Combining Bounding Boxes and JPS to Prune Grid Pathfinding

Steve Rabin
Dept. of Computer Science
DigiPen Institute of Technology
Redmond, WA, USA
steve.rabin@gmail.com

Nathan R. Sturtevant
Dept. of Computer Science
University of Denver
Denver, CO, USA
sturtevant@cs.du.edu

Abstract

Pathfinding is a common task across many domains and platforms, whether in games, robotics, or road maps. Given the breadth of domains, there are also a wide variety of representations used for pathfinding, and there are many techniques which have been shown to improve performance. In the last few years, the state-of-the-art in grid-based pathfinding has been significantly improved with domain-specific techniques such as Jump Point Search (JPS), Subgoal Graphs, and Compressed Path Databases. In this paper we look at a specific implementation of the general idea of *Geometric Containers*, showing that, while it is effective on grid maps, when combined with JPS+ it provides state-of-the-art performance.

Introduction and Background

Pathfinding is a fundamental task for agents that can move about a world, whether real or simulated. This need is common to characters in simulated worlds, such as games or training environments, or in the real world, such as in robots and automated vehicles. Furthermore, in real-time environments it is desirable that paths are returned as quickly as possible, so that the agent can begin moving. In simulations, returning paths quickly allows the simulation to run in real time; the faster the planning, the more agents that can plan simultaneously.

There are many common world representations used across these domains, including grids (Sturtevant 2012), triangulations (Demyen and Buro 2006), graphs (Geisberger *et al.* 2008), as well as other spatial representations such as navigation meshes (Tozour 2002) and quad trees (Finkel and Bentley 1974). The representation used for planning depends strongly on the problem being solved, as there are important trade-offs between accuracy, planning time, and other features involved in each world representation.

Given a representation, there are many possible optimizations that can be used to improve the performance of search. One of the most generic approaches is that of building better heuristics, such as true-distance heuristics (Sturtevant *et al.* 2009) (for polynomial domains) or pattern databases (Culberson and Schaeffer 1998) (for exponential domains). Heuristics label distances in the state space, and states which are too far from the optimal path are pruned.

There are many other ways of pruning the state space that depend on the structure of the space being searched. Contraction Hierarchies (Geisberger *et al.* 2008), for example, build an abstract graph that can be searched more efficiently at runtime while still yielding optimal paths.

In this paper we study another technique from the road network literature, geometric containers (Wagner *et al.* 2005). This approach stores information with edges in the graph that can be used to prune these edges at runtime, proving that an edge will never lead to an optimal solution.

We combine this technique with Jump Point Search (Harabor and Grastien 2014; 2011), an algorithm designed for grid-based pathfinding. JPS uses a canonical ordering on paths in the state space to reduce the number of duplicate paths. Its extension, JPS+, precomputes the set of 'jump points' that allow the search to quickly jump across the search space.

Related to JPS is Subgoal Graphs (Uras *et al.* 2013) which was also designed for grid graphs. Subgoal Graphs use similar structure in the map as JPS. Subgoal Graphs then use a form a contraction hierarchies to significantly improve performance (Uras and Koenig 2014).

We show that the combination of JPS+ and geometric containers achieves state-of-the-art performance in grid-based pathfinding, with performance on par with storing compressed all-pairs shortest path data. The performance is achieved with only a constant amount of data per edge; this is pre-computed and then used at runtime to prune the search.

In this paper we describe the bounding-box version of geometric containers, showing how they can be used to improve performance. More importantly, we show the influence of JPS's canonical ordering on the performance of this technique, and how we can combine them during the preprocessing phase to improve performance.

Problem Definition and Related Work

The problem being solved here can be described as search on a 4-tuple $(\mathcal{G}, s, g, h)$ where $\mathcal{G} = (V, E)$ is a finite directed graph composed of vertices and edges. We use the terms vertex, node, and state interchangeably in this paper. Each edge $e = (v_1, v_2) \in E$ has a positive cost $c(v_1, v_2)$. In the graph,

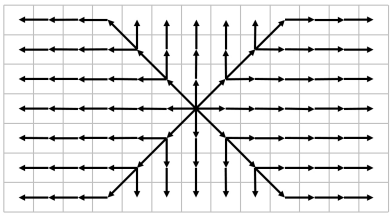
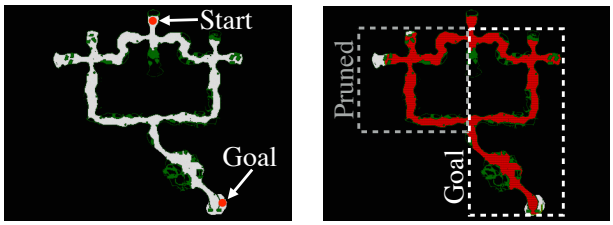


Figure 1: JPS search ordering



(a) (b)

Figure 2: Bounding box example.

$s \in V$ is the start state, $g \in V$ is the goal state, and h is an admissible heuristic function where $h(a,b) \leq c^*(a,b)$ if c^* is the optimal path cost between a and b. The goal is to find a path $P(v_0, v_n) = (v_0, v_1, v_2, \ldots, v_n)$ such that $\forall[i \in 0 \ldots n-1](v_i, v_{i+1}) \in E$, $v_0 = s$, and $v_n = g$. The cost of a path is $c(P) = \sum_{i=0}^{n-1} c(v_i, v_{i+1})$. We designate an optimal path as $P^*(v_0, v_n)$.

There are many metrics that can be used to evaluate the resulting path, including how fast the path is produced and how much precomputation is needed. There are other interesting metrics, such as path quality, but this paper is restricted to algorithms that return optimal paths.

This paper will look at the combination of two primary techniques, Jump Point Search (JPS) and geometric containers, which we explain here in more detail.

Jump Point Search. JPS (Harabor and Grastien 2011) encompasses a set of techniques designed to speed search on uniform cost grids. First, JPS defines a canonical ordering over all actions to minimize the number of transpositions during search. On an empty map JPS's canonical ordering will only generate a single path to each state in the map, as is illustrated in Figure 1(a). In this example, search begins at the middle node and only proceeds along the marked arrows. The canonical ordering generates diagonal moves before cardinal moves, and once cardinal moves are generated, diagonal moves will no longer be considered. Special rules, which we describe later, are needed when obstacles exist to make sure that at least one optimal path is always preserved. Second, instead of expanding one node at a time, it jumps along straight lines without generating the intermediate nodes, saving the cost of adding and removing states from the open list. JPS+ (Harabor and Grastien 2014) precomputes these jumps, storing the distance of the next jump with each edge in the graph.

Geometric Containers Geometric containers (Wagner *et al.* 2005) are a general class of approaches for use in problems that can be mapped to a metric space. In an offline phase, the goal locations reachable by a particular edge (and its neighboring node) in the graph are placed inside a geometric container (e.g. a bounding box). Then, at runtime, edges from nodes are only followed if the goal is in their associated geometric container. The approach can be added to many different search algorithms, but published results (Wagner *et al.* 2005) only use this with Dijkstra's algorithm (Dijkstra 1959) on road and rail networks.

On grid maps, the bounding box version of geometric containers[1] can be conceptualized by the question, "If I leave this grid square traveling downward, what is the furthest row and column I can get to optimally?" If you know that leaving a node in a particular direction can only optimally reach a bounded area defined by a min/max row and column, then if the goal lies outside of the bounded area, there is no reason to explore in that direction.

We illustrate this in Figure 2 with the 'lak308d' map from the Moving AI repository of maps (Sturtevant 2012). Suppose that the start state is at the top of the map, marked by a circle and an arrow in Figure 2(a), and that the goal is at the bottom of the map, similarly marked. Finding the shortest path to this problem using an A* search will require expanding nearly every state in the entire map; these states are shaded (red) in Figure 2(b).

In this example, the optimal path from the start to the goal goes around the right side of the loop. Thus, for an action that moves diagonally down and to the right from the start, the bounding box is shown in the white box labeled 'goal'. An action that moves diagonally down and to the left will have the goal bounds marked in gray (labeled 'pruned'). As the goal is not in this region, this and all similar actions will be pruned and none of these states will be expanded.

Other pruning techniques There are several other pruning techniques that could be applied similarly to the work in this paper:

Swamps are another pruning mechanism (Pochter *et al.* 2010) for general state spaces. A swamp is a group of states that should be considered a dead end and not expanded unless the goal is within that swamp. Formally speaking, a set of vertices is a swamp, S, if, for all vertices $v_i, v_j \notin S$, an optimal path $P^*(v_i, v_j)$ can be constructed that does not contain a state in S. Thus, ignoring the swamp will not affect the shortest path computation. Swamps are similar to geometric containers, as they use constraints to prune the state space. Generally speaking, geometric containers can contain more information than swamps, as they are associated with each node/edge pair in the state space.

Reach (Goldberg *et al.* 2006) is similar, in some ways, to pruning with geometric containers. The idea of reach is to associate a reach value with each node or edge in the graph. If both the start and the goal are farther away than the reach value, then the node/edge will not be part of the shortest path

[1]This approach was independently discovered by the first author of this paper.

between the start and the goal and can be pruned. Reach can be thought of as a circle around each node; if either the start or goal are in the circle the node cannot be pruned, otherwise it can.

Bounding Box Pre-Computation Procedure

In this paper we will focus just on the bounding box version of geometric containers. We will show a general way to pre-compute bounding boxes, and then a JPS-specific method.

Recall that a bounding box is stored for each of the directed edges leaving each state in the state space. There are two ways that we can pre-compute the bounding box for a given state and adjacent edge. The first is to choose a node and query over all other state-edge pairs whether that node should be part of the bounding box of that state and edge (i.e. is it reachable on an optimal path starting at that state and edge). We call this the backwards precomputation because it starts with potential goal states and works backwards to all possible start states. The alternate is to perform a single-source shortest path computation from a node and to directly calculate the bounding box of that node from the resulting search. We call this the forward precomputation because it works forward from the start state to possible goal states. For generic bounding box computations both approaches work, but, as we will see, they are not the same when combined with the canonical ordering of JPS.

Algorithm 1 Compute bounding boxes

```
1:  procedure ComputeBoundingBoxes(map)
2:      // Initialize data
3:      for each node n in map do
4:          for each edge e in n do
5:              for each bounding box axis a in e do
6:                  a.min ← int.MaxValue
7:                  a.max ← 0
8:              end for
9:          end for
10:     end for
11:     for each node s in map do
12:         DijkstraFloodfill(s, map)
13:         for each node n in map do
14:             if n is reachable then
15:                 e ←GetStartingNodeEdge(n)
16:                 for each bounding box axis a in e do
17:                     c ←Coord(n, a)
18:                     a.min ←Min(a.min, c)
19:                     a.max ←Max(a.max, c)
20:                 end for
21:             end if
22:         end for
23:     end for
24: end procedure
```

Forward Precomputation

With a forward computation method and Dijkstra search, we must add an extra piece of information to be tracked by the Dijkstra search. For each node Dijkstra explores, it must relay the original starting node edge that this particular path emanated from, since we will be attributing information back to each starting node edge. Once each Dijkstra search has exhaustively explored all reachable nodes, we loop through all reachable nodes and expand the starting node's bounding box for the edge leading to this node to include the node's location. This preprocessing step has $O(n^2)$ time complexity in the number of nodes within the search space. Fortunately, this process is easily parallelizable.

Algorithm 1 describes the pre-computation. We begin by performing a Dijkstra search from each possible state s in the state space (line 12). After this search is complete, we look at each state n and add the coordinates of n to the bounding box associated with the starting node edge that state n originally emanated from (lines 16 - 20). In the 2D case, each (directed) edge of the graph has its own bounding box defined along two axes.

The function GetStartingNodeEdge returns the starting node edge that led to state n. Note that the cost of reaching n is irrelevant here, so no additional considerations are needed for weighted graphs. While there can be many such edges, typical implementations of Dijkstra's algorithm store a single parent pointer which is determined arbitrarily according to tie-breaking in the data structures. When combining bounding boxes with A*, it doesn't matter which of all possible edges we follow. But, we need to modify this procedure slightly when doing pre-computation for JPS to make sure we break ties for edge generation properly. We will discuss this approach later in the paper.

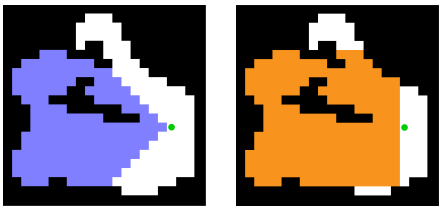
Combining with A* search

Once bounding boxes have been computed for a given map, applying them to an A* search is straightforward and shown in Algorithm 2. The only addition to the A* algorithm is a check to determine if the goal lies within the bounding box of the parent node in the direction of the neighboring node (line 8 of Algorithm 2). If the bounding box check fails, then the neighboring node is discarded and control continues to the next neighboring node in the loop.

Algorithm 2 A* search with bounding boxes

```
1:  procedure AStarSearch(start, goal)
2:      Push(start, openlist)
3:      while openlist is not empty do
4:          n ←PopLowestCost(openlist)
5:          if n is goal then return success
6:          end if
7:          for each neighbor d in n do
8:              if WithinBoundingBox(n, d, goal) then
9:                  // Process d in the standard A* manner
10:             end if
11:         end for
12:     end while
13:     return failure
14: end procedure
```

Figure 3(a) shows an example problem with the start state marked with a (green) circle. The area that can be reached



(a) (b)

Figure 3: (a) Optimal reachability with A* moving left and (b) states in the associated bounding box.

optimally by moving left is highlighted. The bounding box associated with this area is in Figure 3(b). An observant reader might notice that this area is larger than necessary. There are many states that can be reached optimally both by moving left and by moving diagonally down and to the left. In the next section we will show how to avoid this redundancy using the canonical ordering of Jump Point Search.

Combining with JPS+

Jump Point Search is a recent grid-based uniform cost search algorithm developed by Harabor and Grastien (2011). JPS+ is an optimization of Jump Point Search (Harabor and Grastien 2014) that precomputes key map data for even better runtime performance, at a cost of storing 8 values per grid square. For the purposes of this paper we do not need to distinguish between these approaches.

Bounding boxes can be applied to JPS+ to achieve roughly a 1000 times or larger performance increase over A*. Precomputing the bounding boxes for the JPS canonical ordering, however, requires a different procedure than with A* because JPS+ has specific patterns of exploration over symmetric paths that must be matched when performing the bounding box computation.

As we have already illustrated, JPS+ explores the grid search space using the canonical pattern shown in Figure 1. When there are obstacles in the map, the search must explore around these obstacles in a way that guarantees that optimal paths are still preserved. We illustrate this in Figure 4. In this example there are two obstacles (solid boxes). When the search passes an obstacle, forced neighbors are introduced (dashed boxes) which must be explored. Red arrows illustrate edges that are introduced to compensate for forced neighbors. Following a forced neighbor, the canonical ordering of the search is reset, and the search continues in the same manner, shown by the dashed green arrows.

Recall the Dijkstra search used to build bounding boxes. Given a starting state, it finds all reachable nodes, marking each one with the starting node edge that led to that state. A key property in many maps is that there are multiple possible edges that can be traversed to reach any particular node. For A* this doesn't matter, because A* will explore all possible paths. JPS, however, has a canonical ordering of states within a search (following diagonals before cardinal edges to reach a state). Thus, in many cases there is only one possible path that can be followed to reach each node. If the Dijkstra search and JPS search are not coordinated, our bounding boxes could prune an edge that is necessary to reach the goal, thinking that another edge will suffice.

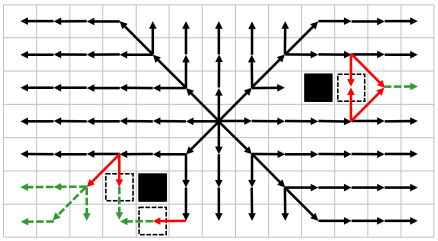


Figure 4: JPS+ example showing how the search strategy changes when obstacles are introduced.

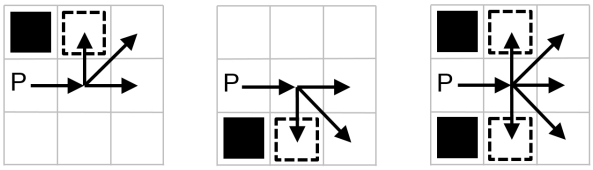
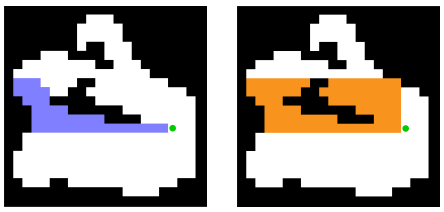


Figure 5: JPS+ forced neighbors (dashed boxes) from parent node P and the resulting neighbor node exploration. Similar cases exist in the other three cardinal directions.

This requires the Dijkstra floodfill search to mimic the canonical ordering used in JPS. We have already seen the basic pattern in Figure 1, but let's focus on the behavior when faced with forced neighbors. JPS explores the forced neighbors according to the cases in Figure 5 (along with their symmetric counterparts). Due to obstacles, these forced neighbors cannot be reached according to the canonical ordering of states. Forced neighbors are added to ensure the whole state space is explored. Once JPS starts exploring along a cardinal direction, it doesn't change the direction of the search unless it reaches one of these cases.

This now justifies why a forward precomputation is preferred to a backward precomputation. The forward precomputation can use the regular canonical ordering to control the search, but a backwards precomputation would need to use a reversed canonical ordering, which is significantly more complex in practice.

In Figure 6, we return to the example from Figure 3 and look at how JPS's canonical ordering improves the bounding



(a) (b)

Figure 6: (a) Reachability with JPS moving left and (b) associated bounding box.

boxes. While the start state and edge being considered are the same, we have significantly improved bounds through using JPS. Thus, JPS can not only search the map faster than A*, it also allows us to use smaller bounding boxes for pruning the state space more efficiently. Because the canonical ordering only removes edges from the graph, the bounding boxes will be strictly smaller than a naive ordering that considers all possible optimal paths. Note that these bounding boxes can be used with A* as well as with JPS, since A* doesn't do any path pruning.

The last detail to ensuring bounding boxes work correctly with JPS+ is to track and propagate ties during the Dijkstra floodfill. While unnecessary and even undesirable for A*, the JPS+ canonical ordering of search ignores non-canonical actions. If there is a tie between two initial edges from the starting node to a set of optimal nodes, the optimal nodes must be attributed to all tied starting edge bounding boxes. Otherwise, in very rare cases the bounding boxes will prevent JPS from finding optimal paths or possibly any path. During the Dijkstra floodfill, this is as simple as identifying when the path to a node ties with the cost from another path. In this situation, the only change to the node is to add the originating starting node edge to the node. If the starting node edges are held as an 8-bit bitfield, a new originating edge is simply or'ed into the bitfield. Once a bitfield contains multiple starting node edges, these are similarly propagated to successive nodes as the Dijkstra search continues to expand, thus attributing all subsequent optimal nodes back to multiple starting node edge bounding boxes.

Further Optimizations with JPS+ Bounding Boxes

On grids using A* variants with an octile heuristic, a very special open list optimization can be performed, first described as A* variant 1 (Sun *et al.* 2009). When an edge is expanded, its cost can never be less than the parent's. Because of this property, all edges with a cost equal to the parent can be placed on a stack instead of the open list. All nodes on the stack are guaranteed to be less than or equal to the cheapest node on the open list. When the cheapest node is needed, the stack is exhausted first before querying the open list. For JPS+ with bounding boxes, not only is the stack much faster, but the open list contains so few nodes that an unordered list has been shown to provide the best performance.

Another significant speed improvement is to identify the permutations of edges and previous states encountered in JPS to create a 2048 entry function pointer look-up table (2^8 edges * 8 previous states). Each entry in the table points to one of the 48 unique cases that are encountered during edge expansion. By directly jumping to a specialized version of expansion, a large number of conditionals are eliminated since each set of edges to expand is partially predetermined.

Experimental Results

Our experimental results have two goals. The first is to illustrate the influence of the JPS canonical ordering on bounding boxes and to understand its overall performance. The second goal is to show how JPS+ with bounding boxes compares to state-of-the-art algorithms.

Table 1: Comparison of A* and JPS+ with bounding on the game maps from the GPPC competition. Time is the average time (in ms) to solve a single problem from this set.

Algorithm	Time (ms)	A* Factor	JPS Factor
A*	15.492	1.0	-
A*+BB+Regular	1.888	8.2	-
A*+BB+Canonical	0.524	29.6	-
JPS+	0.072	215.2	1.0
JPS+BB+Regular	0.014	1106.6	5.1
JPS+BB+Canonical	0.010	1549.2	7.2

Table 2: Comparison of Open List Operations (*in Billions*)

Alg.	Map	Add	Decrease	Ratio
A*	Dragon Age	14,834.69	6,929.67	2.14
JPS+	Dragon Age	1,235.14	95.97	12.87
JPS+BB	Dragon Age	3.01	0.05	60.20
A*	Warcraft III	1,097.72	636.56	1.72
JPS+	Warcraft III	21.33	1.23	17.34
JPS+BB	Warcraft III	3.91	0.01	391.00
A*	StarCraft	65,571.45	36,074.92	1.82
JPS+	StarCraft	4,391.15	308.19	14.25
JPS+BB	StarCraft	200.41	0.78	256.94

We perform our primary comparison on a similar setup to the Grid-Based Path Planning Competition (Sturtevant 2014), a competition that has run since 2012 for the purpose of comparing different approaches to grid-based path planning. There isn't a single winner to the competition; instead there is a pareto-optimal frontier of non-dominated algorithms. An algorithm is non-dominated if there is no other algorithm that has equal or better performance on all metrics. Any particular real-world problem with a utility over the metrics (eg pre-computation limits and speed requirements) will find the best approach among this frontier.

All experiments are performed on maps from the GPPC competition and the Moving AI map repository (Sturtevant 2012). We ran our code on the same server as the GPPC competition (a 2.4 GHz Intel Xeon E5620 with 12 GB of RAM), so timing comparisons against the GPPC competition are meaningful.

Table 3: Per-map results. (Full results over all maps in ms)

	Random	Room	Maze	Starcraft	DA
A*+BB	6.611	7.270	4.992	1.681	0.212
JPS+	213.589	0.582	0.324	0.165	0.048
JPS+BB	1.363	0.038	0.059	0.017	0.008

Influence of Canonical Ordering

We begin by comparing A* and JPS with bounding boxes in Table 1. For each algorithm we denote whether it used bounding boxes (BB) and whether it used a regular ordering or canonical ordering of states. (All later experiments use the canonical ordering.) Besides the average time to solve

Table 4: A comparison between past GPPC entries (taken from http://movingai.com/GPPC/detail.html).

Algorithm	Year	Total Time (s)	Average Path (ms)	RAM (MB)	Storage	Pre-Time (min)
Subgoal	2013	2485.0	1.429	40.00	93 MB	1.0
NLevelSubgoal	2014	1345.2	0.773	42.54	293 MB	2.6
Contraction Hierarchies	2014	630.4	0.362	72.04	2.4 GB	968.8
JPS+BB	new	259.9	0.149	82.43	2.0 GB	3049.0
SRC	2014	251.7	0.145	274.16	52 GB	12330.8

a single problem (in ms), we also report the factor of improvement over a basic A* approach. This experiment is run on all non-artificial GPPC maps, including those from Starcraft, Dragon Age: Origins, and Dragon Age II.

The results show that A* is 8.2 times faster with regular ordering and bounding boxes, and 29.6 times faster when using the canonical ordering. In some sense, using the canonical ordering for bounding boxes partially causes A* to use the same ordering (because it potentially prunes other edges), and thus it gains some of the benefits of JPS without explicitly using the approach. Looking at the performance of JPS+, we see that using the canonical ordering is also significantly more effective, however the margin of gain for bounding boxes and the canonical ordering is less than with A*.

Detailed Results

We provide more detailed results in Table 2 in maps taken from the Moving AI repository. Here we look at the number of open list operations (measured in billions) performed over all search problems on three different map types. What we see is that JPS+ with bounding boxes not only adds relatively few states to the open list, but it also finds shorter paths to states already on the open list far less frequently (measured by decrease key operations). This is an expected result of the canonical ordering.

There is some variance in our results depending on the maps being solved. In Table 3 we investigate performance differences across map types from the GPPC competition. The first three columns are the artificial maps: ones with random obstacles, ones with rooms, and ones with mazes. The last two columns include maps from Starcraft and Dragon Age (Origins and II) respectively. First, we see that the artificial maps are more difficult than the game maps, which isn't surprising because those maps are larger. Second, we see that JPS+ does poorly on random maps. This is because random maps have far more jump points than other map types. But, bounding boxes are able to compensate for JPS+'s poor performance, providing better performance than A* with bounding boxes.

Comparison to State-of-Art

Finally, we compare against the state-of-the-art in Table 4. This table compares the result of JPS+ with bounding boxes to the best optimal algorithms that have appeared in the GPPC, primarily in the 2014 competition (Sturtevant *et al.* 2015). There are many different metrics that we can use to compare these algorithms, and we can clearly see trade-offs in this table. (Not all competition metrics are shown here.) There are 347,868 problems in the entire set, and problems are run five times for statistical significance. The columns measure the total time to solve all problems, the average time to solve a single problem, the RAM usage after solving all problems, the size of the pre-computed storage, and the time spent doing the precomputation.

The pre-computation time of bounding boxes is significantly larger than N-Level Subgoal Graphs (Uras and Koenig 2014), because we compute the full all-pairs shortest path data when doing the bounding. Our implementation is very efficient, however, so we can do this computation much faster than the Single-Row Compression (SRC) entry (Strasser *et al.* 2014), which also computes the all-pairs shortest-path data (compressing it). This is likely because we use the canonical ordering to speed up our Dijkstra search, a full discussion of which is beyond the scope of this paper.

We can see that JPS+ with bounding boxes is faster than all other optimal approaches shown here, with the exception of SRC. It is nearly as fast as the SRC entry, however, which contains the full all-pairs shortest path data and uses 26 times the storage of JPS+ with bounding boxes. The entry does use slightly more RAM than other approaches, and, as mentioned before, takes more time for pre-computation. But, overall, it establishes a new standard for fast grid pathfinding.

Conclusions and Future Work

Bounding boxes are a simple, yet powerful optimization technique that establish state-of-the-art performance when combined with JPS+. Most importantly, the canonical pruning and other ideas which make JPS+ efficient are highly orthogonal to the gains from bounding boxes, meaning that the two ideas are highly efficient when combined. Bounding boxes are especially well suited to static search spaces based in Euclidean space and improve when pre-computed with the canonical ordering of JPS+.

There are many possible optimizations that need to be explored. The most important include (1) finding ways to reduce the cost of pre-computation, (2) computing which bounding boxes are most effective, and only storing these and (3) looking at techniques like metric embeddings (Rayner *et al.* 2011) to re-embed a map in a new metric space that is designed to improve the bounding box performance.

References

J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *National Conference on Artificial Intelligence (AAAI)*, pages 942–947. AAAI Press, 2006.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

R.A. Finkel and J.L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

A.V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *SIAM Workshop on Algorithms Engineering and Experimentation (ALENEX 06)*, page 41, Miami, FL, January 2006. Society for Industrial and Applied Mathematics. also as MSR-TR-2005-132.

Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In Wolfram Burgard and Dan Roth, editors, *AAAI Conference on Artificial Intelligence*. AAAI Press, 2011.

Daniel Damir Harabor and Alban Grastien. Improving jump point search. In Steve Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2014.

N. Pochter, A. Zohar, J. S. Rosenschein, and A. Felner. Search space reduction using swamp hierarchies. In *AAAI Conference on Artificial Intelligence*, 2010.

C. Rayner, M. Bowling, and N. Sturtevant. Euclidean heuristic optimization. In *AAAI Conference on Artificial Intelligence*, pages 81–86, 2011.

Ben Strasser, Daniel Harabor, and Adi Botea. Fast first-move queries through run-length encoding. In Stefan Edelkamp and Roman Barták, editors, *Symposium on Combinatorial Search (SoCS)*. AAAI Press, 2014.

Nathan R. Sturtevant, Ariel Felner, Max Barer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 609–614, 2009.

Nathan R. Sturtevant, Jason Traish, James Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. The grid-based path planning competition: 2014 entries and results. In *Symposium on Combinatorial Search (SoCS)*, 2015.

Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

Nathan R. Sturtevant. The grid-based path-planning competition. *AI Magazine*, 35(3):66–68, 2014.

Xiaoxun Sun, William Yeoh, Po-An Chen, and Sven Koenig. Simple optimization techniques for A*-based search. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 931–936, 2009.

Paul Tozour. AI game programming wisdom 2. In Steve Rabin, editor, *Search Space Representations*, pages 85–113. Charles River Media, 2002.

Tansel Uras and Sven Koenig. Identifying hierarchies for fast optimal search. In Carla E. Brodley and Peter Stone, editors, *AAAI Conference on Artificial Intelligence*, pages 878–884. AAAI Press, 2014.

Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *International Conference on Automated Planning and Scheduling, (ICAPS)*. AAAI, 2013.

Dorothea Wagner, Thomas Willhalm, and Christos D. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM Journal of Experimental Algorithmics*, 10, 2005.