTA files do) is read, and then refused by the constructor, since `*` is the gap. In the case `msa`, we check the names, the lines, the case, the gaps and the blank lines on a small string.

`SequenceAlignment::read_scores( in )` (`SequenceAlignment.h:363`) reads a `ScoreMatrix`, i.e., 625 integers row by row in the order of `amino_symbols`, the gap included, and throws if fewer are read; the score of two gaps is read and ignored, since $c(-,-)$ is always 0. In the case `msa`, we write `pam250` and read it back.

`SequenceAlignment::random_sequences( seed , n , total )` (`SequenceAlignment.h:383`) draws the random instances of section 4.1.3 of [5] (as the header reports them): n sequences (n drawn uniformly in $\{4, \dots, 9\}$ if it is 0), each of a length drawn uniformly among the integers l with $0.9\,T/n < l < 1.1\,T/n$, T being `total` (5000 there), and each residue drawn uniformly among the 20 amino acids, all from `std::mt19937_64` seeded by `seed` with the draws of Lemma 1.9, whence a seed gives the same sequences on all platforms. It throws if $n < 2$, if n exceeds `K`, or if no integer lies in the range of the lengths. For the seeds 1 to 20, the case `msa` checks the number of the sequences, the range of the lengths, the alphabet and the repeatability, and solves a small instance against the dynamic program.

The puzzles by the cells of their tiles

`SlidingTile< W , H >::read_positions( in )` (`SlidingTile.h:751`) reads the format of Burns et al. [3]: the height and the width, the words `starting positions for each tile:`, the cell of each tile from the blank (the tile 0) on, the words `goal positions:` and the goal cell of each tile, the cells counted row by row. It returns the pair of the start board and of the goal board, and throws if the size is not $H \times W$ or if the cells are not a permutation. A `SlidingTile` built from both boards searches toward that goal; `AStarBench` recognizes the format by its words and otherwise reads one board per line. We check the three files of Korf in the case `files`, and the refusal of a board of another size and of a repeated cell.

The boards of Burns et al.

`GridBoard::read( in , from_bottom )` (`Grid.h:196`, the `struct GridBoard`) reads a pathfinding instance in the format of Burns et al. [3]: the height H and the width W, the word `Board:`, H lines of W characters (`#` a blocked cell, any other character a free one; a short line is padded with free cells), a line `Unit` or `Life` (the cost of leaving a cell: 1, or its row y for the life costs), a line `Four-way` or `Eight-way` (the moves; a diagonal move is allowed even if it cuts a blocked corner), and the four numbers x_s, y_s, x_g, y_g of the start and of the goal. The files of Burns et al. count y from the last line of the board up, those of later generators from the first line down: `from_bottom` tells which, and the cells are stored with y counted from the first line, as in a `GridMap`. A blocked start is made free, which changes no shortest path, since a shortest path never comes back to its start (some instances of the authors have one). The `GridBoard` holds the `GridMap`, the start, the goal, the moves and the costs (`GridCost::cells` or `GridCost::life`), and a `Grid< 4 >` or `Grid< 8 >` is built from it with the moves it states (a `Grid< 4 >` refuses a board of 8 moves). In the case `board`, we check the reader on both origins, its refusals (no `Board:`, other costs or moves, a goal outside, missing rows), the heuristics of the life costs, which are exact on the boards without blocked cells, the cut corners, random boards against Dijkstra's algorithm, and the five boards of the data.

The cities

`TravelingSalesman::read_cities( in , metric , first )` (`TravelingSalesman.h:142`) reads the number n of the cities and their coordinates, the format of the random instances of the unit square of Jinnai and Fukunaga, keeps the first `first` cities if it is positive, and returns the matrix of the distances, Euclidean (`Metric::euclidean`, the default) or Manhattan (`Metric::manhattan`, multiplied by 10^4 and truncated to integer costs). It throws on a malformed file or on one with fewer than `first` cities.

The road graphs

The free functions of `DimacsGraph.h` read the road graphs of the 9th DIMACS Implementation Challenge [64]. `read_dimacs_graph< Cost >( in , source , target )` reads a file `.gr` (the line `p sp n m` and m lines `a u v w`, the arc (u, v) of the integer cost w, the nodes numbered from 1, each direction of a road an arc of its own) into an `ExplicitGraph< Cost >` searched from `source` to `target`, and throws if the line `p sp` is missing, if an arc is out of range or of negative cost, or if fewer arcs are read than stated. `read_dimacs_coordinates( in , n )` reads a file `.co` (the line `p aux sp co n` and n lines `v v x y`, the longitude and the latitude in millionths of a degree) into a `DimacsCoordinates`, the two vectors `lon` and `lat`. The heuristic of a node u for the target t is

$$h(u) = \lfloor \kappa\, \delta(u, t) \rfloor, \qquad \kappa = \min_{(u,v):\ \delta(u,v)>0} \frac{w(u, v)}{\delta(u, v)},$$

δ being the great-circle distance in meters, by the formula of the haversine on the sphere of radius 6371008.8 m (`dimacs_distance`); `dimacs_scale( g , c )` computes κ, lowered by the factor $1 - 10^{-9}$ against the rounding

of the distances, and `set_dimacs_heuristic( g , c , kappa )` stores h in the graph for its current target (with κ from `dimacs_scale` if `kappa` is negative) and returns κ.

Proposition 15.10 (The heuristic of the coordinates). If δ satisfies the triangle inequality, h is 0 at the target and consistent, whatever the unit of the costs.

Proof. $h(t) = \lfloor 0 \rfloor = 0$. For an arc (u, v) with $\delta(u, v) > 0$, $\kappa\,\delta(u, v) \leq w(u, v)$ by the definition of κ, and the inequality holds trivially if $\delta(u, v) = 0$. Hence $\kappa\,\delta(u, t) \leq \kappa\,\delta(u, v) + \kappa\,\delta(v, t) \leq w(u, v) + \kappa\,\delta(v, t)$, and since $w(u, v)$ is an integer, $h(u) \leq \lfloor w(u, v) + \kappa\,\delta(v, t) \rfloor = w(u, v) + h(v)$. □

The great-circle distance is a metric, but its value in floating point is not exactly one; the factor $1 - 10^{-9}$ guards against the rounding, and we have not proved that it suffices on all graphs. In the case `dimacs`, we check the consistency of h on each arc of the sample for five random pairs of nodes, and the cost of A* against the blind search.

15.8 Running the instances

The benchmark `AStarBench` (`tools/AStarBench.cpp`) selects the domain by `--domain`, the file by `--file` and the instances of a file by `--ids` (all by default); the options that concern the sets of this chapter are the following.

- `--domain tiles15 | tiles8 | tiles24 --file <file>`: a file of boards, one per line (the lines starting with `#` skipped), or a single instance in the format of `read_positions`, recognized by its words.
- `--domain board --file <board> --origin bottom | top`: a board of Burns et al., with the row 0 of its start and goal on the last line (`bottom`, the default, for the files of the code of PBNF) or on the first (`top`, for the files of the benchmark set of Jinnai); `--block` and `--nblock` give the blocks of AZH and AHDA* (100 by default) and the nblocks of PBNF.
- `--domain msa --file <file.tfa>` or `--random <seed>[:<n>[:<total>]]`: the sequences of a FASTA file, or the random ones of `random_sequences`; the matrix is PAM250 or the 625 scores of `--matrix <file>`, the gap penalty that of the matrix or the linear one of `--gap <P>` (0 by default), `--block` the side of the boxes (4 by default for `msa`) and `--nblock` that of the nblocks (16 by default for `msa`); `--dist HWD` selects the hyperplane distribution, of the thickness `--plane <d>`, `--plane 1/<k>` or `--plane auto` (the default), which takes the rule of `thickness()` for the number of workers of `--threads`.
- `--domain tsp --file <file> [--size <n>] [--metric euclidean | manhattan]`: the first n cities of a file (all of them without `--size`).
- `--domain rubik --file data/rubik/korf10.txt`, `--domain hanoi --file data/hanoi/toh20.txt`, `--domain road --file <name>` (the files `<name>.gr` and `<name>.co`, between the nodes of `--points <s>,<t>` or random ones), `--domain nav2d --file <map>` (with the pairs of points next to the map), and `--domain planning --file <task.sas>` (Chapter 16).

For instance, `AStarBench --domain msa --file data/msa/papers/BBS11035.tfa --algo hda --dist HWD --threads 8` aligns the five sequences of `BBS11035` with HDA* on 8 workers and the thickness of the rule of Kobayashi et al., and prints one line of CSV with the cost, the counters of §1.3, LB, CO and the seconds.

15.9 The benchmark

We compare the work distributions of HDA* on the alignments of `data/msa/papers`, with the costs of PAM250 and the linear gap penalty $P = 8$ of the experiments of Jinnai and Fukunaga: Zobrist hashing (Z), Abstract Zobrist hashing (AZ) and AHDA* (A) with boxes of side 4, and the hyperplanes (HWD) with the thickness of the rule of `thickness()`, each on the threads of C++ with 8 and 32 workers, against the sequential A* on 1 core. All the searches are those of `AStarBench` (`--domain msa --gap 8 --algo hda --dist Z | AZ | A | HWD`); the costs are integers, and the OPEN lists are the bucket queues.

Table 15.3. HDA* with four work distributions on the alignments of `data/msa/papers`, with PAM250 and the gap penalty 8. For each instance and each distribution the table gives the seconds (the median of two runs), the speedup over sequential A*, the search overhead SO, the communication overhead CO and the load balance LB, as defined in §8.2; the machine and the compiler are given in the note of the table.

instance	distribution	workers	seconds	speedup	SO	CO	LB

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The table is written by `manual/build/tables.py` from the CSV of the campaign; until the runs are made on an idle machine, it is a placeholder that says so. The clock is the field `seconds` of the `Solution`, which does not include the reading of the file nor the construction of the domain (the dynamic programs of the pairs, which take $O(\sum_{i<j} L_i L_j)$ time).

15.10 Notes and references

The formulation of MSA as a shortest path on the lattice and the pairwise heuristic are due to Ikeda and Imai [41], and the domain became the benchmark of the searches with a large branching factor with the Partial Expansion A* of Yoshizumi, Miura and Ishida [42]. Kobayashi, Kishimoto and Watanabe [43] evaluated HDA* on it and proposed the hyperplane work distribution, whose bound we restate in Theorem 15.8 (see Remark 15.9); Jinnai and Fukunaga [5] compared the distributions of HDA* on the same instances, with the gap penalty 8 and the boxes of side 4 that the domain takes by default. As for the data, the matrix PAM250 is that of Dayhoff, Schwartz and Orcutt [44], and the instances are those of BAliBASE 3.0 [45]. The claims on [5] and [43] in this chapter (the sections, the penalty, the remark and the rule of the thickness) are those of the comments of the headers, which we have not checked against the papers.

It is worth noting that the domain is the plainest form of the alignment: the costs are linear in the gaps (an affine gap penalty would need the state to remember which sequences are in a gap), the heuristic is the sum of the pairwise alignments (the heuristics from triples, and the pruning of the lattice by an upper bound, are not in the library), and the tables of the pairs are kept whole; the domain is therefore meant for the instances of a few hundred residues per sequence of the papers above, not for the alignment of long sequences. The instance sets of `data/` are, for each domain, those of the papers whose searches the library reproduces (a result of the library can thus be compared with a published one), and their sources and licenses are recorded next to them.

15.11 Exercises

Exercise 15.1 (`ctest -R PBeFS.msa`). Show that, with PAM250 and $P = 0$, $c(\mathtt{W}, \mathtt{C}) = 25$ and $c(\mathtt{C}, \mathtt{C}) = 5$, and that an optimal alignment of `HEAGAWGHEE` and `PAWHEAE` with $P = 8$ costs $17 \cdot 17 - 2\sigma$, σ being the best score with the gap penalty 8 (Proposition 15.2). Check both values against the case.

Exercise 15.2 (`ctest -R PBeFS.msa`). Prove that the lattice of `test5` has 483840 points and that the expansion of its origin generates 127 successors (Proposition 15.5). Why does the case compare the cost of A* with the dynamic program on the lattice only for the instances of 4 sequences?

Exercise 15.3 (`ctest -R PBeFS.msa`). For the instance `test3` (4 sequences, $L = 73$) and 64 workers, compute the bound of Theorem 15.8 for $d = 1, 2, 5, 1/2, 1/4$, and give a state of the lattice of `test3` whose successors reach $\lceil n/d \rceil + 1 = 2$ workers for $d = 5$. Check that the case finds the thickness $1/27$ of the rule of Kobayashi et al.

Exercise 15.4 (`ctest -R PBeFS.msa`). Show that the relation of `for_each_nblock_successor` without its downward boxes satisfies condition (i) of Definition 11.1 but not condition (ii), and count the abstract successors of an inner box for $n = 4$ (Proposition 15.6).

Exercise 15.5 (`ctest -R PBeFS.files`). Read `data/15-puzzle/korf_12` by `read_positions` and explain why the case finds the board of the line 12 of `korf100.txt`; then explain why a board of `instances100.txt` that `SlidingTile::parse` refused would make the case fail.

Exercise 15.6 (`ctest -R PBeFS.board`). Show that freeing a blocked start changes no optimal cost, and that the life heuristic of a board without blocked cells and with 4 moves is exact. Check the costs of the five boards of `data/grid` given in section 15.6.

Exercise 15.7 (`ctest -R PBeFS.dimacs`). Prove Proposition 15.10, and say which inequalities of the proof hold exactly for the distances computed in floating point and which may fail by a rounding error; explain why lowering κ by the factor $1 - 10^{-9}$ makes a failure less likely without excluding it. Check that the case finds h consistent on every arc of the sample.

16. Classical planning

[Source: include/PBeFS/astar/SASTask.h (SASTask, StripsTask), include/PBeFS/astar/SASHeuristics.h (RelaxedHeuristic, LMCut, SASPatternDatabase, SASMergeAndShrink), include/PBeFS/astar/Planning.h (PlanningBase, Planning, PlanningOptions, planning_heuristic), their sources src/SASTask.cpp, src/SASHeuristics.cpp and src/Planning.cpp, the translator tools/pddl2sas.py and the tasks of data/planning]

> **Status — beta.** The code that this chapter describes is in `main` but not yet in a release, and its interface may still change.

We are concerned in this chapter with classical planning as a domain of the library, i.e., with the problem of finding a cheapest sequence of operators that leads a system described by finitely many variables from an initial assignment to an assignment that satisfies a goal. Unlike the domains of Chapter 14 and Chapter 15, which are written by hand for one problem each, a planning task is read from a file, and everything that a search of the library needs (the successors, the heuristic, the features of Zobrist hashing, the abstractions of the distributions and of the nblocks, the operators with an inverse) is computed from the task itself. Hence the planning domain is the test of the domain-independent side of the parallel searches: HDA* was proposed for planning by Kishimoto, Fukunaga and Botea [2], and the automatic work distributions of Jinnai and Fukunaga [138], [5] were first built on the structure of planning tasks.

The tasks are in the SAS+ formalism of Bäckström and Nebel [133], as written by the translator of Fast Downward [134]. The heuristics are those of the literature of optimal planning: $h^{\max}$ of Bonet and Geffner [135] and the landmark-cut heuristic of Helmert and Domshlak [136], both from the delete relaxation, the pattern databases of Edelkamp [137] the merge-and-shrink abstraction of Helmert, Haslum, Hoffmann and Nissim [140] and the Cartesian abstractions refined by counterexamples of Seipp and Helmert [200], [202], together with the inadmissible h^{add} and h^{FF} [101] and the count of the unsatisfied goals. The admissibility and the consistency of a heuristic are those of §2.4, the distributions of HDA* those of §9.4, and the nblocks those of Definition 11.1.

The structure of the chapter is as follows. Section 16.1 defines the tasks, the format of their files and their STRIPS view, and section 16.2 the translator from PDDL of the library. Section 16.3 describes the domain `Planning`, i.e., the packed state, the successor generator, the dead ends and the operators with an inverse. Sections 16.4, 16.5 and 16.6 give the heuristics: those of the delete relaxation, the landmark cut, and the abstraction heuristics, with the proofs of admissibility where they hold. Section 16.7 gives the features and the abstractions for the parallel searches, section 16.8 the goal test at the generation, section 16.9 the searches that run on the tasks and the options of `AStarBench`, and section 16.10 the tasks of the data; sections 16.11 to 16.13 give the benchmark, the notes and the exercises.

16.1 SAS+ tasks and their STRIPS view

Definition 16.1 (SAS+ task). A *SAS+ task* has a set of variables $V = \{0, \dots, n-1\}$, the variable v with the finite domain $D_v = \{0, \dots, |D_v| - 1\}$; a *state* assigns a value to each variable, a *fact* is a pair $v = d$, and a *partial assignment* is a set of facts of distinct variables. Each task has an initial state s_0, a goal G, which is a partial assignment, and a set of operators. An operator o has a set of *prevail conditions*, the facts that must hold before and are not changed, a set of *effects* (v, p, q), each of which sets v to q and requires $v = p$ before unless $p = -1$, and a cost $c(o) \geq 0$. The *preconditions* $\text{pre}(o)$ are the prevail conditions and the facts $v = p$ of the effects with $p \geq 0$; o is applicable in s if s satisfies $\text{pre}(o)$, and then $o(s)$ is s with each v of an effect set to its q. A *plan* is a sequence of operators applicable in turn from s_0 that reaches a state satisfying G, and its cost is the sum of the costs of its operators; an optimal plan is a plan of least cost.

Thus the state space of a task is the graph of the states with an edge $(s, o(s))$ of cost $c(o)$ for each operator

applicable in s, and the search for an optimal plan is a shortest-path problem of Chapter 1, whose graph is given implicitly, and is exponential in the number of variables.

In the library, a task is the class `SASTask` (`SASTask.h`), read from the file `output.sas` that the translator of Fast Downward writes. Its types and its reader are declared as follows:

```
/// the fact \f$ v = d \f$
struct Fact {
 int var;                                   ///< the variable
 int val;                                   ///< the value
 };

/// the effect that sets var to post, asking var = pre if pre >= 0
struct Effect {
 int var;                                   ///< the variable
 int pre;                                   ///< the value before, or -1
 int post;                                  ///< the value after
 };

/// an operator
struct Operator {
 std::string name;                          ///< the name
 std::vector< Fact > prevail;               ///< the prevail conditions
 std::vector< Effect > effects;             ///< the effects
 int cost;                                  ///< the cost
 };

/*----------------------------- READING A TASK -----------------------------*/
 /// the task in the stream, in the format output.sas

 [[nodiscard]] static SASTask read( std::istream & in );

 /// the task in the file of the given name

 [[nodiscard]] static SASTask read_file( const std::string & name );

/*--------------------------- THE PRECONDITIONS ---------------------------*/
 /// the preconditions of the operator o: its prevail conditions and the
 /// values asked by its effects, sorted by variable

 [[nodiscard]] std::vector< Fact > preconditions( std::size_t o ) const;
```

and its data are public members:

```
std::vector< std::string > var_names;   ///< the names of the variables
std::vector< int > range;               ///< | D_v | of each variable
std::vector< std::vector< std::string > > values;
                                        ///< the names of the values
std::vector< std::vector< Fact > > mutexes;   ///< the mutex groups
std::vector< int > init;                ///< the initial value of each v
std::vector< Fact > goal;               ///< the goal
std::vector< Operator > ops;            ///< the operators
bool use_costs = false;                 ///< true if the costs are read
```

A `Fact` is a pair (`var`, `val`), an `Effect` the triple (`var`, `pre`, `post`) with `pre` equal to -1 when the effect requires nothing, and an `Operator` has a name, its prevail conditions, its effects and its cost. `preconditions( o )` returns pre(o) sorted by variable and value, without repetitions; `satisfies_goal( s )` tells whether the values `s`, one per variable, satisfy the goal; and `num_vars()` is n. The members `var_names` and `values` keep the names of the file (the values are the atoms of the translator, e.g., `Atom carry(ball1, left)`), and `range` the sizes $|D_v|$. Then `mutexes` holds the mutex groups of the file (read and kept, not used by the searches), `init` and `goal` the initial state and the goal, `ops` the operators, and `use_costs` tells whether the file has a metric.

The format output.sas and what is refused

`SASTask::read( in )` (src/SASTask.cpp:126) parses the version 3 of the format, which is made of sections. The section `begin_version` holds the number 3, and the section `begin_metric` holds 0 (the costs of the file are then ignored, and each operator costs 1) or 1 (the costs are read). Then come the number of the variables and, for each, a section `begin_variable` with its name, its axiom layer, its range and the names of its values; the number of the mutex groups and their facts; the section `begin_state` with a value per variable, and the section `begin_goal` with its facts. Last come the number of the operators and, for each, a section `begin_operator` with its name, its prevail conditions, its effects (each the line `0 v p q`, whose leading 0 is the number of the conditions of the effect) and its cost, and finally the number of the axioms. Blank lines are skipped, the spaces at the ends of a line are trimmed, and `read_file( name )` opens the file. The reader throws `std::invalid_argument`, with the number of the line, on a malformed file (a missing section, a number out of range, a fact or a value out of the range of its variable), and on what the searches of the library do not support: a version other than 3, the axioms (a variable whose axiom layer is not -1, i.e., a derived variable, or a nonzero number of axioms at the end), the conditional effects (an effect whose number of conditions is not 0), and the negative costs. In the case `planning`, we check each of these refusals and the reading of a task with costs and of one without a metric.

The STRIPS view

The heuristics of the delete relaxation (sections 16.4 and 16.5) work on propositions rather than on variables, and the library gives each task a STRIPS view, the class `StripsTask`:

```
/// a STRIPS operator, the facts sorted
struct Operator {
 std::vector< int > pre;                    ///< the preconditions
 std::vector< int > add;                    ///< the add list
 std::vector< int > del;                    ///< the delete list
 int cost;                                  ///< the cost
 };

/// the view of the task t

explicit StripsTask( const SASTask & t );
```

```
int num_facts = 0;                          ///< the propositions
std::vector< int > offset;                  ///< the first one of each v
std::vector< Operator > ops;                ///< the operators
std::vector< int > init;                    ///< the initial facts
std::vector< int > goal;                    ///< the goal facts
```

Definition 16.2 (The STRIPS view). The proposition of the fact $v = d$ is the number $\mathrm{offset}[v] + d$ (`fact( v , d )`), the offsets numbering the facts of the variables one after the other, and the task has $\sum_v |D_v|$ propositions (`num_facts`). The STRIPS operator of o has the preconditions $\mathrm{pre}(o)$, the add list $\mathrm{add}(o) = \{v = q\}$ over its effects, and the delete list $\mathrm{del}(o)$, which holds $v = p$ for an effect (v, p, q) with $p \geq 0$ and $p \neq q$, and every $v = d$ with $d \neq q$ for an effect with $p = -1$; the three lists are sorted. A state s stands for the set of propositions $F(s) = \{v = s_v : v \in V\}$.

Lemma 16.3 (The STRIPS view is faithful). An operator o is applicable in s if and only if $\mathrm{pre}(o) \subseteq F(s)$, and then $F(o(s)) = (F(s) \setminus \mathrm{del}(o)) \cup \mathrm{add}(o)$.

Proof. The first statement is the definition of applicability. For the second, a variable that no effect of o sets keeps its fact, which is in no delete list of o. For a variable v set by the effect (v, p, q), $F(s)$ holds the single fact $v = s_v$: if $p \geq 0$, then $s_v = p$, and $v = p$ is deleted unless $p = q$, in which case it is added back; if $p = -1$, every $v = d$ with $d \neq q$ is deleted, and $v = q$ is added. In both cases the facts of v in the result are exactly $\{v = q\}$. □

16.2 The translator from PDDL

The tasks of data/planning were written by the script tools/pddl2sas.py, run as `pddl2sas.py domain.pddl problem.pddl output.sas`, which translates a STRIPS task of PDDL with `:typing` (and without negative

preconditions or goals, disjunctions, quantifiers or conditional effects, which it refuses) into the format of section 16.1. It is a small translator in the spirit of that of Fast Downward [134], and it proceeds in three steps.

First, the actions are grounded by relaxed reachability: starting from the atoms of the initial state, each action schema is instantiated by a join over its preconditions with the atoms reached so far (the equalities among the parameters are checked at the end), and the atoms that the ground actions add are added to the reached set, until no new ground action appears. Atoms of the predicates that no action changes are static, and a ground action whose static precondition is false is dropped.

Second, the fluent atoms are grouped into finite-domain variables by *invariants*: a candidate invariant is a set of up to three predicates, each with the positions of the parameters that identify a group, and it is proved if the initial state holds at most one atom of each group and each ground action that adds an atom of a group also requires an atom of that group in its preconditions and deletes it (or adds back the same atom), and deletes an atom of a group only if it requires an atom of the group. By induction on the length of a plan, a proved invariant holds in all the reachable states: each group has at most one true atom. The groups, the largest first, become the variables (each atom in one variable only), with the value `<none of those>` when an action may delete the atom of a group without adding another or the initial state has no atom of the group, and the atoms left out become binary variables, with the values `Atom` and `NegatedAtom`.

Third, each ground action becomes an operator: its preconditions on the variables are the prevail conditions when the operator does not change the variable and the values required by its effects otherwise, its add effects set their variables, a delete effect that is not replaced by an add sets the variable to `<none of those>` (requiring nothing if the action does not require the atom), and each operator costs 1 (the metric is 0). Finally, the mutex groups of the output are the groups of the variables. We have not proved the translation equivalent to the PDDL task in general; the case `planning` checks the optimal costs of the 28 tasks of the data, which agree with the bounds of the public collections (section 16.10).

16.3 The domain Planning

A task becomes a `Domain` of the library through the class template `Planning< W >` (`Planning.h:432`), which derives from the class `PlanningBase` (`Planning.h:182`), the part that does not depend on the size of the state. The heuristic and the parameters of the abstractions are chosen by a `PlanningOptions`:

```
/// the heuristic of a Planning domain

enum class PlanningHeuristic {
 blind ,        ///< 0, admissible
 goal_count ,   ///< the goal facts that do not hold, not admissible
 hmax ,         ///< h^max, admissible
 hadd ,         ///< h^add, not admissible
 hff ,          ///< h^FF, not admissible
 pdb ,          ///< a pattern database, admissible
 lmcut ,        ///< the landmark-cut heuristic, admissible
 mas ,          ///< merge-and-shrink, admissible
 cegar ,        ///< a Cartesian abstraction refined by CEGAR, admissible
 cegar_add      ///< the Cartesian abstractions of the goal facts under
                ///< saturated cost partitioning, admissible
 };

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// the heuristic of the given name: blind, goalcount, hmax, hadd, hff, pdb,
/// lmcut, mas, cegar or cegaradd; throws std::invalid_argument on another
/// name

[[nodiscard]] PlanningHeuristic planning_heuristic( const std::string & n );

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/// the options of a Planning domain

struct PlanningOptions {

 PlanningHeuristic heuristic = PlanningHeuristic::lmcut;  ///< the heuristic
```

```
 /// the pattern of the pattern database, the greedy one of
 /// SASPatternDatabase if empty
 std::vector< int > pattern;

 /// the most abstract states of the greedy pattern
 std::size_t pdb_size = std::size_t( 1 ) << 16;

 /// the most abstract states of merge-and-shrink, the size of the
 /// experiments of Jinnai and Fukunaga (JAIR 2017, section 6.3)
 std::size_t mas_size = 1000;

 /// the most abstract states of cegar, and of all the abstractions of
 /// cegar_add together
 std::size_t cegar_size = 1000;

 /// the most seconds of the refinement of cegar and cegar_add, 0 for no
 /// limit
 double cegar_time = 0;

 /// the fraction of the variables, those of largest fluency, that
 /// Abstract Zobrist hashing leaves out [see PlanningBase]; 0 leaves out
 /// none
 double ignored = 0.3;

 /// the most nblocks of the abstraction of PBNF and of AHDA*
 std::size_t max_nblocks = 10000;

 /// if positive, the largest fraction of the facts that the variables of
 /// the abstraction of the nblocks may hold
 double nblock_facts = 0;

 };  // end( struct( PlanningOptions ) )
```

whose fields are the following.

- `heuristic`: the heuristic of the domain, a `PlanningHeuristic`, LM-cut by default; `blind` is 0, `goal_count` the number of the facts of the goal that do not hold, and the others are those of sections 16.4 to 16.6. `planning_heuristic( name )` maps the names `blind`, `goalcount`, `hmax`, `hadd`, `hff`, `pdb`, `lmcut`, `mas`, `cegar` and `cegaradd` to the values (the last two being `cegar` and `cegar_add`), and throws `std::invalid_argument` on another name.
- `pattern`: the variables of the pattern database; if empty, the greedy pattern of section 16.6 of at most `pdb_size` abstract states (2^{16} by default).
- `mas_size`: the largest number of abstract states of merge-and-shrink (1000 by default, the size of the experiments of Jinnai and Fukunaga, section 6.3 of [5] as the header reports it).
- `cegar_size` and `cegar_time`: the largest number of abstract states of the Cartesian abstraction of `cegar`, and of all the abstractions of `cegar_add` together (1000 by default), and the largest number of seconds of their refinement (0, i.e., no limit, by default).
- `ignored`: the fraction of the variables of largest fluency that the abstraction of Abstract Zobrist hashing leaves out (0.3 by default; section 16.7).
- `max_nblocks` and `nblock_facts`: the largest number of nblocks (10000 by default) and, if positive, the largest fraction of the facts that the variables of the abstraction of the nblocks may hold (section 16.7).

The constructor of `PlanningBase` and its static member `words_for` are declared as follows:

```
 /// constructor, packing the variables into at most max_words words of 64
 /// bits; throws std::invalid_argument if they do not fit

 PlanningBase( SASTask task , const PlanningOptions & o ,
               std::size_t max_words );

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
 /// the words of 64 bits that the state of the task t needs
```

```
[[nodiscard]] static std::size_t words_for( const SASTask & t );
```

The constructor (src/Planning.cpp:283) takes the task by value (it is kept, with its STRIPS view, behind shared pointers, and the copies of a domain share them), packs the variables into at most `max_words` words and throws `std::invalid_argument` if they do not fit, builds the successor generator, the codes of the operators with an inverse, the abstract features and the heuristic of the options, and leaves the nblocks to their first use. It also throws if an operator requires two values of one variable (a precondition that no state satisfies) or if the task has 2^{32} operators or more. `Planning< W >` (`W` is 4 by default) adds to it the state of `W` words of 64 bits, `std::array< std::uint64_t , W >`, and the members of a `Domain`; its constructor `Planning( SASTask task , const PlanningOptions & o = {} )` passes `W` as `max_words`. The domain models `IncrementalFeatureDomain`, `AbstractFeatureDomain`, `FeatureGroupDomain`, `FeatureTransitionDomain`, `AbstractStateDomain`, `NBlockDomain`, `OperatorDomain` and `DiskDomain` (§1.5), and, with `W = 1`, `PerfectHashDomain` (the packed word is the perfect hash, `perfect_hash( s )`); it is neither a `LevelDomain` nor a `PairwiseDomain` nor a `BidirectionalDomain`. A `static_assert` of the tester checks these concepts on `Planning< 4 >` and on `Planning< 1 >`.

The packed state

Proposition 16.4 (The packed state). The variable v takes a field of $b_v = \max\{1, \lceil \log_2 |D_v| \rceil\}$ bits, and the fields fill the words in the order of the variables, a field that does not fit in the rest of a word opening the next one; hence no field crosses two words, and `words_for( t )` is the number of the words so used. The value of a variable is read by a shift and a mask in $O(1)$, an operator is applied in $O(e)$ for e effects, and two states are compared and hashed in $O(W)$.

Proof. The bits of b_v are those of `std::bit_width( |D_v| - 1 )`, at least 1, which hold the values $0, \dots, |D_v|-1$. Here the function `fields` of src/Planning.cpp:207 opens a new word when `used + b > 64`, so that each field lies in one word. `value( s , v )` is `( s[ word ] >> shift ) & mask`, and `apply( s , o )` clears and sets the field of each effect. Both the comparison of `std::array` and the hash, which chains SplitMix64 over the words, take $O(W)$. □

`value( s , v )` reads a variable, `unpack( s )` returns the values of all the variables, and `pack( x )` builds the state of the values `x`. All the 28 tasks of data/planning fit in one word, and `Planning< 1 >` keeps their states in 8 bytes; a task of 65 binary variables needs two words, which the case `planning` checks, together with the refusal of `Planning< 1 >` for it.

The successor generator

An operator is applicable in s if its preconditions hold, and testing each operator in turn costs $O(|O|)$ per expansion. Instead, the library builds the *successor generator* of Fast Downward [134], a decision tree over the variables in the order of their indices: a node tests a variable v, has a child for each value of v required by some operator below it and a *don't care* child for the operators that require nothing of v, and lists the operators whose preconditions have all been tested on the way to it. It is built by the recursive procedure `GenBuilder::build` of src/Planning.cpp:78, applied to all the operators, each with the index 0 of its next precondition in the sorted list pre(o): the operators whose preconditions are exhausted are listed at the node, the node tests the least variable v among the next preconditions of the others, the operators whose next precondition is on v go, with the index advanced, to the child of the value they require, and the others go unchanged to the don't care child. We then enumerate the successors by the following procedure, which transcribes `PlanningBase::visit` (Planning.h:339; the header has no pseudocode of it).

Algorithm 16.1 (The visit of the successor generator).

```
1  procedure VISIT(n, s)
2     for each operator o listed at the node n do report o
3     if n tests no variable then return                    ▷ a leaf
4     v ← the variable tested at n
5     if n has a child c for the value s_v then VISIT(c, s)
6     if n has a don't care child d then VISIT(d, s)
```

Lemma 16.5 (The successor generator). VISIT(root, s) reports each operator applicable in s exactly once, and no other operator; the depth of the tree is at most n, and a visit costs $O(n_s + a)$, n_s being the number of the

nodes visited and a the number of the operators reported.

Proof. Let o be an operator and $(v_1, d_1), \ldots, (v_k, d_k)$ its preconditions sorted by variable. By construction, o follows from the root a single path: at a node that tests v, o goes to the child of d_i if its next precondition is on $v = v_i$, and to the don't care child otherwise (the next variable of o is then larger than v, since the node tests the least next variable); o is listed at the node where its preconditions are exhausted. Along a path, the variables tested increase strictly, whence the depth. VISIT(s) follows, at each node, the child of s_v and the don't care child, i.e., exactly the edges whose condition s satisfies; hence it reaches the node of o if and only if $s_{v_i} = d_i$ for each i, i.e., if and only if o is applicable, and it reaches each node at most once, since the tree has no two paths to a node. □

Figure 16.1 shows the generator of a small task, which we use throughout the chapter: a package p and a truck t, with the variables $p \in \{\texttt{A}, \texttt{B}, \texttt{T}\}$ (the package at A, at B, or in the truck) and $t \in \{\texttt{A}, \texttt{B}\}$, the operators `load-A`, `unload-A`, `load-B`, `unload-B` (each requiring the truck at its place) and `drive-AB`, `drive-BA`, the initial state $p = \texttt{A}$, $t = \texttt{A}$, the goal $p = \texttt{B}$, unit costs, and the optimal plan `load-A`, `drive-AB`, `unload-B` of cost 3.

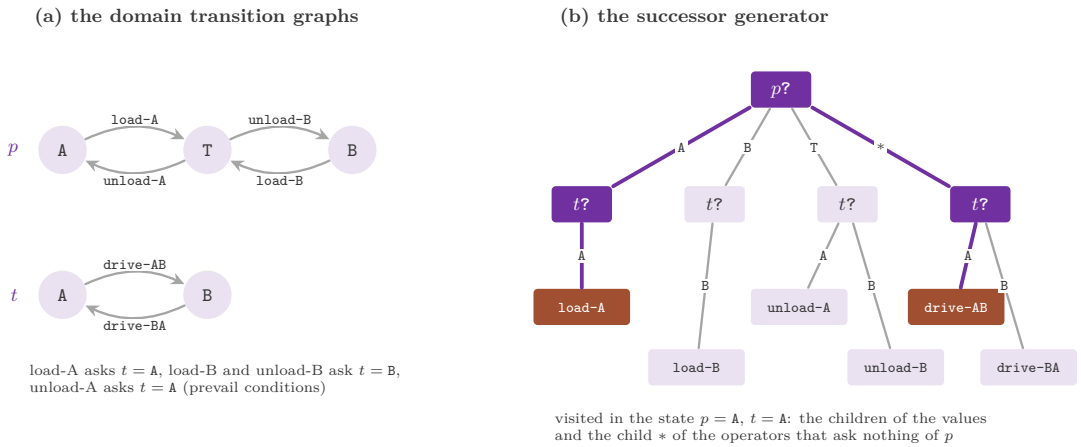


Figure 16.1. A small task. (a) The domain transition graphs of its two variables, the package p and the truck t, an arc labelled by the operator that realizes it. (b) Its successor generator, p (the variable 0) being tested first: the leaves list the operators, and the nodes and the edges in the primary colour are those that the visit of the state $p = \texttt{A}$, $t = \texttt{A}$ follows, which reports `load-A` and `drive-AB` (in the accent colour).

`for_each_successor( s , h , f )` (`Planning.h:540`) visits the generator and, for each operator o reported, computes $t = o(s)$ and its heuristic, and calls `f( t , c(o) , h(t) )` unless t is a dead end (see below); the form `for_each_successor( s , h , op , f )` of `OperatorDomain` also passes the code of o and skips the operators that undo the operator of code `op` (see below). `plan( path )` returns the names of the operators of a path of states, e.g., the `path` of a `Solution`, choosing at each step the cheapest operator that leads from a state to the next, and throws if two consecutive states are not joined by an operator; the tests check each plan by simulating it on the values of the variables (`simulate` of `test/test.h`).

Dead ends

A heuristic of section 16.4 to 16.6 may prove that no goal is reachable from a state, and then it returns the constant `dead_end` (the largest `int`) of `SASHeuristics.h`. `heuristic_or_dead( s )` returns the value of the heuristic of the options with `dead_end`, and `heuristic( s )` returns 0 instead of `dead_end`, which keeps f within the range of the bucket queues (Chapter 3).

Proposition 16.6 (Dead ends). A successor that the heuristic proves a dead end is not generated, and this changes no plan. Hence only the initial state can be a dead end among the states that a search handles; if it is, its value 0 is admissible, and the search expands it alone and ends without a goal.

Proof. A dead end has no path to a goal, and no plan passes through it, and dropping it from the successors removes no plan. Every other state that a search handles is a generated successor, hence not a dead end. A dead end s that is the initial state has $h(s) = 0$, admissible since $h^*(s) = \infty$. Its successors are dead ends for each heuristic that can return `dead_end`: for those of the delete relaxation, because the facts reachable in the relaxation from $o(s)$ are among those reachable from s (the facts of $o(s)$ are those of s or added by o, which is applicable in s); for the pattern database and merge-and-shrink, because the abstract state of $o(s)$ is reached in

the abstraction from that of s (section 16.6). Hence none of them is generated, and the search ends after one expansion. □

The case `planning` checks this on a task whose goal no operator reaches: with $h^{\max}$ the initial state is a dead end, `heuristic` returns 0, and A* returns `unsolvable` after one expansion.

The operators with an inverse

Lemma 16.7 (Inverse operators). Let o and o' be two operators such that (i) o' has an effect on exactly the variables of the effects of o, (ii) each effect (v, p, q) of o has $p \geq 0$, and the effect of o' on v sets v to p, and (iii) each precondition $v = d$ of o' is either on a variable of an effect (v, p, q) of o, with $d = q$, or a precondition of o. Then, for each state s in which o is applicable, o' is applicable in $o(s)$ and $o'(o(s)) = s$.

Proof. Let $t = o(s)$. A precondition $v = d$ of o' of the first kind holds in t, since $t_v = q = d$; one of the second kind is a precondition of o, which holds in s, on a variable that o does not change, and hence it holds in t. So o' is applicable in t. It changes exactly the variables of the effects of o, each to the value p that o required before, i.e., to s_v; the other variables are those of s. □

The function `inverse` of `src/Planning.cpp:230` checks the conditions of Lemma 16.7 (with the same number of effects, each variable of o found among those of o'), and the constructor gives the first 255 operators that have an inverse the codes $0, \ldots, 254$ (`code( o )`), the others the code `no_op`, with the list of the inverses of each code (`undoes( c , o )` tells whether o is an inverse of the operator of code `c`). The form of `for_each_successor` of `OperatorDomain` then skips, at a state generated by the operator of code `op`, its inverses, which lead back to the parent; since the costs are not negative, this skips no cheaper path, as §2.9 shows for the parent pruning in general. In the task of Figure 16.1, `unload-A` is an inverse of `load-A` and `drive-BA` of `drive-AB`. Along an optimal plan of four tasks, the case `planning` checks that a state generated by a coded operator skips its parent and nothing else. `Planning` passes `no_op` for an operator without an inverse (or beyond the first 255), which the concept `OperatorDomain` allows and which only means that nothing is skipped after it.

16.4 The delete relaxation: $h^{\max}$, h^{add} and h^{FF}

The *delete relaxation* of a task ignores the delete lists of its STRIPS view: a relaxed state is a set of propositions that only grows, and a relaxed plan from s is a sequence of operators, each with its preconditions in the facts of s or added before, whose additions cover the goal. Since a plan is a relaxed plan (by Lemma 16.3, the facts of the states of a plan are among those of the relaxed states), the cost $h^+(s)$ of an optimal relaxed plan is an admissible heuristic; it is NP-hard to compute, and the heuristics of this section approximate it.

Definition 16.8 (The heuristics of the relaxation). For a state s, let $h_s(q) = 0$ for the facts $q \in F(s)$ and, for the other facts,

$$h_s(q) = \min_{o : q \in \text{add}(o)} \Big(c(o) + \bigoplus_{r \in \text{pre}(o)} h_s(r) \Big),$$

the minimum of the empty set being ∞ and $\bigoplus$ being the maximum for $h^{\max}$ and the sum for h^{add} (the empty aggregate being 0); then $h^{\max}(s) = \max_{g \in G} h_s(g)$ and $h^{\text{add}}(s) = \sum_{g \in G} h_s(g)$, both ∞ (a dead end) if a goal fact has $h_s(g) = \infty$. For h^{FF}, each fact $q \notin F(s)$ of finite $h_s(q)$ under h^{add} has a *best supporter*, the operator that attains the minimum; the relaxed plan is traced back from the goal facts through the best supporters and their preconditions, and $h^{\text{FF}}(s)$ is the sum of the costs of its operators, each counted once.

In the library, the class `RelaxedHeuristic` computes the three:

```
/// the heuristic of the relaxation
enum class Kind {
 hmax ,     ///< the maximum over the preconditions
 hadd ,     ///< the sum over the preconditions
 hff        ///< the cost of the relaxed plan of h^add
 };

/// constructor, on the STRIPS view of a task, which must outlive it

RelaxedHeuristic( const StripsTask & task , Kind kind );

/// the value on the state of the values s, one per variable
```

```
[[nodiscard]] int operator()( const std::vector< int > & s ) const;
```

Its constructor splits each operator into its *unary operators* $\mathrm{pre}(o) \to q$, one for each $q \in \mathrm{add}(o)$, and indexes them by their preconditions; `operator()( s )` (src/SASHeuristics.cpp:87) evaluates the values `s`, one per variable, by the generalized Dijkstra algorithm of Bonet and Geffner [135], which the following procedure transcribes (the header has no pseudocode of it).

Algorithm 16.2 (The heuristics of the relaxation).

```
 1  procedure RELAXED(s)
 2     h(q) ← ∞ and best(q) ← nil for each fact q
 3     for each q ∈ F(s) do h(q) ← 0;  push (0, q)
 4     for each unary operator u: ∅ → q do relax q to c(u) by u
 5     for each unary operator u do left(u) ← |pre(u)|;  agg(u) ← 0
 6     while a goal fact is unsettled and the queue is not empty do
 7        (k, q) ← the pair of least k, removed;  if q is settled then continue
 8        settle q
 9        for each unary operator u with q ∈ pre(u) do
10           agg(u) ← agg(u) + k   (h^add, h^FF)   or   max(agg(u), k)   (h^max)
11           left(u) ← left(u) − 1
12           if left(u) = 0 then relax eff(u) to agg(u) + c(u) by u
13     if h(g) = ∞ for a goal fact g then return dead_end
14     if h^max then return max over g ∈ G of h(g)
15     if h^add then return Σ over g ∈ G of h(g)
16     trace back from G through best(·) and the preconditions, each operator once
17     return the sum of the costs of the operators traced
```

where "relax q to k by u" sets $h(q) \leftarrow k$ and $\mathrm{best}(q) \leftarrow u$ and pushes (k, q) if $k < h(q)$. Line 6 stops as soon as all the goal facts are settled, which is correct since the values of the settled facts do not change afterwards. All the vectors of the computation are `thread_local`, and several workers may evaluate the heuristic at once.

Theorem 16.9 ($h^{\max}$ is admissible and consistent). Algorithm 16.2 computes the values h_s of Definition 16.8, and for each state s and each operator o applicable in s,

$$h^{\max}(s) \le h^*(s), \qquad h^{\max}(s) \le c(o) + h^{\max}(o(s)),$$

with $h^{\max}(s) = 0$ if s satisfies the goal.

Proof. The aggregation (maximum or sum of nonnegative values) is monotone and not smaller than each of its arguments, and the costs are nonnegative, so that the generalized Dijkstra algorithm settles the facts in the order of their values and computes the least fixpoint of the equations of Definition 16.8, as Dijkstra's algorithm does for a sum of one argument [135]. For the admissibility, let $\pi = o_1, \dots, o_k$ be a relaxed plan from s, C_i the cost of its prefix $o_1, \dots, o_i$, and $C_0 = 0$. We show by induction on i that each fact q added by o_i has $h_s(q) \le C_i$, and that each fact of s has $h_s(q) = 0 = C_0$: the preconditions of o_i are facts of s or added by some o_j with $j < i$, whence $h_s(r) \le C_{i-1}$ for each of them, and $h_s(q) \le c(o_i) + \max_r h_s(r) \le c(o_i) + C_{i-1} = C_i$. A plan is a relaxed plan, and its goal facts are facts of s or added by some operator of it, whence $h^{\max}(s) \le C_k$, the cost of the plan. For the consistency, let $t = o(s)$, and let us show $h_s(q) \le c(o) + h_t(q)$ for each fact q, by induction on the order in which the algorithm settles the facts in t. If $q \in F(t)$, either $q \in F(s)$ and $h_s(q) = 0$, or $q \in \mathrm{add}(o)$ and $h_s(q) \le c(o) + \max_{r \in \mathrm{pre}(o)} h_s(r) = c(o)$, since the preconditions of o hold in s. Otherwise $h_t(q) = c(o') + \max_r h_t(r)$ for an operator o' whose preconditions r are settled before q, and

$$h_s(q) \le c(o') + \max_r h_s(r) \le c(o') + \max_r (c(o) + h_t(r)) = c(o) + h_t(q).$$

Taking the maximum over the goal facts gives the consistency, and the value at a goal state is the maximum of zeros. □

Proposition 16.10 (The inadmissible heuristics). h^{add}, h^{FF} and the count of the unsatisfied goal facts are not admissible.

Proof. Let the goal be $\{g_1, g_2\}$ and the state s hold neither. If a single operator of cost 1 adds both, the optimal cost is 1 while $h^{\mathrm{add}}(s) = 2$ and the goal count is 2. For h^{FF}, let the operators be o_1, adding both of cost 3, and

o_2 and o_3, adding g_1 and g_2 respectively, of cost 2 each, all without preconditions: the best supporters of g_1 and g_2 are o_2 and o_3, and $h^{\mathrm{FF}}(s) = 4 > 3 = h^*(s)$. □

It is worth noting that h^{FF} is the cost of a relaxed plan, and hence at least $h^+(s)$, but the plan traced is not an optimal relaxed one. The three inadmissible heuristics serve the bounded-suboptimal searches of Chapter 6 and the greedy searches of §2.12; on the tasks of the data, the case `planning` checks that A* with each of them returns a valid plan, not cheaper than the optimum.

16.5 The landmark-cut heuristic

A *disjunctive action landmark* of a state s is a set L of operators such that every plan from s contains an operator of L. The landmark-cut heuristic (LM-cut) of Helmert and Domshlak [136] finds landmarks one after the other in the delete relaxation, and lowers the costs in such a way that their contributions add up admissibly. In the library it is the class `LMCut`:

```
/// constructor, on the STRIPS view of a task, which must outlive it

explicit LMCut( const StripsTask & task );

/// the value on the state of the values s, one per variable

[[nodiscard]] int operator()( const std::vector< int > & s ) const;
```

Its constructor adds to the STRIPS view a fact $\top$ that always holds, given as a precondition to the operators that have none, and a fact $G^\star$ added by a *goal operator* of cost 0 whose preconditions are the goal facts; thus the goal is the single fact $G^\star$ and each operator has a precondition. Each evaluation (`src/SASHeuristics.cpp:201`) runs the following rounds, which the header describes and the procedure transcribes, on a copy c' of the costs.

Algorithm 16.3 (LM-cut).

```
 1  procedure LMCUT(s)
 2     c'(o) ← c(o) for each operator o;  h ← 0
 3     loop
 4        compute h^max(q) for each fact q from F(s) ∪ {TOP}, with the costs c'
 5        if h^max(G*) = ∞ then return dead_end
 6        if h^max(G*) = 0 then return h
 7        pcf(o) ← a precondition of o of largest h^max, for each o reached
 8        N* ← the goal zone: the facts from which G* is reached backwards by the arcs
               pcf(o) → q, q ∈ add(o), of the operators with c'(o) = 0
 9        N0 ← the facts reached from F(s) ∪ {TOP} by the arcs pcf(o) → q
               with q ∉ N*
10        L ← the operators o with pcf(o) ∈ N0 that add a fact of N*  ▷ the cut
11        m ← min over o ∈ L of c'(o);  h ← h + m
12        c'(o) ← c'(o) − m for each o ∈ L
```

In the pseudocode, `TOP` stands for $\top$, `N0` for N^0 and `N*` for N^*. The arcs $\mathrm{pcf}(o) \to q$ form the *justification graph*: each operator is reduced to its most costly precondition (the code takes the first of the largest ones, in the order of the sorted preconditions), and $h^{\max}$ of a fact is then the cost of a shortest path in that graph. Figure 16.2 shows the first round on the task of Figure 16.1.

Lemma 16.11 (The cut is a landmark). In a round with $0 < h^{\max}(G^\star) < \infty$, every relaxed plan from s, and hence every plan, contains an operator of L, and $m > 0$.

Proof. The facts of $F(s) \cup \{\top\}$ have $h^{\max} = 0$; if one of them were in N^*, the arcs of cost 0 from it to $G^\star$, each from the most costly precondition of its operator, would give $h^{\max}(G^\star) = 0$. Hence $F(s) \cup \{\top\} \subseteq N^0$. Let π be a relaxed plan from s (with the goal operator at its end), and let o be its first operator that adds a fact of N^*, which exists since $G^\star \in N^*$. We show that each fact reached by π before o is in N^0: by induction, an operator o' of π before o has its preconditions among the facts reached before it, which are in N^0, in particular $\mathrm{pcf}(o')$; its added facts are not in N^* by the choice of o, and they are reached from $\mathrm{pcf}(o')$ by an arc of the justification graph, whence they are in N^0. Then $\mathrm{pcf}(o) \in N^0$, and o adds a fact of N^*, i.e., $o \in L$. Finally, an operator $o \in L$ with $c'(o) = 0$ would put $\mathrm{pcf}(o)$ in N^* by line 8, whereas $\mathrm{pcf}(o) \in N^0$ and $N^0 \cap N^* = \emptyset$; hence $m > 0$. □

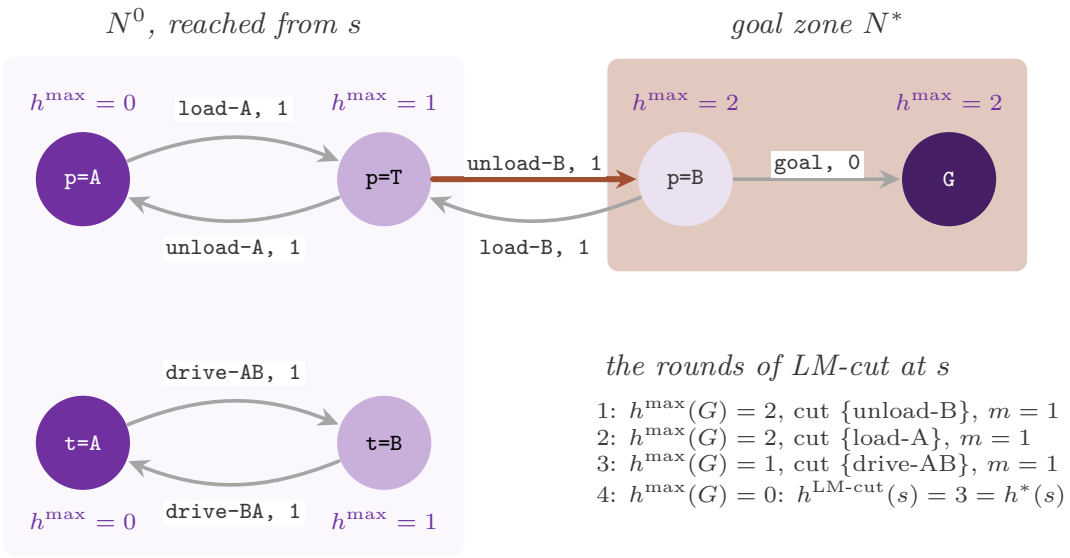


Figure 16.2. The first round of LM-cut on the initial state of the task of Figure 16.1: the facts with their $h^{\max}$, the arcs of the justification graph from the precondition chosen by pcf to each fact added, labelled by the operator and its cost, the goal zone N^* (right) and the facts N^0 reached from the state without entering it (left); the cut is the single operator `unload-B`, in the accent colour. The next rounds cut `load-A` and `drive-AB`, and the heuristic is $3 = h^*$.

The code throws `std::logic_error` if a cut is empty or has $m \le 0$ (`src/SASHeuristics.cpp:307`), which Lemma 16.11 excludes.

Theorem 16.12 (LM-cut is admissible). LM-cut ends after finitely many rounds with integer costs, and its value is at most $h^*(s)$; it returns `dead_end` only if s is a dead end of the relaxation, and hence of the task.

Proof. Each round lowers by $m \ge 1$ the costs of the operators of a nonempty cut and keeps them nonnegative, whence the sum of the costs decreases by at least 1, and there are finitely many rounds. Let $L_1, \dots, L_r$ be the cuts and $m_1, \dots, m_r$ their values, and let π be a plan from s. In the round i, L_i is a landmark by Lemma 16.11, and π contains an operator of L_i. The final costs c' are nonnegative and $c(o) = c'(o) + \sum_{i:o\in L_i} m_i$; hence

$$c(\pi) = \sum_{o\in\pi} c(o) \ge \sum_{o\in\pi} \sum_{i:o\in L_i} m_i \ge \sum_{i=1}^{r} m_i = h,$$

since each L_i meets π (an operator repeated in π is counted as many times, which only adds). Finally, `dead_end` is returned when $h^{\max}(G^\star) = \infty$ in the first round, i.e., when the goal is not reachable in the relaxation, which a plan would contradict. □

Helmert and Domshlak also prove that LM-cut dominates $h^{\max}$ ([136]; we have not reproduced the proof), and the case `planning` checks $h^{\max} \le$ LM-cut along an optimal plan of four tasks. We do not claim that LM-cut is consistent; A* remains optimal with it, since it reopens a closed state reached by a cheaper path (§2.7). Each round costs a computation of $h^{\max}$, i.e., a Dijkstra search over the facts in $O((|F| + U) \log |F|)$ time for $|F|$ facts and U arcs of the operators. The number of the rounds is usually small, but we know no bound on it better than the sum of the costs.

16.6 Abstraction heuristics

An *abstraction* of a task maps its states onto the states of a smaller transition system in a way that preserves the paths; the distances in the abstract system are then an admissible and consistent heuristic (Chapter 5 gives the pattern databases of the permutation puzzles, built on the same principle).

Definition 16.13 (Abstraction). An abstraction is a map α from the states of the task to the states of a transition system $\mathcal{T}$, with transitions labelled by the operators and abstract goal states, such that (i) for each state s and operator o applicable in s, either $\alpha(o(s)) = \alpha(s)$ or $\mathcal{T}$ has a transition $(\alpha(s), o, \alpha(o(s)))$, of cost $c(o)$, and (ii) α maps the goal states to abstract goal states. The heuristic of α is $h^\alpha(s) = d_{\mathcal{T}}(\alpha(s))$, the distance in $\mathcal{T}$

from $\alpha(s)$ to an abstract goal state.

Lemma 16.14 (Abstraction heuristics). The heuristic of an abstraction is admissible and consistent, and it is ∞ only on dead ends.

Proof. By (i), a path from s to a goal of cost C is mapped onto an abstract path from $\alpha(s)$ of cost at most C (the self-loops dropped), which ends at an abstract goal by (ii); whence $h^\alpha(s) \le h^*(s)$, and $h^\alpha(s) = \infty$ implies that s has no path to a goal. For an operator o applicable in s, either $\alpha(o(s)) = \alpha(s)$ and the two values are equal, or the transition of (i) gives $d_\mathcal{T}(\alpha(s)) \le c(o) + d_\mathcal{T}(\alpha(o(s)))$. □

Pattern databases

The *projection* onto a set P of variables (the *pattern*) maps a state onto its values on P, and the abstract system has the partial states on P, the transitions of the operators restricted to P, and the goal restricted to P; it satisfies Definition 16.13. The class `SASPatternDatabase` (src/SASHeuristics.cpp:319) stores the distances of all its abstract states:

```
 /// constructor, building the distances of the projection of task onto
 /// the given variables

 SASPatternDatabase( const SASTask & task , std::vector< int > pattern );

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
 /// a pattern of at most max_size abstract states
 /** The goal variables are added while the size allows, then, in the
  * order of their indices, the variables of the preconditions of the
  * operators that change a variable of the pattern (its parents in the
  * causal graph), as long as one of them fits. */

 [[nodiscard]] static std::vector< int > greedy_pattern(
                              const SASTask & task , std::size_t max_size );
```

The constructor sorts the pattern, numbers the abstract state of the values s by $\sum_{v\in P} s_v m_v$, m_v being the product of the ranges of the variables of P before v (the order of the sorted pattern), and throws `std::invalid_argument` if a variable is out of range or if the abstract space exceeds 2^{40} states. It keeps the operators that change a variable of P, restricted to P, and runs Dijkstra's algorithm backward from all the abstract goal states: an abstract operator is regressed through a state j if j has the values of its effects, the predecessor having the values required before by its preconditions on P (a prevail condition must hold in j as well) and the other values of j, each variable of an effect with no value before taking every value. Abstract states that reach no abstract goal keep the distance `dead_end`. `operator()( s )` looks the distance up in $O(|P|)$, `pattern()` returns the pattern and `size()` the number of the abstract states.

`greedy_pattern( task , max_size )` (src/SASHeuristics.cpp:453) chooses the pattern when the options give none: it adds the goal variables, in the order of the goal, while the product of the ranges stays within `max_size`; then, repeatedly, it adds the first variable, in the order of the indices, that is a parent of a variable of the pattern in the causal graph (a variable of a precondition of an operator that changes a variable of the pattern) and fits, until none fits. Of course, the pattern may be empty if no goal variable fits, and then the database has a single abstract state of distance 0, i.e., it is the blind heuristic.

Proposition 16.15 (The pattern database). The distances of `SASPatternDatabase` are those of the projection, and the heuristic is admissible and consistent; it is built in time $O(N \log N \cdot |O|(|P| + R))$, N being the number of the abstract states and R the largest number of the abstract states that the regression of an operator through a state yields.

Proof. The regression enumerates, for an abstract state j and an abstract operator o whose effects j shows, exactly the abstract states i in which o is applicable and from which it leads to j; hence Dijkstra's algorithm backward from the goals computes the distances of the projection, and Lemma 16.14 applies. Each state settled scans the $|O|$ operators, each in $O(|P|)$ for the test and $O(R)$ for the predecessors, each relaxed in $O(\log N)$ by the binary heap. □

Merge-and-shrink

The class `SASMergeAndShrink` (src/SASHeuristics.cpp:667) builds an abstraction that is not a projection, by the method of Helmert, Haslum, Hoffmann and Nissim [140]:

```
/// constructor, building the abstraction of the task, of at most
/// max_states states

SASMergeAndShrink( const SASTask & task , std::size_t max_states );
```

The *atomic* system of a variable v has the values of v as states, a transition (d, o, q) for each effect (v, p, q) of o and each d with $p = -1$ or $d = p$, a self-loop on d for an operator with the prevail condition $v = d$ and no effect on v, and self-loops everywhere for the operators that do not mention v (which the code marks *irrelevant* rather than storing them); its goal states are the value of v in the goal, or all of them. The *synchronized product* of two systems has the pairs of their states, a transition labelled o from (a, d) to (b, q) if both have a transition labelled o from a to b and from d to q, and the pairs of goal states as goals. The constructor does the following.

1. It takes the variables that are ancestors of the goal in the causal graph (in which u is a parent of v if an operator with an effect on v has a precondition or another effect on u), and orders them for a *linear merge*: the first goal variable, then repeatedly the variable of least index that is a parent of one already taken, or else the next goal variable not taken. All the other variables are left out, which is a further abstraction.
2. For each variable v of the order, it shrinks the current system to at most $\lfloor N/|D_v| \rfloor$ states if it has more (N being `max_states`, clamped to $[1, 2^{24}]$), multiplies it by the atomic system of v, so that the product has at most N states, and prunes from the product the states not reachable from its initial state and those that reach no goal state (the latter are dead ends).
3. The map of a state to its abstract state is kept as one table per step, from the pair (abstract state so far, value of the variable merged) to the abstract state after the step (-1 for a pruned one), each table is renamed when the next shrink merges states; `operator()( s )` follows the tables in $O(k)$ for k variables merged and returns the distance of the final abstract state, or `dead_end` if a table gives -1.

`variables()` returns the variables in the order of the merge and `size()` the number of the final abstract states. The *shrink* is the greedy bisimulation of Nissim, Hoffmann and Helmert [168] (`shrink_map` of src/SASHeuristics.cpp:610): it starts from the partition of the states by their distance h to the goal (the largest values merged into one block if there are more of them than the bound allows), and refines it by the signatures of the states, i.e., by the pairs (label, block of the target) of their transitions (a, o, b) with $h(a) = c(o) + h(b)$, as long as the number of the blocks stays within the bound; the refinement stops at the last partition within the bound, or at a fixpoint, which is then a greedy bisimulation.

Lemma 16.16 (Shrinking by equal distances). Let a partition of the states of a system $\mathcal{T}$, all of which reach a goal, have blocks of states of equal distance h, and let $\mathcal{T}'$ be the quotient system (a transition between two blocks for each transition between their states, and a block is a goal if it holds a goal state). Then the distance of a block in $\mathcal{T}'$ is the common distance of its states in $\mathcal{T}$. If the blocks of the largest distances are merged, the distance of the merged block is at most the least distance of its states.

Proof. The quotient map is an abstraction of $\mathcal{T}$ in the sense of Definition 16.13, and the distance of a block is at most the distance of each of its states (Lemma 16.14). Conversely, a transition (B, o, B') of $\mathcal{T}'$ comes from a transition (a, o, b) with $a \in B$, $b \in B'$, and $h(a) \leq c(o) + h(b)$; since h is constant on the blocks, $h(B) \leq c(o) + h(B')$, and along a path of $\mathcal{T}'$ to a goal block (where $h = 0$, a goal state having distance 0 and the block holding states of equal distance) these inequalities sum to $h(B) \leq$ the cost of the path. For the last statement, the first half of the argument suffices. □

Hence any shrink of the library keeps the distances of the current system exact when the number of the distinct distances is within the bound, and the role of the greedy bisimulation, as Nissim et al. show [168], is to keep the information that the later products need; when it stops before its fixpoint, the abstraction is coarser, and the heuristic remains admissible.

Theorem 16.17 (Merge-and-shrink). On the states reachable from the initial state, the heuristic of `SASMergeAndShrink` is admissible and consistent, and it returns `dead_end` only on dead ends.

Proof. The synchronized product of the atomic systems of a set of variables is the projection onto them (an operator applies to a pair of values exactly when it applies to each, the task having no conditional effects), and each shrink and each pruning is the composition with a map that merges or drops abstract states; the composition of the tables is the map α of the final system. Merging preserves (i) and (ii) of Definition 16.13.

Dropping a state not reachable from the abstract initial state affects no state reachable from the initial state of the task, since by (i) such a state is mapped onto an abstract state reachable from the abstract initial state; dropping a state that reaches no abstract goal maps the states onto it to `dead_end`, which is correct, since a concrete path to a goal would be mapped onto an abstract one. Hence, on the reachable states, α is an abstraction of the task into the final system, and Lemma 16.14 applies. □

Cartesian abstractions refined by counterexamples

The class `SASCartesianAbstraction` (`src/SASHeuristics.cpp:1173`) builds an abstraction by counterexample-guided abstraction refinement (CEGAR), the method of Seipp and Helmert [200], [201]. Instead of fixing a set of variables in advance, as a projection does, it starts from the coarsest abstraction and splits only the abstract states on which an optimal abstract plan fails in the task (the counterexamples).

```
/// the limits of the refinement
struct Limits {
 std::size_t max_states = 1000;           ///< the most abstract states
 std::size_t max_transitions = 1000000;   ///< the most transitions
 double max_time = 0;                     ///< the seconds, 0 for no limit
 };

/// constructor, refining the abstraction of the task within the limits

explicit SASCartesianAbstraction( const SASTask & task ,
                                  const Limits & limits );

/// constructor, refining the abstraction of the subtask of task with the
/// given goal and costs, one per operator, within the limits

SASCartesianAbstraction( const SASTask & task ,
                         std::span< const SASTask::Fact > goal ,
                         std::span< const int > cost ,
                         const Limits & limits );
```

An abstract state is a *Cartesian set* $a = A_0 \times \cdots \times A_{n-1}$, with $\emptyset \neq A_v \subseteq D_v$, and the abstract states partition the states of the task. The system has a transition (a, o, b), $b \neq a$, if o leads from a state of a to a state of b, and a is an abstract goal if it holds a state that satisfies the goal; since a and b are products, both conditions are tested variable by variable. For an operator o and a variable v, let $\mathrm{pre}_v(o)$ be the value that o asks of v (by a prevail condition or by an effect with a value before), if any, and $\mathrm{post}_v(o)$ the value that it gives v, if any.

Lemma 16.18 (Cartesian abstractions). Let a and b be Cartesian sets. Then o leads from a state of a to a state of b if and only if, for each variable v, (i) $\mathrm{pre}_v(o) \in A_v$ when it is defined, and (ii) $\mathrm{post}_v(o) \in B_v$ when it is defined, or else $\mathrm{pre}_v(o) \in B_v$ when $\mathrm{pre}_v(o)$ is defined, or else $A_v \cap B_v \neq \emptyset$. Hence the system of Cartesian sets with these transitions satisfies Definition 16.13, its heuristic is admissible and consistent on every state, and it is `dead_end` only on dead ends.

Proof. The conditions are necessary: a state $s \in a$ in which o applies has $s_v = \mathrm{pre}_v(o)$, and $o(s)_v$ is $\mathrm{post}_v(o)$, or s_v if o does not change v. They are sufficient, since a product lets one choose s_v for each variable apart: $s_v = \mathrm{pre}_v(o)$ where it is defined, a value of $A_v \cap B_v$ for a variable that o neither mentions nor changes, and any value of A_v otherwise; the task having no conditional effects, o applies in s and $o(s) \in b$. Hence (i) of Definition 16.13 holds, with the self-loops dropped, and (ii) holds by the definition of the abstract goals. Hence Lemma 16.14 applies, and it applies to all the states, since no abstract state is pruned. □

Each abstract state remembers its incoming and outgoing transitions and its self-loops, labelled by the operators, and the refinement proceeds as follows.

Algorithm 16.4 (CEGAR).

```
1  procedure CEGAR(task, goal G, costs c)
2     the single abstract state of all the states, every operator a self-loop
3     while no limit is reached
4        a₀ ← the abstract state of s₀;  π ← an optimal abstract plan from a₀
5        if there is no such plan then leave the loop   ▷ s₀ is a dead end
6        s ← s₀;  a ← a₀
```

```
 7       for each step (o, b) of π
 8          if o does not apply in s then
 9             split a by {p} on a v with pre_v(o) = p ≠ s_v;  continue at 3
10          if o(s) ∉ b then
11             split a by A_v ∩ B_v on a v with o(s)_v ∉ B_v;  continue at 3
12          s ← o(s);  a ← b
13       if s misses a fact v = g of G then
14          split a by {g} on v;  continue at 3
15       leave the loop                                  ▷ π is a plan of the task
16    the goal distances h, by Dijkstra's algorithm backward from the goals
```

In the pseudocode, "split a by W on v" replaces a by the two Cartesian sets whose sets of values of v are $A_v \setminus W$ and $A_v \cap W$, and whose other components are those of a. Each split is proper, i.e., both parts are nonempty: at line 9, $p \in A_v$ (the transition of π out of a exists, whence (i) of Lemma 16.18) and $s_v \in A_v \setminus \{p\}$; at line 11, o does not change v (otherwise $o(s)_v = \mathrm{post}_v(o) \in B_v$ by (ii)), and hence $s_v = o(s)_v \notin B_v$, while $A_v \cap B_v \neq \emptyset$ by (ii); at line 14, $g \in A_v$ (a is an abstract goal) and $s_v \neq g$. Since the state s of the trace stays in the abstract state a of the plan (the test of line 10), the split separates s from the part of a whose states admit the step of the plan that failed. When several variables offer a flaw of the same kind, the code splits on the one whose A_v is the smallest fraction of D_v, the least index on a tie (the most refined one, one of the strategies compared in [201]).

The engineering is the following. A split rewires only the transitions of the abstract state split: each incoming transition, outgoing transition and self-loop of a is tested, by the condition of Lemma 16.18 on the variable v alone (the other components did not change), against the two parts; since a transition out of (or into) a part is also one of a, the transitions after the split are those of the new system (cf. Lemma 16.18). The optimal abstract plan of line 4 is found by A* with the goal distances of an earlier abstraction as heuristic, the new part of a split taking the distance of the state split. Indeed, a split may only remove transitions and abstract goals, and therefore these distances remain admissible and consistent; they are computed again whenever the number of the abstract states has grown by a quarter. In turn, the map of a state to its abstract state is the *refinement hierarchy*, a binary tree whose leaves are the abstract states and whose inner nodes are the splits, each testing whether the value of its variable is among those of W; `operator()( s )` descends it in $O(k)$ steps for a leaf of depth k and returns the distance of the leaf. Finally, the `Limits` bound the abstract states (1000 by default), the transitions (10^6) and the seconds of the refinement (none by default), and are tested before each split (hence a limit on the transitions may be exceeded by the transitions of the last split). `size()` and `transitions()` return the numbers of the abstract states and of the transitions, and `exact_init()` is true if the refinement ended at line 15.

Note that the refinement terminates, since each split adds an abstract state and there are at most $\prod_v |D_v|$ of them. If it ends at line 15, the operators of π form a plan of the task of cost $h(a_0)$, and since $h(a_0) \leq h^*(s_0)$ by Lemma 16.18, the value of the initial state is its optimal cost; if it ends at line 5, the initial state is a dead end. Otherwise the abstraction is the one reached within the limits, and its value on the initial state may be below the optimal cost.

Additive Cartesian abstractions

A single abstraction of the task typically spends its abstract states on one goal fact after another; Seipp and Helmert [202] build instead one abstraction for each goal fact and add their values under *saturated cost partitioning*. The class `SASAdditiveCartesian` (`src/SASHeuristics.cpp:1258`) does so:

```
/// constructor, refining the abstractions of the goal facts of task
/// within the limits, max_states and max_time being on their total

SASAdditiveCartesian( const SASTask & task ,
                      const SASCartesianAbstraction::Limits & limits );
```

The second constructor of `SASCartesianAbstraction` refines the abstraction of a subtask, i.e., of the task with another goal and other costs, and `saturated_costs()` returns, for each operator o,

$$\sigma(o) = \max\left\{0, \max_{(a,o,b)} \big(h(a) - h(b)\big)\right\},$$

the inner maximum taken over the transitions labelled o with $h(b)$ finite (the maximum of the empty set being $-\infty$). For the goal facts $g_1, \ldots, g_k$, in the order of the goal, `SASAdditiveCartesian` refines the abstraction h_i of

the subtask whose goal is $\{g_i\}$ with the costs c_i, where $c_1 = c$ and $c_{i+1} = c_i - \sigma_i$, and its value is $h(s) = \sum_i h_i(s)$, or `dead_end` if one of the terms is. The limits on the abstract states and on the time are on the total: each abstraction gets those left, divided by the number of the abstractions left (at least one state), and the construction stops after an abstraction in which the initial state is a dead end.

Theorem 16.19 (Saturated cost partitioning). For each i, $0 \le \sigma_i \le c_i$, the distances of the abstraction h_i are the same under the costs σ_i as under c_i, and $\sum_i \sigma_i \le c$. Hence the heuristic of `SASAdditiveCartesian` is admissible and consistent on every state, and it is `dead_end` only on dead ends.

Proof. A transition (a, o, b) with $h_i(b)$ finite has $h_i(a) \le c_i(o) + h_i(b)$, since the distances are computed under c_i; hence $\sigma_i(o) \le c_i(o)$, and $c_{i+1} \ge 0$. Under σ_i no distance is smaller, since $\sigma_i \le c_i$, and none is larger: each transition (a, o, b) with $h_i(b)$ finite has $h_i(a) \le \sigma_i(o) + h_i(b)$ by the definition of σ_i, and summing these inequalities along a path to a goal gives $h_i(a) \le$ its cost under σ_i. By induction, $\sum_{j \le i} \sigma_j = c - c_{i+1} \le c$. Now let s have a path to a goal of cost C, whose operators o cost $c(o) \ge \sum_i \sigma_i(o)$: by Lemma 16.18 under the costs σ_i, $h_i(s)$ is at most the cost of the path under σ_i, and the sum over i is at most C. For consistency, the same argument on an operator o applicable in s gives $h_i(s) \le \sigma_i(o) + h_i(o(s))$ for each i, and the sum is at most $c(o) + h(o(s))$. A term is `dead_end` only on a dead end of its subtask, which is a dead end of the task, since the goal of a subtask is a part of the goal. □

Of course, the result depends on the order of the subtasks, the first ones taking the costs that they saturate; the code keeps the order of the goal, while Seipp and Helmert also study other orders and the subtasks of the landmarks of the task [201], which the library does not implement.

The case `planning` checks the optimal costs of the tasks with the pattern database, with merge-and-shrink and with the Cartesian abstractions, the admissibility of the first two along an optimal plan and their consistency on each successor of the states of the plan. For the Cartesian abstractions, it enumerates the states reachable in five small tasks and checks, against their distances to the goal computed backward, that both heuristics are 0 on the goals, admissible and consistent, that the additive one is not below each of its terms, and that `exact_init()` implies an exact value of the initial state; it also checks the limits on the abstract states and on the time, the dead end of a task whose goal no operator reaches, and, on four tasks, that A* with each of the two heuristics finds the optimal cost and expands no more states than with $h^{\max}$.

16.7 Features and abstractions for the parallel searches

The distributions of HDA* and the nblocks of PBNF need the structure of the task, which `PlanningBase` derives from it as follows.

The features of Zobrist hashing

The features of a state are its facts, i.e., the propositions of the STRIPS view (`num_features()` is their number, and `for_each_feature( s , f )` calls `f` for the fact of each variable), as in the HDA* for planning of Kishimoto, Fukunaga and Botea [2]. A move changes the facts of the variables that its operator sets, and `for_each_feature_change( s , t , f )` (`Planning.h:606`) finds them by the exclusive or of the words of the two states and the fields of the variables of each word that differs; the incremental Zobrist key of §9.5 costs $O(W + e)$ for e variables changed. In the case `planning`, we check that the facts reported are the symmetric difference of the facts of s and t on each successor of the states of an optimal plan.

The abstract features of FAZHDA*

The abstraction of the features for Abstract Zobrist hashing is that of FAZHDA* of Jinnai and Fukunaga ([138], section 3, and [5], section 4.2, as the header reports them). The *fluency* of a variable is the number of the effects that change it (those with $p \ne q$, an effect with $p = -1$ counting as a change); the fraction `ignored` of the variables of largest fluency (the first $\lfloor ignored \cdot n \rfloor$ in the order of decreasing fluency, ties by index) is left out, and all their facts are mapped to the abstract feature 0, whose key is the same in all the states; their moves never change the owner of a state. The values of each other variable are split into two abstract features by GreedyAFG: on the undirected domain transition graph of the variable (an edge $\{d, q\}$ for each effect that may change d into q), the first half S_1 starts from a value of least degree and grows by the value with the most edges into S_1 (the first one on a tie) until it holds $\lceil |D_v|/2 \rceil$ values; the facts of S_1 and of the rest get two abstract features of their own. A move of such a variable then changes the owner only if it crosses from one half to the other, and the halves are chosen to keep most transitions inside one half. With `ignored` = 0 the abstraction is

that of GAZHDA*. `num_abstract_features()` and `abstract_feature( i )` give the abstraction to `AbstractZobristDistribution`; the case `planning` checks that each abstract feature is in range.

The groups and the transitions

The automatic distributions of Chapter 10 read two further structures of the task. The *groups* of the features are the variables (`num_feature_groups()` is n and `feature_group( i )` the variable of the fact i), whose facts are mutually exclusive, as the concept `FeatureGroupDomain` requires. The *transitions* are the arcs of the domain transition graphs: `for_each_feature_transition( f )` (`Planning.h:269`) calls `f( i , j )` for each operator, each effect (v, p, q) and each fact i of v that the effect may delete ($v = p$, or every $v = d$ with $d \neq q$ if $p = -1$) and the fact j ($v = q$) that it adds; each arc is reported once per operator that realizes it, which gives the weights of the cuts of GRAZHDA*. In the case `planning`, we check that both facts of each arc are of the same variable and that each move of the states of an optimal plan is an arc reported. How the distributions use them is the subject of Chapter 10.

The nblocks and the abstract states

The nblocks of Safe PBNF, which are also the abstract states of AHDA* (`abstract_state( s )` returns `nblock( s )`), are the states of the projection onto a set of variables chosen by the greedy abstraction of structured duplicate detection of Zhou and Hansen [139] (described also in section 2.4 of [3]). `PlanningBase::build_nblocks` (`src/Planning.cpp:511`) starts from the empty set of variables and adds one at each step. The candidates are the variables v not yet taken with $|D_v| \geq 2$ such that the projection onto the set with v has at most `max_nblocks` states and, if `nblock_facts` is positive, the variables of the set hold at most that fraction of the facts (the dynamic threshold of DAHDA*, section 5 of [138] as the header reports it). For each candidate the code builds the directed abstract graph of the projection (an edge from each abstract state matching the preconditions of an operator that changes a variable of the set to the abstract state after its effects), and it takes the variable whose graph has the least largest out-degree, the larger graph on a tie; it stops when no variable fits. The graph of the final set is then made undirected, its edges taken in both directions, and stored in compressed rows: `num_nblocks()` is the number of its nodes, `nblock_vars()` the variables of the projection, `nblock( s )` the index of the projection of s in mixed radix, and `for_each_nblock_successor( b , f )` calls `f` for the neighbours of b. This abstraction is built at its first use, under a `std::once_flag`: a search that needs no nblocks does not pay for it, and several workers may require for it at once.

Proposition 16.20 (The nblocks are an abstraction). The projection chosen by `build_nblocks`, with its undirected graph, satisfies the conditions (i) and (ii) of Definition 11.1. Building it takes $O(n \cdot |O| \cdot N \cdot (k+e) \cdot \log_2 N)$ time, N being `max_nblocks`, k the number of the variables of the projection and e the largest number of effects of an operator.

Proof. (i) Let o be applicable in s. If o changes no variable of the projection, s and $o(s)$ have the same nblock. Otherwise the abstract state of s matches the preconditions of o on the variables of the projection, so that the enumeration of the abstract states matching them (the odometer of `abstract_graph`) reaches it and adds the edge to the abstract state after the effects, which is that of $o(s)$, if the two differ. (ii) The graph is made symmetric, and no node is its own neighbour, since an edge is added only between two different abstract states. For the time, each step multiplies the size of the projection by at least 2, whence there are at most $\log_2 N$ steps; each step tries at most n variables, and each graph enumerates, for each of the $|O|$ operators, at most N abstract states in $O(k + e)$ each. □

The case `planning` checks, along an optimal plan of four tasks, that the nblock of each successor is that of its parent or one of its neighbours, and that the graph is undirected; the parallel cases build the nblocks with at most 512 of them, which keeps the build of the coverage within its time.

16.8 The goal test at the generation

On planning tasks, whose costs are often all 1 and whose heuristics are often weak, the last layer of $f = C^*$ may be large, and the planners test the goal when a node is generated. The sequential A* does so with the option `SearchOptions::goal_at_generation` of `Solution.h`, which §2.11 describes with its proof of optimality (Theorem 2.21): A* keeps the incumbent, the goal of least g among those generated with $h = 0$, and returns it as soon as no entry of OPEN has a smaller f, which saves the nodes of $f = C^*$ that come before a goal in the order of the ties, and all of the last layer when the parent of an optimal goal has $f < C^*$. A successor is tested only if its h is

0, and an informative heuristic pays little for the option; the goals that improve the incumbent are counted in `SearchStats::goals_generated`.

We check the option, in the case `planning`, on the tasks of the data. With the blind heuristic, A* with the option expands exactly the states closer to the initial state than the optimum (counted by a breadth-first search, the costs being 1) on four tasks, and without it at least as many. With LM-cut, the option saves one expansion on each of four tasks: 284 and 283 nodes on `blocks-6-2`, 77 and 76 on `logistics-4-0`, 23 and 22 on `driverlog-03`, 83 and 82 on `gripper-01` (with the default data structures, whose order of the ties these counts depend on). `AStarBench` sets the option by `--gengoal 0 | 1`, by default 1 for `--domain planning` and 0 otherwise; the parallel searches do not honour it.

16.9 The searches on planning tasks

Since `Planning` models the concepts of section 16.3, every search of the library whose concepts it models runs on a task: A* and its variants, IDA*, the search with delayed duplicate detection on the disk (§4.6), the bounded-suboptimal and the greedy searches, and the parallel ones, i.e., SPA*, MQA* and the farm (Chapter 8), HDA* with the distributions Z, AZ, A, H and R, with P for `Planning< 1 >`, and with the automatic distributions of Chapter 10, Safe PBNF and the parallel structured duplicate detection (Chapter 11, Chapter 12), and HDA* over MPI (§9.6). Searches that need a concept that `Planning` does not model refuse it at compile time or, in `AStarBench`, with an exception: e.g., the bidirectional searches (no predecessors), the edge-based ones of PA*SE (no pairwise heuristic), EPEA* (no selection of the operators by Δf), SIW (no goal features) and the hyperplanes (no levels). The case `planning` runs A* on the 28 tasks with each admissible heuristic (but the pairs that take more than a few thousand expansions in the unoptimized build of the coverage) and with the inadmissible ones on eight tasks, IDA* on three tasks and the search on the disk on four. The case `planning-thread` runs HDA* with Z, AZ, A, H and P, the farm, Safe PBNF, SPA* and MQA* with 1, 2, 3 and 8 workers on four tasks, with merge-and-shrink and at most 512 nblocks; `planning-openmp` and `planning-fastflow` do the same on the other executors, the latter without Safe PBNF, SPA* and MQA*. Finally, the case `mpi` runs HDA* over MPI with Z and AZ on `blocks-6-2`.

`AStarBench --domain planning --file <output.sas>` reads a task by `SASTask::read_file` and builds a `Planning< 2 >` if `words_for` gives at most two words and a `Planning< 16 >` otherwise (two sizes only, since each one instantiates every search, and the build of the coverage runs out of memory with three; a task of more than 16 words is refused). Its options for the domain are `--heur blind | goalcount | hmax | hadd | hff | pdb | lmcut | mas | cegar | cegaradd` (`lmcut` by default), `--pdbsize` (the most abstract states of the greedy pattern, 65536), `--massize` (those of merge-and-shrink, 1000), `--cegarsize` and `--cegartime` (those of the Cartesian abstractions, 1000, and the seconds of their refinement, 0 for no limit), `--ignored` (the fraction of the variables left out of the abstraction of AZH, 0.3), `--nblocks` (the most nblocks, 10000), `--nbfacts` (the largest fraction of the facts in the nblocks, 0 for no limit) and `--gengoal`. For instance, `AStarBench --domain planning --file data/planning/logistics-6-0.sas --heur mas --algo hda --dist AZ --threads 8` solves the task with HDA* on 8 workers, the distribution of FAZHDA* and merge-and-shrink.

16.10 The tasks of the data

`data/planning` holds 28 small tasks of the International Planning Competitions, written by `tools/pddl2sas.py` from the PDDL files of the collection of the benchmarks of Fast Downward: the domains gripper of IPC 1998, blocks and logistics00 of IPC 2000, and driverlog of IPC 2002. Its list `optimal.txt` gives the cost of an optimal plan of each (all the costs are 1), marked by `*` when the lower and the upper bound of the public collection planning.domains coincide, and in any case proved by A* with the blind heuristic (as the file states); the IPC tasks carry no license nor copyright notice and are redistributed freely by the public collections of planning benchmarks. Table 16.1 gives their sizes, which we counted on the files.

Table 16.1. The tasks of `data/planning`: the number of the variables, of the facts (the propositions of the STRIPS view), of the operators and of the goal facts, and the optimal costs of `optimal.txt`. All the states fit in one word of 64 bits.

tasks	variables	facts	operators	goal facts	optimal costs
`blocks-4-0` to `-2`	9	34	32	3	6, 10, 6

tasks	variables	facts	operators	goal facts	optimal costs
`blocks-5-0` to `-2`	11	47	50	4	12, 10, 16
`blocks-6-0` to `-2`	13	62	72	5	12, 10, 20
`blocks-7-0` to `-2`	15	79	98	6	20, 22, 20
`driverlog-01`	8	34	88	4	7
`driverlog-02`	9	41	108	7	19
`driverlog-03`	10	46	120	6	12
`gripper-01` to `-04`	7 to 13	24 to 54	34 to 82	4 to 10	11, 17, 23, 29
`logistics-4-0` to `-2`	9	48	78	4	20, 19, 15
`logistics-5-0` to `-2`	9	48	78	5	27, 17, 8
`logistics-6-0` to `-2`	9	48	78	6	25, 14, 25

The tasks are small on purpose: the tests solve them with each heuristic in the unoptimized build of the coverage, and the larger tasks of the same domains are translated by the same script.

16.11 The benchmark

We compare, first, the admissible heuristics under the sequential A* on the 28 tasks, and, second, the parallel searches with merge-and-shrink. Our first table gives, for each task and each of the heuristics `blind`, `hmax`, `lmcut`, `pdb` and `mas`, the expansions and the seconds, with the goal test at the generation (the default of `AStarBench` for planning); the expansions do not depend on the machine, but they depend on the order of the ties of the OPEN list.

Table 16.2. A* with the admissible heuristics on the tasks of `data/planning`: for each task, the optimal cost, and for each heuristic the expansions and the seconds (the median of two runs); the machine and the compiler are given in the note of the table.

task	C*	blind	hmax	lmcut	pdb	mas

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

Table 16.3. The parallel searches on the larger tasks of `data/planning` with merge-and-shrink: HDA* with the distributions Z, AZ (FAZHDA*) and A, and Safe PBNF, on the threads of C++ with 8 workers; for each task and search, the seconds, the speedup over sequential A*, the search overhead SO, the communication overhead CO and the load balance LB of §8.2.

task	search	seconds	speedup	SO	CO	LB

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The tables are written by `manual/build/tables.py` from the CSV of the campaign; until the runs are made on an idle machine, they are placeholders that say so. The clock is the field `seconds` of the `Solution`, which does not include the reading of the task nor the construction of the domain (the pattern database or the merge-and-shrink abstraction, built once by the constructor, and the nblocks, built at their first use within the search, which the clock therefore includes for Safe PBNF and AHDA*).

16.12 Notes and references

The SAS+ formalism is that of Bäckström and Nebel [133], and the format of the files, the successor generator and the causal graph are those of Fast Downward [134]. The heuristics of the delete relaxation are due to Bonet and Geffner [135] ($h^{\max}$ and h^{add}) and to Hoffmann and Nebel [101] (h^{FF}, in the form traced from the best supporters of h^{add} of Keyder and Geffner [173]), the landmark-cut heuristic to Helmert and Domshlak [136], the pattern databases for planning to Edelkamp [137], and merge-and-shrink to Helmert, Haslum, Hoffmann and Nissim [140], with the greedy bisimulation of Nissim, Hoffmann and Helmert [168], and the Cartesian abstractions refined by counterexamples, with their additive form under saturated cost partitioning, to Seipp and Helmert [200], [201], [202]. On the parallel side, HDA* was proposed for planning with the facts as the features of Zobrist hashing by Kishimoto, Fukunaga and Botea [2], and the abstractions of FAZHDA*, GAZHDA* and DAHDA* are those of Jinnai and Fukunaga [138], [141], [5]; the greedy abstraction of the nblocks is that of the structured duplicate detection of Zhou and Hansen [139]. The sections of the papers cited in this chapter are those of the comments of the headers, which we have not checked against the papers.

It is fair to remark that the library is not a planner. Its translator handles STRIPS with types only, without the action costs, the conditional effects, the quantifiers and the derived predicates of PDDL, and its invariant synthesis is a simple search over candidates of up to three predicates; its heuristics are the classical ones, in their basic forms (e.g., the linear merge and a single shrink strategy for merge-and-shrink, one greedy pattern for the pattern database, the subtasks of the goal facts alone, in the order of the goal, for the additive Cartesian abstractions), and its tasks are small. Rather, the domain serves to run the searches of the library, sequential and parallel, on a domain-independent problem whose structure is read from a file, and to compare them with the planners in the literature on the same tasks.

16.13 Exercises

Exercise 16.1 (`ctest -R PBeFS.planning`). Write the task of section 16.1 of one variable of three values with the operators `ab` (0 to 1, cost 5), `bc` (1 to 2, cost 1) and `any-c` (any value to 2, cost 7) in the format `output.sas`, and show that its optimal cost is 6 with each admissible heuristic. Then show why the reader refuses a conditional effect, an axiom, a negative cost, a value out of range and the version 2 of the format.

Exercise 16.2 (`ctest -R PBeFS.planning`). Compute, on the task of Figure 16.1, $h^{\max}$, h^{add}, h^{FF}, the goal count and LM-cut at the initial state, and verify the three rounds of LM-cut given in Figure 16.2. Which of the values equal $h^* = 3$?

Exercise 16.3 (`ctest -R PBeFS.planning`). Prove that, with the blind heuristic, unit costs and the goal test at the generation, A* expands exactly the states whose distance from the initial state is less than C^*, as the case checks by a breadth-first search (use Theorem 2.21); why may the count of the expansions without the option exceed it?

Exercise 16.4 (`ctest -R PBeFS.planning`). Show by Lemma 16.7 that, in the task of Figure 16.1, `unload-A` is an inverse of `load-A` and `drive-BA` of `drive-AB`, and that `unload-B` is not an inverse of `load-A`. Check that the case finds that a state generated by a coded operator skips its parent and nothing else.

Exercise 16.5 (`ctest -R PBeFS.planning`). Prove that the pattern database of the greedy pattern of `blocks-6-2` is consistent (Proposition 16.15) and that merge-and-shrink is consistent on the reachable states (Theorem 16.17); give a state, not reachable from the initial one, on which merge-and-shrink may return `dead_end` although a goal is reachable from it.

Exercise 16.6 (`ctest -R PBeFS.planning-thread`). Show that, under FAZHDA* with `ignored` = 0.3, the moves of the variables left out never change the owner of a state, and that the nblocks of the case (at most 512) satisfy Definition 11.1 (Proposition 16.20). Check that HDA*, the farm, Safe PBNF, SPA* and MQA* find the optimal costs of the four tasks of the case with 1, 2, 3 and 8 workers.

Exercise 16.7 (`ctest -R PBeFS.mpi`). Explain why HDA* over MPI can run on a task whose states are arrays of words with the distributions Z and AZ, which need only the facts and their abstraction, and check that the case finds the optimal cost 20 of `blocks-6-2` with both.

Exercise 16.8 (`ctest -R PBeFS.planning`). Run Algorithm 16.4 by hand on the task of section 16.1 of one variable of three values (the operators `ab`, `bc` and `any-c` of Exercise 16.1), and show that it ends at line 15 with three abstract states and the value 6 of the initial state. Then show, by Lemma 16.18, why the task whose only operator is `ab` gives two abstract states and the value `dead_end` of its initial state, as the case checks.

17. Multi-objective search

[Source: `include/PBeFS/astar/MultiObjective.h` (the concept, the options, the solution, the frontiers, the bucket queue and the indices of the full vectors), `include/PBeFS/astar/MOStructures.h` (the heap and the ordered set of each set of data structures), `include/PBeFS/astar/MOAStar.h` (the sequential searches), `include/PBeFS/astar/ParallelMOAStar.h` (the parallel ones), `include/PBeFS/astar/MultiObjectiveGraph.h` and `include/PBeFS/astar/MultiObjectiveGrid.h` (the domains); `tools/MOABench.cpp`; `test/moa.cpp`]

> **Status — beta.** The code that this chapter describes is in `main` but not yet in a release, and its interface may still change.

We are concerned in this chapter with the searches of a graph whose edges cost a vector of $K \geq 2$ nonnegative components (say, the length, the time and the number of turns of a route), for which no single path is cheapest in general, and the answer is therefore the set of the paths from the start to a goal that no other path improves in every component at once. The searches of the previous chapters order their nodes by one number $f = g + h$; here a search orders paths (not states) by a vector, and it has to decide, for each path it generates, whether a path already known is at least as good in every component (in which case the new one is useless). The algorithms of the literature differ mainly in how fast they answer this question (the *dominance check*), from NAMOA* of Mandow and Pérez de la Cruz [180] to BOA* of Hernández et al. [181], EMOA* of Ren et al. [182] and LTMOA* of Hernández et al. [183], all of which the library implements on one framework. For the parallel searches the library has the search of Ahmadi et al. [184], whose workers run on different orders of the objectives and share their bounds, and a search distributed by the hash of the states, in the manner of HDA* [2] (Chapter 9), which is ours.

The structure of the chapter is as follows. §17.1 gives the problem, the notation and the interface. §17.2 gives the framework of the sequential searches with its proof of correctness, which holds whatever the structure that answers the dominance checks, and §17.3 these structures, i.e., the frontiers of BOA*, LTMOA* and EMOA*, together with the queue of OPEN. §17.4 treats NAMOA* with dimensionality reduction, whose checks are made at the generation of a path. The two parallel searches follow, in §17.5 the searches on several orders of the objectives and in §17.6 the hash-distributed search, with the indices that answer its checks on the full vectors. §17.7 describes the domains of the chapter (the road graphs of DIMACS with several costs, and grids whose moves cost random vectors), and §17.8 to §17.10 give the benchmark, the notes and the exercises. The notation of the single-objective searches is that of Chapter 1, and the conventions of the pseudocode are those of §1.4.

17.1 The problem and its interface

The cost of a path π is the vector $c(\pi) \in \mathbb{R}^K_{\geq 0}$, the sum of the vectors of its edges. We write $u \preceq v$ (u *weakly dominates* v) if $u_i \leq v_i$ for every i, $u \prec v$ (u *dominates* v) if moreover $u \neq v$, and $u <_{lex} v$ for the lexicographic order; clearly $u \preceq v$ implies $u \leq_{lex} v$, and $u \prec v$ implies $u <_{lex} v$, which is the property on which all the searches of the chapter rest. A path to a goal is *Pareto-optimal* if no path to a goal dominates it, and the *Pareto front* $\mathcal{F}$ is the set of the costs of the Pareto-optimal paths. Since several paths may have the same cost, the searches return a *cost-unique Pareto-optimal set*, i.e., one path for each vector of $\mathcal{F}$ [180]. Since the costs are nonnegative, every path to a goal is weakly dominated by a simple one (drop its cycles), and hence by a vector of $\mathcal{F}$, which is finite. The *truncation* $\mathrm{Tr}(v) = (v_2, \ldots, v_K)$ drops the first component of a vector.

A *label* x is a path from s_0 to its state $s(x)$, of cost $g(x)$, with $f(x) = g(x) + h(s(x))$. The heuristic h is a vector too, and it is *consistent* if $h(t) = 0$ at the goals and $h(s) \preceq c(s,t) + h(t)$ for every edge (s,t), i.e., if each component is a consistent heuristic of its own objective (§2.6); it is then admissible in each component, and if Q is a path from $s(x)$ to a goal one has $f(x) \preceq g(x) + c(Q)$. A component equal to infinity marks a state from which no goal is reachable; the searches do not generate such states, which loses nothing. Unlike the searches of

the previous chapters, which keep one node per state, these searches keep many labels per state, one for each cost that is not dominated at that state, and their number usually makes a multi-objective search much "harder" than its single-objective version: already with two objectives the front may have a number of vectors exponential in the size of the graph (although on the road graphs of §17.7 it rarely exceeds a few thousand).

A problem is given to these searches as a type `D` that satisfies the concept `MultiObjectiveDomain` (`MultiObjective.h:123`), i.e., that provides the types `state_type` and `cost_type` (an arithmetic type for the components), the constant `objectives`, the number $K \geq 2$, the methods `initial_state()`, `is_goal( s )` and `hash( s )` of a `Domain` (§1.5), the method `heuristic( s )`, which returns the vector $h(s)$ as a `std::array`, and the method `for_each_successor( s , h_s , f )`, which calls `f( t , c , h_t )` for each successor t of s, c being the vector of the edge and $h_t = h(t)$ (so that a domain that computes h together with the successors computes it once). In the header the concept reads as follows:

```
template< class D >
concept MultiObjectiveDomain =
 requires( const D & d , const typename D::state_type & s ,
           const std::array< typename D::cost_type , D::objectives > & v ) {
  typename D::state_type;
  typename D::cost_type;
  requires std::is_arithmetic_v< typename D::cost_type >;
  requires ( std::size_t( D::objectives ) >= 2 );
  requires std::equality_comparable< typename D::state_type >;
  requires std::copyable< typename D::state_type >;
  { d.initial_state() } -> std::convertible_to< typename D::state_type >;
  { d.is_goal( s ) } -> std::convertible_to< bool >;
  { d.hash( s ) } -> std::convertible_to< std::uint64_t >;
  { d.heuristic( s ) } -> std::convertible_to<
                          std::array< typename D::cost_type ,
                                      D::objectives > >;
  d.for_each_successor( s , v , detail::mo_successor_sink<
                         typename D::state_type , std::array<
                          typename D::cost_type , D::objectives > >{} );
  };
```

The searches return a `ParetoSolution` (`MultiObjective.h:318`), whose member `front` holds one `Path` (the vector `cost` and the `states` along it) for each vector of the front, in lexicographic order, with the costs in the order of the objectives of the domain whatever the order of the search, and whose `status` is `Status::solved` when the front is complete (`Status::unsolvable` if no goal is reachable, the front being then empty) and `Status::node_limit` when the search stopped at its limit; `costs()` gives the vectors without the paths. Its member `stats` is a `MOSearchStats` (`MultiObjective.h:294`), which adds to the counters of §2.10 the labels `extracted` from OPEN (pruned or not), the labels `inserted` into OPEN, the dominance `checks` (the comparisons of two vectors, or of two components for the frontiers that keep one component) and the `solutions` found, which a parallel search may find more than once. The options are a `MOSearchOptions` (`MultiObjective.h:357`):

```
struct MOSearchOptions {

 /// stop with Status::node_limit after storing this many labels (0 = no
 /// limit); in a parallel search the limit holds for each worker
 std::size_t node_limit = 0;

 /// the number of states the search is expected to reach (0 = unknown),
 /// for which the table of the states is sized at the start
 std::size_t expected_states = 0;

 /// no dominance check when a path is generated, only when it is
 /// extracted: the lazy searches of Hernández et al. (IJCAI 2023), which
 /// put into OPEN more paths and check fewer
 bool lazy = false;

 /// the order of the objectives in the lexicographic order of OPEN, a
 /// permutation of \f$ 0 , \ldots , K - 1 \f$, the first one the primary
 /// objective; empty for \f$ 0 , \ldots , K - 1 \f$
```

```
std::vector< unsigned > order;

/// keep the paths of the front, not only their costs
bool paths = true;

};  // end( struct( MOSearchOptions ) )
```

i.e., the limit on the labels stored, the number of states expected (by which the table of the states is sized), the lazy checks of §17.2, the order of the objectives in the lexicographic order of OPEN, and whether the paths are kept besides their costs.

The header also gives the operations on vectors that the searches share, and that a caller may use to check a front: `weakly_dominates( u , v , from )` tests $u_i \leq v_i$ for the components from `from` on (all of them by default), `dominates( u , v )` tests $u \prec v$, `mo_dead_end( v )` tests whether a component is infinite, `pareto_front( v )` returns the vectors of a set that no other one weakly dominates, each once and in lexicographic order (the front by enumeration, on which the tests of the library rest), and `truncate( v )` returns $\mathrm{Tr}(v)$. The order of the objectives is a permutation p, checked by `objective_order( order )`, which returns the identity for an empty order and throws `std::invalid_argument` if the order is not a permutation; `permute( v , p )` gives $(v_{p_0}, \ldots, v_{p_{K-1}})$ and `unpermute( w , p )` its inverse. A search in the order p works on the permuted vectors throughout, and unpermutes the costs of its front at the end; hence, in the rest of the chapter we may assume the identity order without loss of generality. Finally, `MOAStar.h` gives `add_costs( u , v )`, the sum by components, `trace_labels( x )`, the states of the path of a label followed back through its parents, and `finish_front`, which writes the front and the status of a `ParetoSolution` from the labels of the solutions found.

17.2 The framework of the sequential searches

The sequential searches of the library, but NAMOA*dr, are instances of the framework of Ren et al. [182] (its Algorithm 1), which generalizes BOA* [181] to any number of objectives. OPEN holds labels, ordered by the lexicographic order of f; each state s has the *frontier* $G^{Tr}(s)$ of the truncated g of the labels expanded there, and S is the frontier of the truncated f of the solutions found. A frontier answers two questions, i.e., whether one of its vectors weakly dominates a given vector (we then write $v \in \mathrm{dom}\, F$), and the addition of a vector, which removes those that it dominates. The search is the class template `MOAStar< D , F , Q >` (`MOAStar.h:206`), F being the frontier and Q the queue of OPEN, and its pseudocode, which is that of the Doxygen comment of `MOAStar.h`, is the following.

Algorithm 17.1 (The framework of the multi-objective searches).

```
 1  procedure MOA*(s0)
 2     OPEN ← {x0}, with s(x0) = s0, g(x0) = 0;  sols ← ∅
 3     while OPEN ≠ ∅ do
 4        x ← the first label of OPEN by the lexicographic order of f;
             remove it from OPEN
 5        if s(x) is not a goal and Tr(g(x)) ∈ dom G(s(x)) then
             continue                          ▷ the frontier check
 6        if Tr(f(x)) ∈ dom S then continue     ▷ the solution check
 7        if s(x) is a goal then
 8           add Tr(f(x)) to S, removing what it dominates;
             add x to sols;  continue
 9        add Tr(g(x)) to G(s(x)), removing what it dominates
10        for each successor t of s(x) with no infinite h(t) do
11           y ← the label of t with g(y) = g(x) + c(s(x), t), parent x
12           if not lazy and (the check of line 5 or 6 prunes y) then
                continue
13           push y in OPEN
14     return sols
```

The ties of f go to the label pushed first, and a label of a goal is not expanded, since every path through a goal is weakly dominated by its prefix that ends at that goal. Furthermore, with the option `lazy` the checks of line 12 are not made, as in the lazy searches of Hernández et al. [183]: OPEN then receives more labels, and the search makes fewer checks, since a label that is pruned anyway at its extraction is checked once instead of twice; which of the two variants is faster depends on the cost of a check against that of a push and a pop. The truncation at

lines 5 and 6 is the *dimensionality reduction* of Pulido, Mandow and Pérez de la Cruz [185], which the next lemmas justify.

Lemma 17.1 (The order of the extractions). With a consistent heuristic, Algorithm 17.1 extracts the labels by nondecreasing f_1, and by a nondecreasing f in the lexicographic order.

Proof. By consistency $f(y) \succeq f(x)$ for a successor y of x, and hence $f(y) \geq_{lex} f(x)$. A label pushed after the extraction of x descends from labels extracted no earlier than x, and therefore its f is not smaller than $f(x)$, and OPEN gives the smallest of its labels. □

Lemma 17.2 (Dimensionality reduction). If the label y was expanded at s before the label x of s is checked, then $\mathrm{Tr}(g(y)) \preceq \mathrm{Tr}(g(x))$ implies $g(y) \preceq g(x)$; if the solution z was found before x is checked, then $\mathrm{Tr}(z) \preceq \mathrm{Tr}(f(x))$ implies $z \preceq f(x)$.

Proof. If x is checked when it is extracted, then $f_1(y) \leq f_1(x)$ by Lemma 17.1, and $g_1(y) \leq g_1(x)$, since $h(s)$ is the same. If x is checked when it is generated, its parent was extracted after y, and $f_1(y)$ is not larger than the f_1 of the parent, which is not larger than $f_1(x)$. The same argument holds for z, whose f is its cost. □

We call a label x *useful* for $c \in \mathcal{F}$ if a path Q from $s(x)$ to a goal has $g(x) + c(Q) = c$; then $f(x) \preceq c$, and $h(s(x))$ is finite. The root is useful for every c, and we write $|Q|$ for the number of edges of the shortest such Q.

Theorem 17.3 (Correctness of the framework). With a consistent heuristic, Algorithm 17.1 terminates and returns a cost-unique Pareto-optimal set, whatever its frontier and with or without the lazy checks; when it stops at its node limit, every path that it returns is Pareto-optimal.

Proof. Termination: a label expanded at s is not weakly dominated by the labels expanded there before (Lemma 17.2, for its truncation). If its path visited s twice, the prefix that ends at the first visit would have been expanded at s before it, with a g not larger; hence every label expanded is a simple path, of which there are finitely many, and each expansion pushes finitely many labels.

Completeness: let $c \in \mathcal{F}$ never be found, and let x be a label useful for c, among all the labels ever generated, with the least $|Q|$ (the labels are finitely many). If $|Q| = 0$, then $s(x)$ is a goal and $g(x) = c$, and x was either pruned by S, i.e., a solution $z \preceq f(x) = c$ was found (Lemma 17.2), and $z = c$ since c is Pareto-optimal, or became a solution; both contradict the assumption. If $|Q| > 0$, then x was (i) expanded, and its successor on Q, useful with a shorter path, was generated, against the choice of x; (ii) pruned by S, which as above means that c was found; or (iii) pruned by $G^{Tr}(s(x))$, i.e., a label y expanded at $s(x)$ has $g(y) \preceq g(x)$ by Lemma 17.2, so that $g(y) + c(Q) \preceq c$, hence $= c$, y is useful with the same Q, and its successor on Q, useful with a shorter path, was generated when y was expanded, again against the choice of x. OPEN is empty at the end, and no other case is left.

Soundness: let the label z of a goal be extracted and not pruned. If a solution of cost $g(z)$ had been found before, its truncation (or a vector that weakly dominates it) would be in S, and z would be pruned; hence no cost is found twice. Suppose now that $c \prec g(z)$ for some $c \in \mathcal{F}$. A label w useful for c has $f(w) \preceq c \prec g(z) = f(z)$, hence $f(w) <_{lex} f(z)$, and none is in OPEN when z is extracted. The argument of the completeness, applied to the labels generated so far, shows that c has been found, and $\mathrm{Tr}(c) \preceq \mathrm{Tr}(f(z))$ makes S prune z, a contradiction. Hence every solution is Pareto-optimal; the argument does not need the search to have ended, and therefore it holds at the node limit too. □

The proof uses only what the frontiers have in common, i.e., a set of truncated vectors that weakly dominates every vector added to it, and the checks of line 12, when they are made, are those of lines 5 and 6 made earlier; hence the theorem holds for each frontier of §17.3, lazy or not. It does not hold, of course, with an inconsistent heuristic, for which Lemma 17.1 fails, and the library does not check consistency.

A program chooses a search by `mo_search( domain , a , options )` (`MOAStar.h:576`), where `a` is a `MOAlgorithm` (`MOAStar.h:128`): `boa` (BOA*, for two objectives only, `std::invalid_argument` otherwise), `emoa` (EMOA*), `ltmoa` (LTMOA*, and LazyLTMOA* with `lazy`) or `namoa_dr` (NAMOA*dr, §17.4), and `mo_algorithm( name )` gives the `MOAlgorithm` of the names `boa`, `emoa`, `ltmoa` and `namoa-dr`, e.g., of an option of a command line. The following example, which is compiled and run by the case `manual`, finds the front of a graph of four nodes with BOA* and with the hash-distributed search of §17.6 on two workers:

```
// 4 nodes whose arcs cost a distance and a time, searched from 0 to 3
MultiObjectiveGraph< int , 2 > mg( 4 , 0 , 3 );
mg.add_arc( 0 , 1 , { 1 , 3 } );
mg.add_arc( 0 , 2 , { 3 , 1 } );
```

```
mg.add_arc( 1 , 2 , { 1 , 1 } );
mg.add_arc( 1 , 3 , { 1 , 3 } );
mg.add_arc( 2 , 3 , { 1 , 1 } );
mg.set_ideal_heuristic();          // h_i( u ): the distance by i alone
const auto sb = mo_search( mg , MOAlgorithm::boa );
// sb.front: one path for each vector of the Pareto front
ParallelMOOptions mo;
mo.threads = 2;
const auto sd = HDMOAStar< MultiObjectiveGraph< int , 2 > >( mg , mo )
                .solve();
```

Both fronts are $\{(2,6),(3,5),(4,2)\}$, the second vector being the cost of the path through all four nodes, which is longer than the other two in the number of arcs and Pareto-optimal nonetheless.

17.3 The frontiers and the queue of OPEN

The frontiers trade the cost of a check for the cost of their structure, and the three of the library are those of the three algorithms of the framework. With $K = 2$ the truncated vectors are numbers, and the least of them, $g_2^{\min}$, weakly dominates them all: `ScalarFrontier` (`MultiObjective.h:409`) keeps this number alone, and both questions take one comparison, which is the check in constant time of BOA* [181]. For more objectives, `ArrayFrontier` (`MultiObjective.h:448`) keeps the truncated vectors in an array, which a check scans and from which a vector dominated by a new one is removed by moving the last one into its place, as in LTMOA* [183]; the array is small and contiguous, which often (but not always) makes its linear scan faster than a tree, at least for the "small" frontiers of most states. `TreeFrontier` (`MultiObjective.h:510`) keeps them in a balanced tree ordered lexicographically, as in EMOA* [182]: since a dominating vector is lexicographically smaller, a check looks only at the keys $\leq_{lex} \mathrm{Tr}(v)$, and for $K = 3$, when the keys have two components and form a staircase (the second decreasing as the first grows), it answers with the largest key $\leq_{lex} \mathrm{Tr}(v)$ alone, which has the least second component among them, in $O(\log n)$ (TOA* of [182]); the keys that $\mathrm{Tr}(v)$ dominates follow it in a run, which an insertion removes. The aliases `AutoFrontier` (that of BOA* for two objectives, that of LTMOA* otherwise, the default of the searches) and `EMOAFrontier` (that of BOA* for two objectives, the tree otherwise, as in [182]) give the frontier by the number of objectives.

The tree and the heap of OPEN come from `MOStructures.h`, which takes them from the chosen set of data structures (§3.12): `MOHeap` is a priority queue of plain items, and `LexSet` an ordered set of vectors whose operations are those of the checks, i.e., whether a key up to a bound satisfies a predicate (stopping at the first one), the largest key up to a bound, and the removal of the keys from a bound on that satisfy a predicate, all of them or only those of the first run. The default set has the d-ary heap and the AVL tree, whose keys up to a bound are visited from the root down as in the check of [182]; the sets of the standard library and of Boost visit them in order.

OPEN is `MOHeap` by default, and it may be `MOBucketQueue` (`MultiObjective.h:569`) for integer costs, i.e., a bucket for each value of f_1 on a ring that doubles as needed, each bucket a heap in the order of OPEN. A queue of this kind is suggested by Ahmadi et al. [184], whose buckets, however, keep the ties of f_1 in the order of their pushes; that order may expand a label that a later one of the same f_1 dominates at its state, which their search repairs at the solutions. The buckets of the library keep the order of the heap.

Proposition 17.4 (The bucket queue). With integer costs and a consistent heuristic, `MOBucketQueue` returns the labels in the same order as `MOHeap`, and Algorithm 17.1 then makes the same extractions, expansions and checks with either.

Proof. By Lemma 17.1 no label pushed has an f_1 smaller than that of the last label extracted; hence, the least nonempty bucket holds the labels of least f_1, and its heap returns the least of them in the order of OPEN, i.e., the lexicographic order of f with the ties to the first pushed. This is the label that the heap returns; since the rest of the search depends on OPEN only through the order of its extractions, the two runs coincide. □

The case `moa` checks the proposition on random graphs, where the two queues give the same counters, and the traces of the examples of the papers: on the graph of figure 1 of [181] BOA* extracts 10 labels and prunes 2 of its 11 successors when it generates them (the tables 1 and 2 there), and on that of figure 1 of [183] LTMOA* extracts 7 labels and LazyLTMOA* 9 (the tables 1 and 2 there).

17.4 NAMOA* with dimensionality reduction

NAMOA* [180], the earlier of these searches, checks a label mainly when it is generated: besides the frontier $G_{cl}(s)$ of the labels expanded at s, each state keeps the set $G_{op}(s)$ of its labels in OPEN, against which a new label is compared in full, and from which (and from OPEN) the labels that the new one dominates are removed. `NAMOAStarDR< D , F >` (`MOAStar.h:385`) is this search with the dimensionality reduction of NAMOA*dr [185], for any number of objectives.

Algorithm 17.2 (NAMOA*dr).

```
 1  procedure NAMOA*dr(s0)
 2     OPEN ← {x0};  G_op(s0) ← {x0};  sols ← ∅
 3     while OPEN ≠ ∅ do
 4        x ← the first label of OPEN by the lexicographic order of f;
              remove it from OPEN and from G_op(s(x))
 5        if Tr(f(x)) ∈ dom S then continue    ▷ the solution check
 6        if s(x) is a goal then
 7           add Tr(f(x)) to S;  add x to sols;  continue
 8        add Tr(g(x)) to G_cl(s(x))
 9        for each successor t of s(x) with no infinite h(t) do
10           y ← the label of t with g(y) = g(x) + c(s(x), t), parent x
11           if Tr(g(y)) ∈ dom G_cl(t) or g(y) is weakly dominated by
                 a label of G_op(t) or Tr(f(y)) ∈ dom S then continue
12           remove from G_op(t) and from OPEN the labels that y
                 dominates
13           push y in OPEN and in G_op(t)
14     return sols
```

A label extracted is never weakly dominated by a label expanded at its state: a label z expanded there after y was generated was in G_{op} together with y, and the later of the two was checked against the earlier one at line 11. Hence the frontier check of line 5 of Algorithm 17.1 is not needed at the extraction, and only the solution check is made there.

Proposition 17.5 (Correctness of NAMOA*dr). With a consistent heuristic, Algorithm 17.2 terminates and returns a cost-unique Pareto-optimal set.

Proof. The termination and the soundness are those of Theorem 17.3, whose arguments use only the order of the extractions and the checks against G_{cl} and S, both of which are made here at the generation with the same outcome (by Lemma 17.2). For the completeness, let x be, among the useful labels of least $|Q|$, the one generated last. If it was removed from OPEN by a later label, or pruned by a label of G_{op} that is removed later by another one, that label is useful with the same $|Q|$ and was generated later, against the choice of x; if it was pruned by a label of G_{op} that is expanded, or by G_{cl}, the successor of that label on Q is useful with a shorter path. The other cases are those of Theorem 17.3. □

The sets G_{op} are lists, and a label removed from OPEN is marked dead and skipped at its extraction rather than taken out of the heap, which keeps the heap free of deletions at the price of its size.

17.5 Searches on several orders of the objectives

The first parallel search, `MOAStarPar< D , F , Executor , Q >` (`ParallelMOAStar.h:199`), is the parallel MOA* of Ahmadi et al. [184]: each worker j runs Algorithm 17.1 with its own order of the objectives, whose first one is its *primary objective* i_j, and with its own OPEN, frontiers and solutions, so that the workers search the same graph from different "sides" of the front. `objective_orders< K >( n )` gives the first n orders, i.e., the K cyclic ones first, as in the paper, and then the other permutations in lexicographic order, up to $K!$; the workers are as many as the threads. The options are a `ParallelMOOptions` (`ParallelMOAStar.h:135`), i.e., the options of the sequential searches and the parameters of the parallel ones (§1.6), with three fields more:

```
struct ParallelMOOptions : public MOSearchOptions , public ParallelParams {

 /// with false, MOAStarPar shares no bound among its workers (the variant
 /// without upper bounds of Ahmadi et al., ICAPS 2025)
 bool share_bounds = true;
```

```
/// HDMOAStar expands a label only if its \f$ f_1 \f$ (in the order of the
/// search) is at most \f$ L + \mathit{window} \f$, L being the least
/// \f$ f_1 \f$ of the tops of the OPEN lists of the workers; negative for
/// no such limit
double window = 0;

/// HDMOAStar counts the labels that each worker has sent and that their
/// owners have received, and a worker keeps the least f_1 of those still
/// travelling in the f_1 it publishes for the window until all have
/// arrived; with false, it forgets them as soon as its outboxes are pushed
bool acknowledge = true;

};  // end( struct( ParallelMOOptions ) )
```

The workers share two vectors. F_i is the largest f_i that a worker of primary objective i has extracted (with several workers on one primary objective, the largest of theirs), and U_i is an upper bound on the i-th component of the vectors of the front that are still to be found. A worker keeps the costs of its solutions, in its order, in the sets $V_1, \ldots, V_{K-1}$, where V_λ holds vectors of which only the components from the λ-th on (in the order of the worker) are compared, V_1 being the frontier S of Algorithm 17.1.

Algorithm 17.3 (MOA* on several orders of the objectives).

```
 1  procedure MOA*-par(s_0)
 2     F ← 0;  U ← ∞
 3     in parallel, on each worker j with its order (i_j first) do
 4        MOA*(s_0) in the order of j, where
 5           extracting x:  F_{i_j} ← max(F_{i_j}, f_{i_j}(x));
                 if f_{i_j}(x) ≥ U_{i_j} then stop all the workers
 6           the solution check of x (and of a successor, if not lazy):
                 f_i(x) ≥ U_i for some i, or the components from λ on
                 of f(x) weakly dominated by a vector of V_λ, some λ
 7           a solution z:  add z to V_1;  UPDATE
 8     return the union of the solutions of the workers, each cost once
 9  procedure UPDATE                                ▷ on the worker j
10     for λ = 1, ..., K − 2 do
11        i ← the objective of the component λ in the order of j
12        for each v ∈ V_λ with v_i ≤ F_i do
13           move v from V_λ to V_{λ+1}, unless a vector there
                 dominates it, removing those that it dominates
14     if V_{K−1} ≠ ∅ then U_i ← min(U_i, the least component of
           V_{K−1}), i the last objective in the order of j
```

The values of F and U are written with release and read with acquire, so that what a worker did before writing a value happens before what another one does after reading it, and the proofs below reason on this order. We call a vector of V_λ, kept for a solution z, *justified* if for every objective m whose component it no longer compares $z_m \le F_m$ held for a value of F_m read by the worker, i.e., some worker of primary objective m had extracted a label x' with $f_m(x') \ge z_m$ before; the bound $U_m = z_m$ is the vector truncated in every component but m.

Lemma 17.6 (What an extraction certifies). Let a worker of primary objective i extract a label x. Then every $c \in \mathcal{F}$ with $c_i < f_i(x)$ is found by some worker by the end of the search.

Proof. By induction on the order of the events, which is a strict partial order on the finitely many events of a run. Let c be such a vector, not found, and choose the useful label w for c of least $|Q|$ among those that the worker generated before extracting x. Since $f_i(w) \le c_i < f_i(x)$, w is not in OPEN when x is extracted (Lemma 17.1, in the order of the worker), and the cases of Theorem 17.3 apply, with one more: w (or a goal label w of cost c) was pruned by a justified vector of a solution z, i.e., the compared components of z are not larger than those of $f(w) \preceq c$. If $z_m \le c_m$ also for every component m that is no longer compared, then $z \preceq c$, and $c = z$ is found. Otherwise $c_m < z_m \le f_m(x')$ for some m and some label x' extracted by a worker of primary objective m before the pruning, and hence before the extraction of x; by the induction, c is found. □

Theorem 17.7 (Correctness of the search on several orders). With a consistent heuristic, Algorithm 17.3 returns a cost-unique Pareto-optimal set: every vector of $\mathcal{F}$ is found by at least one worker, every solution found is

Pareto-optimal, and the copies found by several workers are merged. With a node limit every path it returns is Pareto-optimal.

Proof. Every worker terminates, as Algorithm 17.1 does. Let the worker j end first. If its OPEN is empty, the argument of Lemma 17.6 applies to all the labels it generated, and every $c \in \mathcal{F}$ is found; if it stopped at a label with $f_{i_j} \geq U_{i_j}$, every label left in its OPEN has $f_{i_j} \geq U_{i_j}$ and is pruned by that bound, which is justified, and the same argument applies. For the soundness, let a worker add the solution z with $c \prec z$ for some $c \in \mathcal{F}$: as in Theorem 17.3, no label useful for c is in its OPEN, and therefore a useful label of the worker was pruned by its frontier of a state (and then a useful label with a shorter path exists) or by its solutions. Every rule of pruning is monotone, i.e., a vector weakly dominated by a pruned one is pruned too at any later time, since the sets V_λ only grow in power (a vector moves to a set that compares fewer components, or is replaced by one that dominates it) and the bounds only decrease; hence $z \succeq c \succeq f(w)$ would have been pruned too. Within a worker no cost is found twice, by its own V_1, and the merge removes the copies of the costs found by several workers. □

Two choices of the implementation are ours: the sets V_λ are updated at every solution, as in [184], and also every `flush_period` extractions when F has grown since, which justifies the truncations earlier, and the check of a label against the solutions starts from V_{K-1}, the smallest and strongest set, which saves comparisons and changes nothing else. With `share_bounds` false no bound is shared, which is the variant without upper bounds of [184]. The frontier `F` may be any of §17.3: with the array (the default for $K \geq 3$) the search is LTMOA*-par, and with two objectives and the frontier of BOA* its two workers are BOA* in the orders $(1, 2)$ and $(2, 1)$, sharing their bounds. A worker beyond the first K adds an order whose primary objective is already taken, and it shares the F_i of that objective.

17.6 Hash-distributed multi-objective search

The second parallel search, `HDMOAStar< D , Executor >` (`ParallelMOAStar.h:573`), divides the states instead of the orders: each state belongs to the worker that a function of its hash assigns it to, as in HDA* (§9.2), the owner keeps the labels of its states in its OPEN, by the lexicographic order of f, together with their frontiers, and the labels generated by a worker travel in batches to the owner of their last state, through the mailbox of §9.3. A solution is kept by the owner of its goal, which sends it to every worker, and each worker prunes with its copy of the solutions known.

Since the workers do not extract the labels in one global order, neither Lemma 17.1 nor Lemma 17.2 holds across the workers: a label expanded may later be dominated by a label that arrives at the same state, and a solution may be dominated by a later one. Hence the frontier of a state keeps the full g of the labels expanded there, a label is pruned if its g is weakly dominated by that frontier or its f by a known solution, and the owner of a goal removes from its solutions those that a new one dominates. This is a *label-correcting* search, in which a label may be expanded and dominated later, while the searches of the previous sections are label-setting (every label expanded is final).

Algorithm 17.4 (The worker of the hash-distributed search).

```
 1  procedure WORKER(i)              ▷ OPEN_i, the frontiers G, sols_i
 2     loop
 3        for each message m received from a worker j do
 4           if m is a solution z then add z to the copy of the
               solutions;  continue
 5           y ← the label of m;  received[j] ← received[j] + 1
 6           if not lazy and (g(y) is weakly dominated by G(s(y))
               or f(y) by a solution) then continue
 7           push y in OPEN_i
 8        publish received[ ];  L_i ← min(f_1 of the top of OPEN_i,
           the least f_1 sent and not yet received);  publish L_i
 9        if OPEN_i = ∅ or f_1(top) > min_k L_k + w then
10           flush the outboxes;  wait for a message or the end;
             continue
11        x ← the top of OPEN_i, removed
12        if g(x) weakly dominated by G(s(x)) or f(x) by a solution
             then continue                    ▷ checked again
13        if s(x) is a goal then
14           remove from sols_i the solutions that x dominates;
             add x to sols_i;  send x to every worker;  continue
```

```
15        add g(x) to G(s(x))
16        for each successor t of s(x) with no infinite h(t) do
17           y ← the label of t, parent x
18           if not lazy and f(y) is weakly dominated by a solution
                then continue
19           if t is owned by i then push y in OPEN_i, as at lines 6-7
20           else put y in the outbox of the owner of t;  sent ← sent + 1
```

A worker never blocks: when it has nothing to do, or its top is outside the window, it flushes its outboxes, pauses briefly and starts the loop again, so that it polls its inbox and publishes its values also while it waits. The window decides the order of the extractions: a worker expands a label only if its f_1 is at most $L + w$ (line 9), where $L = \min_k L_k$ is the least f_1 of the tops of the OPEN lists and of the labels still travelling, as published by the workers, and w is the option `window` (0 by default, negative for none), so that the workers advance together by f_1, close to the order of the sequential search. With `acknowledge` (the default) a worker counts the labels it has sent to each owner, each owner counts the labels it has received from each worker, and the least f_1 of the labels sent stays in the L_i of the sender until all of them have arrived; without it, the sender forgets that f_1 as soon as its outboxes are pushed, and L may then exceed the f_1 of a label still travelling, which the window fails to hold back. The end is detected by the mailbox when no worker has a label and no message travels (§9.3), and the front is then the union of the solutions of the owners, each cost once.

Lemma 17.8 (The window does not stall). Assume that the values published by a worker become visible to the others in finite time. Then, with $w \geq 0$, with or without `acknowledge`, as long as some label is in an OPEN list or travelling, some worker eventually extracts a label or receives one.

Proof. Suppose that from some moment on no worker extracts or receives a label. A label that travels is in an outbox that its sender flushes at its next wait, and its owner receives it at its next poll; hence, after a while, no label travels, and every label is in an OPEN list that no longer changes. Each sender then sees (with `acknowledge`) that all its labels have arrived, or has forgotten them at its flush (without it), and so publishes as L_k the f_1 of its top (∞ if its OPEN is empty). Let L^* be the least f_1 of the tops, and k a worker whose top has $f_1 = L^*$: once the values above are visible, k reads $\min_k L_k = L^*$, its top satisfies $L^* \leq L^* + w$, and it extracts it, a contradiction. □

Theorem 17.9 (Correctness of the hash-distributed search). With a consistent heuristic, without a node limit and under the assumption of Lemma 17.8, Algorithm 17.4 terminates and returns a cost-unique Pareto-optimal set, whatever the order in which its workers extract their labels.

Proof. Termination: a label is expanded at s only if no label expanded there before weakly dominates it, and therefore, as in Theorem 17.3, every label expanded is a simple path; hence the labels and the messages are finitely many, by Lemma 17.8 they are all extracted or received eventually, and the mailbox ends the search when no worker has a label and no message travels. Completeness: let $c \in \mathcal{F}$, and choose the useful label x of least $|Q|$ among all the labels created. If it was expanded, its successor on Q was sent to its owner, which either made a useful label of shorter path, against the choice of x, or pruned it by the frontier of the state, whose label y with $g(y) \preceq$ that of the successor is useful and was expanded (so that its own successor on Q is useful with a path shorter still), or by a solution $z \preceq c$, i.e., $z = c$. The same holds if x itself was pruned, and if $s(x)$ is a goal with $g(x) = c$, then x became a solution unless a solution of cost c was there already. A solution is removed only by one that weakly dominates it, which for c has cost c; hence the owner of some goal ends with a solution of cost c for every $c \in \mathcal{F}$. Soundness: the solutions of each owner are mutually non-dominated, and a solution dominated by a vector of $\mathcal{F}$ found by another owner is removed by the final filter of the union. □

A value of L read late changes the order of the expansions and not the front, by Theorem 17.9. With a node limit the solutions are mutually non-dominated but need not be Pareto-optimal, as in any label-correcting search, and the status `Status::node_limit` says so.

The frontiers of the states and the copies of the solutions keep full vectors, whose checks are dearer than the truncated ones of the sequential search. They rest on one remark.

Lemma 17.10 (Truncation above a threshold). Let a set Z of vectors be kept truncated in a frontier of EMOA*, a truncation being added only if no key dominates it, and let $T = \max_{z \in Z} z_1$. For a vector x with $x_1 \geq T$, a key weakly dominates $\mathrm{Tr}(x)$ if and only if some $z \in Z$ weakly dominates x.

Proof. A key $\mathrm{Tr}(z) \preceq \mathrm{Tr}(x)$ comes from some $z \in Z$ with $z_1 \leq T \leq x_1$, hence $z \preceq x$. Conversely, if $z \preceq x$, then either the truncation of z was added, or a key that dominates it was there, and a key is removed only by one that dominates it, so that some key dominates $\mathrm{Tr}(z) \preceq \mathrm{Tr}(x)$. □

Hence each set keeps its vectors twice. Truncated, as in Lemma 17.10, it answers in one check of the sequential search (one comparison for two and three objectives) the vectors with $x_1 \geq T$, i.e., those of the labels that arrive at a state in the order of f_1, which the window makes the common case. In full, it answers the others: for two objectives, in the tree of EMOA* of the vectors $(0, v)$, whose truncations are the vectors v and form a staircase, in $O(\log n)$; for three, in a `LayerIndex` (`MultiObjective.h:655`), which keeps the vectors by increasing first component in layers of a few of them, with a Fenwick tree [189] over the layers, each node of which keeps the vectors of the layers it covers truncated as in Lemma 17.10. A check of x asks the $O(\log L)$ nodes that cover the layers before that of x, whose vectors have a first component at most x_1 (and therefore each answer is exact by Lemma 17.10), and then compares x with the vectors of its own layer whose first component is not larger; these are the layers of f_1 of the indexed pruning of Xu et al. [187], here on prefixes. For more objectives a node of the Fenwick tree costs too much to keep, and a `BlockIndex` (`MultiObjective.h:787`) keeps the vectors in blocks by increasing first component, with a segment tree of the minima of the blocks, component by component, into whose nodes a check descends only if the minimum weakly dominates x (the block-minimum index of [187]); an insertion compares no vector.

Proposition 17.11 (The indices are exact). For every x, `LayerIndex` and `BlockIndex` report that x is weakly dominated if and only if some vector added to them weakly dominates x.

Proof. Note that neither removes a vector; hence, the question is about all the vectors added. For `LayerIndex`, the vectors of the layers before that of x have a first component at most x_1, and each node of the Fenwick tree answers exactly for its layers by Lemma 17.10, with T the largest first component of the layers it covers; the vectors of the layer of x are compared in full, and the layers after it hold only vectors of first component larger than x_1, which cannot dominate x. For `BlockIndex`, a block or a node of the segment tree whose minimum does not weakly dominate x holds no vector that does, since every vector of it is weakly dominated by the minimum; the search descends into all the others, and compares x with the vectors of the blocks it reaches. □

Keeping the dominated vectors does not change the answers, since a vector weakly dominated by another one added earlier dominates no vector that the other does not dominate as well; it costs memory, though, and `LayerIndex` rebuilds its layers when one of them grows past four times its nominal size, which keeps the scan of a layer short.

17.7 The domains

`MultiObjectiveGraph< Cost , K >` (`MultiObjectiveGraph.h:152`) is a directed graph whose arcs cost a vector, given by `add_arc( u , v , c )`, and searched from a source to a target. Its heuristic is set by `set_heuristic( u , h )` or by `set_ideal_heuristic()`, which computes the *ideal point* of each node, i.e., $h_i(u) = d_i(u, t)$, the cost of a cheapest path from u to the target by the i-th objective alone, by one backward Dijkstra's algorithm per objective (`distances_to`, `MultiObjectiveGraph.h:106`), as in [181]; it is consistent by the triangle inequality of each d_i, and it is infinite at the nodes from which the target is not reachable, which the searches never generate.

`read_dimacs_objectives< Cost , K >( in , extra , source , target , seed )` (`MultiObjectiveGraph.h:310`) builds such a graph from the road graphs of the 9th DIMACS Implementation Challenge [64], one file per objective with the same arcs in the same order (the distances of `USA-road-d` and the times of `USA-road-t`), and appends the objectives `extra`, computed from the graph as the papers do: `ExtraCost::arcs` (1 for each arc, as in [188]), `ExtraCost::degree` (2 for the arcs whose end nodes have on average 4 neighbours or more and 1 otherwise, as in [182]), `ExtraCost::avg_degree` (the average number of neighbours of the two end nodes, as in [184]) and `ExtraCost::random` (an integer drawn uniformly in $[1, 100]$, arc by arc, as in [183]); `extra_cost( name )` gives the `ExtraCost` of the names `arcs`, `degree`, `avg-degree` and `random`. The heuristic is left to 0, and the caller sets the ideal point once the source and the target are known. The script `tools/bench/fetch_dimacs.sh` downloads the graphs, both files of a map with the option `--times`.

`MultiObjectiveGrid< Cost , K >` (`MultiObjectiveGrid.h:80`) is pathfinding on the free cells of a `GridMap` (Chapter 14) with 4 or 8 moves, a diagonal one only if both the cells beside it are free; the i-th cost of the move between two cells is an integer drawn uniformly in $[lo, hi]$ by a hash of the unordered pair of cells, of i and of a seed, so that a move costs as much as its inverse and the grid needs no memory for its costs. With 4 moves and costs in $[1, 10]$ these are the grids of [182], there on the empty grid of 10×10 cells and on maps of the benchmarks of Sturtevant [15]. The heuristic is $h_i(u) = lo \cdot \delta(u, t)$, δ being the Manhattan distance with 4 moves and the Chebyshev distance with 8, which a move lowers by 1 at most while it costs at least lo in each component; hence, h is consistent; `set_ideal_heuristic()` replaces it with the ideal point.

The benchmark `tools/MOABench.cpp` runs any of the searches of the chapter on these domains, a road graph with its queries or a grid with its points, and writes one line of CSV per run with the size of the front, the counters of `MOSearchStats` and, for the parallel searches, the labels sent, the messages and the most labels expanded by one worker.

17.8 Benchmark

The searches are measured by `tools/MOABench.cpp` on the road graph of New York of [64] with the distances, the times and, for three objectives and more, the extra objectives of §17.7, on the queries of the papers. Besides the time, the benchmark reports counts that do not depend on the machine (the labels extracted and expanded, the dominance checks, the most labels expanded by one worker and the messages), and these counts are the measure by which the searches are compared with the codes of the papers, whose fronts and expansions they reproduce when the order of the ties is the same.

Table 17.1. The multi-objective searches on the road graph of New York. A row is a search, with its number of workers; the columns give the number of objectives, the queries solved by every search within the limit, the sum of the sizes of their fronts, the labels expanded, the most expanded by one worker, the dominance checks and the time in seconds, summed over the queries solved by all.

search	workers	objectives	solved	front	expanded	most by a worker	checks	seconds

Placeholder: the results of this table are to be measured on an idle machine and written here by `manual/build/tables.py`; no number is given until then.

The table is a placeholder until it is measured on an idle machine.

17.9 Notes and references

NAMOA* of Mandow and Pérez de la Cruz [180] is the multi-objective A* with consistent heuristics in which every label expanded is final, which is the setting of the whole chapter but §17.6. The dimensionality reduction of Pulido, Mandow and Pérez de la Cruz [185] is the observation behind Lemma 17.2, and BOA* of Hernández et al. [181] its use for two objectives, with the checks in constant time. Ren et al. [182] wrote the framework of Algorithm 17.1 and EMOA*, with the balanced tree and the check in logarithmic time for three objectives, and Hernández et al. [183] the linear frontier of LTMOA* and the lazy checks, with the measure of the checks that the counter `checks` follows. The targeted multiobjective Dijkstra algorithm of Maristany de las Casas et al. [188] is a label-setting search of the same family, from which the objective that counts the arcs is taken.

For the parallel searches, Sanders and Mandow [186] expand at once all the labels whose f is not dominated in OPEN, which keeps the label-setting property at the price of a global synchronization per round; the searches of this chapter have none. Ahmadi et al. [184] proposed the search of §17.5, whose argument is written in full in Lemma 17.6 and Theorem 17.7. The hash-distributed search of §17.6 is close to SIP-MOSP of Xu et al. [187], whose update workers own the frontiers of the vertices assigned to them; here the owner of a state both keeps its frontier and expands its labels, as in HDA* [2], so that no frontier is written by two workers and no worker waits for another. The window of f_1, its acknowledgement and the two indices of the full vectors (the Fenwick tree of Fenwick [189] over the layers, and the minima of the blocks) are the parts of that search that make its checks on full vectors affordable; in our runs on grids and road graphs the search still made several times the checks of EMOA*, while it divided the expansions among the workers rather evenly.

17.10 Exercises

Exercise 17.1 (`ctest -R PBeFS.moa`). Run Algorithm 17.1 by hand, with the frontier of BOA*, on the graph of figure 1 of [181], and check that it extracts 10 labels and prunes 2 successors at their generation; then run it lazily, and explain why the number of extractions grows while that of the checks does not.

Exercise 17.2 (`ctest -R PBeFS.moa`). Show by an example of three nodes that Lemma 17.2 fails with an inconsistent heuristic, and that Algorithm 17.1 may then return a dominated path; the test checks every search

against the front computed by `pareto_front` over all the simple paths of random graphs, with consistent heuristics only.

Exercise 17.3 (`ctest -R PBeFS.moa`). Prove that the search in the order p returns the same front as the search in the identity order, and that its expansions may differ; the test runs every order of the objectives.

Exercise 17.4 (`ctest -R PBeFS.moa`). Prove that for $K = 3$ the keys of a `TreeFrontier` form a staircase, i.e., that the second component decreases as the first grows, and that the largest key $\leq_{lex} \mathrm{Tr}(v)$ therefore answers the check of v alone.

Exercise 17.5 (`ctest -R PBeFS.moa-thread`). In Algorithm 17.3 with two workers on two objectives, show that the worker of primary objective 1 stops at the first label it extracts with $f_1 \geq U_1$, and that by then every vector of the front has been found by one of the two workers (Lemma 17.6).

Exercise 17.6 (`ctest -R PBeFS.moa-thread`). Give a run of Algorithm 17.4 with two workers and no window in which a label is expanded and later dominated at its state by a label that arrives from the other worker. Then show that, if every worker read the current values L_k, with $w = 0$ and `acknowledge` a label expanded could later be dominated only by a label of the same f_1, and explain why values read late weaken this; the test runs the search with and without the acknowledgement and checks the front in both cases.

Exercise 17.7 (`ctest -R PBeFS.moa`). Show that Lemma 17.10 fails for a vector with $x_1 < T$, by an example with two vectors in Z, and explain why `LayerIndex` compares the vectors of the layer of x in full.

Exercise 17.8 (`ctest -R PBeFS.manual`). Change the cost of the arc from 2 to 3 in the example of §17.2 to $(1, 4)$, and find the new front by hand; then check it with the example.

18. Symbolic search

[Source: include/PBeFS/symbolic.h (the header of the whole chapter), include/PBeFS/symbolic/Bdd.h and src/Bdd.cpp (the diagrams and the scope of Sylvan), include/PBeFS/symbolic/SymbolicDomain.h (the concepts), include/PBeFS/symbolic/SymbolicModel.h (the compilation of a domain), include/PBeFS/symbolic/SymbolicSearch.h (the searches), include/PBeFS/symbolic/SymbolicPuzzles.h and include/PBeFS/symbolic/SymbolicSAS.h (the domains); test/symbolic.cpp]

> **Status — beta.** The code that this chapter describes is in `main` but not yet in a release, and its interface may still change.

We are concerned in this chapter with the searches that store sets of states instead of states, each set being represented by the reduced ordered Binary Decision Diagram (BDD) of Bryant [163] of its characteristic function, and that expand a set as a whole by a relational product with the transition relation, as symbolic model checking does [190]. The size of a diagram depends on the structure of the set rather than on its number of states, and a set of billions of states may have a diagram of a few thousand nodes. Hence, these searches may solve, within a given memory, problems whose explicit searches would not fit; the converse also happens (and rather often), since the diagrams of some sets grow exponentially whatever the order of the variables. The searches of the chapter are the breadth-first search, Dijkstra's algorithm and BDDA* of Edelkamp and Reffel [191], in the forward, backward and bidirectional directions, with the heuristics given as sums of small tables and the abstractions of the symbolic planners of Torralba et al. [193], [198], the k cheapest plans in the manner of Speck et al. [195], and the computation of the images of a set in parallel. Its diagrams are those of Sylvan [194], an optional dependency of the library.

The structure of the chapter is as follows. §18.1 gives the diagrams and the scope in which Sylvan runs, and §18.2 the concepts by which a problem is given to the searches. §18.3 shows how a domain is compiled into relations and why the images computed are the intended ones, and §18.4 does the same for the heuristics, together with the check of their consistency. §18.5 treats the pattern databases, the perimeters and the abstractions chosen during the search, §18.6 BDDA* with its proof, §18.7 the bidirectional search and §18.8 the k cheapest plans. §18.9 gives the parallel images, §18.10 the domains, and §18.11 to §18.13 the choice of the library with the limits of the implementation, the notes and the exercises. As for notation, that of the explicit searches is the one of Chapter 1 and Chapter 7.

18.1 The diagrams and their runtime

A `Bdd` (`Bdd.h:63`) is a Boolean function of the variables $0, 1, \ldots$ of Sylvan, ordered by their index, held through a handle that the garbage collector of Sylvan knows (each `Bdd` registers the address of its handle when it is built and releases it when it is destroyed). Since the diagrams are reduced and ordered, equal functions have the same diagram, and `==` compares two functions in constant time. Besides the constants `Bdd::one()` and `Bdd::zero()`, there are `Bdd::variable( v )`, the function of the variable v, `Bdd::cube( vars , values )`, the conjunction of the literals of `vars`, and `Bdd::variables( vars )`, the set of variables that the quantifications and the counts take. The operators `&`, `|`, `-` and `!` are the conjunction, the disjunction, the difference and the negation, `equiv( b )` is the equivalence and `exists( vars )` the existential quantification. A function is queried by `empty()`, `full()`, `disjoint( b )`, `nodes()`, `count( vars )` (the number of assignments of `vars` that satisfy it, as a `double`) and `pick( vars )` (one of them, the first in the order of the variables).

The variables of a state are the even ones of Sylvan, $x_k = 2k$, and those of a successor the odd ones, $x'_k = 2k + 1$; a set of states $S(x)$ is then a diagram on the even variables, and a relation $T(x, x')$ a diagram on both. The free functions of the header give the relational products: `image( S , T , vars )` is $\exists x : S(x) \wedge T(x, x')$ renamed to x, and `preimage( S , T , vars )` is $\exists x' : T(x, x') \wedge S(x')$, where `vars` holds the variables that T reads or

writes, those of a state outside `vars` being left unchanged by T. Then `image_into( S , T , vars , backward , out )` writes the image (or the preimage) into `out`, and it is the only operation that several threads may call at once. The images of S under all the relations `T[ i ]` are computed at once, as tasks of Lace, by `images ( S , T , vars , backward , out )`, and `disjoin( v )` is the disjunction of a vector as a balanced tree of tasks (§18.9); also, `nodes( v )` counts the nodes of several diagrams, those shared counted once.

Sylvan and Lace (its scheduler) are one per process. A `BddRuntime` (Bdd.h:255) starts Lace with its workers (0 for one per core) and Sylvan with its table of nodes and its cache within a given number of bytes, and stops both when it is destroyed. Every `Bdd` must be destroyed before the `BddRuntime`, and a second `BddRuntime` alive at the same time throws `std::logic_error`. Its static members give the workers of Lace (`workers()`), whether a runtime is alive (`running()`), the fraction of the largest table of nodes in use (`filled()`), by which a search decides to stop before the table is full, and a garbage collection (`collect()`).

18.2 The problem and its interface

A problem is given to the searches of the chapter as a type `D` that satisfies the concept `SymbolicDomain` (SymbolicDomain.h:202). Its states are vectors of values of the variables $v = 0, \ldots, n-1$, and v ranges over $D_v = \{0, \ldots, |D_v| - 1\}$. The domain provides the integral type `cost_type`, the methods `num_variables()`, `range( v )` (that is, $|D_v|$) and `initial_values()`, the method `for_each_goal( f )`, which calls `f` with one or more partial assignments (a state being a goal if it satisfies at least one of them), and `for_each_operator( f )`, which calls `f` with every operator. A fact $x_v = d$ is a `SymbolicFact` (SymbolicDomain.h:65), and an operator is a `SymbolicOperator` (SymbolicDomain.h:112), with a cost ≥ 0, a precondition `pre` (a vector of facts) and its effects. Each effect is a `SymbolicEffect` (SymbolicDomain.h:75), built by `assign( v , d )` ($x'_v = d$), by `copy_of( v , u )` ($x'_v = x_u$, the value that u had before the move), or by `map_of( v , u , table )` (x'_v is then the entry of `table` at x_u). An effect whose vector `cond` of facts is not empty is a conditional effect, which happens only where its condition holds before the move. An operator applies all its effects at once, each reading the values before the move, and a variable is written by one effect at most.

Three concepts refine `SymbolicDomain`. A `SymbolicHeuristicDomain` (SymbolicDomain.h:242) gives the terms of an additive heuristic towards the goals by `for_each_heuristic_term( f )`, and a `SymbolicBackwardHeuristicDomain` (SymbolicDomain.h:255) the terms of an estimate of the cost from the initial state by `for_each_backward_heuristic_term( f )`. A term is a `SymbolicTerm` (SymbolicDomain.h:138), i.e., a table of values on the values of a few variables (the first variable varying fastest), and a value equal to `infinite_term< Cost >()` marks the states from which no goal is reachable. A `SymbolicAxiomDomain` (SymbolicDomain.h:227) has derived variables, i.e., two-valued functions of the other variables that no operator writes and that are not part of the state, with `is_derived( v )`, `default_value( v )`, `layer( v )` and `for_each_axiom( f )`. Its axioms are `SymbolicAxioms` (SymbolicDomain.h:122), each of which sets its variable to its value where its condition holds, as the axioms of the planning tasks of the translator of Fast Downward [134]. Finally, a `DecodableSymbolicDomain` (SymbolicDomain.h:268) has the type `state_type` and the method `decode( x )`, which gives the state of an explicit `Domain` (§1.5) whose variables have the values `x`, so that a path found symbolically can be checked (or used) on the explicit domain; the tests rest on it.

All the searches take their options in a `SymbolicOptions` (SymbolicSearch.h:93), namely the `direction` (`forward`, `backward` or `bidirectional`), whether the `heuristic` and its `terms` are used, the `unit_costs` of a breadth-first search, and the `patterns`, the `perimeter` and the `additive` combination of §18.5. Then come the `layout` of the variables, the `abstraction_nodes` of the abstractions chosen during the search, the open list as `list` (§18.6) and the `max_relation_nodes` of the partitions; finally, the way the `images` are computed and the `threads` of the executor (§18.9), the fraction `max_filled` of the table of nodes at which a search stops, and whether the `path` is rebuilt. A search returns a `SymbolicSolution` (SymbolicSearch.h:188), whose `status` is `Status::solved`, `Status::unsolvable` or `Status::node_limit`, with the optimal `cost`, a lower `bound` on it, the `path` (the values of the variables of each state, from the initial state to a goal) and its counters, a `SymbolicStats` (SymbolicSearch.h:148). These count the sets expanded in each direction, the relational products of the expansions and of the path, the states and the nodes expanded, and the largest set and image. They also count the partitions and their nodes, the nodes of the sets of the heuristic, the splits of a set by them, and the products and steps of the abstractions; none of them depends on the number of workers (Lemma 18.9), and they are therefore the measure of these searches.

The class `SymbolicSearch< D , Executor >` (SymbolicSearch.h:248) compiles the domain when it is built and runs a search with `solve()`, the k cheapest plans with `top_k( k )` (§18.8) and the states reachable with `reachable( backward )`. The latter returns a `SymbolicReachable` (SymbolicSearch.h:206), i.e., the number

of states first reached after each number of moves, with the products made and the nodes of the largest layer. Also, `set_executor( ex )` gives the search the executor of §18.9, and `model()` the compiled model. The free functions `symbolic_search( d , o )`, `symbolic_bfs( d , o )` and `symbolic_reachable( d , backward , o )` build the search and run it once. A `BddRuntime` must be alive while a search exists, and the header `PBeFS/symbolic.h` includes all those of the chapter.

18.3 The encoding and the relations

A `SymbolicModel` (`SymbolicModel.h:112`) compiles a `SymbolicDomain` into diagrams. A state $x \in D_0 \times \cdots \times D_{n-1}$ is encoded by writing each x_v in binary on $b_v = \lceil \log_2 |D_v| \rceil$ bits, the most significant first, and by concatenating the codes in the order of the variables, which is that of the domain unless a `SymbolicLayout` (`SymbolicModel.h:72`) gives another `order`, or asks with `optimize` for one found by a local search (§18.11). The bit k of the state before a move is the variable $2k$ of Sylvan and the same bit after the move the variable $2k+1$. The two copies of a bit are thus next to each other, which keeps small the relations of the moves that leave a variable as it is. The codes that are not values (when $|D_v|$ is not a power of two) are excluded by the set V of the valid states, those with $x_v < |D_v|$ for all v. Then, the initial state is the minterm of its code, and the goal is $G = V \cap \bigcup_p \{x : x_u = d \text{ for } (u,d) \in p\}$ over the partial assignments p of the domain. In the model one finds the diagrams of these sets (`initial()`, `goal()`, `valid()`), the set of a value of a variable (`value( v , d , next )`), the set of one state (`state( x )`) and the inverse operations (`decode( S )`, the values of the first state of a set, and `count( S )`, its number of states).

With R_o the variables that the operator o reads or writes and $W_o \subseteq R_o$ those it writes, the relation of o is

$$T_o(x,x') = \mathrm{pre}_o(x) \wedge \bigwedge_{e \text{ of } o} E_e(x,x') \wedge \bigwedge_{v \in R_o \setminus W_o} (x'_v = x_v),$$

where E_e is $x'_w = d$ for an assignment and $x'_w = x_u$ for a copy, the latter written bit by bit as $x'_w \leftrightarrow x_u$ (with the high bits of x'_w set to zero if $b_u < b_w$), and a map is the disjunction over d of $x_u = d \wedge x'_w = \mathrm{table}(d)$. The image of S under T_o on the variables R_o is

$$\mathrm{img}(S,T_o) = \{x' : \exists x \in S,\ T_o(x,x'),\ x'_v = x_v \text{ for } v \notin R_o\},$$

which `image` computes when it is given the variables of R_o in both copies, and the preimage $\mathrm{pre}(S,T_o)$ is the same with x and x' exchanged.

Lemma 18.1 (The images). For any set S of valid states, $\mathrm{img}(S,T_o)$ is the set of the states reached from S by applying o, and $\mathrm{pre}(S,T_o)$ is the set of the states from which applying o reaches S.

Proof. A pair (x,x') is in T_o, with $x'_v = x_v$ outside R_o, if and only if x satisfies pre_o, each written variable of x' has the value given by its effect (computed on x, since each E_e reads the unprimed copy), and each other variable of x' equals that of x (inside R_o by the frame of T_o, outside by the relational product). This is the definition of applying o to x, and both sets follow by projection. □

A conditional effect e with the condition C_e has the relation $(C_e \wedge E_e) \vee (\neg C_e \wedge x'_w = x_w)$, the variables of C_e joining R_o; Lemma 18.1 holds with the same proof, since the relation still describes the move fact by fact. The derived variables of a `SymbolicAxiomDomain` take no bits of the state. Instead, the set D_v of the states where the derived variable v does not have its default value is computed once, layer by layer, as the least fixed point of $D_v = \bigvee_a C_a$ over the axioms a of v; a fact of a condition on a derived variable (of a lower layer, or of the same layer with the value that is not the default) is read from its set. A fact on a derived variable is then the set D_v or its complement, and an operator that reads it reads the variables of the state on which D_v depends.

The operators of equal cost are merged, in the order in which the domain gives them, into partitions $T_P = \bigvee_{o \in P} T_o^{R_P}$ on the union R_P of their variables ($T_o^{R_P}$ is T_o with the frames $x'_v = x_v$ of $R_P \setminus R_o$ added), which `partitions()` returns. A partition is closed when the next operator would bring its diagram above `max_relation_nodes` nodes, and with 0 each partition holds one operator. This is the disjunctive partitioning bounded by size of the symbolic planners [193]: the image of a disjunction is the union of the images, while the diagram of the disjunction of all the operators may be much larger (and usually is) than those of its parts together. By Lemma 18.1 applied to each $T_o^{R_P}$, whose added frames repeat what the relational product does anyway, $\mathrm{img}(S,T_P) = \bigcup_{o \in P} \mathrm{img}(S,T_o)$; hence, the successors of cost c of S are the union of its images under the partitions of cost c.

18.4 The heuristic as a partition of the states

A symbolic search does not evaluate a heuristic on a state; instead, it splits a set by the value of the heuristic, i.e., it intersects the set with the sets $H_k = \{x \in V : h(x) = k\}$. The heuristic of a `SymbolicHeuristicDomain` is a sum of terms $h(x) = \sum_{t=1}^{m} h_t(x_{U_t})$, each a table on the values of a few variables U_t (e.g., one cell for the Manhattan distance of the sliding-tile puzzle, and two pancakes of consecutive sizes for the gap heuristic [60]). For it, `SymbolicModel` computes

$$P_0(0) = \mathbf{1}, \qquad P_t(a) = \bigvee_b \big(P_{t-1}(a-b) \wedge B_t(b)\big), \qquad B_t(b) = \{x : h_t(x_{U_t}) = b\},$$

taking $H_k = P_m(k)$, the list of sets (`levels( backward )`) that BDDA* takes as its heuristic (cf. section 3.3 of [193]).

Lemma 18.2 (The sets of the heuristic). $P_m(k)$ is the set of the states whose terms are finite and add up to k; in particular, the sets H_k are disjoint, and their union holds all the states that are not marked as dead ends.

Proof. By induction on t, $P_t(a)$ is the set of the states whose first t terms are finite and add up to a: this holds for $t = 0$, and a state is in $P_t(a)$ if and only if its term t has some finite value b (only the finite values have a set $B_t(b)$) and its first $t-1$ terms add up to $a-b$. □

Usually, the diagrams stay rather small when the terms are on disjoint variables, ordered as the variables are: a level of the diagram of $P_t(a)$ then has at most as many nodes as there are partial sums, i.e., $O(h_{\max})$, and at most $O(h_{\max}|D_v|)$ when consecutive terms share a variable v (as the pairs of the pancakes do). Of course, a heuristic given as a table on many variables, e.g., a large pattern database of an explicit search, would not fit this construction; the pattern databases of the symbolic searches are computed as sets instead (§18.5).

Since h is never evaluated on a state, its consistency cannot be checked on the states expanded, as an explicit search may do; it can, however, be checked on the reachable states at once, by images.

Lemma 18.3 (The check of consistency). Let Q be the states reachable from the initial state. Then `consisten t( false )` returns true if and only if (i) no goal in Q has $h > 0$ and (ii) for each partition T_P of cost c and each k, $\mathrm{img}(H_k \cap Q, T_P) \cap H_j = \emptyset$ for all $j < k - c$. Under (i) and (ii), h is consistent along the moves between states of Q that are not dead ends, and it is admissible on Q.

Proof. The first claim is the code, which finds Q first by a breadth-first search of the sets. By Lemma 18.1 and Lemma 18.2, condition (ii) means that no move of cost c from a state of Q leads from $h = k$ to $h < k - c$, i.e., $h(x) \le c + h(y)$ for any move (x, y) between states with finite h, $x \in Q$ (and hence $y \in Q$). Summing along a cheapest path from $x \in Q$ to a goal y, whose states reach a goal and are therefore not dead ends, gives $h(x) \le d(x, y) + h(y) = d(x, y)$ by (i). □

The check is restricted to Q because the encoding has codes that are no state of the domain (two pancakes in one position, for instance, on which the gap heuristic is not consistent, although no search ever meets them). Also, the backward heuristic is checked by `consistent( true )` in the same way, by preimages, with the initial state in the place of the goals.

18.5 Pattern databases, perimeters and abstractions

We now raise the heuristic of a direction by heuristics computed with searches on sets, in the original space and in projections of it, which are the symbolic Pattern Databases (PDBs) and the perimeters of the symbolic planners [196], [198] (the explicit abstractions of the library are in §16.6). In `SymbolicModel`, `strengthen( bac kward , patterns , perimeter , additive )` replaces the sets H_k of the direction by those of the largest of its heuristic and of these. A pattern is a set P of variables, and the projection of a state keeps x_P. An operator is projected by dropping its facts and its effects outside P (a copy into P from a variable outside it leaves its variable free), and an operator that writes nothing in P is dropped, since its projection would be a loop. By Lemma 18.1 applied to the projected relations, a move $x \to y$ of cost c is projected onto a move $x_P \to y_P$ of cost c, or onto no move when $x_P = y_P$. Given seeds, i.e., sets Z_j of states with costs γ_j, Dijkstra's algorithm on the projected sets computes the sets of

$$h_P(x) = \min_j \big(\gamma_j + d_P(x_P, (Z_j)_P)\big),$$

where d_P is the distance in the projected space, found by preimages for the heuristic towards the goals and by images for that towards the initial state. With the targets as the single seed, at cost 0, this is the PDB of P. With `perimeter` $= m > 0$, Dijkstra's algorithm in the original space first expands m sets from the targets. Let C be the states it closes, each with its distance $d(x)$, and F those it has reached without closing them, each with the least g found; R is the least of these g (or ∞ if F is empty). The perimeter then gives

$$h_C(x) = \begin{cases} d(x) & x \in C \\ R & x \notin C \end{cases}, \qquad h_{C,P}(x) = \begin{cases} d(x) & x \in C \\ \min_{u \in F} \left(g(u) + d_P(x_P, u_P) \right) & x \notin C \end{cases},$$

the latter being the projected search seeded by the sets of F with their g, in the spirit of the perimeter abstractions of Felner and Ofek [199]. The states from which no seed is reachable in the projection are in no H_k, i.e., they are taken as dead ends.

Lemma 18.4 (The abstractions are consistent). The PDB of a pattern is consistent and zero on the targets, and so are h_C and $h_{C,P}$; the largest of consistent heuristics zero on the targets is consistent and zero on the targets, and hence admissible. The dead ends of all of them are true dead ends.

Proof. For the PDB, a move $x \to y$ of cost c is a projected move of cost c or none; the distance to a set satisfies the triangle inequality, and hence $h_P(x) \le c + h_P(y)$, the targets being seeds at cost 0. For the perimeter, Dijkstra's algorithm closes every state at a distance below R, and the states of F have $g \ge R$. Take a move $x \to y$ of cost c (towards the targets). If both are in C, then $d(x) \le c + d(y)$. If $x \notin C$ and $y \in C$, the expansion of y reached x with $g \le c + d(y)$, and $x \in F$; hence $h_C(x) = R \le g(x) \le c + d(y)$, and $h_{C,P}(x) \le g(x) + 0$ by taking $u = x$. If $x \in C$ and $y \notin C$, then $d(x) < R \le h(y)$ for both, since $h_{C,P}(y) \ge \min_F g \ge R$. If neither is in C, h_C is R on both, and $h_{C,P}$ satisfies the triangle inequality of the projected distance. The targets are in C with $d = 0$, and the largest of heuristics with $h(x) \le c + h(y)$ along each move has the same property; admissibility follows by summing along a path, as in the proof of Lemma 18.3. Finally, let a target be reachable from a state $x \notin C$. On a path from x to a target, the last state outside C is in F, since it was reached by the expansion of the next one; the projection of the path reaches the projection of that state, and hence x is not taken as a dead end. □

In general, the PDBs of several patterns are combined by their largest, which needs no condition on the patterns. Their sum, which would be larger, is admissible only when no operator writes into two of them (as for the additive PDBs of the sliding-tile puzzle [23]). With `additive` the PDBs are added, and `strengthen()` checks that the patterns are disjoint and that no operator writes into two of them.

Lemma 18.5 (Additive pattern databases). Under that condition, the sum of the PDBs of the patterns is consistent and zero on the targets, and hence admissible.

Proof. Let $x \to y$ be a move of cost c by an operator o. For each pattern that o does not write, $x_P = y_P$, and the PDB has the same value on x and on y; for the pattern that o writes, if any, the PDB of x is at most c plus that of y (Lemma 18.4). Summing, the sum at x is at most c plus the sum at y, and on a target every term is 0. □

Note that the moves of the sliding-tile puzzle in the encoding of §18.10 write two cells, and those of the pancakes up to N positions; hence, their encodings admit no additive patterns but trivial ones. The additive PDBs of [23] are on the positions of the tiles, which is a different encoding.

Abstractions chosen during the search. With `abstraction_nodes` $= T > 0$, the bidirectional search starts blind, as SymBA* does [198], and when the next step of both directions is estimated above T nodes it raises, once, the heuristic of each direction by the perimeter of the other (through `strengthen_from( backward , closed , frontier , budget )` of `SymbolicModel`). A blind direction is Dijkstra's algorithm on sets, and hence its closed sets hold every state at a distance below the least g of its open sets: these are the C and F of $h_{C,P}$, with the targets of the other direction. The pattern P starts from all the variables and loses, one at a time, the variable whose removal leaves the smallest projection of the frontier. It stops when the projected search from the frontier keeps all its sets within T nodes, and the open sets of each direction are then split again by its new heuristic. This greedy relaxation by size is ours, while SymBA* relaxes the frontier with abstraction strategies of its own. By Lemma 18.4 the new heuristics are consistent and admissible, with one remark: the bidirectional search does not insert the states with $f \ge U$, which may leave out of the frontier a state whose path to a target enters C there. Such a state has no path cheaper than U, while the proofs of Proposition 18.6 and Proposition 18.7 need the consistency only along an optimal path (whose states have $f \le C^* < U$ as long as $U > C^*$). Also, changing the heuristic between two steps keeps these proofs valid, since they use only the current heuristic and the fact that the closed states have their distances.

18.6 BDDA*

A direction keeps an open list of sets $O_{f,g}$ (the states reached with cost g and $f = g + h$), the closed set C of the states expanded, the sets R_g of the states reached with cost g, and the sequence of the sets expanded with their g. At the start the root (the initial state s, or the goals backward) is split into the sets $O_{k,0}$ by the sets H_k of the direction.

Algorithm 18.1 (A step of BDDA*). Take the nonempty set $O_{f,g}$ of least f and, among those, of least g, and let $F = O_{f,g} \setminus C$. While $F \neq \emptyset$: (i) add F to C; (ii) stop with cost g if F meets the target; (iii) compute $Y_c = \mathrm{img}(F, T_c) \setminus C$ for each cost c, T_c being the partitions of cost c; (iv) add $Y_c \cap H_k$ to $O_{g+c+k,g+c}$ for each k, except $Y_0 \cap H_{f-g}$, which becomes the next F.

These are the buckets of g and h of BDDA* [191], close to the SetA* of Jensen, Veloso and Bryant [192]. With $h = 0$ (the option `heuristic` off, or a domain without terms) the search is the symbolic Dijkstra's algorithm of the planners with costs [196]; with all costs set to 1 (`unit_costs`, `symbolic_bfs()`) it is the breadth-first search, whose layers are the sets $O_{g,g}$. The loop of step (iv) on the moves of cost 0 is the breadth-first search within a bucket that such moves require: by consistency they cannot lower h, and the successors with the same h belong to the bucket being expanded.

Proposition 18.6 (Optimality of BDDA*). Let h be consistent, zero on the target and infinite only on states from which no target is reachable. Then a state is expanded at most once, a state x from which a target is reachable is expanded with $g = d(s, x)$, and the search stops with $g = C^*$ if a target is reachable; otherwise, the open list empties.

Proof. A state is expanded at most once, since it enters C when it is expanded and every set taken from the open list is reduced by C. Suppose that a state from which a target is reachable is expanded with g larger than its distance. Let x be the first one in time, expanded at (f, g) with $g > d(s, x)$, and let P be a cheapest path from s to x. When x is expanded, let y be the first state of P not in C (a target is reachable from y through x, and hence $h(y)$ is finite). The predecessor of y on P was expanded before x, and therefore with its distance; hence, y was inserted in O_{f_y,g_y} with $g_y = d(s, y)$ and, by consistency along the rest of P, $f_y = g_y + h(y) \leq d(s, x) + h(x) < f$. Since $y \notin C$, that set has not been expanded yet, which contradicts the choice of the set of least f (the successors of the set being expanded have $f' \geq f$ by consistency). The same argument, applied to a cheapest path to a target t with $d(s, t) < g$ when the search stops at g, gives a set of the open list with $f_y \leq d(s, t) + h(t) = d(s, t) < g \leq f$, again a contradiction; hence the search stops with $g = C^*$. It ends, since C grows at each step and the states are finite. □

With an inconsistent heuristic the search still returns the cost of a path, since the sets record the costs of real paths, but not necessarily an optimal one: a state may be closed with a g larger than its distance. This is why `consistent()` is there, to check a heuristic before it is used. The states marked as dead ends are in no H_k, and they are dropped when they are reached.

The path. Each set F expanded is recorded with its g, in order. If F_i (with cost g_i) holds the state t that meets the target, a predecessor of t is looked for in the sets F_j, $j < i$, from the last one, with $g_j = g_i - c$ for some cost c and $\mathrm{pre}(\{t\}, T_c) \cap F_j \neq \emptyset$. One state of that intersection is taken, and the walk repeats from it until it reaches the root. The walk cannot fail, since each state of F_i entered its bucket as a successor of a state of a set expanded before, with g lower by the cost of the move; it costs one preimage of a single state per partition and per cost tried (the counter `path_images`). Backward, the walk takes images, and it gives the path from the initial state to a goal directly.

List BDDA*. With the option `list`, the open list keeps one set L_g per g, not split by h, as the List BDDA* of Torralba et al. (section 3.3 of [193]). Before a step, the least h of each L_g is found by intersecting it with $H_0, H_1, \ldots$ from a lower bound kept for it, by increasing g, until g reaches the least f found. The set $L_g \cap H_h$ of the least (f, g) is then moved to the open sets $O_{f,g}$ and expanded as above. The set taken is that of the least (f, g) over all the states reached, as in BDDA*; hence the two expand the same sets, and only the number of intersections with the sets H_k (the counter `splits`) differs, since the states never expanded are never split.

18.7 The bidirectional search

Both directions run as in §18.6, the forward one by images from the initial state with h_F, the backward one by preimages from the goals with h_B. Each set X inserted in the open list of a direction with cost g (the roots and the layers of cost 0 included) is intersected with the sets $R_{g'}$ of the other direction by increasing g', and the first nonempty intersection lowers the incumbent U to $g + g'$ if it is smaller. This is the immediate detection of the

solutions of the explicit searches (§7.2), done on sets. A successor with $f \geq U$ is not inserted, since no path through it costs less than U. Let f_D and g_D be the least f and the least g of the nonempty sets in the open list of the direction D, and ϵ the least cost of an operator. The search stops when an open list is empty or when

$$U \leq \text{LB} = \max\{f_F, f_B, g_F + g_B + \epsilon\}$$

(more precisely, the largest value of LB seen so far).

Proposition 18.7 (The stopping rule). Let h_F and h_B be consistent and admissible. When the search stops, $U = C^*$ if a path exists, and $U = \infty$ otherwise.

Proof. U is always the cost of a path, since g and g' are the costs of paths from s to the state and from the state to a goal. Suppose that $U > C^*$ at some time, and let $P = (s = p_0, \ldots, p_k)$ be an optimal path to a goal. As in the proof of Proposition 18.6, the first state p_i of P not closed forward is in the forward open list with $g = d(s, p_i)$ (it is not pruned, since its f is at most $C^* < U$), and the last state p_j not closed backward is in the backward open list with $g = d(p_j, p_k)$. If $i \geq j$, the state p_j has been inserted by both directions with its optimal g, and the second of the two insertions met the set $R_{g'}$ of the other direction that holds it, giving $U \leq C^*$, a contradiction. Hence $i < j$, and $f_F \leq d(s, p_i) + h_F(p_i) \leq C^*$, $f_B \leq C^*$ likewise, and $g_F + g_B + \epsilon \leq C^*$, since the segment from p_i to p_j has at least one move. Thus $\text{LB} \leq C^* < U$ whenever $U > C^*$, and also for the largest LB seen, since U does not increase; the stop on $U \leq \text{LB}$ therefore gives $U = C^*$. If the forward open list empties, all the states reachable from s that are not dead ends have been closed forward, among them the goal p_k of P, which was inserted with $g = C^*$ and met the set R_0 of the backward direction; hence $U = C^*$, and symmetrically for the backward list. With no path, U stays ∞. □

This is the argument of Theorem 7.6, with sets in the place of nodes. Without the two bounds on f, the bound $g_F + g_B + \epsilon$ is the stopping rule of the bidirectional Dijkstra's algorithm, and that of the bidirectional symbolic search of SymBA* [193], whose heuristics are abstractions computed during the search.

The direction of a step. Any interleaving of the two directions satisfies Proposition 18.7, and the choice is therefore free. A step expands the direction whose next set is estimated cheaper to expand, the estimate being the nodes of the set times the ratio between the nodes of the image and those of the set at the last step of that direction. Torralba et al. estimate instead the time of the next step from the time of the previous one and from the sizes of the two sets (section 3.2 of [193]); we use counts of nodes, so that the search and its counters depend neither on the load of the machine nor on the workers. The path is the forward walk from a state of the intersection that set U (with its g forward), reversed, followed by the backward walk from the same state.

18.8 The k cheapest plans

We now turn to `top_k( k )`, which returns the k cheapest plans, i.e., paths from the initial state to a goal, by nondecreasing cost. A path may visit a state more than once, as in the top-k planning of Speck et al. [195], and two plans differ when their sequences of states or their costs do. Let W be the states from which a goal is reachable, computed first by preimages from the goals. The search keeps, for each g, the set S_g of the states of W reached from s by a path of cost g, without removing the states reached with a smaller g. Starting from $S_0 = \{s\} \cap W$, it takes the least g still open, records S_g, enumerates the plans of cost g, and adds $\text{img}(S_g, T_c) \cap W$ to S_{g+c} for each cost c. A plan of cost g ending in $t \in S_g \cap G$ is enumerated backward: the predecessors of a state u with cost g_u are the states of $\text{pre}(\{u\}, T_c) \cap S_{g_u - c}$ for each c, and the walk ends at s with cost 0.

Proposition 18.8 (The k cheapest plans). Let every operator cost more than 0. Then S_g is the set of the states of W reached from s by a path of cost g, all the plans of cost g are enumerated when S_g is taken, each once, and the plans are returned by nondecreasing cost.

Proof. The first claim is by induction on g. A state y reached by a path of cost $g > 0$ whose states are in W (all its states reach a goal through y) has a predecessor x reached with cost $g - c$, $c > 0$, and x is in S_{g-c}, which is taken before S_g and expanded into S_g; conversely, S_g holds only images of such states. When S_g is taken, every $S_{g'}$ with $g' < g$ is complete, since the costs are positive and the sets are taken by increasing g. A plan of cost g has each of its states u, with the cost g_u of its prefix, in S_{g_u}, and the backward walk therefore finds it; two branches of the walk differ in a state or in a cost, and give different plans. □

With moves of cost 0, S_g would not be complete when it is taken, and a cycle of cost 0 would give infinitely many plans of the same cost; `top_k()` therefore refuses them. The search ends, since either k plans are found, or W has no cycle through the reachable states and the sets S_g become empty, or g exceeds its bound `max_cost`.

Bidirectionally. With the bidirectional direction, `top_k()` (through `top_k_bidirectional()`) also keeps the sets S_b^B of the states from which a goal is reached by a path of cost b, and it extends the direction whose next set has fewer nodes. Let the forward sets be complete up to A and the backward ones up to B. Before the plans of cost c are enumerated, the steps go on until $c \leq A$ or $c \leq A + B + 1$. A plan of cost c is then split after its longest prefix of cost at most A: the prefix goes forward through the sets S^F, and the next state (reached by a move that brings the cost above A) has a rest of cost at most $c - A - 1 \leq B$, which goes through the sets S^B. Each plan has exactly one such split, and it is therefore enumerated once, without the checks of duplicates that a meeting of the two directions at any state would need.

18.9 The parallel images

In Sylvan the operations are parallel inside: each recursive call of a relational product, a conjunction or a quantification spawns its two cofactors as tasks of the work-stealing scheduler Lace, so that one product of a large set can use all the workers. On top of that, the option `images` (a `SymbolicImages`) chooses how the images of the partitions of a set are computed. These are (i) `sequential`, one product after the other, (ii) `lace`, all of them in one task of Lace, which spawns the products of the two halves of the partitions recursively and then joins them by a balanced tree of disjunctions (`images()` and `disjoin()`), and (iii) `executor`, the partitions dealt round-robin to the workers of an executor of ADS (the threads of the library, OpenMP or FastFlow, as in Chapter 8). With `lace` the scheduler balances products of rather different sizes, and with `executor` each worker submits its products to Lace by `image_into()`, Lace accepting them from several threads at once.

Lemma 18.9 (Independence of the workers). The costs, the paths and the counters of `SymbolicStats` other than `images_by` do not depend on the choice of `SymbolicImages`, on the workers of the executor or on those of Lace.

Proof. A reduced ordered BDD is canonical for its function and its order of the variables [163], and the order is fixed by the encoding. Each set computed by the search is a function of the sets before it, whatever way its parts were computed; hence so are its nodes, its states, the choice of the direction (which depends on nodes only) and the state taken from an intersection by the path (the first in the order of the variables). Finally, the products are counted one per partition and per set, whoever computes them. □

In fact, the lemma also makes the counters a fair measure of the work of these searches, since the nodes and the products are the same on any machine while the time of a product is not. Two points of the implementation need care. The garbage collection of Sylvan drops the nodes that no registered handle reaches, and a result received by a thread outside Lace is not registered until it is stored in a `Bdd`. With the executor several threads submit products at once, and `image_into()` therefore stores its result in the registered handle of its output inside the task of Lace, before the task ends. Also, Lace hands a task to a thief with memory fences that the tools which check the synchronization of threads do not model. Hence, the tasks of the library take the data of their job, and give it back when they end, through a flag read with acquire and written with release, and ThreadSanitizer sees the order that the fences give.

18.10 The domains

`SymbolicSlidingTile< W , H >` (`SymbolicPuzzles.h:74`) is the sliding-tile puzzle on $W \times H$ cells (§14.2). The variable c is the cell c, its value the tile in it (0 the blank), and a state therefore takes $WH\lceil \log_2 WH \rceil$ bits, 36 for the 8-puzzle and 64 for the 15-puzzle. The move of the blank from the cell b to the next cell c is the operator with the precondition $x_b = 0$ and the effects $x'_b = x_c$ and $x'_c = 0$, which reads and writes two variables only. The heuristic is the Manhattan distance, a term per cell, and the backward heuristic the same distance from the cells of the start. `SymbolicPancake< N >` (`SymbolicPuzzles.h:223`) is the pancake problem (§14.9), with the position of each pancake as its variable (the inverse of the stack). The flip of the top k pancakes then maps each variable on itself, $x'_p = k - 1 - x_p$ for $x_p < k$ (an effect `map_of`), and the gap heuristic is a sum of terms on the pairs of pancakes of consecutive sizes. Both are `DecodableSymbolicDomain`s, whose states decode to those of `SlidingTile` and `PancakePuzzle`.

`SymbolicSAS< Task >` (`SymbolicSAS.h:67`) is a SAS+ task [133] (§16.1) as a `SymbolicDomain`, whose variables are those of the task and whose goal is the partial assignment of the task. It takes the conditional effects of the operators of the task, and, if the task has axioms (members `axioms` and `axiom_layer`), it is a `SymbolicAxiomDomain` whose derived variables are those of the layers of the translator; the task must outlive the domain.

18.11 The library of BDDs and the limits

PBeFS interfaces an existing package of BDDs instead of implementing the diagrams, and we compared the two that fit a C++ library distributed on three systems, Sylvan [194] and CUDD [197]. Sylvan (Apache-2.0) builds with CMake on Linux, macOS and Windows (MSYS2 and MSVC are in its continuous integration), and its operations are parallel on Lace; CUDD (BSD-3-Clause) builds with the autotools only, has no supported build on Windows, and computes each product on one core, one manager per thread. Neither is packaged by vcpkg, Homebrew or MSYS2 (as of November 2026). We therefore use Sylvan, whose parallel operations are the side of symbolic search that concerns a library of parallel search. Sylvan is optional: CMake finds an installed one with `find_package(sylvan)` or else downloads the commit pinned in `CMakeLists.txt` (unless `PBEFS_FETCH_SYLVAN` is off), and the makefiles take it from `SYLVAN_ROOT`; without it, the library `PBeFS::Symbolic` and the cases `sym-*` of the tester are left out.

Of course, the encodings are not necessarily (and perhaps rarely) the best, and their choice matters a great deal. The stack of pancakes as the state, with a variable per position, makes the flip of k pancakes copy the position i into the position $k-1-i$, whose relation reads the positions in order and must keep up to k values at once. With 8 pancakes, the searches in that encoding fill a table of 2^{24} nodes, while the relations of the encoding of §18.10 have a few nodes per variable. Our partitions follow the order of the operators, and the order of the variables is that of the domain, or one given, or one found with `SymbolicLayout::optimize`, which moves one variable at a time to the place that most lowers the sum, over the operators and the terms of the heuristic, of the squared span of their variables in bits. It is a static order, chosen once (Sylvan has no dynamic reordering), and it is good for that sum only. The k cheapest plans are found by a blind search, as in symk [195], and need positive costs. Finally, a search that fills more than `max_filled` of the largest table of nodes after a collection stops with `Status::node_limit`, with the least f of its open list as its bound, while a single product that fills the table ends the process, as Sylvan does.

18.12 Notes and references

The reduced ordered BDDs are those of Bryant [163], and the relational product with a partitioned relation is that of symbolic model checking of Burch et al. [190]. BDDA* is due to Edelkamp and Reffel [191], and Jensen, Veloso and Bryant [192] gave SetA* and the branching on sets of states. Edelkamp and Kissmann [196] extended the symbolic searches to action costs, with Dijkstra's algorithm and BDDA* on buckets and the symbolic PDBs, and the symbolic planners of Torralba et al. [193], [198] gave the bidirectional search with abstractions chosen during the search, List BDDA*, the partitioning of the relation by size and the choice of the direction by the estimated time of the next step, which §18.7 replaces by a count of nodes. The perimeter seeding the PDBs follows Felner and Ofek [199], and the top-k planning Speck et al. [195], whose planner symk is blind as ours is. The library of diagrams is Sylvan of van Dijk and van de Pol [194].

18.13 Exercises

Exercise 18.1 (`ctest -R PBeFS.sym-astar`). Show by an example of two cells that the copy $x'_w = x_u$ of §18.3 needs the high bits of x'_w set to zero when $b_u < b_w$, and that Lemma 18.1 fails without them; the test compares BDDA* with A* on the 8-puzzle and on four instances of the 15-puzzle.

Exercise 18.2 (`ctest -R PBeFS.sym-astar`). Twice the Manhattan distance is not consistent: find a move of the 8-puzzle on which condition (ii) of Lemma 18.3 fails, and check that `consistent()` refuses it.

Exercise 18.3 (`ctest -R PBeFS.sym-pdb`). Give two patterns of a SAS+ task written together by one operator, and a state at which the sum of their PDBs exceeds the optimal cost; then explain why their largest is still admissible (Lemma 18.4).

Exercise 18.4 (`ctest -R PBeFS.sym-variants`). Show that BDDA* and List BDDA* expand the same sets, and give a case in which the second makes fewer intersections with the sets H_k.

Exercise 18.5 (`ctest -R PBeFS.sym-topk`). Prove that the split of a plan in §18.8 after its longest prefix of cost at most A is unique, and show by an example that splitting at any state where the two directions meet would enumerate some plans twice.

Exercise 18.6 (`ctest -R PBeFS.sym-par-thread`). Run the bidirectional search of the 8-puzzle with 1 and 4 workers of Lace, and check that the counters other than `images_by` agree (Lemma 18.9); explain why the choice of the direction would not be reproducible if it were made by times.

Glossary

A compact, alphabetical reference for the terms used in this manual, each with a pointer to the chapter where it is developed.

Bibliography

The references are those of the README of the repository, with the same numbers, so that a citation [N] means the same work in both; each is given with its doi where it has one.

[1] A. Fukunaga, A. Botea, Y. Jinnai and A. Kishimoto, *A Survey of Parallel A**, arXiv:1708.05296, 2017.

[2] A. Kishimoto, A. Fukunaga and A. Botea, *Evaluation of a simple, scalable, parallel best-first search strategy*, Artificial Intelligence 195, 222-248, 2013.

[3] E. Burns, S. Lemons, W. Ruml and R. Zhou, *Best-first heuristic search for multicore machines*, Journal of Artificial Intelligence Research 39, 689-743, 2010.

[4] E. Burns, M. Hatem, M. J. Leighton and W. Ruml, *Implementing fast heuristic search code*, Proceedings of the Symposium on Combinatorial Search (SoCS) 3, 25-32, 2012.

[5] Y. Jinnai and A. Fukunaga, *On hash-based work distribution methods for parallel best-first search*, Journal of Artificial Intelligence Research 60, 491-548, 2017.

[6] M. Phillips, M. Likhachev and S. Koenig, *PA*SE: Parallel A* for slow expansions*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 24, 208-216, 2014.

[7] R. E. Korf, *Depth-first iterative-deepening: an optimal admissible tree search*, Artificial Intelligence 27, 97-109, 1985.

[8] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980; reprinted by Springer, 1982.

[9] P. E. Hart, N. J. Nilsson and B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics 4(2), 100-107, 1968.

[10] S. Edelkamp and S. Schroedl, *Heuristic Search: Theory and Applications*, Morgan Kaufmann, 2012.

[11] F. Mattern, *Algorithms for distributed termination detection*, Distributed Computing 2(3), 161-175, 1987.

[12] M. Aldinucci, M. Danelutto, P. Kilpatrick and M. Torquati, *FastFlow: high-level and efficient streaming on multicore*, in Programming Multi-core and Many-core Computing Systems, Wiley, 261-280, 2017.

[13] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd edition, Prentice Hall, 2010.

[14] M. Held and R. M. Karp, *A dynamic programming approach to sequencing problems*, Journal of the Society for Industrial and Applied Mathematics 10(1), 196-210, 1962.

[15] N. R. Sturtevant, *Benchmarks for grid-based pathfinding*, IEEE Transactions on Computational Intelligence and AI in Games 4(2), 144-148, 2012.

[16] B. Gendron and T. G. Crainic, *Parallel branch-and-bound algorithms: survey and synthesis*, Operations Research 42(6), 1042-1066, 1994.

[17] D. S. Nau, V. Kumar and L. Kanal, *General branch and bound, and its relation to A* and AO**, Artificial Intelligence 23(1), 29-58, 1984.

[18] Y. Shinano, S. Heinz, S. Vigerske and M. Winkler, *FiberSCIP: a shared memory parallelization of SCIP*, INFORMS Journal on Computing 30(1), 11-30, 2018.

[19] M. Fischetti, M. Monaci and D. Salvagnin, *Self-splitting of workload in parallel computation*, Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2014), Lecture Notes in Computer Science 8451,

394-404, 2014.

[20] R. D. Blumofe and C. E. Leiserson, *Scheduling multithreaded computations by work stealing*, Journal of the ACM 46(5), 720-748, 1999.

[21] J. C. Culberson and J. Schaeffer, *Pattern databases*, Computational Intelligence 14(3), 318-334, 1998.

[22] R. E. Korf and A. Felner, *Disjoint pattern database heuristics*, Artificial Intelligence 134(1-2), 9-22, 2002.

[23] A. Felner, R. E. Korf and S. Hanan, *Additive pattern database heuristics*, Journal of Artificial Intelligence Research 22, 279-318, 2004.

[24] A. Felner, R. E. Korf, R. Meshulam and R. C. Holte, *Compressed pattern databases*, Journal of Artificial Intelligence Research 30, 213-247, 2007.

[25] T. Breyer and R. E. Korf, *1.6-bit pattern databases*, Proceedings of the AAAI Conference on Artificial Intelligence 24, 39-44, 2010.

[26] R. E. Korf, *Minimizing disk I/O in two-bit breadth-first search*, Proceedings of the AAAI Conference on Artificial Intelligence 23, 317-324, 2008.

[27] R. Clausecker and A. Reinefeld, *Zero-aware pattern databases with 1-bit compression for sliding tile puzzles*, Proceedings of the Symposium on Combinatorial Search (SoCS) 10, 35-43, 2019.

[28] F. Yang, J. Culberson, R. Holte, U. Zahavi and A. Felner, *A general theory of additive state space abstractions*, Journal of Artificial Intelligence Research 32, 631-662, 2008.

[29] U. Zahavi, A. Felner, R. C. Holte and J. Schaeffer, *Duality in permutation state spaces and the dual search algorithm*, Artificial Intelligence 172(4-5), 514-540, 2008.

[30] R. C. Holte, A. Felner, J. Newton, R. Meshulam and D. Furcy, *Maximizing over multiple pattern databases speeds up heuristic search*, Artificial Intelligence 170(16-17), 1123-1136, 2006.

[31] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant and Z. Zhang, *Inconsistent heuristics in theory and practice*, Artificial Intelligence 175(9-10), 1570-1603, 2011.

[32] T. Nagahashi and D. Takahashi, *Construction of large zero-aware pattern databases for sliding puzzles on distributed memory machines*, Computational Science and Its Applications (ICCSA 2025), Lecture Notes in Computer Science, 272-284, 2025.

[33] M. Samadi, M. Siabani, A. Felner and R. Holte, *Compressing pattern databases with learning*, Proceedings of the European Conference on Artificial Intelligence (ECAI 2008), Frontiers in Artificial Intelligence and Applications 178, 2008.

[34] W. Myrvold and F. Ruskey, *Ranking and unranking permutations in linear time*, Information Processing Letters 79(6), 281-284, 2001.

[35] E. W. Dijkstra and C. S. Scholten, *Termination detection for diffusing computations*, Information Processing Letters 11(1), 1-4, 1980.

[36] D. Hendler and N. Shavit, *Non-blocking steal-half work queues*, Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), 280-289, 2002.

[37] H. Rihani, P. Sanders and R. Dementiev, *MultiQueues: simple relaxed concurrent priority queues*, Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 80-82, 2015.

[38] A. Postnikova, N. Koval, G. Nadiradze and D. Alistarh, *Multi-queues can be state-of-the-art priority schedulers*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming

(PPoPP), 353-367, 2022.

[39] S. Edelkamp, S. Jabbar and S. Schrödl, *External A**, KI 2004: Advances in Artificial Intelligence, Lecture Notes in Computer Science 3238, 226-240, 2004.

[40] R. E. Korf, *Linear-time disk-based implicit graph search*, Journal of the ACM 55(6), 26:1-26:40, 2008.

[41] T. Ikeda and H. Imai, *Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases*, Theoretical Computer Science 210(2), 341-374, 1999.

[42] T. Yoshizumi, T. Miura and T. Ishida, *A* with partial expansion for large branching factor problems*, Proceedings of the AAAI Conference on Artificial Intelligence 17, 923-929, 2000.

[43] Y. Kobayashi, A. Kishimoto and O. Watanabe, *Evaluations of Hash Distributed A* in optimal sequence alignment*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 22, 584-589, 2011.

[44] M. O. Dayhoff, R. M. Schwartz and B. C. Orcutt, *A model of evolutionary change in proteins*, Atlas of Protein Sequence and Structure 5, supplement 3, 345-352, 1978.

[45] J. D. Thompson, P. Koehl, R. Ripp and O. Poch, *BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark*, Proteins 61(1), 127-136, 2005.

[46] I. Pohl, *Heuristic search viewed as path finding in a graph*, Artificial Intelligence 1(3-4), 193-204, 1970.

[47] E. A. Hansen and R. Zhou, *Anytime heuristic search*, Journal of Artificial Intelligence Research 28, 267-297, 2007.

[48] M. Likhachev, G. Gordon and S. Thrun, *ARA*: Anytime A* with provable bounds on sub-optimality*, Advances in Neural Information Processing Systems (NIPS) 16, 767-774, 2003.

[49] J. T. Thayer and W. Ruml, *Faster than weighted A*: an optimistic approach to bounded suboptimal search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 18, 355-362, 2008.

[50] J. Chen, N. R. Sturtevant, W. Doyle and W. Ruml, *Revisiting suboptimal search*, Proceedings of the Symposium on Combinatorial Search (SoCS) 10, 18-25, 2019.

[51] J. Pearl and J. H. Kim, *Studies in semi-admissible heuristics*, IEEE Transactions on Pattern Analysis and Machine Intelligence 4(4), 392-399, 1982.

[52] J. T. Thayer and W. Ruml, *Bounded suboptimal search: a direct approach using inadmissible estimates*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 22, 674-679, 2011.

[53] D. Gilon, A. Felner and R. Stern, *Dynamic potential search: a new bounded suboptimal search algorithm*, Proceedings of the Symposium on Combinatorial Search (SoCS) 7, 36-44, 2016.

[54] S. Koenig, M. Likhachev and D. Furcy, *Lifelong Planning A**, Artificial Intelligence 155(1-2), 93-146, 2004.

[55] A. Reinefeld and V. Schnecke, *AIDA*: asynchronous parallel IDA**, Proceedings of the Biennial Conference of the Canadian Society for Computational Studies of Intelligence (AI 94), 295-302, 1994.

[56] Z. Bu and R. E. Korf, *A*+IDA*: a simple hybrid search algorithm*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 28, 1206-1212, 2019.

[57] L. Siag, S. S. Shperberg, A. Felner and N. R. Sturtevant, *On parallel external-memory bidirectional search*, Proceedings of the European Conference on Artificial Intelligence (ECAI 2024), Frontiers in Artificial Intelligence

and Applications 392, 4190-4197, 2024.

[58] V. N. Rao and V. Kumar, *Parallel depth first search. Part I. Implementation*, International Journal of Parallel Programming 16(6), 479-499, 1987.

[59] R. E. Korf, *Finding optimal solutions to Rubik's Cube using pattern databases*, Proceedings of the AAAI Conference on Artificial Intelligence 14, 700-705, 1997.

[60] M. Helmert, *Landmark heuristics for the pancake problem*, Proceedings of the Symposium on Combinatorial Search (SoCS) 1, 109-110, 2010.

[61] R. C. Holte, A. Felner, G. Sharon and N. R. Sturtevant, *Bidirectional search that is guaranteed to meet in the middle*, Proceedings of the AAAI Conference on Artificial Intelligence 30, 2016.

[62] B. Bonet, *Efficient algorithms to rank and unrank permutations in lexicographic order*, AAAI Workshop on Search in Artificial Intelligence and Robotics, Technical Report WS-08-10, 18-23, 2008.

[63] H. Dweighter, *Problem E2569*, in *Elementary Problems: E2564-E2569*, The American Mathematical Monthly 82(10), 1009, 1975.

[64] C. Demetrescu, A. V. Goldberg and D. S. Johnson (editors), *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 74, AMS, 2009.

[65] R. B. Dial, *Algorithm 360: shortest-path forest with topological ordering*, Communications of the ACM 12(11), 632-633, 1969.

[66] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics 17(2), 416-429, 1969.

[67] B. Archibald, P. Maier, R. Stewart and P. Trinder, *Replicable parallel branch and bound search*, Journal of Parallel and Distributed Computing 113, 92-114, 2018.

[68] B. Archibald, P. Maier, R. Stewart and P. Trinder, *YewPar: skeletons for exact combinatorial search*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) 25, 292-307, 2020.

[69] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch and M. Winkler, *Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores*, Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) 30, 770-779, 2016.

[70] Y. Xu, T. K. Ralphs, L. Ladányi and M. J. Saltzman, *ALPS: a framework for implementing parallel tree search algorithms*, The Next Wave in Computing, Optimization, and Decision Technologies, 319-334, Springer, 2005.

[71] M. Mezmaz, R. Leroy, N. Melab and D. Tuyttens, *A multi-core parallel branch-and-bound algorithm using factorial number system*, Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) 28, 1203-1212, 2014.

[72] J. Gmys, R. Leroy, M. Mezmaz, N. Melab and D. Tuyttens, *Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms*, Concurrency and Computation: Practice and Experience 28(18), 4463-4484, 2016.

[73] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan and C.-W. Tseng, *UTS: an unbalanced tree search benchmark*, Languages and Compilers for Parallel Computing (LCPC) 19, Lecture Notes in Computer

Science 4382, 235-250, 2007.

[74] E. Tomita and T. Kameda, *An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments*, Journal of Global Optimization 37(1), 95-111, 2007.

[75] P. San Segundo, D. Rodríguez-Losada and A. Jiménez, *An exact bit-parallel algorithm for the maximum clique problem*, Computers & Operations Research 38(2), 571-581, 2011.

[76] C. McCreesh, P. Prosser and J. Trimble, *A partitioning algorithm for maximum common subgraph problems*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 26, 712-719, 2017.

[77] P. C. Gilmore, *Optimal and suboptimal algorithms for the quadratic assignment problem*, Journal of the Society for Industrial and Applied Mathematics 10(2), 305-313, 1962.

[78] R. E. Burkard, S. E. Karisch and F. Rendl, *QAPLIB - a quadratic assignment problem library*, Journal of Global Optimization 10(4), 391-403, 1997.

[79] B. J. Lageweg, J. K. Lenstra and A. H. G. Rinnooy Kan, *A general bounding scheme for the permutation flow-shop problem*, Operations Research 26(1), 53-67, 1978.

[80] E. Taillard, *Benchmarks for basic scheduling problems*, European Journal of Operational Research 64(2), 278-285, 1993.

[81] R. C. Holte, A. Felner, G. Sharon, N. R. Sturtevant and J. Chen, *MM: A bidirectional search algorithm that is guaranteed to meet in the middle*, Artificial Intelligence 252, 232-266, 2017.

[82] G. Sharon, R. C. Holte, A. Felner and N. R. Sturtevant, *Extended abstract: an improved priority function for bidirectional heuristic search*, Proceedings of the Symposium on Combinatorial Search (SoCS) 7, 139-140, 2016.

[83] E. Shaham, A. Felner, J. Chen and N. R. Sturtevant, *The minimal set of states that must be expanded in a front-to-end bidirectional search*, Proceedings of the Symposium on Combinatorial Search (SoCS) 8, 82-90, 2017.

[84] J. Eckerle, J. Chen, N. R. Sturtevant, S. Zilles and R. C. Holte, *Sufficient conditions for node expansion in bidirectional heuristic search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 27, 79-87, 2017.

[85] J. Chen, R. C. Holte, S. Zilles and N. R. Sturtevant, *Front-to-end bidirectional heuristic search with near-optimal node expansions*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 489-495, 2017.

[86] S. S. Shperberg, A. Felner, N. R. Sturtevant, S. E. Shimony and A. Hayoun, *Enriching non-parametric bidirectional search algorithms*, Proceedings of the AAAI Conference on Artificial Intelligence 33, 2379-2386,

2019.

[87] M. Barley, P. Riddle, C. Linares López, S. Dobson and I. Pohl, *GBFHS: A generalized breadth-first heuristic search algorithm*, Proceedings of the Symposium on Combinatorial Search (SoCS) 9, 28-36, 2018.

[88] S. K. Sadhukhan, *Bidirectional heuristic search based on error estimate*, CSI Journal of Computing 2(1-2), S1, 2013.

[89] V. Alcázar, P. J. Riddle and M. Barley, *A unifying view on individual bounds and heuristic inaccuracies in bidirectional search*, Proceedings of the AAAI Conference on Artificial Intelligence 34, 2327-2334, 2020.

[90] J. B. H. Kwa, *BS*: An admissible bidirectional staged heuristic search algorithm*, Artificial Intelligence 38(1), 95-109, 1989.

[91] I. Pohl, *Bi-directional search*, Machine Intelligence 6, 127-140, 1971.

[92] W. Pijls and H. Post, *A new bidirectional search algorithm with shortened postprocessing*, European Journal of Operational Research 198(2), 363-369, 2009.

[93] L. H. O. Rios and L. Chaimowicz, *PNBA*: A parallel bidirectional heuristic search algorithm*, Proceedings of the Encontro Nacional de Inteligência Artificial (ENIA) 8, 2011.

[94] N. R. Sturtevant and J. Chen, *External memory bidirectional search*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 676-682, 2016.

[95] S. Lin and A. Fukunaga, *Revisiting immediate duplicate detection in external memory search*, Proceedings of the AAAI Conference on Artificial Intelligence 32, 2018.

[96] J. E. Doran and D. Michie, *Experiments with the Graph Traverser program*, Proceedings of the Royal Society A 294, 235-259, 1966.

[97] S. Richter and M. Helmert, *Preferred operators and deferred evaluation in satisficing planning*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 19, 273-280, 2009.

[98] G. Röger and M. Helmert, *The more, the merrier: combining heuristic estimators for satisficing planning*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 20, 246-249, 2010.

[99] F. Xie, M. Müller, R. Holte and T. Imai, *Type-based exploration with multiple search queues for satisficing planning*, Proceedings of the AAAI Conference on Artificial Intelligence 28, 2014.

[100] R. A. Valenzano, N. R. Sturtevant, J. Schaeffer and F. Xie, *A comparison of knowledge-based GBFS enhancements and knowledge-free exploration*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 24, 375-379, 2014.

[101] J. Hoffmann and B. Nebel, *The FF planning system: fast plan generation through heuristic search*, Journal of Artificial Intelligence Research 14, 253-302, 2001.

[102] S. Richter and M. Westphal, *The LAMA planner: guiding cost-based anytime planning with landmarks*, Journal of Artificial Intelligence Research 39, 127-177, 2010.

[103] S. Richter, J. T. Thayer and W. Ruml, *The joy of forgetting: faster anytime search via restarting*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 20, 137-144, 2010.

[104] N. Lipovetzky and H. Geffner, *Width and serialization of classical planning problems*, Proceedings of the European Conference on Artificial Intelligence (ECAI 2012), Frontiers in Artificial Intelligence and Applications 242, 540-545, 2012.

[105] N. Lipovetzky and H. Geffner, *Best-first width search: exploration and exploitation in classical planning*, Proceedings of the AAAI Conference on Artificial Intelligence 31, 2017.

[106] V. Vidal, L. Bordeaux and Y. Hamadi, *Adaptive K-parallel best-first search: a simple but efficient algorithm for multi-core domain-independent planning*, Proceedings of the Symposium on Combinatorial Search (SoCS) 1,

100-107, 2010.

[107] R. Kuroiwa and A. Fukunaga, *On the pathological search behavior of distributed greedy best-first search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 29, 255-263, 2019.

[108] R. Kuroiwa and A. Fukunaga, *Analyzing and avoiding pathological behavior in parallel best-first search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 30, 175-183, 2020.

[109] T. Shimoda and A. Fukunaga, *Improved exploration of the bench transition system in parallel greedy best first search*, Proceedings of the Symposium on Combinatorial Search (SoCS) 16, 74-82, 2023.

[110] T. Shimoda and A. Fukunaga, *Parallel greedy best-first search with a bound on expansions relative to sequential search*, Proceedings of the AAAI Conference on Artificial Intelligence 39, 26668-26677, 2025.

[111] T. Shimoda and A. Fukunaga, *Decoupling generation and evaluation for parallel greedy best-first search*, arXiv:2408.05682, 2024.

[112] M. Heusner, T. Keller and M. Helmert, *Understanding the search behaviour of greedy best-first search*, Proceedings of the Symposium on Combinatorial Search (SoCS) 8, 47-55, 2017.

[113] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer and R. C. Holte, *Partial-expansion A* with selective node generation*, Proceedings of the AAAI Conference on Artificial Intelligence 26, 471-477, 2012.

[114] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte and J. Schaeffer, *Enhanced partial expansion A**, Journal of Artificial Intelligence Research 50, 141-187, 2014.

[115] Y. Björnsson, M. Enzenberger, R. C. Holte and J. Schaeffer, *Fringe search: beating A* at pathfinding on game maps*, Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG 2005), 125-132, 2005.

[116] R. E. Korf, *Linear-space best-first search*, Artificial Intelligence 62(1), 41-78, 1993.

[117] R. Zhou and E. A. Hansen, *Beam-stack search: integrating backtracking with beam search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 15, 90-98, 2005.

[118] R. E. Korf, *Real-time heuristic search*, Artificial Intelligence 42(2-3), 189-211, 1990.

[119] S. Koenig and X. Sun, *Comparing real-time and incremental heuristic search for real-time situated agents*, Autonomous Agents and Multi-Agent Systems 18(3), 313-341, 2009.

[120] A. Botea, M. Müller and J. Schaeffer, *Near optimal hierarchical path-finding*, Journal of Game Development 1(1), 7-28, 2004.

[121] S. Mukherjee, S. Aine and M. Likhachev, *ePA*SE: Edge-based parallel A* for slow evaluations*, Proceedings of the International Symposium on Combinatorial Search (SoCS) 15, 136-144, 2022.

[122] S. Mukherjee and M. Likhachev, *GePA*SE: Generalized edge-based parallel A* for slow evaluations*, Proceedings of the International Symposium on Combinatorial Search (SoCS) 16, 153-157, 2023.

[123] H. Yang, S. Mukherjee and M. Likhachev, *A-ePA*SE: Anytime edge-based parallel A* for slow evaluations*, Proceedings of the International Symposium on Combinatorial Search (SoCS) 16, 163-167, 2023.

[124] S. Mukherjee, S. Aine and M. Likhachev, *MPLP: Massively parallelized lazy planning*, IEEE Robotics and Automation Letters 7(3), 6067-6074, 2022.

[125] C. M. Dellin and S. S. Srinivasa, *A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors*, Proceedings of the International Conference

on Automated Planning and Scheduling (ICAPS) 26, 459-467, 2016.

[126] F. Agostinelli, S. McAleer, A. Shmakov and P. Baldi, *Solving the Rubik's cube with deep reinforcement learning and search*, Nature Machine Intelligence 1, 356-363, 2019.

[127] M. Evett, J. Hendler, A. Mahanti and D. Nau, *PRA*: massively parallel heuristic search*, Journal of Parallel and Distributed Computing 25(2), 133-143, 1995.

[128] R. Zhou and E. A. Hansen, *Parallel structured duplicate detection*, Proceedings of the AAAI Conference on Artificial Intelligence 22, 1217-1223, 2007.

[129] R. Zhou and E. A. Hansen, *Breadth-first heuristic search*, Artificial Intelligence 170(4-5), 385-408, 2006.

[130] A. Felner, S. Kraus and R. E. Korf, *KBFS: K-best-first search*, Annals of Mathematics and Artificial Intelligence 39(1-2), 19-39, 2003.

[131] J. W. Romein, A. Plaat, H. E. Bal and J. Schaeffer, *Transposition table driven work scheduling in distributed search*, Proceedings of the AAAI Conference on Artificial Intelligence 16, 725-731, 1999.

[132] U. K. Sarkar, P. P. Chakrabarti, S. Ghose and S. C. De Sarkar, *Reducing reexpansions in iterative-deepening search by controlling cutoff bounds*, Artificial Intelligence 50(2), 207-221, 1991.

[133] C. Bäckström and B. Nebel, *Complexity results for SAS+ planning*, Computational Intelligence 11(4), 625-655, 1995.

[134] M. Helmert, *The Fast Downward planning system*, Journal of Artificial Intelligence Research 26, 191-246, 2006.

[135] B. Bonet and H. Geffner, *Planning as heuristic search*, Artificial Intelligence 129(1-2), 5-33, 2001.

[136] M. Helmert and C. Domshlak, *Landmarks, critical paths and abstractions: what's the difference anyway?*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 19, 162-169, 2009.

[137] S. Edelkamp, *Planning with pattern databases*, Proceedings of the European Conference on Planning (ECP), 13-24, 2001.

[138] Y. Jinnai and A. Fukunaga, *Automated creation of efficient work distribution functions for parallel best-first search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 26, 184-192, 2016.

[139] R. Zhou and E. A. Hansen, *Domain-independent structured duplicate detection*, Proceedings of the AAAI Conference on Artificial Intelligence 21, 1082-1087, 2006.

[140] M. Helmert, P. Haslum, J. Hoffmann and R. Nissim, *Merge-and-shrink abstraction: a method for generating lower bounds in factored state spaces*, Journal of the ACM 61(3), 16:1-16:63, 2014.

[141] Y. Jinnai and A. Fukunaga, *Abstract Zobrist hashing: an efficient work distribution method for parallel best-first search*, Proceedings of the AAAI Conference on Artificial Intelligence 30, 717-723, 2016.

[142] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, Proceedings of the 19th Design Automation Conference, 175-181, 1982.

[143] S. Koenig and M. Likhachev, *D* Lite*, Proceedings of the AAAI Conference on Artificial Intelligence (AAAI) 18, 476-483, 2002; and *Fast replanning for navigation in unknown terrain*, IEEE Transactions on Robotics 21(3),

354-363, 2005.

[144] G. Ramalingam and T. Reps, *An incremental algorithm for a generalization of the shortest-path problem*, Journal of Algorithms 21(2), 267-305, 1996.

[145] U. Meyer and P. Sanders, *Δ-stepping: a parallelizable shortest path algorithm*, Journal of Algorithms 49(1), 114-152, 2003.

[146] A. H. Land and A. G. Doig, *An automatic method of solving discrete programming problems*, Econometrica 28(3), 497-520, 1960.

[147] G. B. Dantzig, *Discrete-variable extremum problems*, Operations Research 5(2), 266-288, 1957.

[148] D. Pisinger, *Where are the hard knapsack problems?*, Computers & Operations Research 32(9), 2271-2284, 2005.

[149] T.-H. Lai and S. Sahni, *Anomalies in parallel branch-and-bound algorithms*, Communications of the ACM 27(6), 594-602, 1984.

[150] W. W. Johnson and W. E. Story, *Notes on the "15" Puzzle*, American Journal of Mathematics 2(4), 397-404, 1879.

[151] D. Lemire, *Fast random integer generation in an interval*, ACM Transactions on Modeling and Computer Simulation 29(1), 1-12, 2019.

[152] A. L. Zobrist, *A new hashing method with application for game playing*, Technical Report 88, Computer Sciences Department, University of Wisconsin, 1970; reprinted in ICGA Journal 13(2), 69-73, 1990.

[153] V. Kumar, K. Ramesh and V. N. Rao, *Parallel best-first search of state-space graphs: a summary of results*, Proceedings of the AAAI Conference on Artificial Intelligence 7, 122-127, 1988.

[154] R. M. Karp and Y. Zhang, *Randomized parallel algorithms for backtrack search and branch-and-bound computation*, Journal of the ACM 40(3), 765-789, 1993.

[155] K. B. Irani and Y. Shih, *Parallel A* and AO* algorithms: an optimality criterion and performance evaluation*, Proceedings of the International Conference on Parallel Processing (ICPP), 274-277, 1986.

[156] R. E. Korf and P. Schultze, *Large-scale parallel breadth-first search*, Proceedings of the AAAI Conference on Artificial Intelligence 20, 1380-1385, 2005.

[157] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd edition, Addison-Wesley, 1998.

[158] D. B. Johnson, *Priority queues with update and finding minimum spanning trees*, Information Processing Letters 4(3), 53-57, 1975.

[159] D. R. Hanson, *Fast allocation and deallocation of memory based on object lifetimes*, Software: Practice and Experience 20(1), 5-12, 1990.

[160] R. K. Ahuja, K. Mehlhorn, J. B. Orlin and R. E. Tarjan, *Faster algorithms for the shortest path problem*, Journal of the ACM 37(2), 213-223, 1990.

[161] A. Reinefeld and T. A. Marsland, *Enhanced iterative-deepening search*, IEEE Transactions on Pattern Analysis and Machine Intelligence 16(7), 701-710, 1994.

[162] T. Miura and T. Ishida, *Stochastic node caching for memory-bounded search*, Systems and Computers in Japan 30(2), 57-65, 1999.

[163] R. E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers C-35(8), 677-691, 1986.

[164] Y. Jinnai, *Introduction to heuristic search* (in Japanese), lecture notes, 175 pages, version of 9 December 2023, available from https://jinnaiyuu.github.io/.

[165] N. R. Sturtevant, *HOG2: Hierarchical Open Graph*, software, Moving AI Lab, University of Denver.

[166] E. L. Lawler, *The quadratic assignment problem*, Management Science 9(4), 586-599, 1963.

[167] A. Felner, U. Zahavi, J. Schaeffer and R. C. Holte, *Dual lookups in pattern databases*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 19, 103-108, 2005.

[168] R. Nissim, J. Hoffmann and M. Helmert, *Computing perfect heuristics in polynomial time: on bisimulation and merge-and-shrink abstraction in optimal planning*, Proceedings of the International Joint Conference on

Artificial Intelligence (IJCAI) 22, 1983-1990, 2011.

[169] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator, *Competitive snoopy caching*, Algorithmica 3(1-4), 79-119, 1988.

[170] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM, 2002.

[171] B. T. Lowerre, *The HARPY speech recognition system*, PhD thesis, Carnegie Mellon University, 1976.

[172] H. Kociemba, *kociemba.org/cube.htm*, web pages on the two-phase algorithm for Rubik's cube and its cubie model, consulted in November 2026.

[173] E. Keyder and H. Geffner, *Heuristics for planning with action costs revisited*, Proceedings of the European Conference on Artificial Intelligence (ECAI), Frontiers in Artificial Intelligence and Applications 178, 588-592, 2008.

[174] D. S. Johnson and M. A. Trick (editors), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 26, American Mathematical Society, 1996.

[175] C. Bron and J. Kerbosch, *Algorithm 457: finding all cliques of an undirected graph*, Communications of the ACM 16(9), 575-577, 1973.

[176] T. C. Koopmans and M. Beckmann, *Assignment problems and the location of economic activities*, Econometrica 25(1), 53-76, 1957.

[177] H. W. Kuhn, *The Hungarian method for the assignment problem*, Naval Research Logistics Quarterly 2(1-2), 83-97, 1955.

[178] S. M. Johnson, *Optimal two- and three-stage production schedules with setup times included*, Naval Research Logistics Quarterly 1(1), 61-68, 1954.

[179] National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, Federal Information Processing Standards Publication 180-4, 2015.

[180] L. Mandow and J. L. Pérez de la Cruz, *Multiobjective A* search with consistent heuristics*, Journal of the ACM 57(5), 1-25, 2010.

[181] C. Hernández, W. Yeoh, J. A. Baier, H. Zhang, L. Suazo, S. Koenig and O. Salzman, *Simple and efficient bi-objective search algorithms via fast dominance checks*, Artificial Intelligence 314, 103807, 2023.

[182] Z. Ren, R. Zhan, S. Rathinam, M. Likhachev and H. Choset, *Enhanced multi-objective A* using balanced binary search trees*, Proceedings of the International Symposium on Combinatorial Search (SoCS) 15, 162-170, 2022.

[183] C. Hernández, W. Yeoh, J. A. Baier, A. Felner, O. Salzman, H. Zhang, S.-H. Chan and S. Koenig, *Multi-objective search via lazy and efficient dominance checks*, Proceedings of the International Joint Conference

on Artificial Intelligence (IJCAI) 32, 7223-7230, 2023.

[184] S. Ahmadi, N. R. Sturtevant, A. Raith, D. Harabor and M. Jalili, *Parallelizing multi-objective A* search*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 35, 131-139, 2025.

[185] F.-J. Pulido, L. Mandow and J. L. Pérez de la Cruz, *Dimensionality reduction in multiobjective shortest path search*, Computers & Operations Research 64, 60-70, 2015.

[186] P. Sanders and L. Mandow, *Parallel label-setting multi-objective shortest path search*, Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS) 27, 215-224, 2013.

[187] X. Xu, N. Zhang, L. Qian, X. Liu, J. Wu and H. Tang, *Asynchronous parallel search for exact multi-objective shortest paths with versioned frontier snapshots and indexed dominance pruning*, arXiv:2609.11944, 2026.

[188] P. Maristany de las Casas, L. Kraus, A. Sedeño-Noda and R. Borndörfer, *Targeted multiobjective Dijkstra algorithm*, Networks 82(3), 277-298, 2023.

[189] P. M. Fenwick, *A new data structure for cumulative frequency tables*, Software: Practice and Experience 24(3), 327-336, 1994.

[190] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, *Symbolic model checking: 10^20 states and beyond*, Information and Computation 98(2), 142-170, 1992.

[191] S. Edelkamp and F. Reffel, *OBDDs in heuristic search*, KI-98: Advances in Artificial Intelligence, Lecture Notes in Computer Science 1504, 81-92, 1998.

[192] R. M. Jensen, M. M. Veloso and R. E. Bryant, *State-set branching: Leveraging BDDs for heuristic search*, Artificial Intelligence 172(2-3), 103-139, 2008.

[193] Á. Torralba, V. Alcázar, P. Kissmann and S. Edelkamp, *Efficient symbolic search for cost-optimal planning*, Artificial Intelligence 242, 52-79, 2017.

[194] T. van Dijk and J. van de Pol, *Sylvan: multi-core framework for decision diagrams*, International Journal on Software Tools for Technology Transfer 19(6), 675-696, 2017.

[195] D. Speck, R. Mattmüller and B. Nebel, *Symbolic top-k planning*, Proceedings of the AAAI Conference on Artificial Intelligence 34, 9967-9974, 2020.

[196] S. Edelkamp and P. Kissmann, *Optimal symbolic planning with action costs and preferences*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1690-1695, 2009.

[197] F. Somenzi, *CUDD: CU Decision Diagram package, release 3.0.0*, University of Colorado at Boulder, 2015.

[198] Á. Torralba, V. Alcázar, D. Borrajo, P. Kissmann and S. Edelkamp, *SymBA*: A symbolic bidirectional A* planner*, The 2014 International Planning Competition: Description of Planners, 105-109, 2014.

[199] A. Felner and N. Ofek, *Combining perimeter search and pattern database abstractions*, Abstraction, Reformulation, and Approximation (SARA), Lecture Notes in Computer Science 4612, 155-168, 2007.

[200] J. Seipp and M. Helmert, *Counterexample-guided Cartesian abstraction refinement*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 23, 347-351, 2013.

[201] J. Seipp and M. Helmert, *Counterexample-guided Cartesian abstraction refinement for classical planning*, Journal of Artificial Intelligence Research 62, 535-577, 2018.

[202] J. Seipp and M. Helmert, *Diverse and additive Cartesian abstraction heuristics*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) 24, 289-297, 2014.