Journal of Artificial Intelligence Research 70 (2021) 631-681 Submitted 06/2020; published 02/2021

Regarding Goal Bounding and Jump Point Search

Yue Hu HUYUE.CSE@GMAIL.COM
College of Systems Engineering
National University of Defense Technology
Kaifu District, Changsha, Hunan, China

Daniel Harabor DANIEL.HARABOR@MONASH.EDU
Monash University
Melbourne, Australia

Long Qin QLDBX2007@SINA.COM
Quanjun Yin* YIN_QUANJUN@163.COM
College of Systems Engineering
National University of Defense Technology
Kaifu District, Changsha, Hunan, China

Abstract

Jump Point Search (JPS) is a well known symmetry-breaking algorithm that can substantially improve performance for grid-based optimal pathfinding. When the input grid is static further speedups can be obtained by combining JPS with goal bounding techniques such as Geometric Containers (instantiated as Bounding Boxes) and Compressed Path Databases. Two such methods, JPS+BB and Two-Oracle Path PlannING (Topping), are currently among the fastest known approaches for computing shortest paths on grids. The principal drawback for these algorithms is the overhead costs: each one requires an all-pairs precomputation step, the running time and subsequent storage costs of which can be prohibitive. In this work we consider an alternative approach where we precompute and store goal bounding data only for grid cells which are also *jump points*. Since the number of jump points is usually much smaller than the total number of grid cells, we can save up to orders of magnitude in preprocessing time and space.

Considerable precomputation savings do not necessarily mean performance degradation. For a second contribution we show how *canonical orderings*, *partial expansion strategies* and *enhanced intermediate pruning* can be leveraged to improve online query performance despite a reduction in preprocessed data. The combination of faster preprocessing and stronger online reasoning leads to three new and highly performant algorithms: JPS+BB+ and Two-Oracle Pathfinding Search (TOPS) based on search, and Topping+ based on path extraction. We give a theoretical analysis showing that each method is complete and optimal. We also report convincing gains in a comprehensive empirical evaluation that includes almost all current and cutting-edge algorithms for grid-based pathfinding.

1. Introduction

Pathfinding on uniform-cost grids is a common problem arising in areas such as robotics and computer games. In recent years there has been a significant amount of attention given to this topic, fuelled in no small part by several international competitions and the

release of a standard set of benchmarks (Sturtevant, 2012). At the 2014 Grid-based Path Planning Competition (GPPC), a number of popular algorithms appeared undominated on the Pareto front of successful entries (Sturtevant et al., 2015). Among these leading techniques is a symmetry breaking method called Jump Point Search (JPS) (Harabor & Grastien, 2011).

JPS can be described as the combination of A* search with a *canonical ordering* applied over the set of available grid moves (Sturtevant & Rabin, 2016). The idea is simple: when moving on a grid JPS prefers to take paths which are *diagonal-first*; i.e. where diagonal moves appear before straight moves whenever possible. Other paths, where diagonal moves appear later, are discarded by JPS. The main advantage from breaking such symmetries is performance: JPS can speed up optimal grid search by ten times and more, all online and without any additional time or memory costs. Further performance gains can be obtained by precomputing jump points (Harabor & Grastien, 2014) and by *goal bounding*: a complementary speedup technique that prunes successors when they cannot possibly appear on an optimal path. Algorithms combining these ideas include JPS+BB (Rabin & Sturtevant, 2016) and Topping (Salvetti et al., 2018), both of which are *up to thousands of times faster* than A* search on grids.

The price for these gains comes in the form of an all-pairs precomputation step, which is necessary to create auxiliary goal-bounding data. This procedure can take up to hours of additional time and hundreds of megabytes of additional memory for just a single grid map. Such overheads can be prohibitive in applications where CPU and memory budgets available for path planning are limited. Unfortunately, this is often the case in robotics and computer games. To the best of our knowledge, there currently exists no pathfinding algorithm that has affordable preprocessing costs and also computes optimal paths at ultra-fast speed (i.e. at the same order of magnitude as JPS+BB and Topping). In this paper we introduce new ideas and algorithms to help bridge this gap. Our contributions include:

(1) **Faster preprocessing:** to reduce offline preprocessing we show that it is only necessary to execute Dijkstra search from grid cells where jump points are located. Since the number of jump points is usually much smaller than the total number of traversable cells this approach can save up to orders of magnitude of preprocessing time and space.

(2) **Faster online queries:** we introduce two families of new ultra-fast algorithms for computing grid-optimal shortest paths: Bounding Boxes based JPS+BB+ and Compressed Pattern Databases based TOPS and Topping+. Both use goal bounding data computed only at jump points. Queries with reduced auxiliary data do not have to lead to reduced performance. In a main experimental result we show that JPS+BB+ can be approximately 30% faster than JPS+BB. Meanwhile, we find that TOPS and Topping+ are comparable with and sometimes faster than the original.

(3) **Comprehensive comparisons:** we compare a wide range of current and state-of-the-art algorithms for grid-based pathfinding. Our lineup includes past winners from the 2014 GPPC and a variety of state-of-the-art techniques appearing since then. We test each method on hundreds of maps and more than 1 million instances, drawn from the competition and from other standard benchmarks.

A subset of our contributions have previously appeared in a short conference paper (Hu et al., 2019). Compared to that work we contribute a variety of new algorithms, new start-successor pruning rules, enhanced intermediate pruning strategy, extended theoretical results and a much larger and more comprehensive empirical section.

2. Problem Definition

We are interested in search problems on 8-connected grid maps. In this domain each cell of the grid is marked as traversable or blocked by an obstacle. There are up to 8 available *move* actions $\vec{v}$ (equiv. *directions*) which serve to transition the search, from each grid cell to one of its cardinally or diagonally adjacent neighbours. The cost of each cardinal (i.e. straight) move (denoted $\vec{c}$) is 1 and the cost of each diagonal move (denoted $\vec{d}$) is $\sqrt{2}$. A move is valid if it does not begin or end in an obstacle. In addition, we enforce *no-corner-cutting* constraints; i.e. we disallow diagonal transitions between two traversable cells if they share a neighbouring obstacle. Each search episode begins at a distinguished cell known as the *start* and is considered successful if it finishes at a distinguished cell called the *target*. The objective is to find a path (i.e. a sequence of valid steps) which is of minimum total cost of all its constituent moves among all paths from start to target.

We sometimes refer to cardinal directions by name: $\{North, South, West, East\}$ (N, S, W and E for short). We also say that each diagonal direction is the result of combining two orthogonal cardinal directions; i.e. $\vec{d} = \vec{c_1} + \vec{c_2}$ with $\vec{d} \in \{NW, NE, SW, SE\}$. Finally, we use the algebra $s' = s + k\vec{d} + m\vec{c}$, with k and m as integers, to indicate a path from s to s' with $k \geq 0$ moves in direction $\vec{d}$ followed by $m \geq 0$ moves in direction $\vec{c}$. We will call such sequences of moves *continuations* of the path. Note that negative integer values for k and m indicate moves in an inverse direction; i.e. $s' = s - N$ is equivalent to $s' = s + S$.

3. Related Work

Pathfinding on grid maps is a classic problem in Artificial Intelligence. It is also a common setup for application areas such as robotics (Choset et al., 2005; Wurman et al., 2008; Sartoretti et al., 2019) and computer games (Sturtevant, 2007; Kring, Champandard, & Samarin, 2010; Rabin & Sturtevant, 2013, 2017). Interest in grid-based pathfinding has grown substantially in recent years, fuelled in part by the release of a standard set of benchmarks (Sturtevant, 2012) and by several editions of the international Grid-based Path Planning Competition (GPPC) (Sturtevant et al., 2015).

In this section we take a broad view of the research literature as it pertains to **optimal pathfinding in static grid maps**. We give a coarse taxonomic classification of works from the literature, and from the GPPC, and we subsequently survey the state of the art in each identified category. The categories are:

- *Fundamental improvements*, referring to works that reduce the effort to maintain the A* Open and Closed lists and which seek to reduce the total sorting workload.

- *Better heuristics*, referring to algorithms, often preprocessing based, that focus on improving the cost-to-go estimates during online search.

- *Symmetry breaking*, referring to algorithms that exploit the topological structure of grid maps to reduce the size of the state space.

- *Goal bounding*, referring to algorithms that prune from consideration edges which cannot possibly belong to an optimal path.

- *Contraction*, referring to graph-augmentation techniques where additional "shortcut edges" are added to the input graph as a way to speed up search.

3.1 Fundamental Improvements

One way to improve the performance of fundamental search algorithms, such as A*, is through the use of specialised data structures for maintaining the Open and Closed lists. Recall that Open is a priority queue where nodes marked for expansion are ranked by their objective or f-value. Recall also that the Closed list is a hashing container used to avoid duplicated expansions; i.e. it stores all nodes that have already been expanded by A* search.

3.1.1 Maintaining the Open List

Perhaps the most common approach to implementing the A* Open list, elsewhere but also in this work, is to use a Binary Heap (Cormen et al., 2001). This data structure maintains the list of expansion candidates as an unbalanced binary tree. It features logarithmic complexity for removing the top-ranked node from the queue, for adding new nodes to the queue and for updating the f-value of nodes already in the queue.

A variety of suggestions appear in the literature for alternative data structures which can improve sorting performance in comparison to Binary Heaps. Among the best known are Fibonacci Heaps (Fredman & Tarjan, 1987) and Pairing Heaps (Fredman, Sedgewick, Sleator, & Tarjan, 1986), both of which achieve constant-time complexity for inserting and updating a node on the A* Open list. The price for these gains comes in the form of expensive remove operations. Though log-time in principle these operations are associated with large "hidden" multiplier costs. These additional costs can often slow performance in practice (Larkin, Sen, & Tarjan, 2014), and for this reason implementers often turn to alternative sorting techniques. For example, at the 2014 GPPC several entrants maintain the Open list using a data structure known as a *bucket list* (Denardo & Fox, 1979; Cazenave, 2006). The idea is to reduce the sorting workload (vs. Binary Heaps) by hashing each f-value to a unique index in an array. With a bucket-based queue the time complexity for inserting and removing nodes from Open is constant. The price comes in the form of substantial memory overheads when searching with real-valued costs (there is one bucket per unique f-value) and a worst-case linear runtime for identifying the next bucket to drain. Bucket lists are a well known optimisation technique and appear in many other contexts besides; for example in domain-independent planning systems such as Fast Downward (Helmert, 2006).

3.1.2 Partial and Immediate Expansions

Another way to improve sorting performance involves reasoning about the f-values of successors before they are added to the list.

Immediate expansion (Sun et al., 2009) is a strategy which avoids adding to Open any successor whose f-value is the same as that of the current node. Because such nodes appear equally as promising they can be treated immediately. The only successors which remain, and which need to be sorted, are those that increase the f-value bound. This simple strategy costs very little and it can often improve performance.

Partial expansion is a related strategy where one tries to avoid generating *surplus nodes* whose f-value lower-bound is larger than the cost of reaching the target node. A variety of strategies have been proposed to mitigate this problem (Yoshizumi et al., 2000; Felner et al., 2012; Goldenberg et al., 2014). The broad strokes are similar: when expanding a node, only the first k successors are generated, with k selected according to some policy. The partially expanded node is then placed back on Open with a new f-value, equal to that of its $k^{th}+1$ successor. Partial expansion can substantially improve the performance of search. The price, in this case, comes in the form of duplicated work if the f-value of the target node is larger than the f-value of the $k^{th}+1$ successor; i.e. nodes may be expanded multiple times, each time with a larger f-value. In Section 7 we consider partial expansions in the context of the algorithm JPS+BB+.

3.1.3 Maintaining Closed and Other Optimisations

Rabin and Sturtevant (2013) discussed a variety of implementation-focused enhancements which can improve the performance of A* in practice. Among the suggestions are memory pre-allocation and direct indexing, both of which are often overlooked and which together can lead to substantially faster implementations of Closed vs. conventional hashing containers. Other notable but often overlooked optimisation techniques include avoiding initialisation overheads when commencing a search and tie-breaking toward larger g-values.

Many of the strategies in this section appear in our own algorithmic implementations and those of the methods against which we compare.

3.2 Better Heuristics

Another way to improve the performance of A* is to improve the accuracy of the heuristic function h. This function computes, for given node n and target node t, an admissible lower-bound on the cost-to-go, from n to t. The standard heuristic for 8-connected grid-optimal search is called Octile Distance. This heuristic ignores all obstacles on the grid and returns a cost incurred by the minimal number of straight and diagonal moves, from n to t.

Differential Heuristics (Sturtevant et al., 2009) are a family of well known estimation techniques that can improve on the Octile Distance baseline. The most popular variant, ALT, first proposed by Goldberg and Harrelson (2005). The idea here is to carefully select a small set of *landmark nodes* L. For every node $l \in L$ one precomputes (using single-source Dijkstra search) a table of exact distances: from l to every other in the graph and also vice versa. Given a set of such landmark nodes it becomes possible to compute an admissible estimate between any pair of grid cells n and m:

$$h(n,m) = \arg\max_{l \in L} \{|d(n,l) - d(l,m)|\}$$

ALT has small preprocessing costs and in many cases it can substantially improve on Octile Distance. It is sometimes described as a *one-dimensional estimator* because each table stores exact distances but always with respect to only a single source node l. ALT represents one extreme along a spectrum of heuristic functions, where further gains can be achieved by considering more than one dimension at a time – i.e. by embedding the nodes of the input grid (or graph) into a higher-dimensional space (Rayner et al., 2011; Cohen et al., 2018). The other end of this spectrum is represented by Hub Labels (Abraham et al., 2011) and CPD Heuristics (Bono et al., 2019): algorithms that compute and compress all-pairs optimal path data. These methods return exact distances between any pair of nodes but they also require substantial and sometimes prohibitive up-front investments, in memory as well as time.

Though heuristics are not part of our contribution in this work, all of the new search-based methods that we describe (e.g. JPS+BB+) can be further enhanced using improved heuristic estimates.

3.3 Symmetry Breaking

In grid-optimal search it is often the case that there exist many *symmetric* paths between the start and target node. Informally, two paths are symmetric if they have the same cost (i.e are *equivalent*) and if one can be derived from the other by simply changing the order of individual grid moves (i.e. up, down, left, right, etc.) as they appear along each path. Symmetries are undesirable because they slow pathfinding search: if there exist several equivalent paths to a given node n then A* can be forced to expand all nodes on all equivalent paths leading to n, even though exploring just one would suffice. Symmetry breaking is a central part of our contribution in this paper and we discuss it in more detail in Section 4.1.

Two popular families of pathfinding algorithms exist which can detect and break grid symmetries: Jump Point Search (JPS) (Harabor & Grastien, 2011, 2014; Sturtevant & Rabin, 2016) and Subgoal Graphs (SGs) (Uras et al., 2013; Uras & Koenig, 2014, 2018). The idea in both cases is similar: identify a set of points through which optimal paths must pass and then connect these nodes to one another via a single representative path. JPS typically constructs its search space on the fly. For this reason the algorithm is typically implemented as a forward A* search which constantly refers to the input grid: to identify successors and to detect the target node. By comparison, SGs break symmetries during preprocessing. The result is a graph that is independent from the grid and which can be searched with any number of different strategies; not just forward A*.

Experimental results show that while JPS can be more effective at breaking symmetries (Harabor & Grastien, 2014), SGs offer more flexibility: they can be more easily integrated with in-principle orthogonal speedup techniques (Uras & Koenig, 2018) and they have been applied to a broader range of problems (Uras & Koenig, 2017). Recent work offers theoretical insights into the connection between these two methods and unifies them into a single generalised framework (Harabor et al., 2019).

3.4 Goal Bounding

Goal Bounding is the name of a broad family of pruning techniques that dynamically reduce the branching factor of runtime search. The general idea is simple to understand: for every node n of the input graph one stores a precomputed label (or set of labels) that tells which of the outgoing edges of n can appear on an optimal path to the target node t.

Constructing a database of such labels requires an All Pairs Shortest Path (APSP) precomputation. A standard approach is to run a complete Dijkstra search for every node in the graph. Though overall complexity is quadratic in the size of the input graph, each Dijkstra search is independent from all the rest. Thus the total running time can be improved by adding more CPUs. Upon termination each Dijkstra search produces a so-called *first move table* where $\mathbf{fm}[s,t] = \{\ldots\}$ tells the set of outgoing edges that appear on an optimal path, from the source node s to any given target node t. Using first-move data one may extract optimal paths between any pair of s and t with just a simple looping procedure:

1. Select any edge $(s,u) \in \mathbf{fm}[s,t]$
2. Let $s = u$
3. Repeat steps 1 and 2 until $s = t$

The main challenge with this approach is that storing first-move APSP data directly is space prohibitive. For this reason a variety techniques have been developed to compress APSP data such that it can fit into working memory. We categorise these compression techniques as being either *lossy* or *lossless*. Both types guarantee to never return a false negative result (i.e. they never prune an optimal outgoing edge) but lossy schemes may return false positives (i.e. they may fail to prune some non-optimal outgoing edges).

3.4.1 Lossy First-Move Compression

Geometric Containers (Wagner, Willhalm, & Zaroliagis, 2005) is a popular approach for compressing first-move data using spatial data structures. The idea is to create a bounding container of minimal size (e.g. a 2d rectangle) which includes all target nodes that can be optimally reached by a given outgoing edge. For every source node s one such container is created and stored with every outgoing edge.

Arc Flags (Köhler, Möhring, & Schilling, 2009) is another popular technique for lossy first-move compression. Here the graph is initially partitioned into a set of K disjoint components, all before any Dijkstra search. Edge labels in this case take the form of bitfields, each of length $|K|$. When the k^{th} bit of a label is set to true it indicates the corresponding edge appears on an optimal path, from the source node s to at least one target node in the k^{th} partition. By adding more partitions during preprocessing, Arc Flags can provide increasingly stronger pruning during runtime search. The main disadvantage is that adding more partitions also increases the space needed to store each label. The result is a size vs. performance tradeoff which must be managed by the practitioner.

One of our contributions in this work is to reduce by orders of magnitude the preprocessing costs (in terms of space and time) of lossy goal bounding techniques. Though our ideas apply generally (i.e. to grid-based path planning with any APSP-based goal bounding

technique) we use as a demonstrative baseline JPS+BB: currently one the fastest known algorithms for grid-based pathfinding and one which combines Geometric Containers with Jump Point Search (Rabin & Sturtevant, 2016). We discuss this topic further in Section 5.

3.4.2 Lossless First-Move Compression

Compressed Path Databases (CPDs) (Botea, 2011; Botea & Harabor, 2013) are a family of techniques which reduce the size of first-move APSP data and which guarantee to identify only optimal first moves: from any source node s toward any target node t. When compared to other works from the literature, CPDs have two compelling advantages:

- They can extract optimal paths at ultra-fast speeds; i.e. without any state-space search.
- They can reduce the size of first-move tables by 2-3 orders of magnitude; i.e. they can be very space efficient in practice and easy to fit into working memory.

Our descriptions in the remainder of this section focus on SRC (Strasser et al., 2014, 2015), a leading CPD variant and overall fastest entry at the 2014 GPPC.

SRC stores first-move data in a matrix form, where each row is a source s and each column stores the set of optimal first moves, from s to any target t. SRC compresses the rows of this matrix using optimal Run Length Encoding (RLE). Despite its apparent simplicity experiments show that SRC can extract entire grid paths on the order of microseconds or less, often improving by substantial margins, on other leading and also precomputation intensive techniques for shortest path and shortest distance queries (Strasser et al., 2014). For other types of queries, such as first move and shortest prefix (i.e. find the first k steps of a shortest path) SRC performs faster still (Sturtevant et al., 2015).

Since the competition, a variety of works have sought to improve SRC. Some contributions focus on better compression (Salvetti et al., 2017), others on faster queries (Salvetti et al., 2018; Bono et al., 2019) and in some cases, authors try to achieve both (Chiari et al., 2019). A main consequence of this sustained research interest is that SRC performance, for compression and speed, has improved substantially beyond the 2014 baseline. Despite substantial progress one area that has not improved is CPD preprocessing time, which remains prohibitive for some applications including robotics and computer games. Our contributions in this paper directly address this gap by developing new CPD-based algorithms with orders of magnitude smaller precomputation times, with smaller storage costs and with comparable or improved performance than currently leading techniques.

3.5 Contraction

Contraction Hierarchies (CH) (Geisberger et al., 2008, 2012) is a family of leading algorithmic techniques for computing shortest paths. Considered as state-of-the-art for routing on road networks, this approach also appears undominated on the Pareto front of optimal methods at the 2014 GPPC.

CH-style algorithms can be understood as making use of a type of embedded graph abstraction. During a preprocessing step additional arcs, called "shortcuts", are inserted into the input graph to help speed up search. Suppose for example there exists a pair of

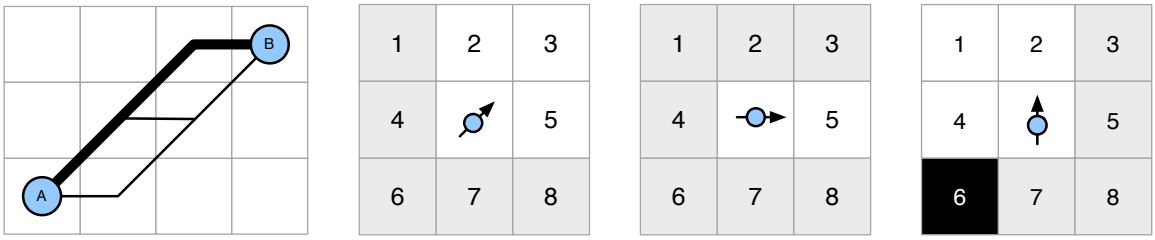


Figure 1: (a) Multiple paths connect A and B with the only difference being where diagonal moves appear. (b) JPS moves diagonally along the grid, from parent node 6. In such cases up to three successors can be considered diagonal-first (marked white) while all the rest (marked gray) are pruned. (c) JPS moves straight along the grid, from parent node 4. In such cases only one successor is considered diagonal-first. (d) Sometimes JPS is *forced* to generate successors such as 1 and 4, because the lexically smaller (i.e. diagonal-first) path from the parent, here node 7, is invalid due to obstacles, such as node 6.

graph edges (u, v) and (v, w) such that the path $\langle u, v, w\rangle$ is unique and optimal. In such situations it is possible to "contract" (equivalently, temporarily remove) the node v and add in its place one or more (depending on other pairs of neighbours of v) optimality-preserving shortcut edges, such as (u, w). The order in which nodes are contracted induces the eponymous hierarchy and strongly determines the number of shortcut edges that will be added. Meanwhile, during the online stage, the augmented graph is searched using a variant of Bi-directional Dijkstra search, where each direction only ever generates successors from higher up in the hierarchy. Since the competition, a number of subsequent works have sought to improve CH, for example by the addition of symmetry breaking (Uras & Koenig, 2018; Harabor et al., 2019) to improve grid performance and by the addition of goal bounding (Harabor & Stuckey, 2018) to improve performance on road networks.

In this work we undertake an extensive experimental comparison that includes a variety of leading grid-based techniques including CH and some subsequent variants.

4. Background

In this section we describe Jump Point Search (Harabor & Grastien, 2011) and two recent preprocessing-based speedup techniques: JPS+BB (Rabin & Sturtevant, 2016) and Topping (Salvetti et al., 2018). The descriptions are a necessary background since in subsequent sections we will show how to improve their preprocessing costs.

4.1 Jump Point Search

Jump Point Search (JPS) (Harabor & Grastien, 2011) is the combination of A* with an online pruning strategy that avoids redundant effort during search. The problem that JPS addresses is the following: in grid domains there often exists between a pair of nodes multiple equivalent paths which differ only with respect to the order in which individual grid moves appear along the path. Figure 1(a) shows an example. Informally, two such paths are *symmetric* if: (i) they share the same start and target and; (ii) they have the same cost

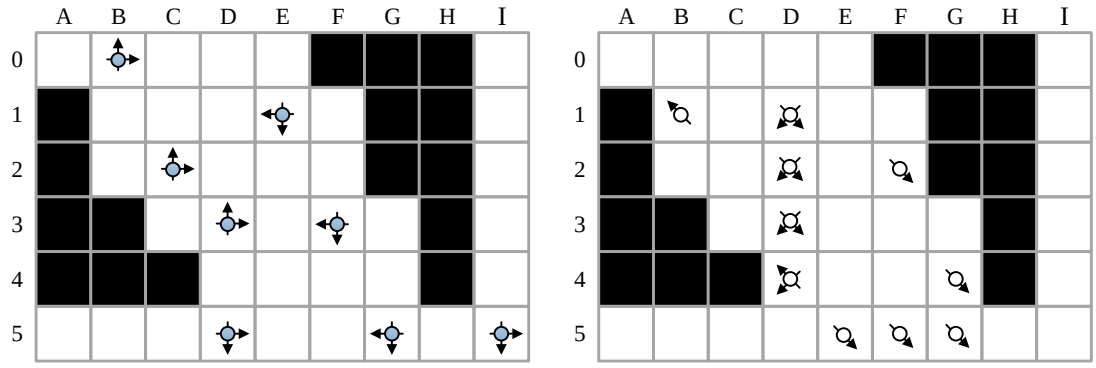


Figure 2: (a) A small grid with (x, y) locations of all straight jump points marked in solid circles and their travel directions as arrows. (b) Diagonal jump points are marked with hollow circles. Each one is produced by a straight jump point in the same row or column.

and; (iii) their constituent moves can be re-ordered to derive one path from the other. To break symmetries JPS considers only optimal paths which are *diagonal-first*:

Definition 1. A path π is *diagonal-first* iff it contains no straight-to-diagonal turning point[1] $\langle n_{k-1}, n_k, n_{k+1} \rangle$ which could be replaced by a diagonal-to-straight turning point $\langle n_{k-1}, n'_k, n_{k+1} \rangle$ to produce a new valid path.

To compute diagonal-first paths JPS applies during search a set of neighbour pruning rules, of the type shown in Figure 1(b) and 1(c). The rules discard any grid successor where the generated path is either locally suboptimal or where the path contains a diagonal move that could have appeared sooner. Maintaining the diagonal-first property helps to reduce the branching factor of each expanded node and also the branching factor of each candidate successor, often to just 0 or 1. Rather than adding such nodes to the Open list, JPS immediately processes them as part of a recursive *jumping* procedure. The only nodes which remain, and which the search must generate and expand, are those where diagonal-first paths can branch. Such nodes are called *jump points* and JPS distinguishes between two different types: straight and diagonal.

Definition 2. A straight jump point is a tuple $(n, \vec{c})$ comprising a grid cell n and a cardinal travel direction $\vec{c}$ such that, for another cardinal direction $\vec{c'} \perp \vec{c}$: **(i)** $n - \vec{c}$ is a valid move; **(ii)** $n + \vec{c'}$ is a valid move and; **(iii)** the path $(n - \vec{c}) + \vec{c'}$ is invalid.

Definition 3. A diagonal jump point is a tuple $(n, \vec{d})$ comprising a grid cell n and a diagonal travel direction $\vec{d} = \vec{c_1} + \vec{c_2}$ such that $n + k_1 \times \vec{c_1}$ or $n + k_2 \times \vec{c_2}$ $(k_1, k_2 \geq 0)$ is a sequence of valid moves that produces a straight jump point or the target.

Each jump point, whether straight or diagonal, has a distinct set of moves which are not pruned by the jumping rules. We refer to these as the *canonical branches* (equiv. *moves* or *directions*) of the jump point. Notice that each branch allows the current path to continue

1. For a more detailed discussion of turning points see (Harabor & Grastien, 2011).

in a way that maintains the diagonal-first property of the path. Figure 2(a) shows a small map with marked locations for all possible straight jump points and with arrows indicating associated travel directions. Meanwhile Figure 2(b) shows all possible diagonal jump points. Notice that every corner point is also a straight jump point. Meanwhile diagonal jump points can be located at any cell where a diagonal-first path can turn to reach a straight jump point. As indicated by these examples, each grid cell can be associated with multiple jump points, including both straight and diagonal. For example, at location $D3$ are found the jump points $(D3, E)$ and $(D3, N)$, which are both straight, and also the jump points $(D3, SE)$ and $(D3, SW)$, which are both diagonal. Note that, when the travel direction is unambiguous, we may for simplicity refer to a jump point using its grid location only.

The final ingredient of JPS is called *target detection*. The idea is to prevent the search from "jumping over" the target node as it scans the grid and moves from one jump point to the next. Target detection is a simple check which JPS performs during each recursive jumping operation. If the check succeeds the procedure returns the target node instead of any subsequent jump point successor. Harabor and Grastien (2011) have shown that for every optimal path to the target there exists an equivalent path of the same cost which is also diagonal-first. Moreover, this path can always be generated by the pruning, jumping and target detection rules. When combined with a best-first exploration of the grid environment the resulting algorithm (i.e. Jump Point Search) is known to be both complete and optimal.

JPS+ and JPS+(P): A principal drawback of JPS is that it scans individual rows and columns of the grid during each expansion operation. JPS+ (Harabor & Grastien, 2014) improves JPS by calculating and storing, for every grid cell n and every outgoing direction $\vec{v}$, the distance from n to the next jump point or the next obstacle in direction $\vec{v}$. Created during an offline preprocessing phase, this cached information improves search performance by up to one additional order of magnitude compared with JPS. Further speedups can be obtained using a strategy known as *intermediate pruning* (Harabor & Grastien, 2014). The resultant algorithm, JPS+(P), treats all diagonal jump points as "intermediate nodes" which can be immediately expanded, thereby allowing the search to make more progress in a single jumping operation. The price for this (effective in practice) optimisation is an increase in the branching factor of the currently expanding node: every intermediate may produce up to two additional successors, all of which need to be added to the A* Open list.

4.2 Faster Jump Point Search using Goal Bounding

Sometimes the grid environment in which agents operate can be assumed to be static. In these cases the performance of JPS can be substantially improved by pre-computing, during an offline step, all-pairs goal-bounding data. Recall from Section 3.4 that such data can be exploited, during a subsequent online phase, where it can substantially speed up pathfinding search. There currently exist two such ideas which exploit all-pairs data in different ways.

JPS+BB: First described by Rabin and Sturtevant (2016), this algorithm combines JPS+ (Harabor & Grastien, 2014) with Geometric Containers (Wagner et al., 2005) instantiated as Bounding Boxes.

During the offline step, JPS+BB stores an empty rectangular bounding box for every valid move (equiv. outgoing edge) in the grid map (equiv. input graph). Each box contains

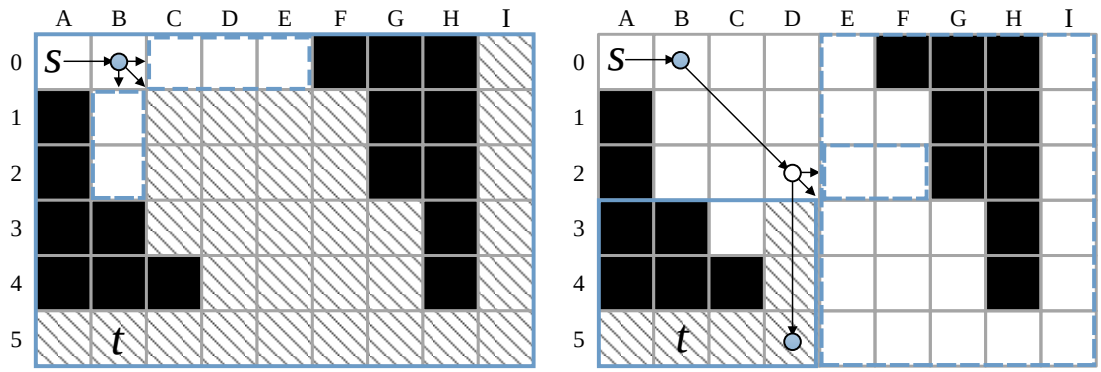


Figure 3: Given start node $A0$ and target $B5$, (a) $B0$ is expanded and the search branches in three canonical directions. We show the associated bounding boxes for each of these moves. The two dashed boxes do not contain the target and corresponding edges can be pruned. Notice that the bounding box for the remaining move, SE, contains false positives; e.g., $A0 - E0$ and $B1 - B2$. (b) At next expansion, JPS+BB checks the bounding boxes of $D2$, only generating the south successor since only that rectangle contains the target.

the set of target nodes that can be optimally reached by following the corresponding grid move. To compute the bounding boxes JPS+BB performs a single-source *canonical Dijkstra search* from every traversable node source node s in the grid; in turn or in parallel. This algorithm combines Dijkstra's well known technique with JPS-style pruning but without recursive jumping. Whenever the search expands a node n it generates all *canonical successors* of n (i.e., diagonal-first) and then it adds n to the bounding box of each optimal first move $\vec{m}$ (equiv. first edge e) associated with the source node s. In other words, from s toward n the first move $\vec{m}$ (equiv. edge e) is optimal.

During the online phase, JPS+BB exploits precomputed data as follows: when expanding a node the only successors generated are those reachable by a canonical move whose bounding box contains the target node. Figure 3 shows an example.

Two Oracle Path Planning (Topping): This algorithm combines strengths from two of the leading entries at GPPC 2014 (Sturtevant et al., 2015): JPS+ and SRC.

Like JPS+BB, Topping relies on pre-computed all-pairs data and its offline phase also involves a single-source canonical Dijkstra search from each grid node. Where the two methods differ is that Topping trades some additional memory in exchange for faster online performance. To compress first-move data, Topping uses Canonical Run-Length Encoding (C-RLE), a variation of the RLE compression scheme used by SRC (Strasser et al., 2015), which tie-breaks in favour moves that are diagonal-first. We discuss C-RLE in Section 5.3 and in Section 9 we analyse in its optimality-preserving characteristics with respect to tie-breaking.

Since C-RLE is a lossless compression scheme the size of the all-pairs data can be larger than what is required for storing bounding box labels. In addition to first-move data Topping also requires a *jump distance table* which records, for every grid cell and every valid move, the number of steps to the next jump point. Precomputation times are similar to

JPS+BB. For its online phase, Topping extracts shortest paths by repeatedly querying two distinct oracles: a first-move oracle and a distance oracle. The first-move oracle specifies which of the available grid moves heads optimally towards the target. The distance oracle then tells how many steps to take in the given direction. In other words, Topping moves from one jump point to the next along an optimal diagonal-first path and at each jump point on the path the first-move oracle reduces the branching factor to 1.

Topping requires no state-space search. Salvetti et al. (2018) reported that this algorithm can be up to one order of magnitude faster than SRC, a leading technique with performance comparable to JPS+BB (Rabin & Sturtevant, 2016). Topping achieves its outstanding performance based on the fact that each query to the distance oracle requires only constant time. For comparison, each query to the first-move oracle (i.e. SRC) requires a binary search through a compressed string of symbols.

4.3 Discussion

JPS+BB and Topping both offer state-of-the-art performance for optimal pathfinding on static grids. The main drawback for both algorithms is that time required for offline preprocessing grows quadratically with the size of the map. As we will see in Section 10, it can take hours of time and hundreds of megabytes of space to precompute on one machine all necessary goal bounding data for a single grid map from the computer game StarCraft. The only recourse currently available to practitioners is to introduce additional CPU resources so that the many necessary Dijkstra searches can be executed in parallel. Once the auxiliary data is computed, storage becomes an important secondary consideration. In the case of StarCraft, the game ships with dozens of different maps which means the storage costs of all the auxiliary data can be up to tens of gigabytes.

5. Reducing the Preprocessing Cost of Goal Bounding

Our contributions in this section stem from the following simple observations:

1. Any diagonal-first optimal path can be fully specified given only the start and target location and the sequence of turning points along the way. Once these locations are known the diagonal-first ordering is sufficient to specify the rest of the path: i.e. how to move from one jump point to the next. We therefore propose to precompute auxiliary goal-bounding data only for straight and diagonal jump points.

2. Many jump points cannot be generated during search, except as successors of specific starting nodes or as direct predecessors of specific target nodes. Therefore, we distinguish between such *dependent* jump points and other jump points which we call *independent*. Since the identity of the start and target node is a priori unknown, we propose to exclude from the precomputation step all dependent diagonal jump points.

Limiting our attention to the set of independent straight and diagonal jump points can reduce offline preprocessing times and storage costs by up to several orders of magnitude.

5.1 Independent Jump Points

We now consider how to identify the set of independent straight and diagonal jump points. First, recall Definition 2 which says that every corner point is also where some straight jump points lie. Since the location of corner points depends only on the topology of the map, and not on any specific start or target node, we can easily see that every straight jump point is also independent. The set of independent diagonal jump points can be similarly identified:

Definition 4. $(n, \vec{d})$ *is an* independent diagonal jump point *if there exists a straight jump point* $(n', \vec{c})$ *such that: (i) the move* $\vec{d} = \vec{c_1} + \vec{c_2}$ *is diagonal-first regarding the incoming move to* $(n', \vec{c})$*; (ii)* $n' + k \times \vec{d} = n$ *is a valid path and; (iii)* $n + m \times \vec{c_1}$ *or* $n + m' \times \vec{c_2}$ *is a valid path that produces a straight jump point.*

In simple words, Definition 4 says that every independent diagonal jump point is a turning point on a direct diagonal-first path between two straight jump points. For example, in Figure 3(b) the grid cell $D2$ is a turning point on the diagonal-first path from $(B0, E)$ to $(D5, S)$. Thus $(D2, SE)$ is an independent diagonal jump point. Notice that $D2$ can be further associated with another diagonal jump point $(D2, SW)$. However this jump point is dependent in the sense that it can only be reached if grid cell $E1$ is a start location.

In the preceding example grid cell $D2$ is associated with only one independent jump point. More generally however grid cells can be associated with multiple independent jump points, both straight and diagonal. The following result characterises the total number of independent jump points, in general and per corner point.

Theorem 1. *The number of independent jump points and the number of grid cells associated with any independent jump point are both worst-case linear in the number of traversable cells found in the input grid map.*

Proof. We prove this by inducing how many jump points can be associated with a single traversable cell. Consider a grid cell whose eight neighbours are all traversable and so it is associated with at most 4 diagonal jump points but no straight jump points because it is not diagonally adjacent to any obstacle. Now block one of its diagonally adjacent cells which invalidates a diagonal jump point attached to it but introduces two straight jump points. Continuing to add an obstacle to the opposite diagonal neighbour takes the same effect and the cell is now associated with two diagonal and four straight jump points. Adding more obstacles to its periphery will only reduce the number of associated jump points. Therefore, an individual grid cell can be a location where at most six independent points are located. Consider the case for all the traversable cells and we can conclude that the total number of independent jump points is worst-case linear in the total of traversable cells. The number of grid cells associated with any independent jump point is even smaller. □

We will show in Section 10 that on a variety of grids of practical interest, including many drawn from real games, the number of jump points is just a small percentage of the total number of grid cells. Moreover, because we only run a canonical Dijkstra search from each associated grid cell (vs. per jump point), the total number of searches required for offline preprocessing is smaller still.

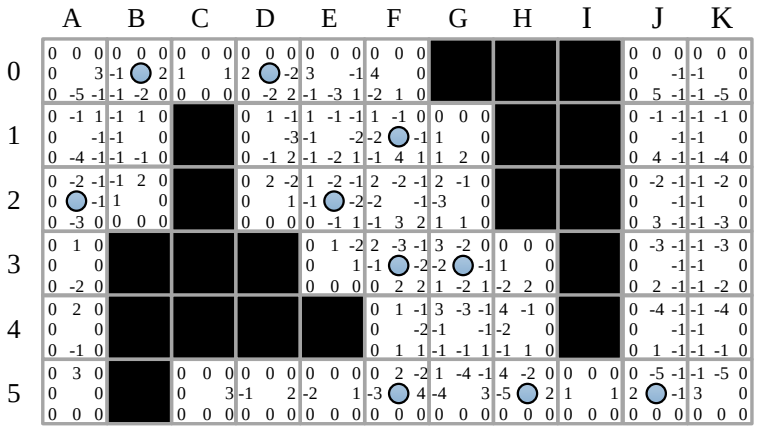


Figure 4: A jump distance table. Eight values are stored with every grid cell indicating the number of steps in each direction to a jump point (positive) or else to an obstacle (negative).

5.2 Computing Independent Jump Points

Algorithm 1 specifies a linear-time procedure for computing the set of independent diagonal jump points. The procedure depends on the set of straight jump points **S** and a table of *jump distances* **T**, both of which can themselves be constructed in linear time. Figure 4 shows an example of the jump distance table. Each table entry $\mathbf{T}[n][\vec{m}]$ indicates with a positive integer value the number of steps from grid cell n to a jump point (not necessarily independent) in direction $\vec{m}$. When no jump point exists the table stores instead a negative integer value which indicates the number of steps from n to an obstacle. There are eight labels per grid cell n, one for each of the eight grid directions.

With the set of straight jump points and a jump distance table in hand, Algorithm 1 proceeds as follows: From every straight jump point $(n, \vec{c})$ (Line 2) we scan the grid in each canonical diagonal direction $\vec{d}$ (Line 3). The jump distance table (Line 4) specifies how far to the next turning point n' where a diagonal-first path can bend to reach another straight jump point. Each turning point thus reached forms an independent diagonal jump point $(n', \vec{d})$ which we add to the set **D** (Line 9). The process continues until there are no more reachable turning points (Line 4). Notice that once a turning point is processed we mark it so as to avoid repeated and redundant work. The number of straight jump points is linear in the size of the map and the number of canonical diagonal continuations per straight jump point is at most 2. Since the diagonal scans never overlap we may reach each grid cell at most eight times (once in each direction) to recognise the independent jump points.

Jump distance tables can also be constructed in linear time and using a similar procedure to the one just outlined. Our implementation uses 2 bytes per distance label. We keep 15 bits for the distance value and one bit for the flag. This limits individual jumps to have at most $2^{15} = 32,768$ steps: a value that is more than 10x larger than any single dimension (i.e. width or height) of any grid map in our benchmark set. For larger grids we could use more bytes per label or compute values recursively. Note that jump distance tables appear as necessary ingredients in a number of prior works (Harabor & Grastien, 2012, 2014; Rabin

Algorithm 1: Identify independent diagonal jump points on a grid map.

Input: Grid map $\mathbf{M}$, the set of straight jump point $\mathbf{S}$ and a jump distance table $\mathbf{T}$

initialize: $\mathbf{D} \leftarrow \emptyset$;
foreach $(n, \vec{c}) \in \mathbf{S}$ **do**
  foreach *canonical diagonal move* $d = \vec{c_1} + \vec{c_2}$ **do**
    while $\mathbf{T}[n][\vec{d}] > 0$ **do**
      $n' \leftarrow n + \mathbf{T}[n][\vec{d}] \times \vec{d}$;
      if $(n', \vec{d})$ *is marked* **then**
        break;
      if $\mathbf{T}[n'][\vec{c_1}] > 0$ **or** $\mathbf{T}[n'][\vec{c_2}] > 0$ **then**
        $\mathbf{D} \leftarrow \mathbf{D} \cup (n', \vec{d})$;
      mark $(n', \vec{d})$;
      $n \leftarrow n'$;

& Sturtevant, 2016; Harabor et al., 2019). They are also closely related to clearance-value tables, which store distance-to-obstacle data (Uras et al., 2013).

5.3 Computing Edge Labels

During the offline phase we propose to compute and store goal bounding data for only a selected set of grid cells rather than for every traversable cell. In particular, we will run one single-source search from each grid cell associated with at least one independent straight or diagonal jump point. Our idea is to trade away some online pruning opportunities in exchange for a smaller database that can be computed offline much faster. The proposed approach is in contrast to currently leading algorithms such as JPS+BB and Topping, both of which compute goal-bounding data for every traversable grid cell but at the cost of a sometimes prohibitive all-pairs offline precompute.

Following other prior works (Sturtevant & Rabin, 2016; Salvetti et al., 2018) we will address this problem by applying during preprocessing an algorithm known as Canonical Dijkstra: a single-source variant of JPS where each recursive "jump" operation stops after the base step. Sturtevant and Rabin (2016) reported that this method can be several times faster than a standard single-source Dijkstra search. Several changes are required to derive this algorithm from Dijkstra's well known technique:

- **Canonical expansion:** when expanding a node, Canonical Dijkstra generates only successors that are diagonal-first from at least one parent. When expanding the source node, every valid move is considered canonical.

- **Multiple parents:** when a node can be reached along multiple canonical paths and with the same cost, Canonical Dijkstra stores each parent instead of tie-breaking arbitrarily among them.

- **First-move propagation:** when generating and relaxing a node Canonical Dijkstra propagates the identity of the first move (equiv. edge) on the path from the source

node. Since a node have multiple optimal and canonical parents, it can similarly be associated with multiple optimal first moves.

At the completion of the Canonical Dijkstra search the first-move data is compressed into a suitable form: e.g. it can be run-length encoded or it can be used to create set of bounding box labels for the source node.

5.4 Compressing First-Move Data

We consider two new techniques for reducing the size of first-move data: Canonical Bounding Boxes and Canonical Run-Length Encoding. There are two main differences between our compression approaches and similar prior works. First, we only store first-move data for source nodes which are associated with independent jump points. Second, if there exist multiple *equivalent first-moves* (i.e. there are several distinct paths from s to a target t, all having the same optimal cost) we tie-break among them during preprocessing using Definition 5. In general, our approach is to prefer canonical first-moves over non-canonical and to select arbitrarily amongst equivalent canonical first-moves. This strategy helps to save space since we store fewer moves. It can also improve online performance since the branching factor of the search is smaller. We present the tie-breaking strategy as a pairwise recursion, which applies one of several tie-breaking rules to an arbitrarily selected pair of optimal moves, and which continues until only one move remains.

Definition 5. Canonical tie-breaking. *Let* $\mathbf{fm}[s,t] = \{\ldots\}$ *be the set of all optimal and canonical moves from the grid cell* s *toward the target grid cell* t*. Also, let* P *be the set of independent jump points associated with* s*. For each pair of moves* $\vec{m_1}, \vec{m_2} \in \mathbf{fm}[s,t]$ *we tie-break as follows:*

1. *The moves* $\vec{m_1}$ *and* $\vec{m_2}$ *are not canonical continuations for any independent jump point in* P*. We are free to choose one move and discard the other.*

2. *The moves* $\vec{m_1}$ *and* $\vec{m_2}$ *are canonical continuations for the same independent jump point* $(s, \vec{u}) \in P$*. We are free to choose one move and discard the other.*

3. *The move* $\vec{m_1}$ *is a canonical continuation for the jump point* $(s, \vec{u}) \in P$ *while* $\vec{m_2}$ *is a canonical continuation for the jump point* $(s, \vec{v}) \in P$*. We are free to choose one move and discard the other.*

4. *The move* $\vec{m_1}$ *is a canonical continuation for the jump point* $(s, \vec{u}) \in P$ *while* $\vec{m_2}$ *is not canonical for any jump point in* P*. We choose move* $\vec{m_1}$ *and discard* $\vec{m_2}$*.*

In Theorem 4 we prove that offline canonical tie-breaking preserves both optimality and completeness of online pathfinding. We also settle a prior conjecture proposed by Rabin and Sturtevant (2016). In that work authors argued that any tie-breaking among equivalent first-moves can "in rare cases" (and therefore in general) lead to suboptimality and incompleteness for JPS+BB. We show that the conjecture is not true. In the remainder of this section we consider how to further compress first-move data, including with the help of canonical tie-breaking.

5.4.1 Canonical Bounding Boxes

We store for each source node s a set of bounding boxes. This approach is similar to JPS+BB (Rabin & Sturtevant, 2016) but the total storage requirements are smaller since we consider fewer source nodes. Another difference is tie-breaking. When multiple first-moves exist, from s toward a target t, we tie-break according to which move increases its corresponding bounding box the least (in a relative sense). Smaller bounding boxes have fewer false positives which can improve online performance. We explore this issue in Appendix A.

5.4.2 Canonical Run-Length Encoding

Run-Length Encoding (RLE) is a foundational technique for lossless compression. Given a string of symbols such as `SSSSSWWWWWEEEE` the idea is to represent the same data more compactly as a series of *runs*: `0S 5W 10E`. Each run tells two pieces of information: (i) the index in the original string where the run begins and; (ii) the symbol being encoded. Notice that the length of each run can be inferred by looking at the index of the next run.

We apply RLE using the same compression principles as suggested in (Strasser et al., 2015) and we refer the reader to that work for a more complete description. In brief: we consider the first-move data as a matrix $\mathbf{M}$ of size $A \times B$. The A columns correspond to the set of all grid cells. The B rows correspond to the set of grid locations with at least one independent jump point. Each row of the matrix can be computed and compressed independently of all the rest. The columns of the matrix are organised according to a "DFS ordering" scheme. Row compression proceeds greedily from left to right. Whenever an entry $\mathbf{M}[i, j]$ contains multiple optimal first moves we proceed as follows: if the target has a non-canonical first-move we discard that symbol and extend the run using the remaining canonical first moves. Whenever a target has two canonical first moves, we choose to store whichever move produces a longer run. We refer to this canonical-only compression scheme as C-RLE. Arguments for the completeness and optimality-preserving properties of this tie-breaking policy are the same as for Bounding Boxes and discussed in Section 9.

5.4.3 Run-Length Encoding for Jump Points

C-RLE stores one compressed row per grid cell. An alternative approach is to store one compressed row per independent jump point. This increases the number of rows but the number of runs can be smaller since, for any given jump point, many nodes cannot be optimally reached with any canonical first-move.

During the compression phase we assign to each target that cannot be optimally reached by a canonical first move a special "don't care" wildcard symbol. Wildcards are common in modern CPD implementations where they can be used to extend any run. These special symbols allow us to connect together two otherwise separate runs into a single long run. This approach strictly improves on the space performance of C-RLE but, as we will see in Section 10, it complicates path extraction and introduces a space-performance tradeoff. We will refer to this approach as C-RLE-JPW.

6. Pathfinding with Goal Bounding Data

With auxiliary data in hand we proceed to the online stage where we solve point to point shortest paths queries. We consider three distinct approaches:

- **JPS+BB+** is a new search-based algorithm that exploits Bounding Box labels. Closely related to JPS+BB (Rabin & Sturtevant, 2016) this method offers large improvements in offline preprocessing costs. We also introduce new partial expansion techniques that often improve online performance.

- **Two-Oracle Pathfinding Search (TOPS)** is a new search-based algorithm that exploits compressed first-move data. Each time the search expands a jump point the database reduces the branching factor to 1, providing substantial speedup. Further gains can be achieved by also pruning nodes with non-canonical first-moves.

- **Topping+** is a new extraction-based algorithm that identifies optimal paths using only compressed first-move data and no state-space search. Closely related to Topping (Salvetti et al., 2018) this algorithm substantially improves preprocessing costs while offering similarly excellent run-time performance.

Each new algorithm instantiates a common problem solving schema that proceeds as follows: (i) we scan the grid to identify the jump point successors of the starting node; (ii) we move from one jump point to the next, with the help of a precomputed jump distance table; (iii) during every jump we perform a target detection operation. In the remainder of this section we outline each of these common procedures. In Section 7 we consider in more detail the JPS+BB+ algorithm and in Section 8 we describe in detail TOPS and Topping+. In Section 9 we consider the theoretical properties of these new methods and in Section 10 we will show that, despite working with a much smaller auxiliary database, each of our new algorithms is competitive with the state-of-the-art.

6.1 Computing Start Successors

The starting node can be any cell from the grid and is usually not an independent jump point. Without the aid of goal bounding data to reduce the branching factor we proceed by considering as canonical every outgoing move. The successors of the starting node are then found by performing recursive grid scanning operations (i.e. *jumping*) in each of these directions. We give a full description of our start successor identification approach in Algorithm 2. To begin, we check if the target can be reached (Line 2-3) by following a path which is *diagonal-first free-space reachable* (or DFFR, for short). The following definition was proposed by Harabor et al. (2019):

Definition 6. *The* diagonal-first freespace-shortest path *from a cell n to n' is a freespace-shortest path where all diagonal moves appear before cardinal ones.*

Testing for a DFFR path is trivial using the jump distance table **T**. If such a path exists we generate the target as the only successor of the starting node and return. Notice that the search will terminate immediately thereafter, having expanded the target node.

Algorithm 2: Compute start successors. Blue text indicates differences from JPS+BB.

Input: start node s, target node t, goal bounding data $\mathbf{L}$, jump distance table $\mathbf{T}$
Output: the set of jump point successors for the start node

```
1  Function start_successors(s, t, L, T):
2      if t is diagonal-first freespace reachable from s then
3          return {t} ;
4      successors ← ∅;
5      foreach cardinal direction c⃗ do
6          if T[s][c⃗] > 0 then
7              find a straight jump point: n ← s + T[s][c⃗] × c⃗;
8              foreach m⃗ ∈ canonical_moves(n, c⃗) do
9                  if labelTesting(n, m⃗, t, L) then
10                     successors ← successors ∪ {n};
11     foreach diagonal direction d⃗ = c⃗1 + c⃗2 do
12         set current node: n ← s;
13         while T[n][d⃗] > 0 do
14             find a diagonal intermediate: n ← n + T[n][d⃗] × d⃗;
15             foreach c⃗ ∈ {c⃗1, c⃗2} do
16                 if T[n][c⃗] > 0 then
17                     find a straight jump point: n ← n + T[n][c⃗] × c⃗;
18                     foreach m⃗ ∈ canonical_moves(n, c⃗) do
19                         if labelTesting(n, m⃗, t, L) then
20                             successors ← successors ∪ {n};
21     return successors ;
```

When there exists no DFFR path to the target, we jump. For each of the four cardinal and diagonal directions (Lines 5, 11), the jump distance table $\mathbf{T}$ tells with a positive value the number of steps to the next jump point. In keeping with JPS+(P) (Harabor & Grastien, 2014), we treat each diagonal jump point as an intermediate node (Line 13). Intermediate nodes are expanded immediately which allows the recursion to continue. In place of each intermediate node we generate up to two independent straight jump points successors (Line 15). We discuss intermediate pruning in more detail in Section 7.1.

Before adding any candidate successor to Open list we apply the function `labelTesting` and check whether the candidate has any canonical continuation which can produce an optimal path to the target t. If the function returns true we say that the candidate is a successor of s and add it to the Open list. The recursion stops when no further diagonal jump points can be reached.

Example 1. *We show how to identify and prune jump point successors of the starting node. In Fig. 5(a) we exploit bounding box data and in Fig. 5(b) compressed first-move data. To begin, we recurse from s in all directions and find three jump points: $(H5, S)$, $(B0, W)$ and $(F5, S)$. Notice locations $(D0, NW)$ and $(H4, SE)$ are "intermediate" and expanded immediately during recursion. Using bounding boxes we prune $(B0, W)$ and $(F5, S)$ because these nodes have no canonical move whose associated box contains the target. Using compressed*

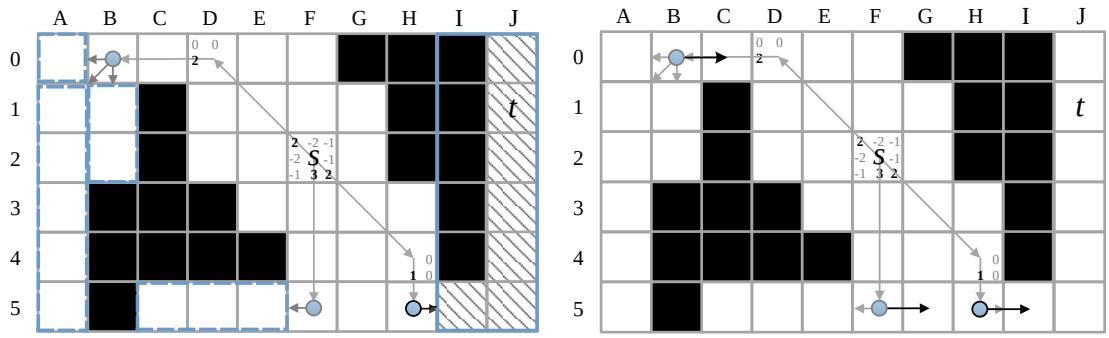


Figure 5: The starting node s has three diagonal-first freespace reachable jump points: $(B0, W)$, $(F5, S)$ and $(H5, S)$. We show the diagonal-first path to each jump point and we show precomputed jump distances. Short outgoing arrows at each jump point indicate canonical moves. (a) We show bounding boxes for each canonical move. Only $H5$ has a move, E, whose bounding box contains the target. (b) We show optimal first moves, from each grid cell associated with a jump point. Only $H5$ has an optimal-first move, E, which is also canonical.

first-move data we find that the optimal first-moves, from $B0$ and $F5$ toward t, are not canonical moves. We thus generate in both cases only a single successor: $(H5, S)$.

Our start successor approach is similar to both JPS+ (Harabor & Grastien, 2014) and JPS+BB (Rabin & Sturtevant, 2016). Compared to JPS+, we add aggressive pruning of the successor set (cf. generating all jump point successors). Compared to JPS+BB:

- We treat diagonal jump points as intermediate nodes and never add them to the OPEN list. This means each diagonal jump recurses further than JPS+BB and we can reach the target sooner. However, we may generate more start successors than JPS+BB since we usually lack auxiliary data for the starting node.
- When recursing diagonally, we only refer to auxiliary data at straight jump points. This is because diagonal jump point successors of the starting node are seldom independent jump points; i.e. we have no auxiliary data to exploit.
- Our `labelTesting` function checks, for each candidate start successor, whether any subsequent canonical move $\vec{m}$ can optimally reach the target (Lines 9, 19). This additional "lookahead" reduces the branching factor of s and is in contrast to JPS+BB which only checks auxiliary data for moves at s.

6.2 Moving Between Jump Points

With start successors in hand we next invoke an optimal pathfinding procedure to connect the start and target. This procedure can be search-based (this is the case for JPS+BB+ and TOPS) or it can be purely extraction-based (this is the case for Topping+). Either way, pathfinding advances by moving from one jump point to the next. At each jump point we exploit goal bounding data to prune the set of canonical moves and then jump anew in

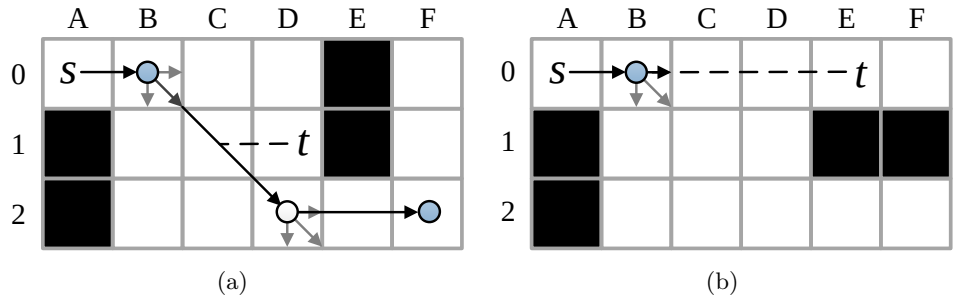


Figure 6: Two instances of target detection. Black arrows indicate branches along which some jump point successors are generated. Grey arrows indicate moves pruned by goal bounding. Dashed lines mark successful detection. (a) When expanding $(B0, E)$ we jump in the canonical direction SE. During this jump the target is detected as being DFFR via $C1$ and we therefore generate it as a successor. (b) Target detection for cardinal jumps.

each remaining direction, thus generating new successors. The jumping procedure is made fast with the aid of the jump distance table. The identification of canonical moves is made fast by the aid of a function-pointer lookup table which eliminates from the implementation grid-reasoning conditionals (i.e. if-this-then-that statements). We use the same 2048-entry table as described in JPS+BB (Rabin & Sturtevant, 2016).

6.3 Detecting the Target

Like the starting node, the target can be any cell from the grid. However the search only moves from one jump point to the next. To ensure we always reach the target we apply a target detection procedure originally described by Harabor and Grastien (2014). The idea is to generate the target as a successor as soon as it becomes diagonal-first freespace reachable.

The target detection procedure is trivial and fast with the help of jump distance table. Taking Fig. 6(a) for example, the current jump from the $B0$ to $D2$ crosses the row, where the target t lies, at cell $C1$. Through a simple comparison, it is discovered that the distance between $C1$ and t is within the eastern jump distance of $C1$, the direction to t. This indicates that the target is immediately reachable by a diagonal-cardinal transition and could be generated as a successor. This same target detecting approach is also applicable to cardinal moves, as demonstrated in Fig. 6(b).

6.4 Discussion

Our general pathfinding approach is described in Algorithm 3. We use the generic term *frontier* (Line 2) to refer to the set of candidate jump points which have yet to be considered. In the case of JPS+BB+, the frontier takes the form of of a binary heap sorted by the usual A* f-cost function and using the standard octile heuristic for estimating the cost to go. We discuss this approach in Section 7. For Topping+, the frontier is a recursively generated stack. We discuss this approach in Section 8.2. TOPS meanwhile is a hybrid algorithm that proceeds in the manner of A* search (and therefore similar to JPS+BB+) but it relies

Algorithm 3: General framework for path queries.

```
   Input: start node s, target node t
   Output: path from s to t as a vector P
 1 initialize: P ← ∅;
 2 frontier ← start_successors(s, t, L, T);
 3 while frontier is not empty do
 4 |   next node from the frontier: n;
 5 |   if n == t then
 6 |   |   P' ← BacktrackPath(n);
 7 |   |   P ← current_best(P, P');
 8 |   |   if P cannot be improved upon then
 9 |   |   |   return P ;
10 |   |   else
11 |   |   |   goto line 3;
12 |   foreach optimal and canonical move m⃗ do
13 |   |   if target detection succeeds then
14 |   |   |   add to frontier: t;
15 |   |   else
16 |   |   |   add to the frontier the jump point successors of n in direction m⃗ ;
17 return P;
```

on a CPD for goal bounding, which means the number of optimal and canonical moves per expanded node is always 1 (similar to Topping+). We discuss this method in Section 8.1.

Notice (Lines 12, 13) that we constantly refer to the input grid during pathfinding search: to identify next moves and to detect the target. Recent work shows that such grid-based reasoning during search can be almost entirely eliminated by constructing a graph of jump points (Harabor et al., 2019). Such "Jump Point Graphs" require that we perform all symmetry breaking a priori. The main advantage is that other graph-based speedup techniques, such as Contraction Hierarchies (Geisberger et al., 2008), can be easily integrated for further performance improvement. In the experimental section we explore this issue in detail from the perspective of run-time performance and offline preprocessing. We also discuss challenges and opportunities for future works that consider to integrate together Symmetry Breaking, Goal Bounding and graph-based speedup techniques.

7. Pathfinding with JPS+BB+

JPS+BB+ is a novel goal-bounding algorithm based on JPS+BB (Rabin & Sturtevant, 2016). Both algorithms combine Jump Point Search with Bounding Boxes and both aim to reduce the branching factor during search. Compared to the original, JPS+BB+ offers substantial savings in preprocessing costs and faster online performance. Like the original work, JPS+BB+ instantiates the search schema from Section 6. It differs by adding two new ingredients: (i) a partial expansion strategy which can further reduce the branching factor and; (ii) an enhanced version of intermediate pruning, which can reduce the number of expansions required per search episode.

7.1 Analysis for Intermediate Pruning

Intermediate pruning is an online speedup technique (Harabor & Grastien, 2014). The idea is simple: when recursing in a diagonal direction the search immediately expands and never adds to OPEN list any diagonal jump points found along the way. Straight jump points, reachable from any such "intermediate diagonals", are instead generated as successors of the current node. We saw an application of this technique in Example 1.

Intermediate pruning allows diagonal recursions to continue further and to make more progress toward the target. The main drawback is a potentially large increase in the branching factor of the search. To better understand the advantages and disadvantages we make an analysis of the branching factor per jump point, with and without intermediate pruning.

Theorem 2. *Without intermediate pruning, the branching factor per independent jump point is not more than 5.*

Proof. First we derive the largest number of outgoing moves of a straight jump point that can canonically lead to independent-jump-point successors. Definition 2 describes a straight jump point as a tuple $(n, \vec{c})$, indicating that $\vec{c}$ is a canonical move. The move $n - \vec{c}$ cannot be canonical since the path is suboptimal. The two remaining cardinal directions $\vec{c_1} \perp \vec{c}$ and $\vec{c_2} \perp \vec{c}$ can both be canonical branches according to Definition 2, if we allow that $n - \vec{c} + \vec{c_1}$ and $n - \vec{c} + \vec{c_2}$ are both obstacles. Since $\vec{c_1}$ and $\vec{c_2}$ are canonical it follows that $\vec{d_1} = \vec{c} + \vec{c_1}$ and $\vec{d_2} = \vec{c} + \vec{c_2}$ are also canonical as per Definition 2. The two remaining diagonal directions are invalid and cannot be canonical. Moreover, adding or removing any obstacles adjacent to n can only decrease the number of canonical successors. In sum, the five branches $\vec{c}$, $\vec{c_1}$, $\vec{c_2}$, $\vec{c} + \vec{c_1}$ and $\vec{c} + \vec{c_2}$ are all possible to produce an independent-jump-point successor.

We now consider diagonal jump points. The tuple $(n, \vec{d})$ satisfies Definition 3 if there exists a straight jump point reachable by recursion in direction $\vec{c_1}$ or $\vec{c_2}$ with $\vec{d} = \vec{c_1} + \vec{c_2}$. Together with $\vec{d}$, this brings the branching factor to ≤ 3. Other straight or diagonal directions $\vec{m}$ cannot be canonical since any path $n + \vec{d} + \vec{m}$ is not diagonal-first. □

Theorem 3. *With intermediate pruning, the branching factor per independent jump point is not more than 4H-7, assuming the input grid has size $W \times H$ with $H \leq W$.*

Proof. By Theorem 2 there exist for each independent jump point at most 5 canonical branching directions: two diagonal and three cardinal. Each cardinal branch takes $k \geq 2$ steps and produces at most one successor. Each diagonal branch takes between 1 and $(H - 1)$ diagonal steps and after every step, until the last two steps, the horizontal and vertical moves can both produce one straight jump point successors. For the last two steps only the horizontal move can produce a straight jump point. The number of successors for n is therefore at most $3 + 2 + 2 \times 2(H - 3) = 4H - 7$. □

With up to $4H - 7$ successors per independent jump point we see that intermediate pruning can quickly explode the number of candidate paths in the search space. The worst-case is pathological however and on a variety of practical benchmarks intermediate pruning is known to speed JPS+ by up to several factors (Harabor & Grastien, 2014).

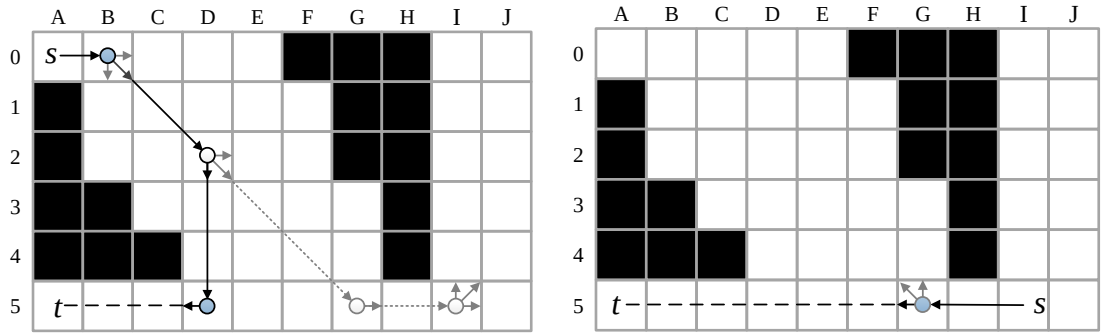


Figure 7: Two pathfinding problems with different starts but the same target. Canonical moves are indicated as short arrows. In grey we show moves (and potential recursions, dotted) pruned by goal bounding. In black we show moves (and recursions) that are not.

7.2 New Partial Expansion Strategies for Faster Intermediate Pruning

Intermediate pruning can be enhanced by exploiting goal bounding data. The idea is simple: during each diagonal recursion we check if the target is inside the bounding box of each considered canonical move. Diagonal recursions are now slower, because of the additional target tests, but if a move is not useful for reaching the target optimally the search never generates any corresponding jump point successor. In other words, we reduce the size of the Open list which allows the search to make progress towards the target overall faster.

Example 2. *Figure 7(a) shows a diagonal recursion from* $(B0, E)$*. At each diagonal step we propose to test with goal bounding data of every canonical move in turn. From the diagonal jump point* $(D2, SE)$ *only* S *can lead to the target so we generate a successor* $(D5, S)$*. Notice that this successor is DFFR from* $(B0, E)$*; i.e. there is no need to store any backpointer data to* $(D2, SE)$ *since the diagonal-first order is sufficient to specify the entire path. We are now free to continue jumping but the recursion stops because the next diagonal move is pruned. Without goal bounding, intermediate pruning continues scanning the grid to cell* $G5$ *and subsequently returns one additional candidate successor* $(I5, E)$*.*

Our strategy here can be understood as a type of partial expansion because we never have to add to Open or even evaluate a subset of possible successors. Another benefit is that diagonal recursions terminate sooner, avoiding redundant grid scanning operations. A main difference between our approach and other partial expansion techniques (Yoshizumi et al., 2000; Felner et al., 2012) is that we rely on constraints to avoid successor nodes rather than on the f-value bound. The two approaches are however complementary. Consider: when goal bounding data is available we can choose to exploit it and avoid generating surplus successors. When the goal bounding is ineffective (e.g. it might tell a move leads to the target when it does not, such as the DFS-based bounding box computation scheme proposed by Harabor & Stuckey, 2018) the successor might still be reduced by partial expansion with f-value bounding. Such topics suggest an interesting direction for future work.

Algorithm 4: (JPS+BB+ only) Compute successors with intermediate pruning.

Input: node n, incoming move $\vec{v}$, target t, bounding box data $\mathbf{L}$
Output: the canonical moves that are not pruned, from n to t

Function `compute_optimal_moves_jpspbbp`(n, $\vec{v}$, t, $\mathbf{L}$)**:**
  $\mathbf{B} \leftarrow \emptyset$;
  foreach $\vec{m} \in$ `canonical_moves`$(n, \vec{v})$ **do**
    if `labelTesting`$(n, \vec{m}, t, \mathbf{L})$ **then**
      $\mathbf{B} \leftarrow \mathbf{B} \cup \{\vec{m}\}$;
  return $\mathbf{B}$;

Input: node n, incoming move $\vec{v}$, target t, jump distances $\mathbf{T}$, bounding box data $\mathbf{L}$
Output: successors of n

Function `compute_successors_jpspbbp`(n, $\vec{v}$, t, $\mathbf{T}$, $\mathbf{L}$)**:**
  $\mathbf{B} \leftarrow$ `compute_optimal_moves_jpspbbp`$(n, \vec{v}, t, \mathbf{L})$;
  foreach cardinal move $\vec{c} \in \mathbf{B}$ **do**
    `jump_straight_jpspbbp`$(n, \vec{c}, t, \mathbf{T}, \mathbf{L})$;
  foreach diagonal move $\vec{d} \in \mathbf{B}$ **do**
    `jump_diagonal_jpspbbp`$(n, \vec{d}, t, \mathbf{T}, \mathbf{L})$;

Procedure `jump_straight_jpspbbp`(n, $\vec{c}, t, \mathbf{T}, \mathbf{L}$)**:**
  while $\mathbf{T}[n][\vec{c}] > 0$ **do**
    $n \leftarrow n + \mathbf{T}[n][\vec{c}] \times \vec{c}$;
    $\mathbf{B} \leftarrow$ `compute_optimal_moves_jpspbbp`$(n, \vec{c}, t, \mathbf{L})$;
    if $\vec{c} \notin \mathbf{B}$ **or** $\vec{m} \in \mathbf{B}$ such that $\vec{m}$ is diagonal or $\vec{m} \perp \vec{c}$ **then**
      return $(n, \vec{c})$
  return $\emptyset$

Procedure `jump_diagonal_jpspbbp`(n, $\vec{d}, t, \mathbf{T}, \mathbf{L}$)**:**
  $Succ \leftarrow \emptyset$;
  while $\mathbf{T}[n][\vec{d}] > 0$ **do**
    $n \leftarrow n + \mathbf{T}[n][\vec{d}] \times \vec{d}$;
    $\mathbf{B} \leftarrow$ `compute_optimal_moves_jpspbbp`$(n, \vec{d}, t, \mathbf{L})$;
    foreach *cardinal direction* $\vec{c} \in \mathbf{B}$ **do**
      $Succ \leftarrow Succ \cup$ `jump_straight_jpspbbp`$(n, \vec{c}, t, \mathbf{L})$;
    if $\vec{d} \notin \mathbf{B}$ **then**
      return $Succ$
  return $Succ$

7.3 New Intermediate Pruning for Straight Jump Points

Goal bounding data can also be used to extend the applicability of intermediate pruning: from diagonal jump points only to straight and diagonal. During straight jumps, if the only remaining move is dead ahead, the recursion can continue.

Example 3. *Consider Figure 7(b) and suppose we begin a recursive jumping operation from s in direction W. After two steps the recursion identifies the straight jump point $(G5, W)$. Notice the path now has several canonical continuations: W, N and NW. Here we propose to exploit goal bounding data and we will test whether direction-changing moves, such as N and NW, can possibly appear on an optimal path. If they cannot, the recursion continues. In this case we continue jumping and we avoid two otherwise unnecessary operations on the* Open *list (one push, one pop) due to $(G5, W)$.*

Algorithm 4 sketches our successor generation function with intermediate pruning and partial expansion. At Lines 3-5 we identify optimal and canonical moves for the currently expanding jump point $(n, \vec{v})$; i.e. those moves not pruned by goal bounding. We then jump in each direction looking for straight and diagonal successors (Lines 10, 12). During each jump we recursively prune candidate successors using goal bounding (Lines 16 and 24). We continue recursing until the main straight or diagonal move is no longer part of the set of optimal canonical moves (Lines 18 and 28 respectively) or until the jump table (Lines 14 and 22) indicates there exist no further jump points; whichever occurs sooner.

8. Pathfinding with Two Oracles

Two Oracle Path Planning is an idea first explored by Salvetti et al. (2018). Recall that in that work authors combine two distinct databases. The *first-move oracle* (implemented as a CPD) tells which of the available moves at a given grid cell leads optimally toward the target. The *distance oracle* meanwhile (implemented as a jump-distance table) tells how many steps until the next turning point on the optimal path. In combination these two oracles produce an extremely fast grid-based path planner that requires no state-space search but which depends on APSP preprocessing step to construct auxiliary data. In this section we generalise the two oracle schema into a family of related algorithms that require only partial first-move data.

8.1 TOPS

TOPS (short for Two Oracle Pathfinding Search) is an optimal search-based algorithm that instantiates the pathfinding schema described in Section 6. In other words, it scans the grid to identify the start successors; it uses an Open list to prioritise expansions and it relies on a standard octile-distance heuristic for estimating cost-to-go and; it advances the search frontier by moving from one jump point to the next. Although TOPS and JPS+BB+ appear similar there are some notable differences:

1. TOPS uses a CPD to reduce the branching factor of each jump point to exactly one. Any node whose first-move is not a canonical move can be pruned. This is presented in function `compute_optimal_moves_cpd` in Algorithm 5.

2. When the Open list is reduced to exactly one node TOPS simply extracts the rest of the path using the CPD.

3. TOPS moves from one jump point to the next using only the information in the jump distance table. It also applies the strategy of partial expansion like JPS+BB+ but

Algorithm 5: (TOPS, Topping+ only) Computing optimal moves and successors.

```
   Input: node n, incoming move v⃗, target t, CPD L
   Output: the optimal first move, from n toward t
1  Function compute_optimal_moves_cpd(n, v⃗, t, L):
2      m⃗ ← L[n][t] ;
3      if m⃗ ∉ canonical_moves(n, v⃗) then
4          return ∅;
5      return {m⃗};
   Input: node n, incoming move v⃗, target t, jump distance table T, CPD L
   Output: the next jump point, from n in direction m⃗
6  Function compute_successors_cpd(n, v⃗, t, T, L):
7      m⃗ ← compute_optimal_moves_cpd(n, v⃗, t, L);
8      foreach cardinal move c⃗ ∈ B do
9          jump_straight_cpd(n, c⃗, t, T, L)
10     foreach diagonal move d⃗ ∈ B do
11         jump_diagonal_cpd(n, d⃗, t, T, L)
```

prunes successors relying on CPD, so we will not spell out the related procedures in Algorithm 5.

In the experimental section we will consider two variants of this algorithm which are differentiated by the types of compression schemes used to encode the first-move data. When using C-RLE the set of start successors is usually small which leads to an overall faster search as there are fewer candidate paths to consider. When using C-RLE-JPW the size of the CPD is reduced, since we assign a valid first-move symbol to every potential target node, including those that cannot be optimally reached by any canonical first move. This does not affect the optimality of the search but it may increase the branching factor of the starting node.

8.2 Topping+

It is often the case that, after identifying and pruning the set of start successors, only a small number of candidates remain. Moreover, the number of jump points, from each successor to the target, can itself be small, relative to the number of individual grid steps. This motivates us to explore an alternative type of pathfinding approach which is purely based on CPD extraction. In other words, rather than exploring tentative using best-first search, we propose instead to exploit the partial CPD and to extract a series of complete paths, from each start successor to the target. Among all extracted paths we then select and return as the optimal solution the one having the overall smallest cost. Our work is similar to Topping (Salvetti et al., 2018), another extraction-based optimal path planner. The main difference is that we extract multiple paths instead of just one. Due to the similarity with the original we will refer to this new method as Topping+.

A pseudo-code description of Topping+ appears in Algorithm 6. We begin as usual by identifying the set of canonical jump point successors for the starting node (Line 4). We

Algorithm 6: Path extraction for Topping+.

Input: start node s, target node t, first-move database $\mathbf{L}$, jump distance table $\mathbf{T}$
Output: an optimal path from s to t, assuming one exists to begin with

```
1  curIter ← curIter + 1;
2  s.iter ← curIter, s.cost_to_t ←INF;
3  t.iter ← curIter, t.cost_to_t ← 0;
4  succSet ← start_successors(s, t, L, T) ;
5  foreach curSucc ∈ succSet do
6      curNode ← curSucc, v⃗ ←incoming move, preNode ← s;
7      while curNode.iter ≠ curIter do
8          curNode.iter ← curIter;
9          next ← compute_successors_cpd(curNode, v⃗, t, T, L);
10         if next is ∅ then
11             goto line 5 and process next start successor;
12         if target detection succeeds then
13             next ← t;
14         preNode ← curNode;
15         curNode ← next;
16         preNode.succ ← next;
17     while preNode ≠ s do
18         preNode.cost_to_t ← curNode.cost_to_t+ dist(curNode, preNode);
19         curNode ← preNode;
20         preNode ← preNode.parent;
21     tempCost ← curNode.cost_to_t+dist(curNode, s);
22     if tempCost < s.cost_to_t then
23         s.cost_to_t ← tempCost;
24         s.succ ← curNode;
25 return ForwardPath(s);
```

then extract a concrete path of jump points, from each start successor to the target (Lines 6-16). The different paths may coincide and continue together toward the target, as shown in Figure 8. To avoid extracting the same subpath multiple times we keep a bookkeeping label of iterations in case of linear-time initialisation for all nodes. The label is initialised lazily by way of a simple counter which we increment at the beginning of each new path planning episode (Line 1). It works by stopping the path extraction process when the next jump point is one which has been previously encountered during the the current episode (Line 7). Besides, every node keeps a heretofore known shortest cost to the target and these values of s and t are initialised to infinity and zero, respectively (Line 2-3). When an extraction operation successfully completes we update the cost cache backwards along the path (Lines 17-20) and we compare the overall cost of the newly discovered path against the best known cost, updating the incumbent as necessary (Lines 22-24). The best incumbent is eventually returned as the optimal path (Line 25).

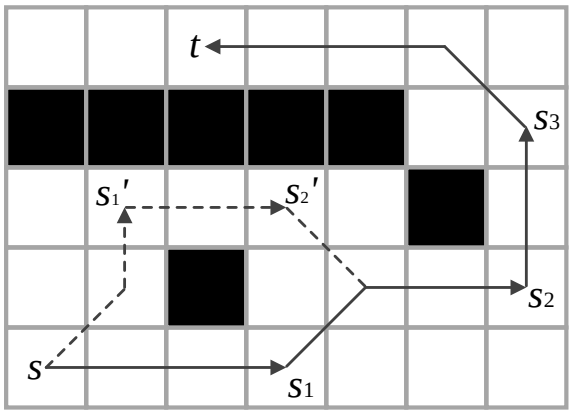


Figure 8: Example of Topping+. There are two start successors s_1 and s'_1, each of which may appear on an optimal path from start to the target. The two paths eventually meet at s_2 and share a common subpath thereafter. The algorithm will extract both paths and compare their costs, eventually returning $\langle s, s_1, \ldots, t \rangle$ as having the overall smallest cost.

8.3 Termination Conditions for TOPS and Topping+

TOPS and Topping+ both assume that a path exists from start to target. The case where no path exists can be handled separately as follows: During the preprocessing stage we identify (in linear time) the connected components of the input grid. At the cost of one additional label per grid node we can store the id of the component with each grid cell. A path from start to target can only exist if both cells belong to the same connected component. This check can be performed in constant time and prior to calling the main pathfinding procedure described in Algorithm 3.

Assuming a path exists, TOPS and Topping+ will both terminate after having considered a path toward the target via each start successor. Moreover, each individual path extraction is itself guaranteed to terminate, in one of three cases which are sufficient to cover every possible outcome:

- The path from the current node to the target has been previously extracted and is now recorded by the cache;
- The target is detected on the way to the next jump point;
- The next jump is invalid, which means that either the first-move is not canonical or there exists no reachable jump point in the specified direction. When first-move data is encoded with C-RLE compression such invalid jumps can occur only at start successors. When using C-RLE-JPW, invalid jumps can occur at any node.

Topping+ guarantees to return the optimal path. Associated proofs are given in Section 9. Similar to TOPS, we also consider a number of variants of the Topping+ algorithm. These are differentiated from one another by the encoding scheme used to compress first-move data. Each of the tradeoffs described in Section 8.1 applies similarly to Topping+. In Section 10 we explore these issues from an empirical perspective.

9. Theoretical Analysis

Each of our proposed algorithms (resp. JPS+BB+, TOPS and Topping+) is based on Jump Point Search (Harabor & Grastien, 2011). This algorithm (JPS) considers all optimal paths from start to target and reasons about equivalences amongst them on-the-fly. When two optimal paths have the same cost JPS breaks the tie by choosing canonical paths over non-canonical ones. This case was considered by Harabor and Grastien (2011) and shown to be optimality-preserving. When there exist multiple canonical paths, from start to target, JPS breaks the tie arbitrarily. This case is also optimality-preserving and was implied by Harabor and Grastien (2011). In this section we give a first explicit proof, shedding further light on this popular algorithm.

The main difference between JPS and our proposed methods is that we pre-compute auxiliary goal-bounding data. That means if there exist several first-moves from start to target, each being optimal, we choose amongst them offline, according to a canonical tie-breaking strategy which we outline in Definition 5. The offline decisions made by canonical tie-breaking need to agree with the pruning decisions made during online search, such that at least one optimal path to the target always remains in the online search space. The following result shows that this is indeed the case.

Theorem 4. *Canonical Tie-breaking among equivalent first-moves guarantees completeness and optimality when pathfinding with the diagonal-first invariant.*

Proof. Let **fm** be a first-move table, constructed offline, where we apply Canonical Tie-breaking (Definition 5) to the set of optimal first-moves of every entry $\mathbf{fm}[i, j]$. We will use the table to construct an optimal canonical path from any source cell s to any target cell t; i.e. $\pi = \langle s = c_0, \ldots, t = c_k \rangle$. For each node $c_i \in \pi$ we then analyse the set of possible canonical continuations, as computed by online JPS. We will show that no matter how JPS reaches c_i the path π remains in the JPS search space.

(1) We consider the base case $c_i = s$. Any of the canonical tie-breaking rules can apply here but eventually only one move $\vec{m} \in \mathbf{fm}[c_i, t]$ remains. Since there is no incoming/parent direction the online diagonal-first order does not prune any valid move at c_i. Following the DFFR continuation of each valid move at s, including $\vec{m}$, eventually produces $c_1 \in \pi$. Clearly then π remains in the search space.

(2) We have $c_i \neq s$ and there exist from c_i two optimal moves toward t: $\vec{m_1}$ and $\vec{m_2}$.

First, suppose both $\vec{m_1}$ and $\vec{m_2}$ are canonical continuations of an independent jump point $(c_i, \vec{u})$. Canonical tie-breaking rule 2 applies here and move $\vec{m_1}$ is chosen arbitrarily. Following its DFFR continuation eventually produces $c_{i+1} \in \pi$.

Next, suppose $\vec{m_1}$ is canonical for the independent jump point $(c_i, \vec{u})$ while $\vec{m_2}$ is not. Canonical tie-breaking rule 4 applies and chooses $\vec{m_1}$. Following its DFFR continuation eventually produces $c_{i+1} \in \pi$.

Finally suppose neither $\vec{m_1}$ or $\vec{m_2}$ are canonical continuations of any independent jump point at c_i. Canonical tie-breaking rule 1 applies here and chooses $\vec{m_1}$ arbitrarily. Notice however that both moves are pruned by the diagonal-first ordering. Here we derive a contradiction: π cannot be a canonical path since there exists no canonical

continuations from $c_i \in \pi$. In other words, c_i cannot appear on any JPS path from s to t.

(3) We have $c_i \neq s$ and there exist from c_i two equivalent optimal moves toward t: the move $\vec{m_1}$ being a canonical continuation of the independent jump point $(c_i, \vec{u})$ and the move $\vec{m_2}$ being a canonical continuation of $(c_i, \vec{v})$.

Now, it could be the case that the canonical continuations of the two jump points overlap and that both $\vec{m_1}$ and $\vec{m_2}$ are considered as canonical for either $(c_i, \vec{u})$ or $(c_i, \vec{v})$. This is the same as case (2). Alternatively, $\vec{m_1}$ is only canonical for $(c_i, \vec{u})$ and $\vec{m_2}$ is only canonical for $(c_i, \vec{v})$. Canonical tie-breaking rule 3 applies here and arbitrarily chooses $\vec{m_1}$. In this case we derive a contradiction to show that π is suboptimal or non-canonical.

π is suboptimal: the path from s to $(c_i, \vec{u})$ is locally suboptimal if the sum of costs for $\vec{u}$ and $\vec{m}$ is less than the optimal cost from c'_{i-1} to c_{i+1}. In this case $\vec{u}$ and $\vec{m}$ form at c_i a cardinal-to-cardinal or diagonal-to-diagonal turning point or a "switchback" cardinal-to-diagonal point, all of which are trivially suboptimal. But existence of this subpath contradicts the supposed optimality of π.

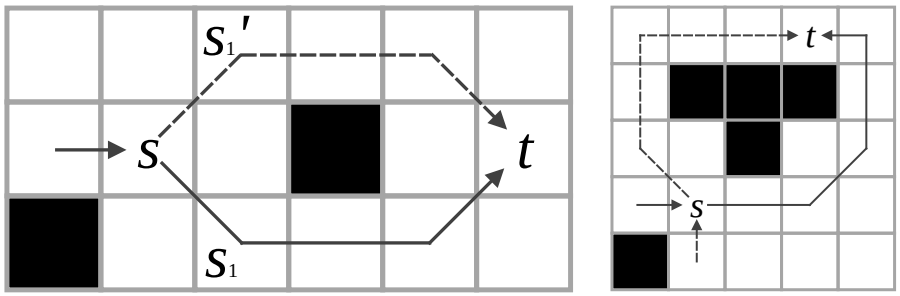
π is not canonical: the path π is not canonical if the moves $\vec{u}$ and $\vec{m}$ form at c_i an optimal cardinal-to-diagonal turning point. Similar to Case 2 we rewrite this path so the diagonal move appears sooner and run the argument anew. This new path is either canonical, and therefore covered by one of the three cases, or we rewrite it further.

□

Rabin and Sturtevant (2016) (p.5) suggested that offline tie-breaking can lead to suboptimality or incompleteness for online pathfinding "in rare cases". If this claim were generally true it would mean we must store all equivalent first moves so they can all be considered during the online phase. Theorem 4 shows the claim is not true in general. The following examples provide further insight. Example 4 in particular shows a potentially problematic case where tie-breaking arbitrarily (cf. canonical tie-breaking as per Definition 5) can introduce suboptimality. Such instances may correspond to the "rare cases" identified by Rabin and Sturtevant (2016) (no concrete examples were given).

Example 4. *In Figure 9(a) there exists two optimal moves from s toward t. These are SE and NE. Notice that move SE is canonical for the independent jump point (s, E) but the move NE is not canonical for any independent jump point at s. That means NE cannot appear on any optimal canonical path from s toward t except when s is the start node. In such cases we can safely discard the move NE entirely and without storing any corresponding bounding box. To handle queries where s is the start node and NE is an optimal canonical move we rely on an online insertion procedure described in Section 6.1.*

Example 5. *In Figure 9(b) there exists two optimal canonical moves from s toward t. The move NW is canonical for the independent jump point (s, N) while move E is canonical for the independent jump point (s, E). Now notice that for any s' adjacent to s the optimal path toward t does not pass through s. In other words in the example s cannot appear on an optimal canonical path toward t except when s is the start node. This is true for all s and*



(a) (b)

Figure 9: (a) The source node s is associated with two straight jump points, both of which have distinct diagonal-first paths to the target t. (b) Two equivalent paths connect s and t. However the dashed path is not diagonal-first from any jump point associated with s.

t pairs where such "disjoint" canonical moves exist (see case 3 in the proof of Theorem 4). Thus we are free to tie-break arbitrarily.

Theorem 5. *Given a pair of reachable node s and t, Algorithm 3 guarantees to return an optimal diagonal-first path.*

Proof. Let $\pi = \langle s = c_1, \ldots, c_k = t \rangle$ be an optimal path from s to t. We suppose π is canonical and has the diagonal-first property. If it does not, we can always reorder the individual moves of π to derive a new equivalent cost path, π', which is canonical. There may exist several such canonical paths (see Figure 10) and we will show that at least one optimal canonical path always remains in the search space of Algorithm 3. First, suppose s and t are diagonal-first freespace reachable (DFFR). This case is handled by Algorithm 2 (Lines 2-3) which generates a DFFR trajectory from s to t and tests it for feasibility. This path is exactly π.

Next suppose that s and t are not DFFR. This means the optimal path π must pass through one or more independent jump points on the way to t, since these points form a DFFR Shortest Path Cover (Harabor et al., 2019). We divide path π into three segments: first, middle and last. The first segment connects the start point s to c_i, the first independent jump point in π. The node c_i is guaranteed to be in the search space due to Algorithm 2. Our procedure considers all valid moves at s and exhaustively enumerates all DFFR continuations, eventually producing c_i as a successor. Now, we consider the middle segment. This is a possibly empty path from the first independent jump point $c_i \in \pi$ to the last independent jump point $c_j \in \pi$. Moving from one independent jump point c_i to the next independent jump point c_{i+1} requires the application of $k \times \vec{m}$ canonical moves with $k \geq 1$ and as indicated by the jump distance table. Algorithm 3 considers all canonical moves for each independent jump point. However the algorithm also prunes, with goal bounding data, those canonical moves which cannot optimally reach the target. Thus two kinds of moves remain at each independent jump point c_i: false positives and optimal moves. False positive moves cannot belong to any optimal s to t path and so $\vec{m}$ cannot be one of these. Therefore $\vec{m}$ must be among the remaining optimal moves. Although $\vec{m}$ is optimal it is still possible that $\vec{m}$ can be pruned, at preprocessing time due to canonical tie-breaking rules 2 and 3. However there must remain at c_i an equivalent optimal move $\vec{m'}$ which can be used

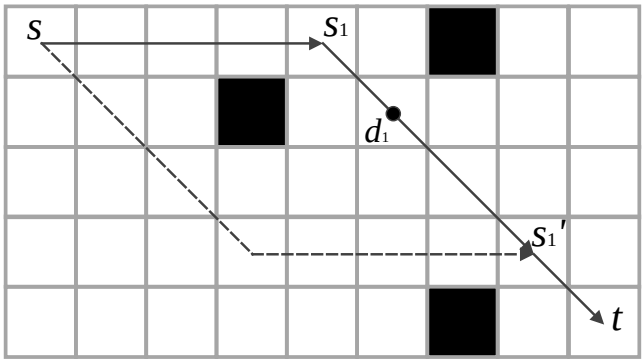


Figure 10: There exist two equivalent paths between s and t. Notice that both paths have the same cost and along each path diagonal moves appear as early as possible.

to reach the target along another equivalent canonical path π' which has the same cost as π (Theorem 4). Thus we are guaranteed that *some* optimal diagonal-first path to t remains in the search space. Finally we consider the last segment. This is a path that connects the last independent jump point $c_j \in \pi$ to the target point t. The move $\vec{m}$, optimal from c_j toward t, is again guaranteed to be among the optimal canonical moves remaining at c_j. Target detection applies here and guarantees to generate t as a successor of c_j in addition to any subsequently reachable jump point in the optimal direction $\vec{m}$. □

10. Empirical Evaluation

We evaluate our ideas in three distinct experiments: (i) partial goal bounding with Geometric Containers; (ii) partial goal bounding with Compressed Path Databases; (iii) performance comparisons vs. a variety of baseline algorithms on the benchmarks from Sturtevant's repository (Sturtevant, 2012) and full set of benchmark problems appearing at the 2014 Grid-based Path Planning Competition[2].

Implementation: Our implementations are based on codes made freely available; by the original authors of JPS+BB (Rabin & Sturtevant, 2016) (available from GitHub[3]) and by the original authors of SRC (Strasser et al., 2014) (available from the GPPC-14 organisers[1]). We also follow their suggested optimisations: (1) we use buckets indexed by cost, which speeds up sorting of candidates on the Open list to accelerate Dijkstra search. (2) we enhance the Open list, implemented as a binary heap, with a hash function, which speeds up membership tests and update operations. (3) we use a function-pointer lookup table, which eliminates some conditional instructions and speeds up the identification of canonical moves (Sturtevant et al., 2015).

2. https://movingai.com
3. https://github.com/SteveRabin/JPSPlusWithGoalBounding

Table 1: Information about benchmarks from Sturtevant's repository (Sturtevant, 2012) and from the GPPC-2014 competition. Note that the Rooms and Mazes have various sizes and corridor widths, respectively.

Benchmark	**Map Data**			**# Problem**
	# Maps	**Min size**	**Max size**	**Instances**
StarCraft	75	384 × 384	1024 × 1024	198230
Dragon Age: Origins (DAO)	156	30 × 21	1104 × 1260	159465
Baldur's Gate II (BG)	75	512 × 512	512 × 512	93160
Warcraft III	36	512 × 512	512 × 512	45101
Rooms	40	512 × 512	512 × 512	84350
Mazes	60	512 × 512	512 × 512	627000
GPPC-14 competition maps	132	19 × 35	1550 × 1550	347868

Baseline algorithms: We compare each of our new algorithms against a range of recent and state-of-the-art methods for grid-based path planning:

- **JPS+BB**, which combines JPS with Goal Bounding implemented as Bounding Boxes. First described by Rabin and Sturtevant (2016), this algorithm requires a full APSP precompute. We use C++ codes made available by the original authors.
- **Topping**, which combines JPS with Goal Bounding implemented as Compressed Path Databases. First described by Salvetti et al. (2018), this algorithm requires a full APSP precompute. We implement this algorithm ourselves in C++ but we use CPD code from the reference (Strasser et al., 2014), which was the fastest overall method at GPPC-2014.
- **CH-SG**, which combines Simple Subgoal Graphs with Contraction Hierarchies. First described by Uras and Koenig (2018), this algorithm performs offline symmetry breaking among equivalent paths which are all *freespace reachable*. We use C++ codes made available by the original authors[4].
- **CH-JP**, which combines Jump Point Graphs with Contraction Hierarchies. First described by Harabor et al. (2019), this algorithm performs offline symmetry breaking among equivalent paths which are all *diagonal-first freespace reachable*; We use C++ codes made available by the original authors[5].
- **SRC**, which uses CPD path extraction to identify shortest paths. First described by Strasser et al. (2014), this method requires a full APSP precompute. It was the overall fastest entry at the 2014 Grid-based Path Planning Competition (Sturtevant et al., 2015). We use codes from the original authors, as submitted to the competition.

Setup: The set of maps and instances used in our experiments are summarised in Table 1. All codes are compiled with GCC 4.8.5 and all experiments are run on Intel Xeon(R) CPU

4. `http://idm-lab.org/sg-ch`
5. `https://bitbucket.org/dharabor/pathfinding/other/jump_point_graphs`

Table 2: Preprocessing results for all maps per benchmark set, excluding those from GPPC-2014, including the number of Dijkstra calls (in millions), total preprocessing time (in hours) and total space required to store all the data (in MB).

	JPS+BB			**JPS+BB+ (new!)**		
	#Dijk (M)	Time (h)	Space (MB)	#Dijk (M)	Time (h)	Space (MB)
StarCraft	19.8	237.3	1207.5	2.4	25.3	148.3
DAO	3.3	3.5	203.0	0.5	0.6	30.6
BG	5.5	9.4	338.4	0.1	0.3	9.0
Warcraft III	4.0	11.1	247.2	0.2	0.6	12.4
Rooms	9.3	53.6	563.8	0.4	2.7	26.1
Maze	12.5	48.7	762.1	1.0	3.7	64.3

E5-2678 v3 @ 2.50GHz × 48 with 64 GB of RAM. We run each online experiment as a single thread in a single core when no threads of other users are running in the machine, in case that interruptions and shared cache create problems. In order to mitigate the impacts of CPU timing variations, we run each pathfinding instance for five times. As suggested by Sturtevant et al. (2015), the repetitions are not consecutive as the cache may store some data used by one query to unfairly accelerate data reading for the next sequential query. Note that for experiments in Section 10.1 and 10.2 all competitors return compact paths that specify only jump points. In Section 10.3 we follow competition rules from GPPC 2014 in the sense that all competitors need to explicitly refine every path.

10.1 Experiments with Geometric Containers

In this experiment we consider grid-based pathfinding with auxiliary goal bounding data implemented as bounding boxes. Our baseline algorithm is JPS+BB (Rabin & Sturtevant, 2016) which we compare with our new method JPS+BB+. Recall that JPS+BB requires an all-pairs precomputation step and stores one bounding box label per grid edge. The new method computes bounding box data only for independent jump points and for canonical edges. We consider the offline preprocessing costs of these two methods then we examine their online performance.

Offline Preprocessing: In Table 2 we report more than one order of magnitude improvement for preprocessing time and space vs. JPS+BB. This saving is attributed to the fact that there exist far fewer independent jump points than there are cells in the grid. This can be directly observed in column "Dijk" where we report the total number of calls to the Canonical Dijkstra algorithm. The total savings depend on the topology of the map and in some cases (benchmark BG) we approach two orders of magnitude improvement.

Online Pathfinding: In Table 3 we compare pathfinding performance across several distinct metrics: query time, heap operations and the size of the Open list. The table divides algorithms into two distinct families: those methods requiring an all-pairs precompute, and those which require only a partial precompute. The different variants are derived by adding on top of a reference implementation each of our suggested pruning techniques

Table 3: Online performance for pathfinding with bounding boxes. We consider a variety of algorithms, each produced by incrementally adding pruning techniques on top a common baseline. "+IP", "+PE' and "+SSP" are short for Intermediate Pruning, Partial Expansion and Start Successor Pruning. Column T indicates average time per instance (in microseconds), *#HO* measures average number of heap operations (nodes pushed, popped and relaxed from the Open list) while column $|O|$ measures the average size of Open at each search step. The best results under each metric are shown in bold.

	StarCraft			**DAO**			**BG**		
	T	#HO	$\|O\|$	μs	#HO	$\|O\|$	T	#HO	$\|O\|$
All-Pairs Precompute									
JPS+BB	23.2	257.9	**2.4**	10.9	138.2	**1.7**	3.0	36.2	**1.5**
+IP (new!)	48.3	268.0	26.5	18.1	137.7	9.2	3.8	27.4	5.3
+IP+PE (new!)	**11.0**	**80.1**	**2.4**	**4.8**	**53.2**	**1.7**	**1.8**	**15.0**	**1.5**
Partial Precompute									
JPS+BB+ (new!)	38.8	301.7	12.8	15.3	157.3	5.0	6.7	39.2	4.1
+IP	56.5	293.1	32.1	21.4	147.1	10.9	5.6	30.0	6.3
+IP+PE	19.7	113.5	13.2	8.0	65.0	4.9	3.5	20.0	4.1
+IP+PE+SSP	**14.2**	**85.8**	**2.9**	**6.4**	**55.4**	**2.1**	**2.7**	**15.9**	**1.7**

	Warcraft III			**Room**			**Maze**		
	T	#HO	$\|O\|$	μs	#HO	$\|O\|$	T	#HO	$\|O\|$
All-Pairs Precompute									
JPS+BB	5.2	54.5	**2.0**	54.3	413.0	5.6	83.0	1179.7	**1.4**
+IP (new!)	7.6	46.7	11.2	60.3	380.7	9.6	75.1	922.0	1.7
+IP+PE (new!)	**3.3**	**18.1**	2.1	**35.0**	**220.3**	**5.1**	**70.6**	**850.7**	**1.4**
Partial Precompute									
JPS+BB+ (new!)	9.2	63.7	9.0	57.8	433.0	6.3	83.9	1180.6	1.5
+IP	10.0	53.5	15.0	63.5	391.4	10.1	77.4	923.8	1.7
+IP+PE	5.8	29.4	9.0	37.6	239.4	5.7	75.4	902.2	1.5
+IP+PE+SSP	**4.2**	**19.7**	**2.7**	**37.4**	**237.9**	**5.3**	**73.2**	**899.8**	**1.4**

from Section 7: Intermediate Pruning (+IP), Partial Expansion (+PE) and our one-step lookahead that further prunes the set of candidate start successors (+SSP).

The headline comparison is between JPS+BB and JPS+BB+, denoted in the table as New!+IP+PE+SSP to indicate that it combines all our new approaches for online and offline enhancements. We report convincing gains with runtimes being reduced by anywhere from 10% (BG) to almost 40% (StarCraft). Observe also that +IP+PE together can further improve the baseline algorithm, JPS+BB. Here we see that the combination of full APSP data and aggressive online pruning achieves the strongest overall result with more than 50% runtime improvement in some cases (StarCraft, DAO). Consequently, if a user has the time and memory space to compute labels for all traversable cells, then one should prefer this approach. If not, JPS+BB+ shows similar performance with much smaller overhead costs. We observe in particular that both JPS+BB (+IP+PE) and JPS+BB+ approach the 1μs performance threshold for some benchmark domains (DAO, BG, WC3).

It is interesting to observe that adding only +IP to JPS+BB or to the New! baseline is strictly worse. This can appear surprising since Harabor and Grastien (2014) reported intermediate pruning yields positive results. We can understand the discrepancy as follows: intermediate pruning helps the search by allowing each recursive jumping procedure to

Table 4: Preprocessing costs for computing full and partial CPDs. Column *Dijk* indicates the total number of Canonical Dijkstra per benchmark (in millions), *Time* indicates the total preprocessing time (in hours) and *Space* indicates total storage costs (in MB).

	Full CPD (Topping)			**Partial CPD** (new!)			
	#Dijk (M)	Time (h)	Space (MB)	#Dijk (M)	Time (h)	Space (MB)	
						C-RLE	C-RLE-JPW
StarCraft	19.8	243.5	31199.4	2.4	26.9	2182.9	822.5
DAO	3.3	3.8	2038.2	0.5	0.6	176.8	85.3
BG	5.5	11.2	7685.3	0.1	0.3	76.6	32.8
Warcraft III	4.0	11.4	5308.6	0.2	0.6	164.5	63.0
Rooms	9.3	54.8	9125.5	0.4	2.7	366.2	101.7
Mazes	12.5	49.3	3829.6	1.0	3.9	48.8	41.3

make more progress toward the target. The price is an increase in the branching factor, which is usually offset by avoiding the explicit expansion of intermediate nodes. When goal bounding data is available however, the number of expansions are smaller and the search is guided quickly towards the target. Here the increase in branching factor is overall negative as many of the candidate successors cannot possibly lead to the target. The further addition of partial expansion stops recursions early and reduces the number of candidate successors.

10.2 Experiments with Compressed Pattern Databases

In this experiment we consider grid-based pathfinding with auxiliary goal bounding data implemented as Compressed Path Databases. Our baseline algorithm in this section is Topping (Salvetti et al., 2018), a two oracle extraction-based path planner which combines CPDs with Jump Point Search and which is among the fastest currently known methods for grid-based path planning. Recall that Topping requires an all-pairs precompute to create the CPD. We compare this algorithm with two new methods, TOPS and Topping+, both of which rely only partial first-move data that is computed only for independent jump points. We compare these three approaches in terms of preprocessing costs and then we examine their online performance.

Offline Preprocessing: In Table 4 we report the total preprocessing costs for each benchmark set excluding GPPC-2014 (we evaluate this separately in Section 10.3). We see that computing partial first-move data yields more than one order of magnitude reduction for total time and total number of calls to Canonical Dijkstra. These gains are similar to those reported in Section 10.1 but the times are slightly larger due to compression overheads. We also report up to *two orders of magnitude* improvement for total space.

For storing partial data we further compare two compression schemes: C-RLE and C-RLE-JPW. We described these approaches in Sections 5.4.2 and 5.4.3. Recall that C-RLE compresses together first-moves for all jump points associated with a particular grid cell while C-RLE-JPW compresses and stores separately first-move data for each associated jump point. We see that C-RLE-JPW improves space performance by up to several factors over C-RLE. Specifically, it requires less than one third of the space on room maps compared

Table 5: Compression results of variants based on CPD as measured across all maps. Column *#row* denotes average row numbers (in thousands) of all maps in each set, while *#run* indicates average run numbers (in millions). The best results under each metric are shown as bold and results on both metrics are better when they are smaller.

	StarCraft		**DAO**		**BG**		**Warcraft III**		**Room**		**Maze**	
	#row	#run	#row	#run	#row	#run	#row	#run	#row	#run	#row	#run
Topping	263.8	108.8	21.3	3.4	73.9	26.8	112.5	38.5	232.8	59.6	207.9	16.5
Partial CPD (new!)												
C-RLE	**32.4**	7.6	**3.2**	0.3	**2.0**	0.3	**5.7**	1.2	**10.1**	2.4	**17.2**	0.2
C-RLE-JPW	42.7	**2.8**	4.7	**0.1**	2.8	**0.1**	7.3	**0.5**	19.4	**0.6**	31.3	**0.1**

with C-RLE. These gains are due to wildcard (don't care) symbols, which this method assigns to any grid cell that cannot be optimally reached by any canonical first-move. Because wildcard symbols are compatible with any run, this method produces longer runs and requires fewer runs overall per row.

Table 5 gives further insight. Notice that C-RLE not only stores fewer rows per map on average than Topping, but also that the average number of runs per row is smaller. In other words, the first-move tables of independent jump points compress better than first-move tables for other source nodes. Because we only consider canonical moves at each source node (cf. any valid move at all), the total number of moves which can appear as an optimal first move is also smaller. With fewer symbols, compression can be more effective (assuming a good ordering of the columns). Regarding C-RLE-JPW: the total number of rows is higher than C-RLE, since we store one row per independent jump point. However the average runs per row is substantially smaller, thus resulting in an overall improvement.

Online Pathfinding: Table 6 shows detailed results on four representative metrics. To begin, we focus on the TOPS algorithm and report the influence on performance from different online pruning enhancements. Similar to the results with geometric containers, we see that effective start successor pruning (+SSP) (this time by eliminating candidates with non-canonical first-moves) considerably reduces the number of branches at the beginning of search, especially on game maps. Intermediate Pruning (+IP) and Partial Expansion (+PE) also help and together these three enhancements serve to reduce heap operations by up to several factors. When the Open list is reduced to just one node, we see that simply following the CPD to the target (+X) leads to further improvement. The combination of all these ingredients yield the complete TOPS algorithm denoted in the table as +IP+PE+SSP+X (C-RLE compression) and TOPS (+all) (C-RLE-JPW).

Our second algorithm, Topping+, also benefits from +SSP but instead of searching for a path in best-first fashion it simply follows the CPD, from each start successor to the target, similar to +X. We can see the extraction approach usually gives faster overall performance on partial CPDs compressed with C-RLE. To further reduce the database size we can apply C-RLE-JPW compression. Here the branching factor of the start node is increased and in this setting TOPS has an advantage. False positive branches can be pruned by bound, which is often cheaper than many full path extractions. Notice the results (for time) with C-RLE-JPW are very similar to C-RLE, showing that only small performance gains are sacrificed in exchange for substantial space savings.

Table 6: Online performance of variants based on both CPDs and jump distance table, as measured across all instances from different map sets. Column T indicates average time (in microseconds) for each path query, $\#S$ measures average number of successors generated from the start, $\#HO$ presents average sum of operations on the Open list, while $\#C$ counts the average number of CPD calls. The best results under each metric are shown as bold.

	StarCraft				**DAO**				**BG**			
	T	#S	#HO	#C	T	#S	#HO	#C	T	#S	#HO	#C
Topping (full CPD)	**12.0**	/	/	**87.1**	8.0	/	/	**57.3**	2.7	/	/	16.4
Partial CPD (C-RLE)												
TOPS Basic (new!)	33.5	20.1	214.8	102.5	13.9	8.4	132.8	64.4	5.3	4.7	35.0	**15.4**
+IP+PE	21.9	20.1	85.9	103.8	7.7	8.4	54.0	65.9	3.9	4.7	18.0	15.7
+IP+PE+SSP	17.0	**2.1**	57.4	115.2	6.2	**2.2**	43.9	72.1	3.2	**1.4**	13.5	19.6
+IP+PE+SSP+X	15.4	**2.1**	**24.6**	115.2	4.8	**2.2**	**12.5**	72.1	2.7	**1.4**	**6.2**	19.6
Topping+ (new!)	14.6	**2.1**	/	129.0	**3.8**	**2.2**	/	75.1	**2.5**	**1.4**	/	23.5
Partial CPD (C-RLE-JPW)												
TOPS (+all) (new!)	23.4	12.3	66.6	139.4	8.0	5.8	30.6	83.6	3.8	3.6	14.3	22.2
Topping+ (new!)	20.8	12.3	/	214.7	5.8	5.8	/	96.9	4.1	3.6	/	39.0
	Warcraft III				**Room**				**Maze**			
	T	#S	#HO	#C	T	#S	#HO	#C	T	#S	#HO	#C
Topping	**4.3**	/	/	**20.5**	11.5	/	/	**64.6**	**28.0**	/	/	497.5
Partial CPD (C-RLE)												
TOPS Basic (new!)	9.2	9.9	50.7	21.1	17.1	2.8	144.0	70.8	79.4	2.2	995.0	**496.5**
+IP+PE	6.7	9.9	23.8	21.6	15.8	2.8	78.1	71.3	70.8	2.2	686.5	**496.5**
+IP+PE+SSP	5.1	**1.8**	13.6	30.2	14.8	**1.5**	75.9	73.4	68.2	**1.0**	684.1	497.6
+IP+PE+SSP+X	5.1	**1.8**	**9.0**	30.2	12.2	**1.5**	**23.6**	73.4	32.5	**1.0**	**2.0**	497.6
Topping+ (new!)	5.2	**1.8**	/	40.5	**11.4**	**1.5**	/	73.9	32.9	**1.0**	/	497.6
Partial CPD (C-RLE-JPW)												
TOPS (+all) (new!)	6.7	7.3	21.0	34.8	15.1	2.8	68.4	80.1	39.3	1.6	10.0	500.0
Topping+ (new!)	8.6	7.3	/	82.4	**11.4**	2.8	/	92.4	40.9	1.6	/	500.0

In broad overview we see that each new approach, no matter the compression scheme, is at least competitive with the baseline Topping algorithm, which requires a full CPD. For example, TOPS and Topping+ combine with C-RLE to deliver average performance that is never more than 20% from the baseline. On two benchmarks (BG, Room) query times are similar to the baseline and in one benchmark (DAO) the new methods can even be faster, with average runtime reductions of 40-50%. For C-RLE-JPW compression we report an overall increase in query times but never more than a factor of two from the baseline and usually less. Recall the main advantage for these methods is reduced space, with database sizes being several factors smaller vs. C-RLE, as seen in Table 4.

10.3 Experiments on GPPC-2014

In this section we evaluate JPS+BB+, TOPS and Topping+ on the set of maps and instances appearing at the 2014 Grid-based Path Planning Competition (Sturtevant et al., 2015). This benchmark contains a diverse variety of challenging problems with 132 gridmaps and 347,868 instances in total. 105 grid maps are drawn from different games (11 sc1, 27 dao and 67 da2). The remaining 27 maps represent a variety of synthetic benchmarks

Table 7: Results of pre-computing databases and query times of the compared approaches on GPPC-14 maps. We report results for all problems appearing at the competition and for different subsets, with the number of maps and instances being indicated. Rows *Setup Time* and *Setup Space* indicate offline preprocessing costs of each algorithm in total (in hours and MBytes). We also report online query times (resp. average and median, in microseconds). The names of algorithms first appearing in this paper are in boldface so as more easily distinguished from those existing priors. Due to the space limit, we vertically split the results into this table and the following one. Best results per row across the two tables are marked in bold.

		JPS+BB	**JPS+BB+**	CH-SG-R	CH-JP
all 132 maps (347868)	Setup Time (h)	1591.7	198.4	**0.7**	2.1
	Setup Space (MB)	1742.3	287.8	**242.4**	784.9
	Query Time (avg)	292.5	230.3	**45.1**	53.3
	Query Time (med)	40.0	29.4	34.0	28.0
105 game maps (142534)	Setup Time (h)	30.8	3.9	0.5	< **0.1**
	Setup Space (MB)	309.3	**37.1**	102.1	352.1
	Query Time (avg)	15.3	11.0	30.6	19.3
	Query Time (med)	9.0	7.4	23.0	16.0
11 sc1 maps (29970)	Setup Time (h)	28.8	3.7	0.5	< **0.1**
	Setup Space (MB)	193.5	**23.8**	31.5	89.2
	Query Time (avg)	25.5	**18.6**	57.6	31.6
	Query Time (med)	18.4	**14.2**	53.0	31.0
27 dao maps (44414)	Setup Time (h)	1.4	0.1	0.02	< **0.1**
	Setup Space (MB)	50.7	**6.3**	28.4	103.4
	Query Time (avg)	19.9	13.3	31.3	21.2
	Query Time (med)	14.6	11.4	29.0	20.0
67 da2 maps (68150)	Setup Time (h)	0.6	0.1	< **0.1**	< **0.1**
	Setup Space (MB)	65.1	**7.0**	42.2	159.5
	Query Time (avg)	7.8	6.1	18.3	12.7
	Query Time (med)	6.2	5.2	17.0	13.0
9 random maps (32228)	Setup Time (h)	441.1	152.1	**0.1**	**0.1**
	Setup Space (MB)	364.1	215.0	**164.5**	481.1
	Query Time (avg)	2537.6	2062.1	**130.6**	270.8
	Query Time (med)	1492.5	1208.2	**120.0**	238.0
9 room maps (27130)	Setup Time (h)	766.0	39.8	**0.1**	2.0
	Setup Space (MB)	553.0	**29.7**	41.0	160.3
	Query Time (avg)	191.4	59.0	57.4	67.9
	Query Time (med)	86.4	40.2	44.0	57.0
9 maze maps (145976)	Setup Time (h)	353.8	2.6	< **0.1**	< **0.1**
	Setup Space (MB)	515.9	**6.0**	36.9	143.5
	Query Time (avg)	86.2	71.9	39.9	**35.8**
	Query Time (med)	82.2	67.6	37.0	**34.0**

(room, maze, random) and account for a majority of the total instances. Our baselines in this experiment are several recent and leading grid-based path planners from the literature including SRC, the overall fastest method at the competition. Recall that in this experiment we follow competition rules which requires every competitor to return a refined path (cf. jump-points-only).

Table 8: A continuation of Table 7.

				C-RLE		C-RLE-JPW	
		SRC	Topping	**TOPS**	**Topping+**	**TOPS**	**Topping+**
all 132 maps (347868)	Setup Time (h)	2527.6	1805.2	276.9	276.9	297.4	297.4
	Setup Space (MB)	26585.4	56968.7	7580.7	7580.7	2965.6	2965.6
	Query Time (avg)	105.5	48.4	46.8	46.9	48.3	46.3
	Query Time (med)	59.2	24.4	24.2	**22.8**	28.8	26.2
105 game maps (142534)	Setup Time (h)	47.7	34.7	3.8	3.8	4.1	4.1
	Setup Space (MB)	2462.5	5779.9	393.1	393.1	151.4	151.4
	Query Time (avg)	23.2	10.8	10.6	**9.3**	14.8	11.7
	Query Time (med)	13.8	7.0	7.0	**6.0**	9.4	7.2
11 sc1 maps (29970)	Setup Time (h)	44.7	33.2	3.6	3.6	3.9	3.9
	Setup Space (MB)	2085.3	4794.2	331.2	331.2	120.0	120.0
	Query Time (avg)	47.6	20.0	21.3	19.3	29.6	25.1
	Query Time (med)	41.8	18.0	17.6	16.2	24.2	21.8
27 dao maps (44414)	Setup Time (h)	2.0	1.1	0.1	0.1	0.1	0.1
	Setup Space (MB)	204.6	553.0	34.6	34.6	17.0	17.0
	Query Time (avg)	25.6	12.2	11.5	**9.9**	16.5	12.3
	Query Time (med)	19.8	9.4	10.0	**8.4**	13.8	10.6
67 da2 maps (68150)	Setup Time (h)	1.0	0.4	0.1	0.1	0.1	0.1
	Setup Space (MB)	172.6	432.7	27.3	27.3	14.4	14.4
	Query Time (avg)	10.8	5.9	5.3	**4.6**	7.2	5.4
	Query Time (med)	9.0	5.0	4.6	**4.2**	6.4	4.8
9 random maps (32228)	Setup Time (h)	920.8	672.5	232.7	232.7	251.5	251.5
	Setup Space (MB)	7072.7	8275.0	5925.0	5925.0	2420.3	2420.3
	Query Time (avg)	203.7	161.1	162.9	166.3	147.8	147.5
	Query Time (med)	171.2	138.2	136.6	141.6	126.6	125.0
9 room maps (27130)	Setup Time (h)	1018.5	750.7	37.9	37.9	39.1	39.1
	Setup Space (MB)	16052.7	40275.2	1638.5	1638.5	382.5	382.5
	Query Time (avg)	143.2	**31.1**	47.9	67.1	41.5	51.2
	Query Time (med)	131.0	**28.0**	39.4	52.5	36.4	43.4
9 maze maps (145976)	Setup Time (h)	540.6	347.3	2.5	2.5	2.7	2.7
	Setup Space (MB)	997.5	2638.6	17.2	17.2	11.4	11.4
	Query Time (avg)	157.2	63.3	57.7	53.3	60.3	57.6
	Query Time (med)	135.6	56.2	51.6	47.2	54.6	51.8

Table 8 and Table 7 give a summary. We group competitors as follows: those based on Bounding Boxes, those based on CPDs and other types of techniques. We see that the new methods (resp. JPS+BB+, TOPS and Topping+) compete well, having much cheaper preprocessing costs and often improving query times. Specifically, JPS+BB+ dominates JPS+BB on average, not just for setup costs but also runtime, with up to a 50% improvement (Rooms).

For partial CPDs compressed with C-RLE we report overall improvements vs. Topping, including for query times, except on the Rooms subset. We further observe that CPD-based methods generally have better online performances than their counterparts based on geometric containers (eg. Topping vs. JPS+BB and TOPS/Topping+ vs. JPS+BB+). This is especially pronounced for random maps where JPS+BB+ experiences many false positives. In these cases Topping+ query times can be more than ten times faster than

JPS+BB+, but this advantage comes at the cost of much larger precomputed data. When comparing TOPS (C-RLE) to Topping+ (C-RLE), we see that the latter can sometimes be slower, such as on room maps. Here Topping+ requires roughly two times the number of CPD calls to extract all candidate paths, which is not reported in the table. For partial CPDs compressed with C-RLE-JPW we again see reductions in total space requirements. However, though competitive in general, query performance on game maps lags behind other competitors. On artificial maps however, these methods have faster access to precomputed data and can be up to 10% faster, including vs. Topping (full CPD).

Among all the algorithms, CH-SG-R and CH-JP are the best at random and maze maps, respectively, but no more than two times better than Topping and Topping+. In general, our new partial goal bounding methods are competitive with and often substantially improve on the existing baselines. Further and more detailed results, for all other maps in our benchmark set, are considered in Appendix B. In Appendix A we analyse in detail how canonical tie-breaking improves overall space consumption and online performance (these improvements are all included in our main experiments).

11. Conclusion

Goal Bounding (Wagner et al., 2005) and Jump Point Search (Harabor & Grastien, 2011, 2014) are two successful approaches that independently speed up grid-based pathfinding. Previous work has attempted to combine their orthogonal advantages to develop even faster algorithms, but always at the price of an All Pairs precomputation step. In this work we re-examine Goal Bounding for Jump Point Search and we develop two new families of algorithms that have the same strong performance as prior methods but which have only a fraction of the total preprocessing costs. While prior methods compute goal bounding data by running a Dijkstra search from every grid cell, we create a partial goal bounding database by running a Dijkstra only from every cell associated with at least one independent jump point. Because the set of such cells is a much smaller set, we obtain savings of more than one order of magnitude for total time and up to two orders of magnitude for total space.

Besides considerable savings in preprocessing we also introduce new online pruning techniques that exploit goal bounding data more effectively, leading to further online performance improvements, despite a much smaller auxiliary database. We undertake the most comprehensive comparison of grid-based path planners since GPPC 2014 (the last competition in the area) and we show that our new methods convincingly improve on the state of the art, including in comparison to the overall fastest method appearing at the competition and on the same set of maps and instances.

One possible direction for further research is to consider the integration of other orthogonal speedup methods for grid-based pathfinding. There exists four broad families of optimality-preserving speedup techniques in this area: stronger heuristics, goal bounding, symmetry breaking and contractions. In this paper we explore symmetry breaking and goal bounding. Other recent works explore symmetry breaking and contraction (Uras & Koenig, 2018; Harabor et al., 2019), contraction and goal bounding (Harabor & Stuckey, 2018) and strong heuristics with goal bounding (Bono et al., 2019). Combinations such as symmetry breaking, contraction and goal bounding are as yet unexplored and appear promising for still further gains, possibly breaking the 1μs threshold for pathfinding on static grids. We

Table 9: Counts of canonical moves of grid cells associated with at least one independent jump point. We calculate the average of such a measurement in each measured map and present a five-number summary of all map sets. The bottom row shows the average numbers of non-canonical moves discarded from consideration.

		StarCraft	DAO	BG	Warcraft III	Room	Maze
Numbers of canonical moves	min	3.41	3.47	3.35	3.28	3.47	2.34
	Q1	3.47	3.66	3.52	3.40	3.48	3.83
	Q2	3.51	3.74	3.64	3.46	3.52	4.00
	Q3	3.55	3.87	3.79	3.48	3.57	4.03
	max	3.67	6.00	4.14	3.55	3.65	4.18
Mean discards		4.49	4.21	4.34	4.56	4.47	4.28

Table 10: Comparison of online performances when arbitrary tie-breaking among canonical moves is applied or not. The variants denoted by *non-a* and *arb* are those without and with the tie-breaking strategy, respectively. The definition of metrics *#HO* and $|O|$ used in this table are the same as Table 3. And bold numbers indicate better results.

Variants	StarCraft		DAO		BG		Warcraft III		Room		Maze	
	#HO	$\|O\|$	#HO	$\|O\|$	#HO	$\|O\|$	#HO	$\|O\|$	#HO	$\|O\|$	#HO	$\|O\|$
JPS+BB+(non-a)	88.8	3.1	59.0	2.2	16.0	**1.7**	20.1	2.8	246.7	5.4	902.4	**1.4**
JPS+BB+(arb)	**85.8**	**2.9**	**55.4**	**2.1**	**15.9**	**1.7**	**19.7**	**2.7**	**237.9**	**5.3**	**899.8**	**1.4**

regard this level of performance, on average and using conventional desktop machines, as a notable milestone and a currently open challenge in the area.

Acknowledgements

The work described in this paper was sponsored by the National Natural Science Foundation of China under Grant No. 61273300 and Natural Science Foundation of Hunan Province under Grant No. 2017JJ3371. Daniel Harabor is funded by the Australian Research Council under grants DE160100007, DP190100013, DP200100025 and also by an Amazon Research Award.

Appendix A. Effects by Canonical Tie-breaking

For each independent jump point, the proposed Canonical Tie-breaking throws any non-canonical moves out of all eight candidate directions. Table 9 gives a simple summary of the number of remaining moves which are consistent with the diagonal-first ordering for any independent jump point $(n, \vec{c})$ or $(n, \vec{d})$. Similar to Theorem 2, the numbers show roughly that we can retain fewer than a half of outgoing edges for grid cells associated with jump points, from which the Canonical Dijkstra floods during preprocessing. Non-canonical moves screened, JPS+BB+ has no need to keep modifying their bounding boxes for goal bounding while RLE can compress symbols of only canonical moves for diagonal-first ordering of paths.

Table 11: Comparison of run numbers (in thousands) when arbitrary tie-breaking among canonical moves is applied or not. The columns denoted by *non-a* and *arb* show results without and with the tie-breaking strategy, respectively. We calculate the number of compressed runs of each map and present a five-number summary across all map sets. The last row gives the ratios of reductions of mean run numbers. In the table, bold numbers indicate better results, namely fewer runs.

		StarCraft		**DAO**		**BG**		**Warcraft III**		**Room**		**Maze**	
		non-a	arb	non-a	arb	non-a	arb	non-a	arb	non-a	arb	non-a	arb
Numbers of runs	min	1116.9	**1107.4**	**0.0**	**0.0**	29.3	**28.9**	325.4	**316.4**	87.6	**81.2**	39.5	**35.9**
	Q1	3293.1	**3223.4**	17.3	**16.4**	119.0	**115.1**	653.6	**629.5**	412.4	**395.4**	75.7	**69.3**
	Q2	4933.7	**4805.4**	116.7	**112.7**	180.5	**174.0**	1005.0	**966.6**	1349.3	**1278.6**	164.7	**147.9**
	Q3	8237.5	**8016.9**	296.3	**281.2**	298.3	**287.7**	1528.9	**1462.7**	3532.4	**3298.6**	285.7	**249.6**
	max	45808.0	**41041.4**	5947.9	**5153.7**	1744.5	**1662.5**	3658.9	**3463.8**	7899.9	**6672.4**	535.8	**494.7**
Mean reductions(%)		5.0%		8.9%		4.0%		4.7%		8.0%		7.5%	

Moreover, our strategy also involves an arbitrary tie-breaking among moves that are all canonical for an individual independent jump point. For bounding boxes based approaches, we assign the target node to the box whose relative size increase is the least due to this addition. Smaller containers incur fewer false positives. The results in Table 10 demonstrate that with the strategy, the numbers of operations on Open list decrease across all map sets and the size of Open list becomes smaller. However, the small gap cannot bring about obvious performance improvements.

For first-move data based approaches, this arbitrary strategy allows longer extension of runs and therefore introduces fewer runs. Table 11 specifies a comparison of run numbers of the C-RLE compression approach. Despite its superiority of compression power, the strategy decreases numbers of runs only by relatively limited ratios all under 10%. This is because the tie-breaking principle applies only on a small number of grid cells, which are associated with individual independent jump points with different canonical first moves towards a target. Its effect on path-query performance on these variants is similar to that shown in Table 10, so we will not include the results.

In sum, Canonical Tie-breaking involves discarding non-canonical moves and arbitrary tie-breaking among canonical ones. It reduces computation costs during preprocessing, and meanwhile for path queries avoids violation of diagonal-first ordering and makes further improvements on performance.

Appendix B. Further Experimental Comparisons

This section presents a thorough comparison of all competing approaches on the set of non-competition maps and instances from Table 1. We report results for each of our three new methods (resp. JPS+BB+, TOPS and Topping+) and we compare against each of the five currently leading methods in Section 10. Table 12 (preprocessing) shows that JPS+BB+, and also our two approaches for creating partial CPDs (resp. C-RLE and C-RLE-JPW), are competitive with existing methods for preprocessing space, and orders of magnitude better than APSP-based path planners. One interesting observation is that Topping finishes its

Table 12: Preprocessing results as measured across all maps. Column *Dijk* indicates the total number of Dijkstra calls (in millions), *Time* indicates the total preprocessing time (in hours) and *Mem* indicates total space requirements for all bounding boxes, compressed pattern databases, or the graph/hierarchy and clearance values (in megabytes). Bold values under each metric denote the best results among these methods.

	StarCraft			**DAO**			**BG**		
	Dijk(M)	Time(h)	Mem(MB)	Dijk(M)	Time(h)	Mem(MB)	Dijk(M)	Time(h)	Mem(MB)
JPS+BB	19.8	237.3	1207.5	3.3	3.5	203.0	5.5	9.4	338.4
JPS+BB+	**2.4**	25.3	**148.3**	**0.5**	0.6	**30.6**	**0.1**	0.3	**9.0**
SRC	19.8	368.0	13134.0	3.3	6.3	840.7	5.5	28.0	3092.1
Topping	19.8	243.5	31199.4	3.3	3.8	2038.2	5.5	11.2	7685.3
C-RLE	**2.4**	20.5	2182.9	**0.5**	0.4	176.8	**0.1**	0.2	76.6
C-RLE-JPW	**2.4**	22.5	822.5	**0.5**	0.5	85.3	**0.1**	0.2	32.8
CH-SG-R	/	2.7	194.3	/	0.08	95.2	/	0.02	81.0
CH-JP	/	**0.3**	574.5	/	**0.03**	337.0	/	< 0.01	306.0
	Warcraft III			**Room**			**Maze**		
	Dijk(M)	Time(h)	Mem(MB)	Dijk(M)	Time(h)	Mem(MB)	Dijk(M)	Time(h)	Mem(MB)
JPS+BB	4.0	11.1	247.2	9.3	53.6	563.8	12.5	48.7	762.1
JPS+BB+	**0.2**	0.6	**12.4**	**0.4**	2.7	**26.1**	**1.0**	3.7	64.3
SRC	4.0	18.8	2257.9	9.3	91.1	6582.1	12.5	89.3	1408.1
Topping	4.0	11.4	5308.6	9.3	54.8	9125.5	12.5	49.3	3829.6
C-RLE	**0.2**	0.5	164.5	**0.4**	2.4	366.2	**1.0**	3.4	48.8
C-RLE-JPW	**0.2**	0.6	63.0	**0.4**	2.5	101.7	**1.0**	3.6	**41.3**
CH-SG-R	/	0.02	40.7	/	< 0.01	50.8	/	< 0.01	80.4
CH-JP	/	< 0.01	149.8	/	0.03	187.2	/	0	282.6

precomputation at around a half of time required by SRC. This is due to the Canonical Dijkstra algorithm which breaks grid symmetries and finishes sooner. We can see however that because SRC extends its runs without considering symmetry breaking it compresses data more effectively than Topping, often halving the total space requirements. Overall, JPS+BB+ and CH-SG-R take the cheapest preprocessing overheads in terms of the time and memory required, respectively.

Figure 11 illustrates the query times of the approaches across all the instances as box plots to see the distribution of online performances. Since SRC and CH-SG-R both return final solutions as cell-based paths (cf. jump point paths), the results of all approaches reported in the figure include the time used for path refinement. Besides, we use the algorithm Topping+ with C-RLE to compare with other methods. As demonstrated in Table 3, JPS+BB+ runs faster than JPS+BB and its IP variant, named as JPS+(P+BB), and also the values of its query times have relatively smaller variances. With offline APSP precomputaion and online combination of SRC and JPS oracles, Topping manifests itself as an ultra-fast competitor for most problems. However, Topping+ which saves substantial preprocessing time and space not only surpasses SRC in terms of average query time, but

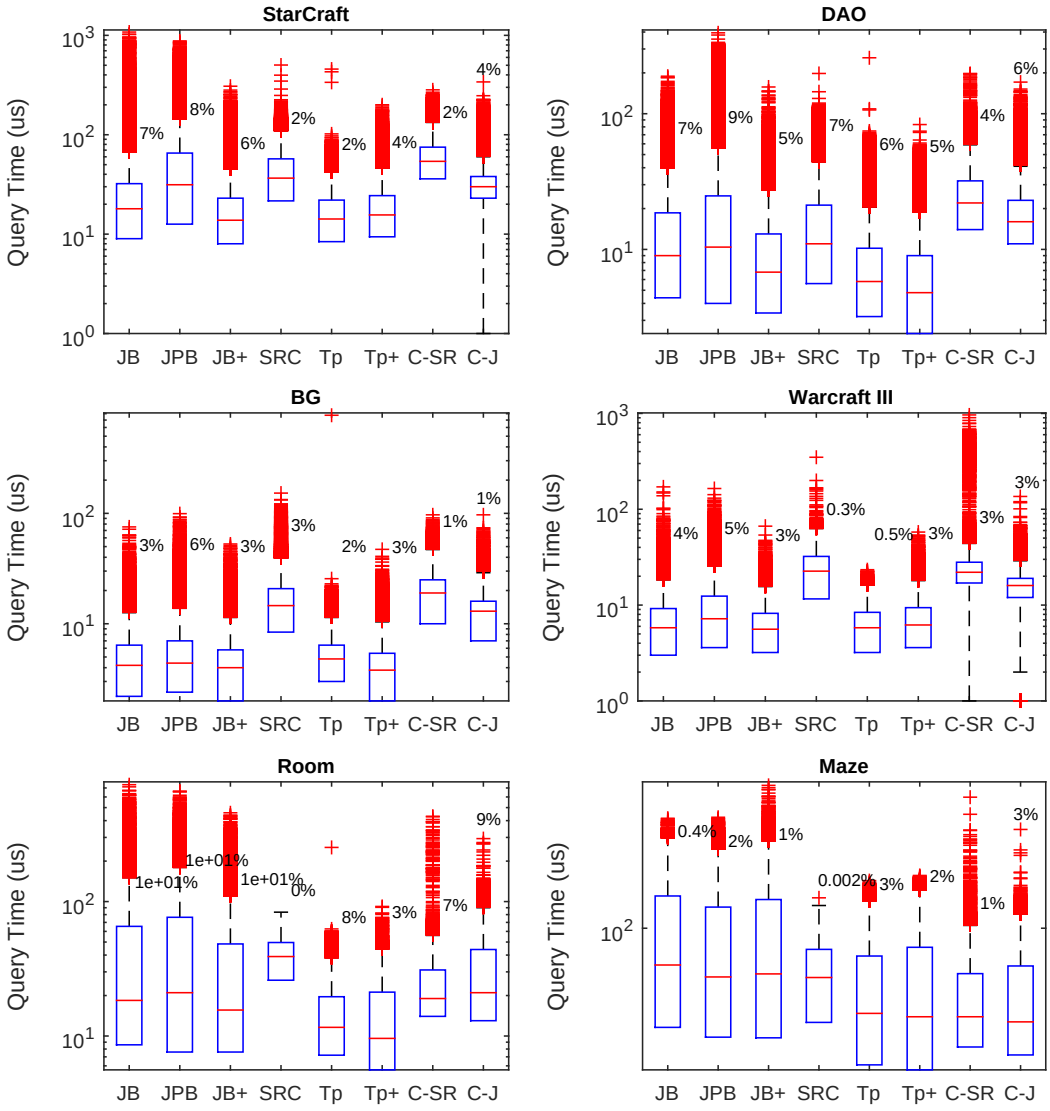


Figure 11: Box plots that show online performances of eight competing algorithms and their variances across all instances on each set of maps. The short labels below each plot represent algorithms of JPS+BB, JPS+(P+BB), JPS+BB+, SRC, Topping, Topping+ with C-RLE, CH-SG-R and CH-JP in order. The query time includes the total time used for connection, search and refinement, and the values on y-axis have been transformed into a logarithmic scale. Besides a box without a tail occurs when its minimal value is rounded to 0. We have marked the outliers larger than $Q3 + 1.5(Q3 - Q1)$ and labelled the percentage of their numbers in the total instances.

more remarkably has a very close performance to its original algorithm Topping and finds paths slightly faster on some map sets. While we cannot easily distinguish their performance gap on StarCraft, Warcraft III and maze maps, Topping+ has smaller quantile values than Topping on queries from other sets of maps. Regardless of precomputed information loss, it seemingly experiences faster memory accesses because a small database can already be in cache. Another approach developed in this paper, JPS+BB+, responds to path queries much faster than SRC on most maps and embraces similar speed to Topping. Also, the two algorithms we propose are much faster than CH-SG-R and CH-JP for both game maps and synthetic maps, except that JPS+BB+ answers queries on mazes a little slower than the others.

References

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2011). A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Symposium on Experimental and Efficient Algorithms*, pp. 230–241, Berlin, Heidelberg. Springer.

Bono, M., Gerevini, A. E., Harabor, D. D., & Stuckey, P. J. (2019). Path Planning with CPD Heuristics. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1199–1205. International Joint Conferences on Artificial Intelligence Organization.

Botea, A. (2011). Ultra-Fast Optimal Pathfinding without Runtime Search. In *the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 122–127, Palo Alto, California, USA.

Botea, A., & Harabor, D. (2013). Path Planning with Compressed All-Pairs Shortest Paths Data. In *the Twenty-Third International Conference on Automated Planning and Scheduling*, pp. 293–297, Rome, Italy.

Cazenave, T. (2006). Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *IEEE Symposium on Computational Intelligence and Games*, pp. 27–33.

Chiari, M., Zhao, S., Botea, A., Gerevini, A. E., Harabor, D., Saetti, A., Salvetti, M., & Stuckey, P. J. (2019). Cutting the size of compressed path databases with wildcards and redundant symbols. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pp. 106–113.

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., & Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press.

Cohen, L., Uras, T., Jahangiri, S., Arunasalam, A., Koenig, S., & Kumar, T. K. S. (2018). The fastmap algorithm for shortest path computations. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pp. 1427–1433.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms* (Second edition). MIT Press, Cambridge, MA.

Denardo, E. V., & Fox, B. L. (1979). Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, *27*(1), 161–186.

Felner, A., Goldenberg, M., Sharon, G., Stern, R., Beja, T., Sturtevant, N., Schaeffer, J., & Holte, R. C. (2012). Partial-Expansion A* with Selective Node Generation. In *the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pp. 471–477, Toronto, Ontario, Canada.

Fredman, M. L., Sedgewick, R., Sleator, D. D., & Tarjan, R. E. (1986). The pairing heap: A new form of self-adjusting heap. *Algorithmica*, *1*(1-4), 111–129.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, *34*(3), 596–615.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *International Workshop on Experimental and Efficient Algorithms*, pp. 319–333, Berlin, Heidelberg. Springer.

Geisberger, R., Sanders, P., Schultes, D., Vetter, C., Geisberger, R., Sanders, P., Schultes, D., & Vetter, C. (2012). Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, *46*(3), 388–404.

Goldberg, A. V., & Harrelson, C. (2005). Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 156–165.

Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N., Holte, R. C., & Schaeffer, J. (2014). Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research*, *50*, 141–187.

Harabor, D., & Grastien, A. (2014). Improving Jump Point Search. In *Twenty-Fourth International Conferenc on International Conference on Automated Planning and Scheduling*, pp. 128–135, Portsmouth, New Hampshire, USA.

Harabor, D. D., Uras, T., Stuckey, P. J., & Koenig, S. (2019). Regarding Jump Point Search and Subgoal Graphs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1241–1248, Macau, China.

Harabor, D. D., & Grastien, A. (2011). Online Graph Pruning for Pathfinding On Grid Maps. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 1114–1119, San Francisco, California, USA.

Harabor, D. D., & Grastien, A. (2012). The jps pathfinding system.. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*.

Harabor, D. D., & Stuckey, P. J. (2018). Forward Search in Contraction Hierarchies. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pp. 55–62.

Harabor, D. D., Uras, T., Stuckey, P. J., & Koenig, S. (2019). Regarding jump point search and subgoal graphs. In Kraus, S. (Ed.), *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 1241–1248. ijcai.org.

Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Hu, Y., Harabor, D., Qin, L., Yin, Q., & Hu, C. (2019). Improving the combination of JPS and geometric containers. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019.*, pp. 209–213.

Köhler, E., Möhring, R. H., & Schilling, H. (2009). Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *74*, 41–72.

Kring, A. W., Champandard, A. J., & Samarin, N. (2010). Dhpa* and shpa*: Efficient hierarchical pathfinding in dynamic and static game worlds. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010, October 11-13, 2010, Stanford, California, USA*.

Larkin, D. H., Sen, S., & Tarjan, R. E. (2014). A Back-to-basics Empirical Study of Priority Queues. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pp. 61–72, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Rabin, S., & Sturtevant, N. R. (2013). Pathfinding architecture optimization. *Game AI Pro: Collected Wisdom of Game AI Professionals*, *1*, 241–252.

Rabin, S., & Sturtevant, N. R. (2016). Combining Bounding Boxes and JPS to Prune Grid Pathfinding. In *AAAI Conference on Artifitial Intelligence*, pp. 746–752, Phoenix, Arizona, USA.

Rabin, S., & Sturtevant, N. R. (2017). *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, chap. Faster A* with Goal Bounding, pp. 281–288. CRC Press.

Rayner, D. C., Bowling, M. H., & Sturtevant, N. R. (2011). Euclidean heuristic optimization. In *AAAI*.

Salvetti, M., Botea, A., Gerevini, A. E., Harabor, D., & Saetti, A. (2018). Two-Oracle Optimal Path Planning on Grid Maps. In *International Conferenc on International Conference on Automated Planning and Scheduling*, pp. 227–231, Delft, The Netherlands.

Salvetti, M., Botea, A., Saetti, A., & Gerevini, A. E. (2017). Compressed Path Databases with Ordered Wildcard Substitutions. In *the Twenty-Seventh International Conference on Automated Planning and Scheduling*, pp. 250–258, Pittsburgh, Pennsylvania, USA.

Sartoretti, G., Kerr, J., Shi, Y., Wagner, G., Kumar, T. S., Koenig, S., & Choset, H. (2019). Primal: Pathfinding via Reinforcement And Imitation Multi-Agent Learning. *IEEE Robotics and Automation Letters*, *4*(3), 2378–2385.

Strasser, B., Botea, A., & Hararbor, D. (2015). Compressing Optimal Paths with Run Length Encoding. *Journal of Artifitial Intelligence Research*, *54*(1), 593–629.

Strasser, B., Harabor, D., & Botea, A. (2014). Fast First-Move Queries through Run-Length Encoding. In *the Seventh Annual Symposium on Combinatorial Search*, pp. 157–165, Prague, Czech Republic.

Sturtevant, N. (2012). Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144 – 148.

Sturtevant, N. R. (2007). Memory-efficient abstractions for pathfinding.. *AIIDE*, *684*, 31–36.

Sturtevant, N. R., Felner, A., Barrer, M., Schaeffer, J., & Burch, N. (2009). Memory-based Heuristics for Explicit State Spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 609–614.

Sturtevant, N. R., & Rabin, S. (2016). Canonical Orderings on Grids. In *International Joint Conference on Artifitial Intelligence*, pp. 683–689, New York, USA.

Sturtevant, N. R., Traish, J. M., Tulip, J., Uras, T., Koenig, S., Strasser, B., Botea, A., Harabor, D., & Rabin, S. (2015). The Grid-Based Path Planning Competition: 2014 Entries and Results. In *Annual Symposium on Combinatorial Search*, pp. 241–251, Ein Gedi, the Dead Sea, Israel.

Sun, X., Yeoh, W., Chen, P.-A., & Koenig, S. (2009). Simple Optimization Techniques For A*-based Search. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009, Budapest, Hungary, May 10-15, 2009, Volume 2*, pp. 931–936.

Uras, T., & Koenig, S. (2014). Identifying Hierarchies for Fast Optimal Search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pp. 878–884, Québec City, Québec, Canada.

Uras, T., & Koenig, S. (2017). Feasibility study: Subgoal graphs on state lattices. In *Tenth Annual Symposium on Combinatorial Search*, pp. 100–108.

Uras, T., & Koenig, S. (2018). Understanding Subgoal Graphs by Augmenting Contraction Hierarchies. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1506–1513, Stockholm, Sweden.

Uras, T., Koenig, S., & Hernández, C. (2013). Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids. In *International Conference on Automated Planning and Scheduling*, pp. 224–232, Rome, Italy.

Wagner, D., Willhalm, T., & Zaroliagis, C. (2005). Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, *10*(1), 1–30.

Wurman, P. R., D'Andrea, R., & Mountz, M. (2008). Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, *29*(1), 9–20.

Yoshizumi, T., Miura, T., & Ishida, T. (2000). A* with Partial Expansion for Large Branching Factor Problems. In *the Seventeenth National Conference on Artificial Intelligence*, pp. 923–929, Austin, Texas, USA.