Canonical Orderings on Grids

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

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

Abstract

Jump Point Search, an algorithm developed for fast search on uniform cost grids, has successfully improved the performance of grid-based search. But, the approach itself is actually a set of diverse ideas applied together. This paper decomposes the algorithm and gradually re-constructs it, showing the component pieces from which the algorithm is constructed. In this process, we are able to define a spectrum of new algorithms that borrow and repurpose ideas from Jump Point Search. This decomposition opens the door for applying the ideas from Jump Point Search in other grid domains with significantly different characteristics from two dimensional grids.

1 Introduction

Grids are a common state space representation for map-based movement and planning tasks, as they are simple, easy to implement, and have many other desirable properties. But, grids are also sometimes avoided because they represent a space using a uniform decomposition, regardless of the size of the space. This results in dense spaces with many transpositions — meaning that best-first algorithms like A* must be used over linear space algorithms like IDA*. These transpositions can be costly to search, as a significant amount of time is spent generating and detecting transpositions.

Recently, a number of different algorithms have been proposed that take advantage of the underlying structure of grids to significantly improve the performance of search on grids. These include approaches like Jump Point Search (JPS) [Harabor and Grastien, 2011], Subgoal Graphs (SG) [Uras *et al.*, 2013], and Transit Routing for grid maps (TR) [Antsfeld *et al.*, 2012]. To improve performance, all of these approaches explicitly limit the optimal paths that will be followed in a map. JPS does this by enforcing a canonical ordering over shortest paths in the map. TR does this by adding small random weights to edges in the map, which provides implicit preference between paths when many paths have the same cost. SG does this with a *direct-h-reachable* relationship.

While these techniques have successfully increased performance in grid-based maps, the underlying parts of their success are not well-studied. In this paper we look particularly at JPS. The original JPS work [Harabor and Grastien, 2011] describes the algorithms with a set of local rules for movement with 'forced neighbors'. This and follow-up work [Harabor and Grastien, 2014] mention A*, but do not explore the relationship with A* search.

This paper shows how JPS can be broken into three components: a best-first search, a canonical ordering of states, and a specialized successor function. Clearly delineating these components of the algorithm allows us to modify them individually. We introduce Canonical A*, which uses just the best-first search and the canonical ordering. We introduce Bounded JPS, which uses a parameterized successor function to improve performance. The Canonical Dijkstra's algorithm improves the performance of single-source shortest-path searches. Finally, we can modify the best-first ordering to create weighted (suboptimal) variants of these algorithms.

In addition to studying the performance of these approaches, we also consider the correctness of the canonical ordering. The canonical ordering used in JPS is a special case of work on symmetry reduction that has been applied elsewhere [Holte and Burch, 2014], and is not generally correct in best-first search. We describe how one can validate the correctness of the pruning in JPS.

1.1 Search Assumptions

There are many settings in which search algorithms are applied; each setting has its own set of related constraints on memory, pre-processing, optimality, and other performance metrics. This paper focuses on algorithms that do not use pre-processing before search begins, but discover the environment online during search. This setting is suitable for dynamic environments, or applications that have limited memory or time for pre-computation.

The work here is applicable to regular search environments, such as grids, that contain significant symmetry. Two-dimensional pathfinding problems are an obvious example application and the primary domain described in previous work. Other possible domains include higher-dimension grids, such as 3D voxels, or problems that have underlying grid structure, such as the configuration space of a robotic arm [Russell and Norvig, 2010].

In comparing the performance of different domains, the primary difference between domain types will be the cost of

Algorithm 1 Best-First Search

BFS($start$, $goal$())

```
 1: Add start to open
 2: while open not empty do
 3:    Get best from open
 4:    if best is goal then
 5:       Return path
 6:    end if
 7:    for all successors s of best do
 8:       if s in open then
 9:          Update cost to s in open
10:       else if s not in closed then
11:          Add s to open
12:       end if
13:    end for
14:    Add best to closed
15: end while
```

generating successors. Most two-dimensional grids are stored explicitly, and generating successors is as simple as checking the eight adjacent neighbors of a given cell. In a robotic arm, however, the domain is usually described implicitly, and thus checking successors requires more expensive checks to see if the line segments representing the robotic arm have collided with any of the line segments representing the environment. Even explicit domains may have significant costs as the number of dimensions increase – due to the cache locality of the neighbors in memory. The algorithms in this paper will be particularly applicable in domains with different performance characteristics than two-dimensional grids.

2 Decomposing Jump Point Search

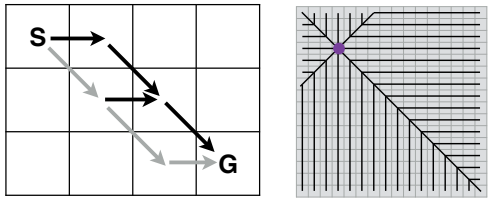
The first contribution of this paper is primarily pedagogical: a clear decomposition of JPS into its component pieces. Given an understanding of this decomposition, we can then easily introduce a variety of new algorithms by describing their specific changes to one of these components.

2.1 Best-First Search

In this paper we are concerned with best-first search algorithms. We review the concepts behind best-first search and then discuss the characteristics of search on grids.

A best-first search expands states according to a cost function that determines the best possible state to expand next. A*, for instance, is best-first search according to $f = g + h$, where the g-cost of a state is the cost of the path from the start state and the h-cost is an admissible (non-overestimating) estimate of the cost to the goal. We also assume that h is consistent, obeying the triangle inequality.

Pseudo-code for best-first search is in Algorithm 1. The search maintains $open$ and $closed$ lists. States on $open$ have been generated, but not expanded. States on $closed$ have been expanded. A state, s, is *expanded* when it is removed from $open$, the successors of s are generated, and s is placed on $closed$. A successor function *generates* the children of s by determining the states that can be reached from s and adding them to $open$. This paper will use several variants of best-first search. The full JPS algorithm is best-first search with a custom method of generating successors.



(a) (b)

Figure 1: Canonical Ordering in JPS

2.2 Canonical Ordering

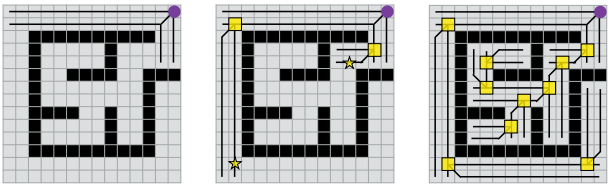
On open (convex) maps without obstacles, such as in Figure 1(a), there can be many different optimal paths between points. For the start (S) and goal (G) states shown, there are three possible optimal paths between the start and the goal. The difference between these paths is whether the action that moves to the right is taken first, second, or last. If we were to widen the grid by adding more grid cells, but keep the goal in the lower-right corner, the number of possible paths would grow exponentially. A* will not be forced to explore all of these paths, since with a consistent heuristic it will only expand each state once. But, generating the same states along different paths can still be costly.

One component of JPS is a canonical ordering which serves to eliminate many of these redundancies. The canonical ordering is a total ordering over paths; the preference is for paths that contain diagonal actions (which move in two or more axes) before cardinal actions (which only move in one axis) whenever possible along a path. Thus, among the three paths in Figure 1(a) JPS will always return the path in gray, as diagonal actions are taken before cardinal actions. JPS does not have to generate all paths and select between them to get the canonical ordering; it can generate them incrementally by considering the action used to reach a state.

On an empty convex map, these rules will provide a unique path from any start state to every other state in the map. We illustrate the canonical ordering for a given start state in Figure 1(b). The start state is in the upper-left portion of the map marked with a circle. Lines are drawn from each state to its possible successors. These lines proceed diagonally from the start state, with vertical and horizontal portions extending from the diagonals. Each state in the map is reached by exactly one line. Thus, the search space is no longer a graph; instead it is a tree.

We call this a *basic* canonical ordering; it is sufficient for finding paths in convex state spaces with no obstacles. This basic canonical ordering can be implemented by annotating each state, s, with the action that was used to reach s. Multiple actions are allowed – the start state, for instance, is considered to have been reached by every legal action. The canonical ordering determines the legal actions at each state. These can be specified by two rules: (1) If action a to arrive at s is one of the four cardinal directions, the only legal action at s is a. (Assuming a is valid.) (2) If the action a to arrive at s is a diagonal, with a_1 and a_2 cardinal components, the legal actions at s are a, a_1, and a_2.

The basic canonical ordering rules fall short when maps are



(a) (b) (c)

Figure 2: Extended the Canonical Ordering with forced neighbors

non-convex or contains obstacles, as the basic ordering rules are not sufficient to provide a path to every state in the state space. Consider the maze in Figure 2(a) with the start state marked with a circle in the upper-right corner of the map. The basic canonical ordering from this state will follow the lines shown, but cannot reach the rest of the map. The basic ordering assumes that the unreached states will be reached by other paths, but this is prevented by the presence of obstacles.

To address this, JPS introduces the notion of *jump points*. After a canonical path passes an obstacle, a jump point is placed on the corner of the obstacle. Here, the canonical ordering is reset to allow the canonical ordering to explore the portions of the state space that would otherwise be missed due to the presence of obstacles. These added successors at the jump points are called *forced neighbors*. Figure 2(c) adds the jump points to Figure 2(a), marking them with squares. We call this a *full* canonical ordering.

Jump points are only added when the search is proceeding in a cardinal direction. If an action orthogonal to the current movement direction (e.g. south when the canonical ordering is expanding to the west) is blocked due to an obstacle, a jump point is added when that orthogonal action is no longer blocked by the obstacle. In the top of Figure 2(b) the search proceeds west in the row above the obstacle. Since south is blocked, a jump point is added when the obstacle ends so that the search can proceed south around the obstacle.

In the maze map of Figure 2(c) the full canonical ordering is able to reach all states, and it will reach every state with exactly one path. This is because there is only one path to each state in the larger structure of the maze. In Figure 3(a) there are multiple paths around the obstacles in the map. In this example the jump points will direct the search around each obstacle indefinitely if we do not check for duplicates. The exact mechanism by which this occurs is part of JPS's specialized successor function, which we address next.

2.3 Successor Function

The final piece of JPS is a successor function that jumps over states in the state space, significantly reducing the number of states that are added to the $open$ and $closed$ lists. JPS generates[1] many states along the paths allowed by the canonical ordering (computing the g-cost of each), but only adds two types of states to $open$, goal states and jump points. The goal state must be added to ensure the search terminates with the correct solution, as a best-first search terminates when the goal is expanded. Jump points are added to keep the search from going into an infinite loop as the full canonical ordering wraps around obstacles.

So, to generate the successors of a state s, JPS follows the basic canonical ordering out of s until the goal or a jump point is reached, in which case the state is added to $open$, or a wall is reached, in which case nothing is added to $open$. We illustrate this in Figure 2. Part (a) of the figure shows the states that will be generated from the start state. Along these paths JPS will find two jump points, which are marked in part (b) with squares. In this case the start state has two successors that are added to $open$. A best-first ordering is used to choose which will be expanded next. Each of these states also has exactly one successor, shown in part (b) marked with stars. In summary, the successor function follows the canonical ordering from the parent, but only returns jump points and goal states as successors.

2.4 Summary

Put together, we can now describe JPS as a best-first search algorithm that generates states according to a canonical ordering. A basic canonical ordering is enhanced by jump points, creating a full canonical ordering, and ensuring that every state in the state space will be reached. To reduce the number of states added to $open$, JPS only adds jump points and goal states to $open$. JPS is equivalent to Algorithm 1 where *best* is measured by $f = g + h$ and the successor function (line 7) generates successors according to the full canonical ordering with jumping until a jump point or the goal is reached.

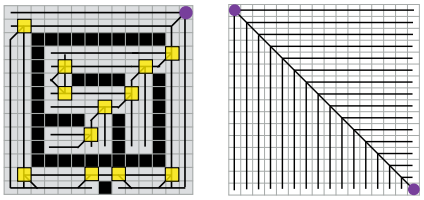
We observe that JPS has better performance than A* not because it explores fewer states, but because it avoids costly $open$ list operations. JPS, similarly to A*, must generate or expand every state with f-cost less than the optimal f-cost in order to prove that the optimal solution has been found. The advantage for JPS is that it will only generate many states that A* expands.

3 Canonical Ordering Correctness

Previous work has focused on showing that the canonical ordering itself is correct [Harabor and Grastien, 2011], but there is a secondary issue which has not been addressed, the correctness of using the canonical ordering with a best-first search such as A*. A canonical ordering is a special case of general move pruning techniques which can be automatically detected and applied at runtime [Holte and Burch, 2014]. Unfortunately, these approaches are not always correct in a best-first search - we can create canonical orderings or move pruning rules that are locally correct, but prune all legal paths between two states. The important question is whether the canonical ordering used by JPS is correct.

To prove the correctness of the JPS canonical ordering in best-first search, we must consider the case of a state s on $open$ reached by action a_1 with cost c that is later reached by action a_2 also with cost c. Maintaining the s in $open$ twice, once with each parent action, blows up the state space, negating the efficiency of the canonical ordering. So, we must

[1]When describing JPS we use a broader definition of generate, referring to all states that are checked along the canonical ordering, whether or not they are added to $open$.



(a) (b)

Figure 3: Canonical ordering starting in the upper-left corner for an empty map.

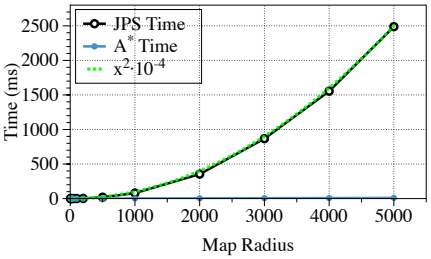


Figure 4: JPS running on an empty map.

show that maintaining only a single parent of s still guarantees that all states in the state space can be reached with optimal cost. We have verified that this holds by enumerating all pairs of actions that can reach a state and confirming that even if we do not maintain two parent actions, all states will still be reached with optimal cost. The full details are omitted due to space constraints. Thus, when JPS reaches a state via a different action with the same (or greater) cost, it can ignore this new action and maintain correctness.

4 New Algorithms

Given the decomposition of JPS into best-first search, a canonical ordering, and a specialized successor function, we can now make small changes to each of these pieces in order to observe its impact on JPS's behavior.

4.1 Canonical A*

The first algorithm we propose is A* with a canonical ordering, or Canonical A*. This algorithm takes the best-first search and the canonical ordering from JPS, but omits the jumping step and only generates the next successor of each state during search. That is, CA* is equivalent to Algorithm 1 where *best* is measured by $f = g + h$ and the successor function (line 7) generates successors according to the full canonical ordering without jumping. This approach will, in most cases, expand exactly as many nodes as A*; it will just be more efficient, because it won't generate as many successors for each state, avoiding redundant lookups on $open$. CA* sometimes expands slightly more states near the goal because it is restricted by the canonical ordering and regular A* is not. As we will see in experimental results, about half the performance gain from JPS comes from the canonical ordering.

4.2 Bounded JPS

This practice of jumping directly to the jump points reduces the number of states added to and removed from $open$. But, there is a significant drawback to this approach, which hasn't been explored previously. We illustrate this in Figure 3(b). In this figure, the start is in the upper-left corner of the map and the goal is in the lower-right corner. When solving this problem, the only successor of the start state that will be added to $open$ is the goal, which will then be expanded, terminating the search. Thus, no matter the size of the map, JPS will only perform two node expansions. But, JPS will generate every state in the entire map while checking for jump points.

We validate this behavior in Figure 4. We created a series of problems on open maps of size $r \times r$, where the start is at $(0, 0)$ and the goal is at $(r - 1, r - 1)$. We then solve the problems with JPS and with A* and plot the time it took to find the solution. We also plot the curve $10^{-4} \times x^2$ for comparison purposes. What we find is that the time required for JPS to find a solution grows with the size of the map (r^2), while A*'s performance grows with the radius (r). This is because A* has a perfect heuristic and thus only generates the optimal path from the start to the goal. This suggests that JPS's policy of generating many states but only inserting jump points and the goal into $open$ may not always be the best approach. Based on this observation we introduce the bounded jump point search (BJPS) algorithm.

BLJPS is equivalent to Algorithm 1 where *best* is measured by $f = g + h$ and the successor function (line 7) generates successors according to the full canonical ordering with jumping until a jump point or the goal is reached, or the jump distance exceeds a threshold determined by a bounding function. Our bounding function takes a parameter b; a state is out of bounds if it is more than distance b away from its parent; other bounding functions are possible.

BJPS will clearly fix the r^2 performance seen in Figure 4, as it will generate rb^2 states, where b is expected to be constant. The exact choice of b will depend on the properties of the domain. Interestingly, BJPS is a hybrid algorithm that interpolates between CA* and JPS. With a bound of 0, BJPS is CA*. With a bound of infinity, BJPS is the same as JPS. As we will see in the experimental results, BJPS offers better performance than both CA* and JPS.

4.3 Canonical Dijkstra

BJPS reduces the length of the jumps performed by JPS and increases performance as a result. But, there is one setting where performing longer jumps is reasonable. This is when we are performing a single-source shortest-path computation. In this context we are required to visit every state in the state space, so jumping long distances will not necessarily be wasted overhead.

We can modify any optimal best-first search algorithm to find the shortest path to all states in the state spaces, but for historical reasons we describe this approach applied to Dijkstra's algorithm [Dijkstra, 1959], and name the approach Canonical Dijkstra (CD). CD starts with a version of Algorithm 1 where *best* is measured by $f = g + h$ and the successor function (line 7) generates successors according to the

full canonical ordering with jumping until a jump point or the goal is reached.

The following additional modifications must be made: First, when successors are being generated along the canonical ordering, their g-costs must be written to the closed list. Unlike a traditional Dijkstra search, these g-costs are not guaranteed to be optimal. It is possible that we may revisit these states when jumping from a different parent later in the search. If, when jumping in the successor function, we find a state with lower or equal g-cost, we can immediately terminate the current jump. If we find a state with higher g-cost, we update the g-cost and continue the jumping process. If the state that was updated is a jump point (whether a state is a jump point is determined by the direction from which we reach it), we must take the state off of the $closed$ list and put it back on $open$ with updated parent information from the canonical search. This search continues until $open$ is empty.

4.4 Suboptimal Variants

The final modification we propose to JPS is to use a different metric for best. JPS traditionally orders its expansions by $g + h$ cost, but weighted A* [Pohl, 1970] uses a cost function of $g + w \cdot h$, where w is a parameter that trades suboptimality for search speed.[2] Weighted JPS and weighted CA* are equivalent to Algorithm 1 where *best* is measured by $f = g + w \cdot h$ and the successor function generates successors according to the respective rules for JPS and CA*.

If we consider that an algorithm like JPS is just a best-first search on a graph formed by the start state, jump points, and the goal, it is clear that we can modify the best-first metric used to order expansions without touching the canonical ordering or the successor generation rules. Thus, these weighted variants have the same solution quality bounds as weighted A*.

5 Experiments

We have already shown (Figure 4) that the worst-case performance of JPS can require the algorithm to visit every state in the state space, when a perfect heuristic is able to guide the search directly to the goal. The goal of our experimental results is to measure the performance of our new search algorithms, CA*, BJPS, and CD, as well as the weighted counterparts of CA* and BJPS. We will evaluate these algorithms by looking at the total time to solve a problem (in ms), the number of expansions, the number of node generations (this includes states that were analyzed during successor generation but not added to $open$), and the number of states on $open$ at the end of the search. These metrics paint a clear picture of the benefits and drawbacks of each approach.

All of our experiments were run on a 2.3 GHz Intel Core i7 laptop with 16 GB of RAM. Our priority queue is implemented with a heap, with grid-based $open$ and $closed$ lists. Our implementation of CD and CA* are based upon existing A* code. Our implementation of BJPS is based on a custom JPS implementation. All searches use the octile distance heuristic and ties are broken towards larger g-costs.

[2]Far more advanced suboptimal search algorithms have been proposed [Thayer *et al.*, 2012]; we only explore the baseline comparison here.

5.1 Optimal and Suboptimal Search

In this section we look at the performance of A*, CA* and JPS along with the weighted (suboptimal) variants of these problems. Our experiments are run on the Dragon Age: Origins maps from the Moving AI repository [Sturtevant, 2012]. The results of these experiments are found in Table 1.

First, we look at the comparative performance of A*, CA* and JPS when finding optimal solutions (weight = 1). Our average speedup isn't as great as reported in the original JPS work [Harabor and Grastien, 2011]. We have implemented numerous optimizations in our code, but it is clear that additional optimizations could improve performance further. These do not, however, change the number of expansions or generations. Thus, we still draw broader conclusions about the best approach according to the relative cost of node generation and expansion in a given domain/implementation.

Our results suggest that about half of the performance of JPS comes from the canonical ordering, and the other half comes from the successor generation policy. Both CA* and A* expand very similar numbers of states; JPS expands over 50x fewer states on average. Although JPS generates fewer states than A*, it generates four times more states than CA* does. This is because JPS generates large numbers of states when it is jumping that CA* does not expand. When finding a solution, JPS has fewer states remaining on $open$ than the other two algorithms.

Increasing the weight used in the best first search increases the number of states on $open$, while reducing the total number of expansions and generations. CA* finds the highest quality solutions between the algorithms given a particular search weight. This is because the canonical ordering prevents CA* from quickly going around obstacles like weighted A* would. The canonical ordering does not restrict JPS as much as CA*, because JPS is searching over jump points not grid cells.

JPS has the best performance of the three algorithms for any particular search setting. But, the large number of generations motivates BJPS which can reduce node generations by restricting jumping. This will be particularly important in domains where generations are more expensive than open-list operations.

5.2 BJPS

In Tables 2 we look at the performance of BJPS with different values of the search bound. The problems are taken from Dragon Age: Origins and Starcraft maps respectively. On both sets of maps there is a saddle point with a bound between 4 and 16 where the time is minimized, although this generally corresponds to the setting that has the most nodes on $open$. Clearly the JPS successor generation policy is not always the best policy. The gains from using BJPS are larger on Starcraft maps than on Dragon Age maps. This is because the Starcraft maps are larger and tend to have larger open areas than the Dragon Age maps. Thus, the overhead of generating successors and performing the jumping is much higher in Starcraft, and it is worth reducing this cost. Results in maze

Table 1: Average work by A* and Canonical A* with a weighted heuristic.

	Time (ms)			Expansions			Generations			Open			Quality		
Weight	A*	CA*	JPS	A*	CA*	JPS	A*	CA*	JPS	A*	CA*	JPS	A*	CA*	JPS
1	5.325	2.647	1.895	13295	13302	228	99483	13301	59325	305	122	37	1.00	1.00	1.00
2	3.156	1.475	1.165	7720	8002	147	57325	8001	36476	526	164	39	1.04	1.01	1.02
5	2.175	1.057	0.919	5406	6003	115	39881	6002	29053	634	188	42	1.09	1.03	1.05
10	1.907	0.973	0.843	4800	5574	106	35298	5573	26552	669	196	43	1.10	1.04	1.07

Table 2: BJPS results on Dragon Age and Starcraft maps.

Dragon Age				
Bound	Time (ms)	Expanded	Generated	Open
0	2.74	13037	13036	122
1	2.01	7387	15021	134
4	1.61	3963	21133	166
16	1.76	1869	36889	243
64	2.05	471	52777	159
∞	2.05	227	59324	36
Starcraft				
Bound	Time (ms)	Expanded	Generated	Open
0	13.02	48549	48548	363
1	9.06	26297	53061	390
4	6.54	12295	63550	441
16	6.29	5455	100096	668
64	6.89	1594	142105	734
∞	9.38	400	204099	101

maps (not shown) show that there is no significant gain to using BJPS on these problems. This is because there are no significantly large open areas in mazes that are costly for JPS to generate.

Our results show the trade-off between generations and expansions in BJPS. If we want to consider extending BJPS to other domains, the key question will be the cost of generating versus expanding successors. BJPS can be tuned to balance these costs and maximize performance.

5.3 Single Source Shortest Path

In this section we experiment with the single-source shortest-path computation across a variety of maps types, shown in Table 3. In these maps we only show the performance of an A* implementation, a CA* implementation, and a CD implementation (which uses the JPS successor generation policy). BJPS is omitted because experimental results showed that it had the same or slightly worse performance than CA*.

Our results show that the CA* can speed up a single-source shortest-path computation by a factor of 2, and CD up to 4.4 times faster than A*. This computation has many purposes, such as building heuristics [Rayner *et al.*, 2013], so improving the performance is a valuable contribution.

5.4 Discussion and Related Work

Canonical orderings have surfaced in the literature many places, but have not been deeply studied. We highlight a few areas of work here. Several papers suggest the importance of canonical paths in passing, but do not study this in detail. A short paper looking at hierarchy and grid graphs [Storandt, 2013b] alludes to many of the ideas studied here, as does work on contraction hierarchies in grids [Storandt, 2013a], but neither approach studies canonical orderings more deeply. Older work on reach [Goldberg *et al.*, 2006] discusses the importance of canonical orderings in road networks, but did not explore canonical orderings for experiments on grids later in the paper. Work on transit routing on grids [Antsfeld *et al.*, 2012] essentially creates a canonical ordering of states via small randomizations on edges. It is an open question whether they would get the same performance using the canonical ordering from JPS, or whether their approaches benefit from the randomness of their orderings.

In real-time search, Sturtevant's work on f-LRTA* [Sturtevant and Bulitko, 2011] dynamically built canonical orderings to prune paths in real-time search algorithms, but the work never considered *a priori* applying a canonical ordering for pruning purposes. There is likely more work that can be done in this direction. Related to this, the FRIT algorithm [Rivera *et al.*, 2014] looks at the ideal tree of shortest paths to the goal, which has a strong connection to canonical orderings.

6 Conclusions and Future Work

In this paper we have decomposed the performance of JPS, showing that the algorithm can be broken down into a best-first search, a canonical ordering, and a successor function that jumps between jump points to reduce the number of states on $open$. We use this breakdown to derive several new algorithms that are variants of JPS. Detailed experimental results reveal that JPS performs a significant number of node generations in order to reduce total node expansions. These results suggest that JPS may not be applicable to state spaces where there is a high cost of node generation relative to the cost of adding and removing states from $open$. More work is needed on a broader range of domains.

Table 3: Single-source shortest-path computation using A*, CA*, and Canonical Dijkstra. Times are in ms; improvement factor is over and above A*.

	A*	CA*		CD		
Maps	Time	Time	Ratio	Time	Ratio	|Map|
DA2	6.8	3.3	2.1	2.0	3.3	15,911.2
DAO	9.1	4.3	2.1	2.6	3.4	21,322.5
Mazes	86.8	40.9	2.1	26.4	3.3	207,940.7
Rand.	95.0	54.5	1.7	38.2	2.5	185,864.1
Rooms	124.7	60.5	2.1	28.3	4.4	232,785.2
SC	138.5	63.4	2.2	34.8	4.0	263,782.5

References

[Antsfeld *et al.*, 2012] Leonid Antsfeld, Daniel Damir Harabor, Philip Kilby, and Toby Walsh. TRANSIT routing on video game maps. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-12, Stanford, California, October 8-12, 2012*, 2012.

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

[Goldberg *et al.*, 2006] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*, pages 129–143, 2006.

[Harabor and Grastien, 2011] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

[Harabor and Grastien, 2014] Daniel Damir Harabor and Alban Grastien. Improving jump point search. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014.

[Holte and Burch, 2014] Robert C. Holte and Neil Burch. Automatic move pruning for single-agent search. *AI Commun.*, 27(4):363–383, 2014.

[Pohl, 1970] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artif. Intell.*, 1(3):193–204, 1970.

[Rayner *et al.*, 2013] D. Chris Rayner, Nathan R. Sturtevant, and Michael Bowling. Subset selection of search heuristics. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.

[Rivera *et al.*, 2014] Nicolas Rivera, Leon Illanes, Jorge A. Baier, and Carlos Hernández. Reconnection with the ideal tree: A new approach to real-time search. *J. Artif. Intell. Res. (JAIR)*, 50:235–264, 2014.

[Russell and Norvig, 2010] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[Storandt, 2013a] Sabine Storandt. Contraction hierarchies on grid graphs. In *KI 2013: Advances in Artificial Intelligence - 36th Annual German Conference on AI, Koblenz, Germany, September 16-20, 2013. Proceedings*, pages 236–247, 2013.

[Storandt, 2013b] Sabine Storandt. The hierarchy in grid graphs (extended abstract). In *Proceedings of the Sixth Annual Symposium on Combinatorial Search, SOCS 2013, Leavenworth, Washington, USA, July 11-13, 2013.*, 2013.

[Sturtevant and Bulitko, 2011] Nathan R. Sturtevant and Vadim Bulitko. Learning where you are going and from whence you came: h- and g-cost learning in real-time heuristic search. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 365–370, 2011.

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

[Thayer *et al.*, 2012] Jordan Tyler Thayer, Roni Stern, Ariel Felner, and Wheeler Ruml. Faster bounded-cost search using inadmissible estimates. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 2012.

[Uras *et al.*, 2013] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 2013.