Faster Grid Pathfinding with Approximate Bounding Boxes

Mark Carlson, Daniel D. Harabor, Peter J. Stuckey

Department of Data Science and AI, Monash University
mark.carlson@monash.edu, daniel.harabor@monash.edu, peter.stuckey@monash.edu

Abstract

Grid-based path planning is an important application used widely in computer games and robotics. One of the fastest current approaches to optimal grid-based path planning, JPS+BB+, relies on extensive pre-computation to compute, for each jump point node, pruning bounding boxes containing all points reachable optimally by leaving the node in each direction. With this, the search for a shortest path is almost direct, only rarely considering more than one leaving direction at each node. But the pre-computation is expensive, requiring a full Dijkstra search from each jump point node to every other node in the graph. For the largest grid maps, this may require over 8 hours. In this paper we explore how to speed up this pre-computation, at the expense of a decrease in the accuracy of the bounding boxes created. We show we can improve pre-computation time by up to two orders of magnitude while only increasing average search time by at most 60% but often less than 10%. The faster bounding box computation can also be used to compute bounding boxes for every node in the map, decreasing average search time by up to 20% with typically only a 10% increase in pre-processing time.

Introduction

Path planning is an important and long studied problem in AI, having broad application to problems in robotics and computer games. In many practical examples, the environment is defined as a grid of traversable and non-traversable cells, with each traversable cell connected to its orthogonal and diagonal traversable neighbours. Path planning in such grid domains is well studied, with many competing approaches. The Grid-Based Path Planning Competition (GPPC^2) (Harabor et al. 2026) is a forum for tracking and disseminating state-of-the-art progress in the area. According to GPPC^2, the current fastest optimal approach to grid-based path planning is JPS+BB+ (Hu et al. 2021), requiring on average 27 microseconds to generate an optimal path across the benchmark suite. However, JPS+BB+ has by far the greatest pre-processing requirements of any method, requiring 12,936 minutes (over 215 hours) to process the benchmark maps. This is despite JPS+BB+ having a significant pre-processing advantage over its predecessor JPS+BB (Rabin and Sturtevant 2016), which computes a bounding box for each leaving direction from every cell in the grid. The pre-processing requirements for JPS+BB greatly exceed the limits imposed by GPPC^2.

The pre-processing step of JPS+BB+ computes, for each jump point node, bounding boxes containing all cells optimally reachable by leaving the cell in each direction. This requires a Dijkstra search from each jump point node to every grid cell. These bounding boxes are then used to prune the search, significantly reducing search effort; we will refer to them as *pruning bounding boxes*. It is not uncommon for this pruning to result in a single unique exit direction from a jump point node towards the target.

In this paper, we present new methods using JPS and CJPS to compute approximate pruning bounding boxes in a fraction of the time of a grid-level Dijkstra search. We apply these methods to reduce the JPS+BB+ pre-processing cost by up to two orders of magnitude, in exchange for a small increase in search time. Additionally, an alternative application of these methods allows complete pruning bounding box data to be computed in only a little more time than JPS+BB+, resulting in a small decrease in search time.

Background

Grid pathfinding restricts to graphs where each node is a traversable cell in a grid. Orthogonally adjacent traversable cells are connected by an edge of cost 1, and diagonally adjacent traversable cells are connected by an edge of cost $\sqrt{2}$ if both orthogonally adjacent cells are also traversable.

Jump Point Search (JPS) (Harabor and Grastien 2011, 2014) is a pathfinding method for grid maps which enforces a canonical ordering of moves to prune the search. The order requires that when possible, diagonal moves precede cardinal moves. To maintain this ordering, the search forbids diagonal moves from being generated immediately after a cardinal move, except when this is required to navigate around an obstacle; such exceptions occur at *jump points*. During search, only jump points are placed on the open and closed lists, significantly reducing the number of open list operations compared to grid search. Conceptually, this is achieved by immediately expanding any node which is not a jump point. In practice, an efficient online scanning procedure is used to quickly identify the successor jump points, resulting in a search time reduction of up to two orders of magnitude. JPS+ (Harabor and Grastien 2014) modifies the scan-

ning procedure to use a lookup table containing the distance to the next jump point, allowing it to locate the next jump point or diagonal turning point immediately. Formally, the *jump distance table* $jumpdb[n, \vec{d}] = (dist, block)$ records for each node n and direction $\vec{d}$ the distance $dist$ to either the next jump point $block = 0$ or the first obstacle $block = 1$ in the straight line from n in direction d. This can be computed at map load time in time linear in the map size.

Although JPS is a substantial improvement over grid search, it suffers from some pathological behaviours due to only searching jump points (Zhao, Harabor, and Stuckey 2023). First, the same grid cell may be scanned multiple times over the course of the search, particularly when there are multiple ways to navigate around an obstacle. Second, it is possible for the search to expand jump points with sub-optimal paths. This happens because whether a grid cell is considered a jump point depends on the path used to reach it. If a grid cell is a jump point on a sub-optimal path but not on any optimal path, then the search does not discover that the jump point is dominated and wastes time expanding it. The authors propose Constrained JPS (CJPS) which mitigates both of these issues, but it fails to translate into a search time reduction on most of the standard benchmarks. However, when a small number of random obstacles are introduced, CJPS can be several times faster than JPS.

Geometric containers (Wagner, Willhalm, and Zaroliagis 2005) is a search pruning method which removes many edges that are not on an optimal path to the goal from consideration. A pre-processing step computes, for each directed edge e, the set of grid cells $S(e)$ for which e is on an optimal path to the grid cell. When considering an edge e during search, if the goal is not in $S(e)$ then e is pruned. Storing these sets directly would require $O(N^2)$ space in the number of grid cells, which is often a prohibitively large space requirement. Instead, a geometric object requiring constant memory, such as a bounding box, containing at least every grid cell in $S(e)$ is stored. This reduces the space requirement to $O(N)$ in exchange for less precise pruning. This method reduces search time by an order of magnitude, but requires an $O(N^2)$ pre-processing step.

JPS+BB (Rabin and Sturtevant 2016) combines the JPS+ and geometric containers methods to achieve an order of magnitude improvement in search times compared to JPS+. When expanding a node and at diagonal turning points along a diagonal jump, only the directions for which the target is inside the associated pruning bounding box are considered. The combination requires a slight alteration to the computation of $S(e)$ to ensure that it is compatible with the canonical ordering of JPS+. JPS+BB+ (Hu et al. 2021) reduces the pre-processing cost of JPS+BB by computing partial pruning bounding box data; pruning bounding boxes are only computed for jump points and turning points on the way from one jump point to another. Since there are often significantly fewer of these than there are grid cells, pre-processing time is typically reduced by an order of magnitude. Furthermore, the missing pruning bounding boxes only affect the expansion of the start node due to the operation of JPS+, resulting in only a small increase in search time.

Algorithm 1 Algorithm for pre-computing bounding boxes of scanned cells.

procedure COMPUTEJUMPBB($n, jumpdb$)
  $r \leftarrow$ empty bounding boxes
  for valid direction $\vec{d}$ at n **do**
    $r[\vec{d}].grow(n + \vec{d})$
    $dist, block \leftarrow jumpdb[n, \vec{d}]$
    if $\vec{d}$ is cardinal **then**
      $r[\vec{d}].grow(n + (dist - block)\vec{d})$
    else
      $p \leftarrow 0$
      repeat
        $dist, block \leftarrow jumpdb[n + p\vec{d}, \vec{d}]$
        for $i = p + 1..p + dist - block$ **do**
          $dc, bc \leftarrow jumpdb[n + i\vec{d}, cl(\vec{d})]$
          $r[\vec{d}].grow(n + i\vec{d} + (dc - bc)cl(\vec{d}))$
          $dcc, bcc \leftarrow jumpdb[n + i\vec{d}, ccl(\vec{d})]$
          $r[\vec{d}].grow(n + i\vec{d} + (dcc - bcc)ccl(\vec{d}))$
        $p \leftarrow p + dist$
      until $block$
  return r

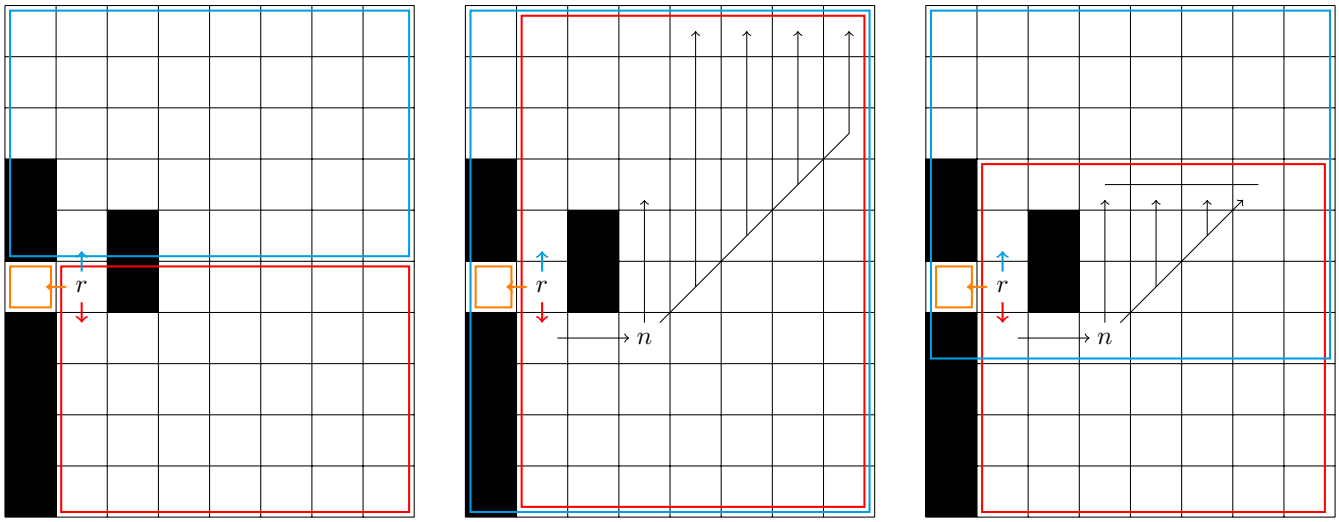
Approximating Bounding Boxes with JPS

To compute the pruning bounding boxes for the outgoing edges from a grid cell r, JPS+BB+ does a grid-level Dijkstra search using the canonical ordering starting at r. When expanding a node $n \neq r$, the first directed edge e on the optimal path from r to n is identified. Since $n \in S(e)$ by definition of $S(e)$, n must appear in the pruning bounding box enclosing $S(e)$; therefore, the pruning bounding box associated with e is enlarged to include n. When the Dijkstra search exhausts the open list, the pruning bounding boxes for the outgoing edges from r are complete.

Our Approximate Bounding Box (ABB) computation replaces the grid-level Dijkstra search with a JPS+ Dijkstra search. Since JPS+ only expands jump points, the original strategy for enlarging pruning bounding boxes would result in pruning bounding boxes which enclose optimally reachable jump points, rather than optimally reachable grid cells. Since the target is rarely a jump point, this will often result in inadmissible pruning, preventing the optimal solution from being found. We therefore modify the pruning bounding box enlargement procedure to fix this.

To allow this strategy to compute correct pruning bounding boxes, we conservatively assume that any grid cell scanned by the scanning procedure is optimally reached via the node being expanded. We modify the JPS+ scanning procedure to build a bounding box containing all of the grid cells it scanned. The pruning bounding box associated with e is then enlarged to contain this entire bounding box. We implement this efficiently by pre-computing for each jump point the bounding boxes containing all of the grid cells scanned by the scanning procedure when jumping in each direction.

The algorithm for computing the bounding boxes of scanned cells for node n in each direction is given in Al-



(a) (b) (c)

Figure 1: Pruning bounding boxes for r computed by different methods, (a) Original grid-level Dijkstra, (b) Approximate Bounding Boxes, and (c) Constrained Approximate Bounding Boxes. (b) and (c) additionally show the jump which grows the red pruning bounding box beyond the size of the original.

gorithm 1. Starting with an empty bounding box r, we first scan in each valid direction from n. We immediately grow the bounding box r by 1 in each valid direction $\vec{d}$, and then use the jump distance table $jumpdb[n, \vec{d}]$ to return the next jump point or turning point in that direction. For cardinal directions, we grow the bounding box to include the last reachable point in this direction. For diagonal directions, we have to scan both cardinal directions along the diagonal up to the jump distance. To do this, we iterate i along the jump segment, computing the jump distance in the clockwise cardinal direction from $\vec{d}$, $cl(\vec{d})$ and enlarging the bounding box, and similarly in the counter-clockwise cardinal direction $ccl(\vec{d})$. We repeat this process, tracking the start of the next jump segment in p, until we arrive at a blocked cell.

The correctness of ABB follows from the fact that JPS+ is an optimal search algorithm. For any reachable cell n, JPS+ will eventually scan n during the expansion of some jump point j with an optimal and canonical path. Therefore, $n \in S(e)$, where e is the first edge along this path. Since n was scanned in the expansion of j, the bounding box of scanned cells includes n by definition. The pruning bounding box associated with e is then enlarged to include this entire bounding box, and therefore also includes n, as required. Note that sometimes cells are scanned with sub-optimal paths, and the procedure does not act differently in this case. The result is merely that the pruning bounding box associated with e may be larger than is necessary to enclose $S(e)$; it does not violate the correctness requirements.

While this method significantly reduces the time required to compute pruning bounding boxes for JPS+BB+, the pruning bounding boxes it produces are often larger than necessary. This can be seen in Figure 1(b). The grid cells scanned by the north-east scan from n shown are assumed to be optimally reached from r via the red move, resulting in the pruning bounding box for the red move being extended to the top of the map. The reason for this is the pathological behaviour identified by Zhao, Harabor, and Stuckey (2023), in particular the fact that the same grid cell may be scanned multiple times. In this case, the north-east section of the map is scanned twice. It is scanned with optimal paths during the expansion of a jump point reached via the blue move from r, and then scanned again with sub-optimal paths via n.

Approximating Bounding Boxes with CJPS

Zhao, Harabor, and Stuckey (2023) propose Constrained JPS (CJPS) to mitigate the pathological behaviours they identify. The first pathological behaviour is re-scanning of grid cells, which they mitigate with constraints on the lengths of jumps. A constraint is created when CJPS performs a cardinal scan and finds a jump point that already has a lower g-value. Conceptually, a constraint is a barrier extending away from this jump point in the direction perpendicular to the cardinal scan direction. Diagonal scans and its cardinal branches will stop at the barrier. However, if a cardinal scan stops before the barrier, then some validity requirements cannot be ascertained and the constraint is discarded.

While constraints reduce redundant scanning, they do not address the second pathological behaviour, sub-optimal node expansion. Sub-optimal node expansion occurs when a jump point node is generated at a grid cell for which all of the canonical and optimal paths to that grid cell do not consider the cell to be a jump point. Since they do not consider it a jump point, they do not place the optimal g-value on it, resulting in the search never discovering that the sub-optimal node is sub-optimal. To fix this, CJPS identifies *corner points*, cells which are jump points when reached in some directions but not others. During scanning, if a corner

point is encountered which already has a better g-value, then the scan is terminated and a constraint is created. If it does not, then its g-value is updated and the sub-optimal jump point node is removed from the OPEN list.

We can use CJPS to create a better pruning bounding box approximation, Constrained Approximate Bounding Boxes (CABB), which replaces the grid-level Dijkstra search with a CJPS Dijkstra search. Like in ABB, we conservatively assume that any scanned cell is optimally reached via the node being expanded. The CJPS scanning procedure is modified to build a bounding box of scanned cells, which is then used to enlarge the relevant pruning bounding box in the pruning bounding box enlargement procedure. Unlike in ABB, the set of cells scanned by a jump cannot be pre-computed for CABB as it now depends on g-values found during search. Additionally, diagonal scans in CJPS cannot make use of the JPS+ jump database since branching cardinal scans are required at every diagonal step to ensure that any constraints remain valid. These limitations mean that CABB generally takes longer than ABB.

The benefit of the CABB method is illustrated in Figure 1(c). The north scan during the expansion of n finds the jump point at the north-east corner of the obstacle. This jump point has a better g-value via a path from the blue move, resulting in the creation of a constraint. This constraint is depicted in the figure as a barrier line extending east from the jump point. The northward branch scans of the north-east diagonal scan from n, as well as the diagonal scan itself, stop at the barrier. This prevents the red pruning bounding box from growing beyond the top of the obstacle. While the resulting pruning bounding box is still larger than the one produced by the original grid-level Dijkstra method (Figure 1(a)), it is much smaller than the one produced by ABB.

Accelerating Non-Jump Point Bounding Box Computation

While jump points and turning points are the grid cells for which pruning bounding box data is the most valuable, Hu et al. (2021) report search time increases of typically 30% as a result of missing pruning bounding box data for the remaining cells. While ABB and CABB are fast enough that they can compute the full pruning bounding box data more quickly than grid-level Dijkstra can compute the partial pruning bounding box data, the lower quality of the pruning bounding boxes still results in slower search much of the time. However, the JPS-like operation of the approximate methods provide the opportunity to leverage partial pruning bounding box data in the computation of full pruning bounding box data. We therefore propose Bounded Approximate Bounding Boxes (BABB), which modifies ABB to make use of partial pruning bounding box data to both increase quality compared to ABB and improve pre-processing time.

During the search which computes the pruning bounding boxes for the outgoing edges of a grid cell r, BABB maintains for each node n a *relevancy box*, which is the intersection of the pruning bounding boxes associated with each directed edge along the path from r to n. If a grid cell is outside of the relevancy box of n, then it must be outside the pruning bounding box of some edge along the path from r to n. Because this pruning bounding box contains at least all of the grid cells optimally and canonically reached using that edge, the grid cell is not optimally and canonically reached via any path through n. This establishes the key property of the relevancy box: no grid cell outside of it can be optimally and canonically reached via n. BABB uses relevancy boxes in three ways:

1. The relevancy box of n is intersected with the bounding box of scanned cells in the expansion of n before being passed to the pruning bounding box enlargement procedure.
2. Any successor of n which is outside of the relevancy box of n is suboptimal and consequently discarded.
3. If the relevancy box of n is entirely contained within the pruning bounding box n contributes to, then n is discarded.

The first and second uses mitigate the overscan and sub-optimal node expansion issues present in ABB, and their correctness follows directly from the key property of relevancy boxes. The third use prunes entire subtrees which cannot contribute to the expansion of their pruning bounding box. This pruning rule is applied after the pruning bounding box n contributes to is enlarged using the bounding box of scanned cells. Recall that this bounding box is retrieved from a lookup table, and so does not require expanding n to obtain.

Experimental Setup

We implemented ABB, CABB, and BABB in Rust, building on an existing implementation of JPS+BB+ in the `mkpath` pathfinding library (this implementation is the one on GPPC[2]). Full code is available online[1]. We will refer to the original JPS+BB pre-processing method as BB, and whenever a method is used to compute partial pruning bounding box data we will append a plus symbol, in keeping with the JPS+BB+ naming scheme. Since BABB requires existing partial pruning bounding box data, we will refer to it by appending BABB to the name of the method used to compute the partial data. We tested configurations ABB+ (using ABB for jump points), CABB+ (using CABB for jump points), CABB (using CABB for all cells), CABB+BABB (using CABB for jump points and BABB for the remainder), and BB+BABB (using BB for jump points and BABB for the remainder), primarily comparing against BB+ (Hu et al. 2021) and including BB (Rabin and Sturtevant 2016) for reference.

Experiments were run on an Ubuntu 22.04 system with an AMD Ryzen 9 5950X CPU and 32GB of memory. We used the Moving AI (Sturtevant 2012) and Iron Harvest (Harabor, Hechenberger, and Jahn 2022) benchmark instances. The Moving AI benchmark maps rarely exceed 1024×1024 in size, with 512×512 the most common size, while the Iron Harvest maps often exceed 3000×3000. The Room, Maze, and Random maps are synthetic maps, while the remaining map sets are derived from real game maps or street map data.

[1] https://github.com/MinusKelvin/2026-jpsbb-approx

	Pre-processing Time (min)						
	Partial: Jump Points Only			Full: All Cells			
Benchmark	BB+	ABB+	CABB+	CABB	CABB+BABB	BB+BABB	BB
Dragon Age 2	0.100	0.0119 −88.1%	0.0140 −86.0%	0.0718 −28.5%	0.0409 −59.2%	0.121 +20.1%	0.857 +754%
Baldur's Gate II	0.460	0.0229 −95.0%	0.0301 −93.5%	0.378 −17.8%	0.154 −66.6%	0.557 +21.1%	19.0 +4020%
Warcraft III	1.13	0.0337 −97.0%	0.0422 −96.3%	0.627 −44.4%	0.205 −81.8%	1.24 +9.58%	21.6 +1820%
Dragon Age: Origins	1.20	0.129 −89.2%	0.141 −88.3%	0.697 −41.9%	0.351 −70.7%	1.33 +11.2%	6.98 +482%
StarCraft	51.8	1.69 −96.7%	1.94 −96.3%	14.0 −72.9%	5.17 −90.0%	54.1 +4.48%	401 +674%
Iron Harvest	2930	27.1 −99.1%	38.4 −98.7%	1420 −51.6%	205 −93.0%	3070 +4.66%	—
Street	51.4	0.787 −98.5%	0.948 −98.2%	17.6 −65.7%	3.58 −93.0%	52.8 +2.83%	1320 +2470%
Room	5.42	0.642 −88.1%	0.786 −85.5%	7.25 +33.8%	5.44 +0.472%	9.99 +84.4%	122. +2150%
Maze	10.3	3.32 −67.6%	2.79 −72.8%	13.4 +30.6%	5.70 −44.5%	13.1 +27.2%	151 +1370%
Random	107	170 +59.7%	109 +1.74%	160 +49.5%	326 +205%	317 +197%	156 +45.7%

Table 1: Total preprocessing time (minutes) for each of the variants, as well as percentage change relative to the BB+ baseline.

For each method, we report the total wall-clock time for pre-processing the maps in each benchmark set using 16 threads, as well as the average time per path across each benchmark set. When measuring search time, all variants used exactly the same code; only the pruning bounding box data was different. We did not collect data for the BB method on the Iron Harvest benchmark set; we estimate the pre-processing time would exceed 3 months on our benchmarking system.

Experimental Results

We present the total pre-processing times for each method in Table 1 and the average search time per path for each method in Table 2. We also give the percentage change of each method relative to BB+. We first compare the variants that only compute pruning bounding box data for jump points: BB+, ABB+ and CABB+. Then we compare methods that compute pruning bounding boxes for every cell: CABB, CABB+BABB, BB+BABB and BB.

Partial Bounding Box Data

ABB+ and CABB+ both achieve significant reductions in pre-processing time compared to BB+, typically around an order of magnitude. The sole exception are the Random maps, which JPS-style algorithms are known to perform poorly on. The largest map we tested, `scene_sp_endmaps` from the Iron Harvest benchmarks, saw a reduction from 9.6 hours with BB+ to just 3.8 minutes with ABB+ and 5.5 minutes with CABB+; a two orders of magnitude improvement. Every map not in the Iron Harvest and Random benchmark sets took less than 10 seconds to pre-process with CABB+, except for StarCraft's `TheFrozenSea` map, which took 14 seconds. We can also see that CABB+ typically requires between 15% and 30% more pre-processing time than ABB+, which is expected due to the additional work CABB+ requires to handle constraints.

ABB+ and CABB+ also both see increases in average search time compared to BB+, however ABB+ is significantly more affected than CABB+. The relative increase in search times for ABB+ is often 3 to 5 times larger than for CABB+. While ABB+ has three benchmarks where the average search time was double that of BB+, CABB+ has none, with only a single benchmark where average search time was more than 50% longer than BB+. This demonstrates the effectiveness of CJPS constraints in improving pruning bounding box quality.

The largest search time increases were observed on the Iron Harvest and Street benchmarks. These maps feature many circumnavigable obstacles, which produce situations such as the one shown in Figure 1, leading to larger pruning bounding boxes. The Random benchmark, which features maps consisting almost entirely of circumnavigable obstacles, do not exhibit a similar search time increase because they do not have large open spaces for the pruning bounding boxes to be unnecessarily enlarged to contain. In contrast, the Maze benchmark contains no circumnavigable obstacles and sees no meaningful search time change.

Full Bounding Box Data

Computing pruning bounding box data for every cell in the map generally requires much more pre-processing time than computing pruning bounding box data only for the jump points and turning points. It is therefore interesting to see

	Search Time (μs/path)						
	Partial: Jump Points Only			Full: All Cells			
Benchmark	BB+	ABB+	CABB+	CABB	CABB+BABB	BB+BABB	BB
Dragon Age 2	1.46	1.69 +15.3%	1.51 +3.44%	1.27 −13.0%	1.28 −12.6%	1.23 −15.7%	1.21 −16.9%
Baldur's Gate II	1.34	1.85 +38.8%	1.44 +7.46%	1.14 −14.5%	1.18 −11.8%	1.09 −18.1%	1.04 −22.0%
Warcraft III	1.87	3.96 +112%	2.43 +29.9%	1.89 +1.03%	2.02 +8.08%	1.49 −20.3%	1.32 −29.6%
Dragon Age: Origins	2.39	3.69 +54.8%	2.60 +9.09%	2.06 −13.5%	2.12 −11.1%	1.97 −17.5%	1.88 −21.4%
StarCraft	5.47	10.6 +93.9%	6.54 +19.6%	5.53 +1.23%	5.77 +5.53%	4.63 −15.3%	4.47 −18.3%
Iron Harvest	28.9	77.2 +168%	43.2 +49.8%	37.8 +31.0%	39.9 +38.3%	25.1 −13.0%	—
Street	4.18	12.1 +191%	6.52 +56.1%	5.49 +31.5%	5.92 +41.7%	3.59 −14.1%	3.11 −25.4%
Room	14.9	16.9 +12.9%	15.6 +4.20%	14.8 −0.747%	14.8 −0.682%	14.3 −4.38%	14.2 −5.19%
Maze	31.3	31.3 −0.059%	31.3 −0.051%	31.2 −0.496%	31.2 −0.429%	31.3 +0.009%	31.2 −0.333%
Random	113	127 +12.4%	116 +3.25%	115 +2.29%	117 +3.77%	112 −0.199%	112 −0.641%

Table 2: Search time per path (microseconds) for each of the variants, as well as percentage change relative to the BB+ baseline.

that CABB required less pre-processing time than BB+ on all of the non-synthetic benchmarks, with three examples where it took less than half the time of BB+. Remarkably, CABB achieved between 10% and 15% lower average search time than BB+ on three of the benchmarks, but other benchmarks show as much as 30% longer search times. Whether CABB has lower search times or not largely depends on whether the improvement from the additional pruning bounding boxes exceeds the search time cost of having lower quality pruning bounding boxes at the jump points and turning points.

CABB+BABB uses BABB to compute the remaining cells after using CABB to compute pruning bounding boxes for the jump points and turning points, allowing us to compare the BABB and CABB methods for computing the remaining pruning bounding boxes. CABB+BABB often halves the pre-processing time compared to CABB, with notable outliers in the Iron Harvest, Street, and Random maps which see about one seventh, one fifth, and double the pre-processing time respectively. However, the pruning bounding boxes produced by BABB are lower quality than the ones produced by CABB, with small increases in average search time across all benchmark sets of less than 10%. Compared to BB+, CABB+BABB achieves up to an order of magnitude pre-processing time improvement with average search times similar to CABB.

Unlike the methods discussed above, BB+BABB does not improve pre-processing time compared to BB+ as it simply performs the BB+ pre-computation and then computes pruning bounding boxes for the rest of the grid cells using BABB. On most benchmarks, the pre-processing time increase is marginal; it is less than 5% on StarCraft, Iron Harvest, and Street, with only the synthetic maps seeing more than a 25% increase. However, BB+BABB consistently produces faster searches than BB+, often around 15% to 20% faster. In all but three benchmarks, the average search times of BB+BABB are within 5% of BB, which represents the fastest search time achievable with pruning bounding box data.

Conclusion

We developed new algorithms for computing pruning bounding boxes for a JPS+BB+ search which trade minor increases in search time for substantial reductions in pre-processing time, often by more than one order of magnitude. CABB+ allows the computation of pruning bounding boxes for almost all maps in an amount of time that could be absorbed into loading the map, with a search time cost of often less than 10% and never more than 60%. We believe this creates an attractive trade-off for applications where maps are static but not known ahead of time, for example when iterating on a map design in game development, or when loading a user-generated map. When more expensive pre-processing is acceptable, our BB+BABB algorithm improves search times to nearly that of JPS+BB, with a pre-processing cost only marginally higher than that of JPS+BB+. For maps such as the Iron Harvest maps where JPS+BB pre-processing is prohibitively expensive, this brings a new performance level into reach.

Acknowledgements

This research is supported by an Australian Government Research Training Program (RTP) Scholarship.

References

Harabor, D.; and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI conference on artificial intelligence*, volume 25, 1114–1119.

Harabor, D.; and Grastien, A. 2014. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 128–135.

Harabor, D.; Hechenberger, R.; and Jahn, T. 2022. Benchmarks for pathfinding search: Iron harvest. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 218–222.

Harabor, D.; Sturtevant, N.; Chen, Z.; and Hechenberger, R. 2026. Grid-based Path Planning Competition. https://gppc.search-conference.org/. Accessed: 2026-03-23.

Hu, Y.; Harabor, D.; Qin, L.; and Yin, Q. 2021. Regarding goal bounding and jump point search. *Journal of Artificial Intelligence Research*, 70: 631–681.

Rabin, S.; and Sturtevant, N. 2016. Combining bounding boxes and JPS to prune grid pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

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

Wagner, D.; Willhalm, T.; and Zaroliagis, C. 2005. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)*, 10: 1–3.

Zhao, S.; Harabor, D.; and Stuckey, P. J. 2023. Reducing Redundant Work in Jump Point Search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 16, 128–136.