Improving Jump Point Search

Daniel Harabor and **Alban Grastien**
NICTA and The Australian National University
firstname.lastname@nicta.com.au

Abstract

We give online and offline optimisation techniques to improve the performance of Jump Point Search (JPS): a recent and very effective symmetry-breaking algorithm that speeds up pathfinding in computer games. First, we give a new and more efficient procedure for online symmetry breaking by manipulating "blocks" of nodes at a single time rather than individual nodes. Second, we give a new offline pre-processing technique that can identify jump points apriori in order to further speed up pathfinding search. Third, we enhance the pruning rules of JPS, allowing it to ignore many nodes that must otherwise be expanded. On three benchmark domains taken from real computer games we show that our enhancements can improve JPS performance by anywhere from several factors to over one order of magnitude. We also compare our approaches with SUB: a very recent state-of-the-art offline pathfinding algorithm. We show that our improvements are competitive with this approach and often faster.

Introduction

Grid-based pathfinding is a problem that often appears in application areas such as robotics and computer games. Grids are popular with researchers because the encoding is simple to understand and apply but the process of finding optimal paths between arbitrary start-target pairs can be surprisingly challenging. At least one reason for this difficulty can be attributed to the existence of symmetries: myriad in grid maps but less common in other domains such as road networks. A path is considered symmetric when its individual steps (or actions) can be permuted in order to derive a new and equivalent path that has identical cost. In the presence of symmetry classical algorithms such as A* will waste much time looking at permutations of all shortest paths: from the start node to each expanded node.

Jump Point Search (JPS) (Harabor and Grastien 2011) is a recent and very effective technique for identifying and eliminating path symmetries on-the-fly. JPS can be described as the combination of A* search with two simple neighbour-pruning rules. When applied recursively these rules can improve the performance of optimal grid-based pathfinding by an order of magnitude and more – all without any pre-processing and without the introduction of any memory overheads.

The efficiency of JPS depends on being able to quickly scan many nodes from the underlying grid map in order to identify jump points. On the one hand such a procedure can typically save many unnecessary node expansions. On the other hand the same operation proceeds in a step-by-step manner and it can scan the same node multiple times during a single search. Consider Table 1, where we give a comparative breakdown of how JPS and A* spend their time during search. The results are obtained by running a large set of standard instances on three realistic game benchmarks that appeared in the 2012 Grid-based Path Planning Competition. Observe that JPS spends ∼90% of its time generating successors (cf. ∼40% for A*) instead of manipulating nodes on the open and closed lists – i.e. searching.

In this paper we propose a number of ideas to improve the performance of Jump Point Search. We focus on: (i) more efficient online symmetry breaking that reduces the time spent scanning the grid; (ii) pre-computation strategies for breaking symmetries offline; (iii) more effective online pruning strategies that avoid expanding some jump points. We evaluate our ideas on three realistic grid-based benchmarks and find that our enhancements can improve the performance of Jump Point Search by anywhere from several factors to over

	A*		JPS	
	M.Time	G.Time	M.Time	G.Time
D. Age: Origins	58%	42%	14%	86%
D. Age 2	58%	42%	14%	86%
StarCraft	61%	39%	11%	89%

Table 1: A comparative breakdown of total search time on three realistic video game benchmarks. M.Time is the time spent manipulating nodes on open or closed. G.Time is the time spent generating successors (i.e. scanning the grid).

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

one order of magnitude.

Related Work

Efficiently computing optimal paths is a topic of interest in the literature of AI, Algorithmics, Game Development and Robotics. We discuss a few recent results.

TRANSIT (Bast, Funke, and Matijevic 2006) and Contraction Hierarchies (Geisberger et al. 2008) are two highly successful algorithms that have appeared in recent years. Both of these approaches operate on similar principles: they employ pre-processing to identify nodes that are common to a great many shortest paths. Pre-processed data is exploited during online search to dramatically improve pathfinding performance. Though very fast on road networks these algorithms have been shown to be less performant when applied to grids, especially those drawn from real computer games; e.g. (Sturtevant and Geisberger 2010; Antsfeld et al. 2012; Storandt 2013).

Swamps (Pochter et al. 2010) is an optimal pathfinding technique that uses pre-processing to identify areas of the map that do not need to be searched. A node is added to a set called a swamp if visiting that node does not yield any shorter path than could be found otherwise. By narrowing the scope of search Swamps can improve by several factors the performance of classical pathfinding algorithms such as A*. Such pruning approaches are orthogonal to much of the work we develop in this paper. We will revisit the connection in the concluding section of this paper.

SUB (Uras, Koenig, and Hernàndez 2013) is a recent and very fast technique for computing optimal paths in grid-map domains. This algorithm works by precomputing a grid analogue of a visibility graph, called a subgoal graph, which it stores and searches instead of the original (much larger) grid. A further improvement involves directly connecting pairs of nodes for which the local heuristic is perfect (this operation is similar to graph contraction (Geisberger et al. 2008) in that it has the effect of pruning any intermediate nodes along the way). To avoid a large growth in its branching-factor SUB prunes other additional edges from the graph but this latter step makes the algorithm sub-optimal in certain cases. We compare our work against two variants of SUB in the empirical evaluation section of this paper.

Jump Point Search

Jump Point Search (JPS) is the combination of A* search with simple pruning rules that, taken together and applied recursively, can identify and eliminate many path symmetries from an undirected and 8-connected grid map. There are two sets of rules: pruning rules and jumping rules.

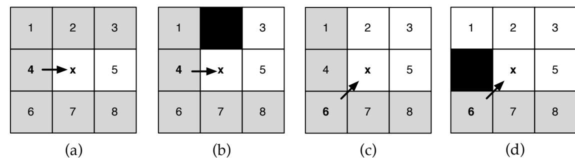

Figure 1: When the move from p to x is straight (as in (a)) only one natural neighbour remains. When the move from p to x is diagonal (as in (c)), three natural neighbours remain. When obstacles are adjacent to x some neighbours become forced; we illustrate this in for straight moves (q.v. (b)) and for diagonal moves (q.v. (d)).

Pruning Rules: Given a node x, reached via a parent node p, we prune from the neighbours of x any node n for which one of the following rules applies:

1. there exists a path $\pi' = \langle p, y, n \rangle$ or simply $\pi' = \langle p, n \rangle$ that is strictly shorter than the path $\pi = \langle p, x, n \rangle$;
2. there exists a path $\pi' = \langle p, y, n \rangle$ with the same length as $\pi = \langle p, x, n \rangle$ but π' has a diagonal move earlier than π.

We illustrate these rules in Figure 1(a) and 1(c). Observe that to test each rule we need to look only at the neighbours of the current node x. Pruned neighbours are marked in grey. Remaining neighbours, marked white, are called the *natural* successors of node x. In Figure 1(b) and 1(d) we show that obstacles can modify the list of successors for x: when the alternative path $\pi' = \langle p, y, n \rangle$ is not valid, but $\pi = \langle p, x, n \rangle$ is, we will refer to n as a *forced* successor of x.

Jumping Rules: JPS applies to each forced and natural neighbour of the current node x a simple recursive "jumping" procedure; the objective is to replace each neighbour n with an alternative successor n' that is further away. Precise details are given in (Harabor and Grastien 2011); we summarise the idea here using a short example:

Example 1 *In Figure 1(a) pruning reduces the number of successors of x to a single node $n = 5$. JPS exploits this property to immediately and recursively explore n. If the recursion stops due to an obstacle that blocks further progress (which is frequently the case), all nodes on the failed path, including n, are ignored and nothing is generated. Otherwise the recursion leads to a node n' which has a forced neighbour (or which is the goal). JPS generates n' as a successor of x; effectively allowing the search to "jump" from x directly to n' – without adding to the open list any intermediate nodes from along the way. In Figure 1(c) node x has three natural neighbours: two straight and one diagonal. We recurse over the diagonal neighbour only if both straight neighbours produce failed paths. This ensures we do not miss any potential turning points of the optimal path.*

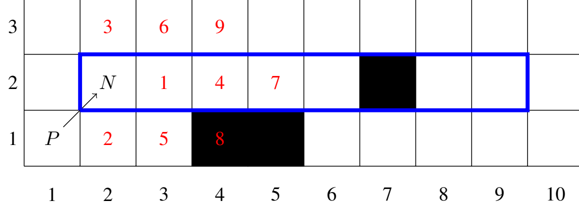

Figure 2: A current search state (the grid is assumed larger than the part presented). The red numbers show in which order the traversability of the nodes is tested. The blue rectangle represents the byte that is returned when we scan the grid to read the value of location $N = \langle 2, 2 \rangle$.

By jumping, JPS is able to move quickly over the map without inserting nodes in the A* open list. This is doubly beneficial as (i) it reduces the number of operations and (ii) it reduces the number of nodes in the list, making each list operation cheaper. Notice that JPS prunes nodes entirely online; the algorithm involves no preprocessing and has no memory overhead.

Block-based Symmetry Breaking

In this section we will show how to apply the pruning rules from Jump Point Search to many nodes at a single time. Such block-based operations will allow us to scan the grid much faster and dramatically improve the overall performance of pathfinding search. Our approach requires only that we encode the grid as a matrix of bits where each bit represents a single location and indicates whether the associated node is traversable or not.

For a motivating example, consider the grid presented in Figure 2 (this is supposed to be a small chunk of a larger grid). The node currently being explored is $N = \langle 2, 2 \rangle$ and its parent is $P = \langle 1, 1 \rangle$. At this stage, the horizontal and vertical axes must be scanned for jump points before another diagonal move is taken. As it turns out, a jump point will be found at location $\langle 5, 2 \rangle$. When looking for a horizontal jump point on row 2, Jump Point Search will scan the grid more or less in the order given by the numbers in red, depending on the actual implementation. Each one of these nodes will be individually tested and each test involves reading a separate value from the grid.

We will exploit the fact that memory entries are organised into fixed-size lines of contiguous bytes. Thus, when we scan the grid to read the value of location $N = \langle 2, 2 \rangle$ we would like to be returned a byte B_N that contains other bit-values too, such as those for locations up to $\langle 9, 2 \rangle$. In this case we have:

$$B_N = [0, 0, 0, 0, 0, 1, 0, 0] \tag{1}$$

Each zero valued bit in B_N represents a traversable node and each set bit represents an obstacle. In a similar fashion, and using only two further memory operations, we can read the values for all nodes immediately above and immediately below those in byte B_N:

$$B_{\uparrow} = [0, 0, 0, 0, 0, 0, 0, 0] \tag{2}$$
$$B_{\downarrow} = [0, 0, 1, 1, 0, 0, 0, 0] \tag{3}$$

Note that in practice we read several bytes at one time and shift the returned value until the bit corresponding to location $\langle 2, 2 \rangle$ is in the lowest position[1].

Note also that our implementation uses 32-bit words but for this discussion we will continue to use 8-bit bytes as they are easier to work with.

When searching recursively along a given row or column there are three possible reasons that cause JPS to stop: a forced neighbour is found in an adjacent row, a dead-end is detected in the current row or the target node is detected in the current row. We can easily test for each of these conditions via simple operations on the bytes B_N, $B_{\uparrow}$ and $B_{\downarrow}$.

Detecting dead-ends: A dead-end exists in position $B[i]$ of byte B if $B[i] = 0$ and $B[i+1] = 1$. We can test for this in a variety of ways; CPU architectures such as the Intel x86 family for example have the native instruction `ffs` (find first set). The same instruction is available as a built-in function of the GCC compiler. When we apply this function to B_N we find a dead-end at position 4.

Detecting forced neighbours: A potential forced neighbour exists in position i of byte B, or simply $B[i]$, if there is an obstacle at position $B[i-1]$ and no obstacle at position $B[i]$. We test for this condition with the following bitwise operation (assuming left-to-right travel):

```
forced(B) = (B<<1) & !B        (4)
```

When we apply this procedure to $B_{\downarrow}$ we find a potential forced neighbour at bit position 4; $\texttt{forced}(B_{\uparrow})$ yields no potential forced neighbours.

In order to minimise the number of branching instructions (important to prevent CPU stalling) we do not test the individual results of operations: rather we combine their results by way of bitwise disjunction into a single byte B_S (S for stop). For the example in Figure 2 we have

$$B_S = \text{forced}(B_{\uparrow}) \quad | \quad \text{forced}(B_{\downarrow}) \quad | \quad B_N \tag{5}$$
$$B_S = [0, 0, 0, 0, 1, 1, 0, 0] \tag{6}$$

Because $B_S \neq 0$, we know that the search has to stop. Using the `ffs` command we extract the position of the first set bit in B_S (call this bit b_S) and compare it

[1]We will assume throughout this work that bytes are zero-indexed and organised in little-endian format; i.e. the lowest bit is always in the left-most position and the highest bit is always in the right-most position. This means that when we apply logical shift operators such as << and >> the individual bits move in the opposite literal direction to the operator.

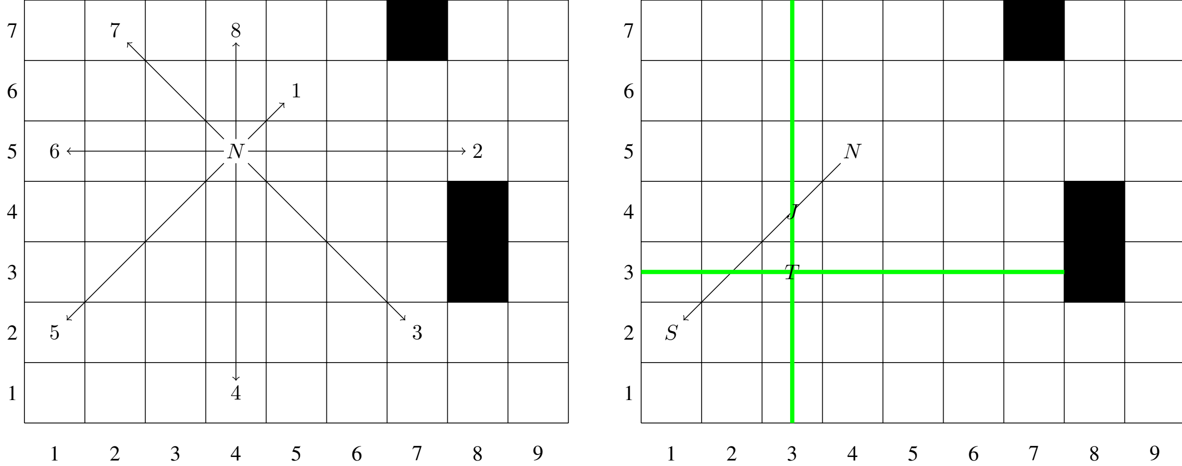

Figure 3: (a) A jump point is computed in place of each grid neighbour of node N. (b) When jumping from N to S we may cross the row or column of the target T (here, both). To avoid jumping over T we insert an intermediate successor J on the row or column of T (whichever is closest to N).

with the position of the first set bit in B_N (call this bit b_N). If $b_S < b_N$ we have discovered a jump point at location $B_N[b_S - 1]$; otherwise we have hit a dead-end. Alternatively, if B_S evaluates to zero there is no reason to stop: we jump ahead 7 positions (not 8) and repeat the procedure until termination.

Detecting the target node: To avoid jumping over the target node we compare its position to the position of the node where the block-based symmetry-breaking procedure terminated – either due to finding a successor or a reaching dead-end. If the target lies between these two locations we generate it as any other jump point successor.

Additional Considerations

The procedure we have described in our running example is applicable in the case where JPS scans the grid left-to-right. The modification for right-to-left scanning is simple: we shift the current node into the most significant position of B_N and replace `ffs` with the analogous instruction `msb`. In the case of up-down travel we have found it helpful to store a redundant copy of the map that is rotated by 90 degrees. Other possibilities also exist that do not incur an overhead: for example the map may be stored as blocks of size $m \times n$ that each contain bits from several rows and columns. Such an approach would also benefit diagonal travel (which in our implementation remains step-by-step) but may reduce the step size during horizontal scanning.

Preprocessing

In the preceding section we have suggested a strategy for enhancing the online performance of Jump Point Search. In this section we give an offline technique which can improve the algorithm further still. First, recall that Jump Point Search distinguishes between three different kinds of nodes:

- Straight jump points. Reached by travelling in a cardinal direction these nodes have at least one forced neighbour.
- Diagonal jump points. Reached by travelling in a diagonal direction, these nodes have (i) one or more forced neighbours, or (ii) are intermediate turning points from which a straight jump point or the target can be reached.
- The target node. This is a special node which JPS treats as a jump point.

We will precompute for every traversable node on the map the first straight or diagonal jump point that can be reached by travelling away from the node in each of the eight possible cardinal or diagonal directions. During this step we do not identify any jump points that depend on a specific target node. However, as we will show, these can be easily identified at run-time.

The precomputation is illustrated on Figure 3(a). The left side of the figure shows the precomputed jump points for node $N = \langle 4, 5 \rangle$. Nodes 1–3 are typical examples of straight or diagonal jump points. The others, nodes 4–8, would normally be discarded by JPS because they lead to dead-ends; we will remember them anyway but we distinguish them as *sterile jump points* and never generate them (unless they lead to the target).

Consider now Figure3(b), where T is the target node. Travelling South-West away from N, JPS would normally identify J as a diagonal jump point because it is an intermediate turning point on the way to T. However

J was not identified as a jump point during preprocessing because T was unknown. Instead, the sterile jump point S is recorded in the precomputed database. We use the location of S to determine whether the jump from N to S crosses the row or column of T (and where) and then test whether T is reachable from that location. This procedure leads us to identify and generate J as a diagonal jump point successor of N. We apply the same intersection test more broadly – to all successors of N. This is sufficient to guarantee both completeness and optimality. We call this revised preprocessing based algorithm JPS+.

Properties

JPS+ requires an offline pre-processing step that has worst-case quadratic time complexity and linear space requirements w.r.t the number of nodes in the grid. The time bound is very loose: it arises only in the case where the map is obstacle free and one diagonal jump can result in every node on the map being scanned. In most cases only a small portion of the total map needs to be scanned. Moreover, if we pre-compute using block-based symmetry breaking the entire procedure completes very quickly. We will show that on our test machine (a midrange desktop circa 2010) pre-processing never takes more than several hundred milliseconds, even when the procedure is applied to large grid maps containing millions of nodes.

Advantages and Disadvantages

The main advantage of JPS+ is speed: instead of scanning the grid for jump points we can simply look up the set of jump point successors of any given location on the grid in constant time. On the other hand, preprocessing has two disadvantages (i) jump points need to be recomputed if the map changes (some local repair seems enough) and (ii) it introduces a substantive memory overhead: we need to keep for each node 8 distinct labels (one for each successor). In our implementation we use two bytes per label. The first 15 bits indicate the number of steps to reach the successor and the final bit distinguishes the successor as sterile or not.

We can use less memory if we store labels for intermediate locations instead of actual jump points: for example using one byte per label we could jump up to 127 steps at one time. The disadvantage of this approach is that more nodes may be expanded during search than strictly necessary. A hybrid algorithm that combines a single-byte database with a recursive jumping procedure is another memory-efficient possibility.

Improved Pruning Rules

We have seen that JPS distinguishes between jump points that have at least one forced neighbour and those that have none. The former can be regarded as the grid

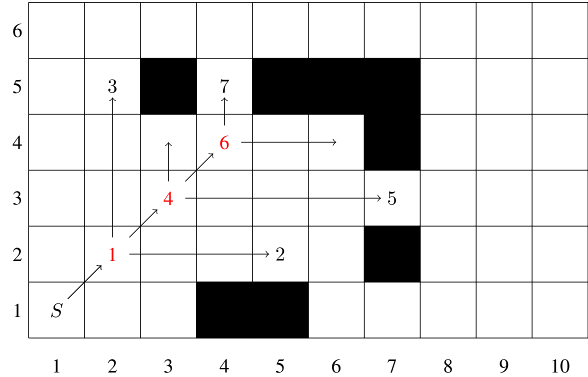

Figure 4: We prune all intermediate jump points (here nodes 1, 4 and 6) and instead generate their immediate successors (nodes 2, 3, 5 and 7) as children of the node from where initiated the jump (i.e., S).

equivalent of a visibility point (Lozano-Pérez and Wesley 1979): they are adjacent to at least one obstacle, and they are on at least one optimal path between two neighbours that are not mutually observable. If JPS prunes any of these nodes it is entirely possible that it will not return an optimal path. The second type of jump points are not adjacent to any obstacle. They are simply intermediate locations where the optimal path can change direction in order to reach the first type of jump point or the goal.

We argue that intermediate jump points can be safely pruned without affecting the optimality of Jump Point Search. Each intermediate jump point has at most three successors: the first is a jump point that is reachable horizontally, the second is a jump point that is reachable vertically and the third is the next intermediate jump point that can be reached without changing direction. When we prune an intermediate jump point we store its successors in a list and generate them in its stead. We apply this procedure recursively to any successors which are also intermediate jump points and terminate only when a dead-end is reached. Figure 4 shows an example.

To see that our strategy is optimality preserving we need only observe that for each intermediate jump point that is pruned the g-value of any of its successors remains unchanged. We simply generate these nodes earlier without first expanding any intermediary location. Once we have pruned such a node, the parent of each newly orphaned successor becomes the starting location from where we initiated the jump. To extract a concrete path we simply walk from one jump point on the final path to the next in a diagonal-first way: i.e. we follow the octile heuristic but take all diagonal steps as early as possible. Such a path is guaranteed to be valid and thus obstacle-free.

Our pruning strategy is applied entirely online; it

	StarCraft		Dragon Age: Origins		Dragon Age 2	
	Time (μs)	Branches	Time (μs)	Branches	Time (μs)	Branches
JPS 2011	19.89	3.76	6.36	3.31	4.54	3.25
JPS (B)	1.85	3.76	0.93	3.31	0.85	3.25
JPS (B+P)	7.10	22.72	1.96	8.11	1.54	6.62
JPS+	0.38	3.76	0.21	3.31	0.20	3.25
JPS+ (P)	1.56	22.72	0.52	8.11	0.46	6.62

Table 2: A comparison of the average time required to expand one million random starting nodes from each map in our three benchmark sets. We aggregate the figures by benchmark and give results for the average time to expand a single node, the average standard deviation and the average branching factor. Times are given in microseconds.

does not require any special data structures and does not store nor compute any additional information. It can be combined with other suggestions discussed in the current work and it allows us to jump further than we otherwise might. The cost is up to two additional open list operations for each node that we prune. Nevertheless, we will show empirically that pruning intermediate nodes from the search tree improves the performance of Jump Point Search.

Experimental Setup

We implemented Jump Point Search and a number of variants as described in this this paper. All source code is written from scratch in C++. For all our algorithms we have applied a number of simple optimisations that help to effectively stride through memory and reduce the effect of cache misses. This means that (i) we store the input map as a vector of bits, one bit for each node; (ii) we store the map twice, once in row major order and once in column major order; (iii) we pre-allocate memory in 256KB chunks.

We run experiments on a 2010 iMac running OSX 10.6.4. Our machine has a 2.93GHz Intel Core 2 Duo processor with 6MB of L2 cache and 4GB of RAM. For test data we selected three benchmark problem sets taken from real video games:

- **Dragon Age: Origins**; 44,414 instances across 27 grids of sizes ranging 665 to 1.39M nodes.
- **Dragon Age 2**; 68,150 instances across 67 grids of sizes ranging 1369 to 593K nodes.
- **StarCraft**; 29,970 instances across 11 grids of sizes ranging 262K to 786K nodes.

Instances are sampled from across all possible problem lengths on each map. All have appeared in the 2012 Grid-based Path Planning Competition.

In keeping with the rules of the competition we disallow diagonal corner-cutting transitions in all our experiments. This change requires a slight modification to the JPS algorithm. Principally it means that we no longer test for forced neighbours when jumping diagonally. Jumping straight is also simplified. In the terminolgy of our block-based jumping rules this change means we stop at location $B_N[b_S]$ rather than $B_N[b_S - 1]$. These modifications do not affect the optimality or correctness of the algorithm. The argument is identical to the one in (Harabor and Grastien 2011). Source codes for algorithms developed in this paper are available from `http://ddh.googlecode.com`.

Comparison with JPS 2011

We first analyse the impact on JPS search performance for various combinations of our optimisation techniques: JPS (B), which adds block-based jumping, JPS (B + P) which combines block-based jumping with improved pruning, JPS+ which adds pre-processing and JPS+ (P) which combines pre-processing with improved pruning. To avoid confusion we will denote the original algorithm as JPS 2011.

We have previously observed that the bottleneck of the JPS 2011 algorithm is individual node expansion operations. In Table 2 we give results for the average time required to expand one million random starting nodes from each input map in each benchmark set. We find that block-based jumping and pre-processing jump points each improve average times by one and two orders of magnitude respectively vs. JPS 2011. Pruning intermediate jump points from the search tree increases the average branching factor by several times but the time-per-expansion is still much better.

In Figure 5 we give a summary of search performance, in terms of time and node expansions, for all GPPC instances on each of our three benchmark sets. In each case and for each metric we consider relative improvement vs. JPS 2011. We show the spread of results after having assigned all test instances into buckets of similar length. We observe that any of our alternative approaches is strictly faster than the original. Moreover, JPS (B) and JPS (B + P) have all the same advantages as JPS 2011: they are fast, optimal, online and require, in principle at least, no extra memory vs. JPS 2011 (recall that in practice we store a rotated copy of map to improve memory access patterns). A summary of pre-

	Database Size (MB)					Prep Time (seconds)				
Algo	Min	Q1	Med	Q3	Max	Min	Q1	Med	Q3	Max
Dragon Age: Origins										
JPS+	0.02	0.08	1.15	5.76	21.65	0.00	0.00	0.02	0.17	0.47
SUB-S	0.00	0.02	0.32	1.64	5.92	0.00	0.00	0.00	0.01	0.04
SUB-TL	0.00	0.02	0.32	1.58	5.84	0.00	0.00	0.02	0.25	1.05
Dragon Age 2										
JPS+	0.03	0.54	1.74	4.16	9.26	0.00	0.01	0.04	0.09	0.20
SUB-S	0.01	0.15	0.49	1.13	2.55	0.00	0.00	0.00	0.01	0.01
SUB-TL	0.01	0.15	0.48	1.12	2.49	0.00	0.01	0.02	0.05	0.76
StarCraft										
JPS+	4.14	4.14	9.24	9.24	12.28	0.18	0.27	0.43	0.73	0.85
SUB-S	1.22	1.33	2.98	3.18	5.25	0.01	0.01	0.03	0.04	0.07
SUB-TL	1.18	1.25	2.79	2.87	4.20	0.27	1.14	2.84	9.39	23.77

Table 3: Preprocessing results for JPS+ and the two variants of SUB that we compare against. We present figures for the size of the resultant database (in MB) and the amount of precomputation time needed (in seconds). Columns Q1 and Q3 indicate values for the first and third quartile of each data set.

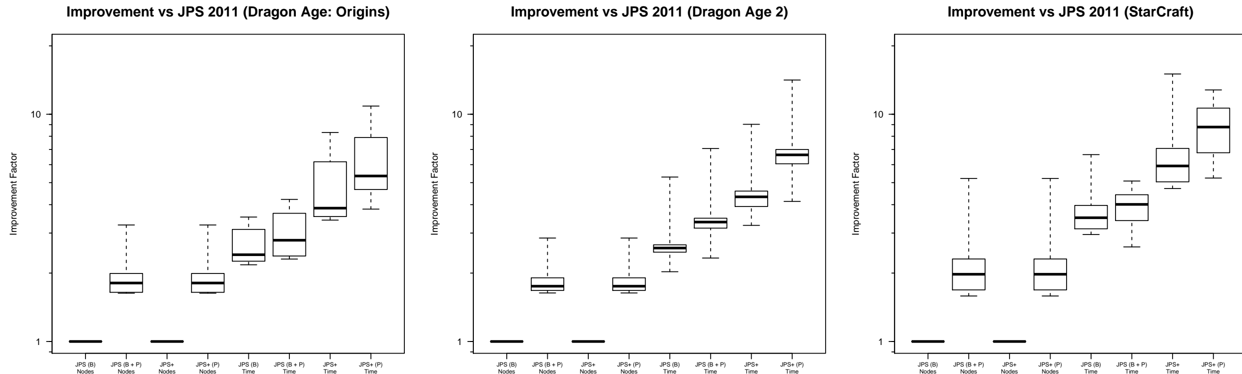

Figure 5: We measure the relative performance (or improvement factor) of each of our new JPS variants against the original. We consider two metrics: nodes expanded and search time. An improvement factor of 2 for search time means twice as fast; for node expansions it means half as many nodes expanded. Higher values are always better.

processing requirements is given in Table 3. Note that JPS+ and JPS+ (P) have the same requirements and are not listed separately.

Comparison with SUB

SUB (Uras, Koenig, and Hernàndez 2013) is a recent pathfinding technique from the literature. As one of the joint winners of the 2012 Grid-based Path Planning Competition (GPPC) SUB has been shown to be very fast and is considered the current state of the art. We compare against two variants described by the original authors: SUB-S (S for Simple) and SUB-TL (TL for Two Level). The former is guaranteed optimal while the latter is not. To evaluate SUB-S and SUB-TL we used the authors' original C++ implementation which we obtained from `http://gppc-2012.googlecode.com/svn/trunk/entries/SUB-a/`.

Table 3 compares the pre-processing requirements of SUB-S and SUB-TL with JPS+ (JPS+ (P) has identical requirements and is not shown). We observe that both JPS+ and SUB are able to pre-process most maps in well under a second and in most cases using less than 10MB of memory. A small number of notable exceptions arise for both JPS+ and SUB-TL. In Figure 6 we compare our four JPS variants with SUB-S and SUB-TL across all game map instances from the 2012 GPPC. We find that JPS (B) and JPS (B + P) are both competitive with, and often faster than, SUB-S. Meanwhile, JPS+ (P) appears competitive with SUB-TL for a large set of instances. Across our three benchmarks, DA:O, DA2 and SC, we measured an improvement for JPS (B + P) vs. SUB-S in 92%, 84% and 89% of tested instances respectively. For JPS+ (P) vs. SUB-TL we measured an improvement in 43%, 77% and 68% of tested instances respectively.

Discussion

The results demonstrate the superiority of the approaches presented in this paper. In JPS (B) and JPS (B + P) we have improved the performance of JPS 2011 by several factors all while retaining the same advantages inherent to the original algorithm: completeness, optimality and little-to-no memory overhead. Such results are remarkable as JPS 2011 has itself been shown to improve the performance of classical search algorithms such as A* by up to one order of magnitude and sometimes more.

We have shown with JPS+ and JPS+ (P) that further improvements are also possible. In our experiments we employ an offline pre-processing step together with a small amount of memory (10MB or less in most cases) and identify apriori all jump point successors of each grid node. The main advantage is performance: JPS+ and JPS+ (P) can improve the search times of JPS 2011 by up to one order of magnitude. The main disadvantage is that if the map changes the pre-processed database needs to be re-computed. We have shown that each such pre-computation can be performed very fast – usually requiring only tens or hundreds of milliseconds. Moreover the pre-computation can be easily parallelised over several time slices with JPS (B + P) employed as a fallback algorithm in the interim.

We have compared our JPS-based approaches against two variants of SUB (Uras, Koenig, and Hernàndez 2013). The first variant, SUB-S guarantees optimality; the second, SUB-TL, does not. We find that across most benchmark instances JPS (B) and JPS (B + P) are not only competitive with but faster than SUB-S. When we compare JPS+ (P) and SUB-TL we find the two algorithms often have complementary strengths: JPS+ (P) always has low pre-processing requirements, always finds the optimal path and is faster on a majority class of

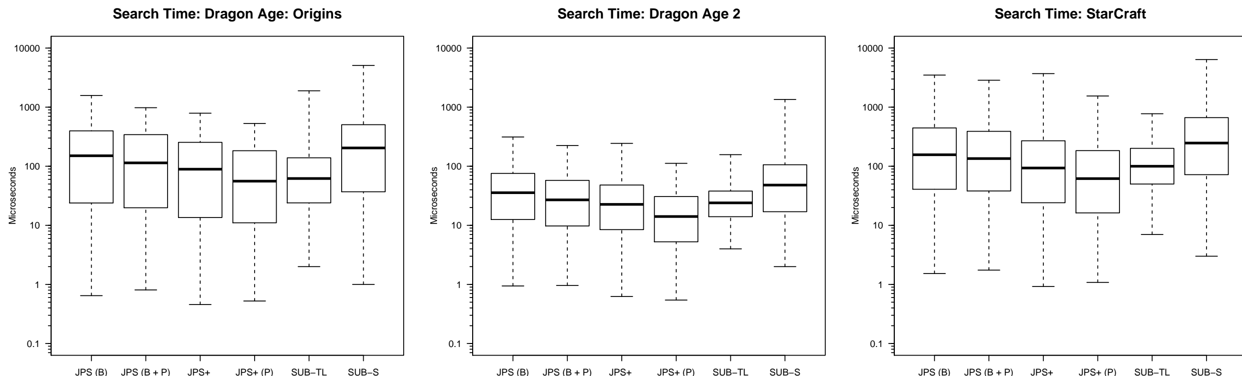

Figure 6: We compare the raw search time performance of our improved JPS variants (both online and offline) with two recent and very performant algorithms: simple subgoal graphs (SUB-S) and two-level subgoal graphs with local edge pruning (SUB-TL). All JPS variants and SUB-S are provably optimal. SUB-TL is not.

tested instances; SUB-TL has low space requirements and quickly finds optimal or near-optimal solutions to a large class of remaining instances.

Conclusion

We study several techniques for improving Jump Point Search (JPS). The first improvement is motivated by the observation that JPS spends a majority of its time scanning the grid for successors rather than manipulating nodes from the open and closed list (i.e., searching). We give a new procedure to detect jump points more efficiently by considering sets of nodes from the grid at one time (cf. one at a time). The second improvement is motivated by the observation that most jump points are goal-independent. We give a new pre-processing strategy which computes and stores such jump points for every node on the map. Our third improvement is motivated by the observation that some jump points are simply intermediary locations on the grid. We give a new pruning strategy that avoids expanding such nodes.

There are several interesting directions for further work. One possibility is stronger pruning rules that will allow us to jump over some of the remaining nodes in the graph. For example: we might consider pruning a node n if all of its successors have an f-value that is not larger than $f(n)$. A stronger variant of this idea is to keep jumping as long as we are heading in the same direction as when we reached n — or in a new direction which is a component of the one used to reach n. It is likely that this procedure will increase the branching factor at n but we posit that fewer node expansions will be required overall because we do not need to stop each time the path turns due to an obstacle.

Combinations of JPS with existing grid-based speedup techniques appears to be another fruitful direction for further research. A number of well-known speedup techniques, both optimal and sub-optimal, work by limiting the scope of grid search to a corridor of nodes relevant for the instance at hand; e.g. Swamps (Pochter et al. 2010), HPA* (Botea, Müller, and Schaeffer 2004) or any number of pruning-based heuristics e.g. (Björnsson and Halldórsson 2006; Goldenberg et al. 2010). JPS can be trivially combined with such approaches to further speed up search.

One strength of the JPS family is that it performs very well even online. We have shown however that pre-computed jump points accelerate significantly the search. It is possible to combine both approaches, by populating a database with jump points as they are discovered online. We want to apply this feature in dynamic environments where obstacles appear or disappear, immediately rendering any pre-processed information obsolete.

Acknowledgements

We thank Patrik Haslum for taking the time to read and comment on an early revision of this paper. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

Antsfeld, L.; Harabor, D. D.; Kilby, P.; and Walsh, T. 2012. Transit routing on video game maps. In *AIIDE*.

Bast, H.; Funke, S.; and Matijevic, D. 2006. Transit ultrafast shortest-path queries with linear-time preprocessing. In *In 9th DIMACS Implementation Challenge*.

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. In *AIIDE*, 9–14.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *J. Game Dev.* 1(1):7–28.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 319–333.

Goldenberg, M.; Felner, A.; Sturtevant, N.; and Schaeffer, J. 2010. Portal-based true-distance heuristics for path finding. In *SoCS*.

Harabor, D., and Grastien, Al. 2011. Online graph pruning for pathfinding on grid maps. In *AAAI*.

Lozano-Pérez, T., and Wesley, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM* 22(10):560–570.

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

Storandt, S. 2013. The hierarchy in grid graphs. In *SoCS*.

Sturtevant, N. R., and Geisberger, R. 2010. A comparison of high-level approaches for speeding pathfinding. In *AIIDE*, 76–82.

Uras, T.; Koenig, S.; and Hernàndez, C. 2013. Subgoal graphs in for optimal pathfinding in eight-neighbour grids. In *ICAPS*.