Improving the Combination of JPS and Geometric Containers

Yue Hu, [1] **Daniel Harabor,** [2] **Long Qin,** [1] **Quanjun Yin,** [1] **Cong Hu** [1]
[1]National University of Defense Technology, [2]Monash University
huyue.cse@gmail.com, daniel.harabor@monash.edu, qldbx2007@sina.com
yin_quanjun@163.com, hccz95@163.com

Abstract

The JPS family of grid-based pathfinding algorithms can be improved with preprocessing methods such as Geometric Containers. However, such enhancements require a Dijkstra search for every node in the grid and the space and time costs of all this additional computation can be prohibitive. In this work we consider an alternative approach where we run Dijkstra only from every node where a jump point is located. We also compute and store geometric containers only for those outgoing edges which are consistent with the diagonal-first ordering in JPS. Since the number of jump points on a grid is usually much smaller than the total number of grid cells, we can save up to orders of magnitude of time and space. In addition to improving preprocessing overheads, we also present a partial expansion strategy which can improve the performance of online search by reducing the total number of operations on the open list.

Introduction

Pathfinding on uniform-cost grid is a common search 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 and a number of popular algorithms appear undominated on the Pareto front of successful entries at the 2014 Grid-based Path Planning Competition (Sturtevant et al. 2015). Among these winning methods is JPS+ (Harabor and Grastien 2014), a preprocessing-based symmetry breaking search technique that can improve the efficiency of grid-based A* by up to hundreds of times.

When moving on a grid, JPS-based methods apply a canonical ordering (Sturtevant and Rabin 2016) which takes diagonal moves before straight moves whenever possible. Since every node on the grid can be optimally reached by a path that satisfies the rule, JPS+ can discard from consideration any paths where diagonal moves appear later. The pruning is applied recursively, which means that many nodes can be processed immediately and without being explicitly added to the open list (e.g. whenever the successor set of a candidate node is reduced to 0 or 1 neighbours). The only nodes which remain, and which must be generated and searched, are known as *jump points*.

In its original form JPS runs entirely online and has no memory overhead. A main disadvantage is that it must scan individual rows and columns of the grid each time it expands a node. JPS+ improves its performance by storing, for every traversable cell, the first jump point or an obstacle that can be reached in every direction.

A related variant, JPS+BB (Rabin and Sturtevant 2016), further reduces search effort using Geometric Containers (Wagner, Willhalm, and Zaroliagis 2005), a well known and target-oriented pruning technique. The addition of containers to JPS+ can improve performance tenfold. The price for these gains comes in the form of additional time and memory which must be invested during preprocessing. To calculate all containers JPS+BB requires an all-pairs precomputation step that takes up to hours of additional time and dozens of megabytes of additional memory. These overheads can be prohibitive in applications where CPU and memory budgets available for path planning are limited. Such is often the case in robotics and computer games.

This paper aims to improve JPS+BB in two ways: (1) preprocessing and (2) online performance. To improve preprocessing it is only necessary to precompute containers for grid cells which are also jump points with the incoming directions and then only for outgoing edges which can appear on some diagonal-first path. Since the number of jump point locations is usually much smaller than the number of grid cells, and since the total number of outgoing edges is dramatically reduced, this approach can save orders of magnitude of preprocessing time and space. To improve online performance we describe a partial expansion strategy which can help to reduce operations on the open list, sometimes by up to several factors. The combination of (1) and (2) yields a new algorithm, JPS+BB+, and a corresponding new state of the art for optimal pathfinding on static grids.

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 as an obstacle. There are 8 available *move* actions $\vec{v}$ (equiv. *directions*) which serve to transition the search, from one cell to the next. The cost of each cardinal (i.e. straight) move (denoted $\vec{c}$) is 1 and the cost of each

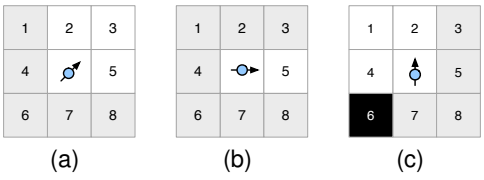

Figure 1: (a) When moving diagonally JPS always prunes all but 3 successors. (b) When moving straight JPS often prunes all but one successor. (c) Sometimes JPS is forced to consider more than one straight successor. Here, the lexically smaller path from node 7 to node 1 and to node 4 is blocked because node 6 is an obstacle.

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 cost among all paths from start to target.

We will sometimes refer to the cardinal directions by name: $\{Up, Down, Left, Right\}$ (U, D, L and R for short). We will also say that each diagonal direction is the result of combining two orthogonal cardinal directions; i.e. $\vec{d} = \vec{c_1} + \vec{c_2} \in \{UL, UR, DL, DR\}$. Finally, we use the algebra $s' = s + k\vec{d} + m\vec{c}$, with k and m as integers, to say that the cell s' is reached from s with k moves in direction $\vec{d}$ followed by m moves in direction $\vec{c}$. Negative coefficients in the algebra indicate moves in the opposite directions.

The JPS Family

Jump Point Search (JPS) (Harabor et al. 2011) is the combination of A* with an online symmetry breaking procedure that avoids redundant effort during search. We will say that two paths are *symmetric* if they share the same start and target and; if they have the same cost and; their constituent moves can be re-ordered to derive one path from the other.

To break symmetries JPS employs canonical ordering known as *diagonal-first*. This rule is applied during search to prune the set of successors, as illustrated in Figure 1. Notice that each pruned successor can be reached from the parent of the current node along a path which has smaller cost or which has the same cost but a smaller lexical prefix (i.e. diagonal moves appear sooner). Diagonal-first pruning reduces not only the branching factor of each expanded node but 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. Jumping allows the search to generate and expand only a distinguished set of nodes called *jump points* at which diagonal-first paths can branch.

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 and 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 the runtime performance of search by up to one additional order of magnitude. Further performance benefits can be obtained using a strategy known as *intermediate pruning* (Harabor and Grastien 2014). This algorithm, JPS+(P), treats all so-called diagonal jump points as "intermediate nodes" which can themselves be "jumped over".

JPS+BB: This algorithm combines JPS+ with Geometric Containers (Wagner, Willhalm, and Zaroliagis 2005) instantiated as Bounding Boxes. Geometric Containers are a type of target-oriented speedup technique which involves computing a 2D label for every edge in the graph. The labels indicate whether the edge appears on an optimal path, from its source node to a given target node.

JPS+BB works as follows: during an offline step an empty bounding box is created for every edge in the graph. A canonical best-first search, that combines Dijkstra's well known algorithm with diagonal-first pruning is then performed (in turn or in parallel) from every traversable node in the grid. Each time a search reaches a node with optimal cost it adds that node to the bounding box of the first arc appearing on the path to the node. Once computed, when expanding a node the only successors generated are those reached by an edge whose bounding box contains the target node. A description of this algorithm appears in (Rabin and Sturtevant 2016). In this work we use the same implementation as the original authors.

JPS+BB is highly effective in practice and improves performance of JPS+ by up to one additional order of magnitude (Rabin and Sturtevant 2016). Its main weakness is that the time requirements of the preprocessing step grows quadratically with the size of the map. Meanwhile its space requirements are always linear in the size of the map. We explore these issues further in our experimental evaluation.

Preprocessing the Grid

Our first contribution is a preprocessing improvement to JPS+BB which allows us to only perform a Dijkstra search from source nodes that are also jump points. Furthermore, we show that it is not necessary to store bounding boxes for every outgoing edge of each jump point but rather only those edges which allow an incoming path to continue from the source node in a way that is diagonal-first. On grids of practical interest (e.g. those drawn from real games) the number of jump points is usually orders of magnitude smaller than the total number of grid cells. This allows our new algorithm, JPS+BB+, to complete its preprocessing using just a small fraction of the total space and time required by JPS+BB.

The idea behind JPS+BB+ resembles that of TRANSIT which stores global information merely in a small set of important states without loss of online efficiency (Bast, Funke, and Matijević 2006). A main difference is that the set of transit nodes are determined algorithmically in road networks while the set of jump points are determined by the topology of the input grids.

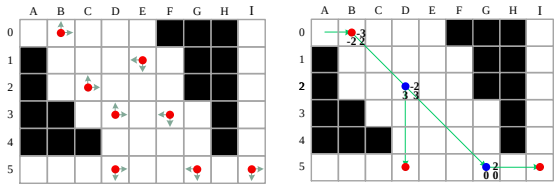

Figure 2: All the straight jump points and their cardinal travel direction in the small map are marked as red nodes and gray arrows in (a), respectively. There can be many diagonals from which their associated cardinal moves produces straight nodes. Figure (b) only shows the two marked as blue, which appear as successors of $(B0, R)$.

Definition 1. A *straight jump point* is a tuple $(n, \vec{c})$ comprising a grid cell n and a cardinal *travel* direction $\vec{c}$ such that: **(i)** $n - \vec{c}$ is a valid move; **(ii)** $n + \vec{c'}$ is a valid move; **(iii)** $\vec{c'}$ and $\vec{c}$ are orthogonal; **(iv)** The path $(n - \vec{c}) + \vec{c'}$ is invalid.

Definition 2. A *diagonal jump point* is a tuple $(n, \vec{d})$ where n is a grid cell and $\vec{d} = \vec{c_1} + \vec{c_2}$ is a diagonal *travel* direction such that $n + k_1\vec{c_1}$ or $n + k_2\vec{c_2}$ is a sequence of valid moves that produces a straight jump point or the target, where k_1 and k_2 are positive.

Identifying jump points: JPS distinguishes two types of jump points as definitions above indicate. Stated in simple words, Definition 1 says that every corner point is also a straight jump point location whereby Figure 2(a) recognizes all the tuples of straight nodes in the given small map. Note that a cell can correspond to more than one jump point such as $(C2, R)$ and $(C2, U)$. Meanwhile Definition 2 says that a diagonal jump point is any cell where a diagonal-first path can turn to reach a straight jump point.

Using these descriptions we develop a simple procedure to identify the set of all diagonal jump points. From each traversable grid cell n, we "jump" in each outgoing direction $\vec{v}$ and we record, in a *jump distance table*, the distance from n to the first straight or diagonal jump point in direction $\vec{v}$. If no jump point exists we store instead the distance from n to the first obstacle in direction $\vec{v}$. The example in Figure 2(b) briefly shows how these distances can help pinpoint the locations of the diagonal successors of a certain straight jump point, where negative integers indicate the distances to obstacles. In that case, $(D2, DR)$ and $(G5, DR)$ are two diagonal jump points associated with $(B0, R)$.

The proposed algorithm is linear in the size of the map provided we consult the jump distance table during recursion. Specifically, the computation of jump distances guarantees to never visit a grid cell more than eight times: i.e. once in each direction.

Space requirements are also linear in the size of the input map: we store 8 distances per grid cell and one flag per distance that indicates whether the node that will be reached is a jump point or not. Our implementation uses 2 bytes per 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 byes per label or compute values recursively.

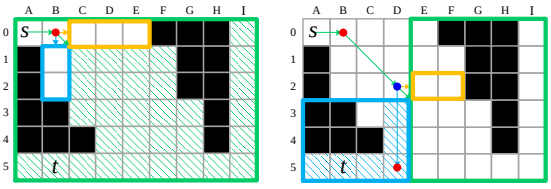

Figure 3: Given start $A0$ and target $B5$, (a) $B0$ has one bounding box label per diagonal-first move. We mark each move and box with the same color. Besides shaded nodes that $B0$ optimally reaches via the edge the rectangle bounding box for the diagonal move contains false positives; e.g. $A0 - E0$ and $B1 - B2$. (b) Since that box contains the target, the search proceeds to the first diagonal successor $D2$ where partial expansion is called to prune irrelevant edges.

Computing bounding boxes: JPS+BB performs a (Canonical) Dijkstra search from every unblocked cell. We also invoke this algorithm but only from every straight jump point and every diagonal jump point that appears as a successor of at least one other jump point. The idea here is to reduce the number of Dijkstra calls while still computing enough labels to retain the speed advantages of JPS+BB.

After each Dijkstra search all outgoing edges of each jump point are labelled with a bounding box. However, not all edges are necessary. In particular if we consider the incoming directions of each jump point we may realise that some edges will never be relaxed during search, except if the current Dijkstra source is also the start node. Figure 3(a) shows an example where only three edges from $B0$ follow the diagonal-first ordering. We exploit this observation to discard all unnecessary bounding boxes. The number of bounding boxes is reduced by at most 1 for locations of straight jump points and at most 5 for diagonals, respectively.

Online Search

The online search of JPS+BB+ is based on JPS+(P): i.e. we generate all straight jump point successors and we recursively prune any diagonal jump points. There exist two important differences between our approach and JPS+BB: (1) if the start node is not a jump point then there are no bounding box labels available and we cannot prune any immediate successors until the next expansion; (2) the branching factor of any jump point expanded by JPS+BB is fixed and at most 5 while we can produce up to two straight successors each time we "jump over" an intermediate diagonal jump point. This can happen up to $\min(H, W)$ times, where H and W are short for the height and width of a given map. We will

Table 1: Preprocessing results. 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 (in Megabytes). The number of maps in each set is given in the parenthesis. The best results under each metric are shown as bold numbers.

•	StarCraft (75)			DAO (156)			Random (60)			Room (30)			Maze (60)		
•	Dijk	Time	Mem	Dijk	Time	Mem	Dijk	Time	Mem	Dijk	Time	Mem	Dijk	Time	Mem
JPS+BB	19.8M	99.0h	1174MB	3.3M	3.0h	192MB	10.7M	44.7h	483MB	7.2M	37.0h	418MB	12.5M	37.6h	625MB
JPS+BB+	**2.4M**	**11.6h**	**65MB**	**0.5M**	**0.5h**	**15MB**	**6.9M**	**30.0h**	**235MB**	**0.1M**	**0.6h**	**3MB**	**1.0M**	**2.9h**	**26MB**

Table 2: Online performance as measured across all instances. Column *Time* indicates average time (in microseconds), *Gen* measures average nodes generated, while Expa measures average nodes expanded. The total number of instances for each set of maps is shown in parenthesis.

•	StarCraft (198230)			DAO (159465)			Random (137780)			Room (59550)			Maze (627000)		
•	Time	Gen	Expa	Time	Gen	Expa	Time	Gen	Expa	Time	Gen	Expa	Time	Gen	Expa
JPS+BB	26.1	122.4	119.8	13.6	61.7	60.2	553.2	1369.6	1311.0	29.1	94.8	89.0	153.8	629.6	628.4
JPS+(P+BB)	48.1	155.3	112.8	25.4	77.0	61.1	953.6	1854.8	1684.6	28.4	84.0	72.2	**123.2**	508.0	506.4
JPS+BB+v1	39.8	145.9	135.7	18.9	70.9	67.4	619.9	1403.8	1344.1	30.8	98.9	92.9	148.4	630.0	628.9
JPS+BB+	**18.6**	**61.1**	**51.0**	**10.0**	**33.4**	**30.0**	**397.8**	**898.8**	**843.8**	**17.8**	**56.2**	**50.1**	132.6	**497.5**	**496.5**

investigate how these differences impact performance in experiments.

Partial Expansion: This section presents a partial expansion strategy that improves the efficiency of online search. Our idea exploits containers stored with diagonal moves $\vec{d}$ as follows: If the bounding box of $\vec{d}$ contains the target then we recursively scan along the diagonal generating successors as long as the bounding box of the associated cardinal or diagonal move leading to the successor also contains the target. Alternatively, if $\vec{d}$ does not contain the target we do not recurse at all and thus generate zero successors.

Taking Figure 3(a) for instance, JPS+BB+ firstly discards the two cardinal branches from $B0$ and proceeds diagonally to $D2$ as JPS+BB does. As Figure 3(b) shows, the target is no longer in the diagonal bounding box of $D2$. The search can safely prunes edges that will never lead to the target and turns to the promising successor $D5$. Because we also apply intermediate pruning the method will not generate $D2$ but jump ahead directly for an additional saving.

Our idea is similar to methods such as Enhanced Partial Expansion A* (Goldenberg et al. 2014) and also avoids generating surplus nodes. A main difference is that we reason with geometric containers instead of f-value bounds.

Experimental Setup

Our work builds directly on JPS+BB (Rabin and Sturtevant 2016) and we compare against that algorithm using C++ codes made freely available by the original authors[1]. We use a binary heap as our priority queue and octile distance as the heuristic function of A* algorithm. And each of four coordinates of each bounding box is stored by 16 bits. In experiments we compare the following methods:

- JPS+BB: the original work.
- JPS+(P+BB), which is the combination of JPS+BB with online intermediate pruning and involves no edge pruning by container testing at diagonal jump points (Harabor and Grastien 2014).
- JPS+BB+: our improved method which includes intermediate pruning, partial expansion and bounding box labels that we store only at selected jump points and then only for moves which are known to be diagonal-first.
- JPS+BB+v1: a variant of JPS+BB+ without partial expansion.

We run experiments on a 3.60 GHz Intel Core i7-4790 CPU with 16 GB of RAM. We select for testing a wide range of maps from Sturtevant's collection (Sturtevant 2012) which are all available from https://movingai.com. We choose all maps and run all scenarios from the following benchmark sets: StarCraft and DAO (both drawn from real games), Random, Room and Maze (all synthetic).

Results

Table 1 presents preprocessing results across all tested maps. We show that JPS+BB+ calls Dijkstra search substantially fewer times. Consequentially, this method saves orders of magnitude in both computation time and space requirements vs JPS+BB. Note that JPS+(P+BB) and JPS+BB+v1 share the same preprocessing as JPS+BB and JPS+BB+, respectively.

In Table 2 we give results for online performance. We notice that for Room and Maze the addition of intermediate pruning helps JPS+(P+BB) to have fewer operations and run faster than JPS+BB. Meanwhile, it hinders rather than helps this algorithm on StarCraft and DAO, and shows itself ineffective on Random. In synthetic domains such as rooms and mazes, diagonal moves produce short recursions and few surplus nodes (i.e. nodes generated but never expanded). In these cases JPS+BB incurs extra overhead from constantly evaluating diagonal jump points by generating and expanding them. But on games maps diagonal moves produce long

[1] https://github.com/SteveRabin/JPSPlusWithGoalBounding

recursions and the overhead from exploring those straight successors without edge pruning in JPS+(P+BB) can be substantial.

Our proposed method possesses both of their online merits with edge pruning but no operations at diagonal jump points. This can easily explain that JPS+BB+ dominates in all three indicators for almost all sets of maps. On StarCraft, one of the most difficult of our chosen benchmark sets, JPS+BB+ generates less than half the nodes required by JPS+BB and runs about 30% faster. By contrast, JPS+BB+v1 answers all the queries much slower. The comparison shows that the partial expansion strategy is an effective way of accelerating the search and serves to mitigate the initial disadvantage from not storing bounding boxes with the start node. The proposed algorithm reduces the runtime indicator to a narrower range and its variance on query time also gets smaller compared with JPS+BB.

Conclusions

This paper presents an improved algorithm JPS+BB+ which improves the combination of JPS+ with Geometric Containers implemented as Bounding Boxes. To compute labels JPS+BB+ only runs Dijkstra search from grid cells which are jump points rather than all traversable cells. We also never store labels for non-canonical grid moves. We show this approach can save up to two orders of magnitude of additional preprocessing time and additional space vs JPS+BB, a previous attempt at combining these approaches.

Besides savings in preprocessing we introduce a simple partial expansion strategy that further exploits stored labels and which improves the performance of online search by up to 30% on average.

Through the combination of these ideas our JPS+BB+ achieves a new state of the art for optimal pathfinding on static grids. Future work could improve the combination of JPS with other preprocessing-based speedup techniques such as CPDs (Salvetti et al. 2018).

Acknowledgments

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.

References

Bast, H.; Funke, S.; and Matijević, D. 2006. TRANSIT: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research* 50:141–187.

Harabor, D., and Grastien, A. 2014. Improving jump point search. In *Twenty-Fourth International Conferenc on International Conference on Automated Planning and Scheduling*, 128–135.

Harabor, D. D.; Botea, A.; Kilby, P.; et al. 2011. Path symmetries in undirected uniform-cost grids. In *Symposium on Abstraction Reformulation & Approximation*, 58–61.

Rabin, S., and Sturtevant, N. R. 2016. Combining bounding boxes and jps to prune grid pathfinding. In *AAAI Conference on Artifitial Intelligence*, 746–752.

Salvetti, M.; Botea, A.; Gerevini, A. E.; Harabor, D.; and Saetti, A. 2018. Two-oracle optimal path planning on grid maps. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, 227–231.

Sturtevant, N. R., and Rabin, S. 2016. Canonical orderings on grids. In *International Joint Conference on Artifitial Intelligence*, 683–689.

Sturtevant, N. R.; Traish, J. M.; Tulip, J.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The grid-based path planning competition: 2014 entries and results. In *Annual Symposium on Combinatorial Search*, 241.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *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. *J. Exp. Algorithmics* 10:article 1.3.