Online Graph Pruning for Pathfinding on Grid Maps

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

Abstract

Pathfinding in uniform-cost grid environments is a problem commonly found in application areas such as robotics and video games. The state-of-the-art is dominated by hierarchical pathfinding algorithms which are fast and have small memory overheads but usually return suboptimal paths. In this paper we present a novel search strategy, specific to grids, which is fast, optimal and requires no memory overhead. Our algorithm can be described as a macro operator which identifies and selectively expands only certain nodes in a grid map which we call *jump points*. Intermediate nodes on a path connecting two jump points are never expanded. We prove that this approach always computes optimal solutions and then undertake a thorough empirical analysis, comparing our method with related works from the literature. We find that searching with jump points can speed up A* by an order of magnitude and more and report significant improvement over the current state of the art.

Introduction

Widely employed in areas such as robotics (Lee and Yu 2009), artificial intelligence (Wang and Botea 2009) and video games (Davis 2000; Sturtevant 2007), the ubiquitous undirected uniform-cost grid map is a highly popular method for representing pathfinding environments. Regular in nature, this domain typically features a high degree of path symmetry (Harabor and Botea 2010; Pochter et al. 2010). Symmetry in this case manifests itself as paths (or path segments) which share the same start and end point, have the same length and are otherwise identical save for the order in which moves occur. Unless handled properly, symmetry can force search algorithms to evaluate many equivalent states and prevents real progress toward the goal.

In this paper we deal with such path symmetries by developing a macro operator that selectively expands only certain nodes from the grid, which we call *jump points*. Moving from one jump point to the next involves travelling in a fixed direction while repeatedly applying a set of simple neighbour pruning rules until either a dead-end or a jump point is reached. Because we do not expand any intermediate nodes between jump points our strategy can have a dramatic positive effect on search performance. Furthermore, computed solutions are guaranteed to be optimal. Jump point pruning is fast, requires no preprocessing and introduces no memory overheads. It is also largely orthogonal to many existing speedup techniques applicable to grid maps.

We make the following contributions: (i) a detailed description of the jump points algorithm; (ii) a theoretical result which shows that searching with jump points preserves optimality; (iii) an empirical analysis comparing our method with two state-of-the-art search space reduction algorithms. We run experiments on a range of synthetic and real-world benchmarks from the literature and find that jump points improve the search time performance of standard A* by an order of magnitude and more. We also report significant improvement over Swamps (Pochter et al. 2010), a recent optimality preserving pruning technique, and performance that is competitive with, and in many cases dominates, HPA* (Botea, Müller, and Schaeffer 2004); a well known sub-optimal pathfinding algorithm.

Related Work

Approaches for identifying and eliminating search-space symmetry have been proposed in areas including planning (Fox and Long 1999), constraint programming (Gent and Smith 2000), and combinatorial optimization (Fukunaga 2008). Very few works however explicitly identify and deal with symmetry in pathfinding domains such as grid maps.

Empty Rectangular Rooms (Harabor and Botea 2010) is an offline symmetry breaking technique which attempts to redress this oversight. It decomposes grid maps into a series of obstacle-free rectangles and replaces all nodes from the interior of each rectangle with a set of macro edges that facilitate optimal travel. Specific to 4-connected maps, this approach is less general than jump point pruning. It also requires offline pre-processing whereas our method is online.

The *dead-end heuristic* (Björnsson and Halldórsson 2006) and *Swamps* (Pochter et al. 2010) are two similar pruning techniques related to our work. Both decompose grid maps into a series of adjacent areas. Later, this decomposition is used to identify areas not relevant to optimally solving a particular pathfinding instance. This objective is similar yet orthogonal to our work where the aim is to reduce the effort required to explore any given area in the search space.

A different method for pruning the search space is to identify *dead* and *redundant* cells (Sturtevant, Bulitko, and

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Björnsson 2010). Developed in the context of learning-based heuristic search, this method speeds up search only after running multiple iterations of an iterative deepening algorithm. Further, the identification of redundant cells requires additional memory overheads which jump points do not have.

Fast expansion (Sun et al. 2009) is another related work that speeds up optimal A* search. It avoids unnecessary open list operations when it finds a successor node just as good (or better) than the best node in the open list. Jump points are a similar yet fundamentally different idea: they allow us to identify large sets of nodes that would be ordinarily expanded but which can be skipped entirely.

In cases where optimality is not required, hierarchical pathfinding methods are pervasive. They improve performance by decomposing the search space, usually offline, into a much smaller approximation. Algorithms of this type, such as HPA* (Botea, Müller, and Schaeffer 2004), are fast and memory-efficient but also suboptimal.

Notation and Terminology

We work with undirected uniform-cost grid maps. Each node has ≤ 8 neighbours and is either traversable or not. Each straight (i.e. horizontal or vertical) move, from a traversable node to one of its neighbours, has a cost of 1; diagonal moves cost $\approx \sqrt{2}$. Moves involving non-traversable (obstacle) nodes are disallowed. The notation $\vec{d}$ refers to one of the eight allowable movement directions (up, down, left, right etc.). We write $y = x + k\vec{d}$ when node y can be reached by taking k unit moves from node x in direction $\vec{d}$. When $\vec{d}$ is a diagonal move, we denote the two straight moves at 45 deg to $\vec{d}$ as $\vec{d_1}$ and $\vec{d_2}$.

A path $\pi = \langle n_0, n_1, \ldots, n_k \rangle$ is a cycle-free ordered walk starting at node n_0 and ending at n_k. We will sometimes use the setminus operator in the context of a path: for example $\pi \setminus x$. This means that the subtracted node x does not appear on (i.e. is not mentioned by) the path. We will also use the function len to refer the length (or cost) of a path and the function $dist$ to refer to the distance between two nodes on the grid: e.g. $len(\pi)$ or $dist(n_0, n_k)$ respectively.

Jump Points

In this section we introduce a search strategy for speeding up optimal search by selectively expanding only certain nodes on a grid map which we term *jump points*. We give an example of the basic idea in Figure 1(a).

Here the search is expanding a node x which has as its parent $p(x)$; the direction of travel from $p(x)$ to x is a straight move to the right. When expanding x we may notice that there is little point to evaluating any neighbour highlighted grey as the path induced by such a move is always dominated by (i.e. no better than) an alternative path which mentions $p(x)$ but not x. We will make this idea more precise in the next section but for now it is sufficient to observe that the only non-dominated neighbour of x lies immediately to the right. Rather than generating this neighbour and adding it to the open list, as in the classical A* algorithm,

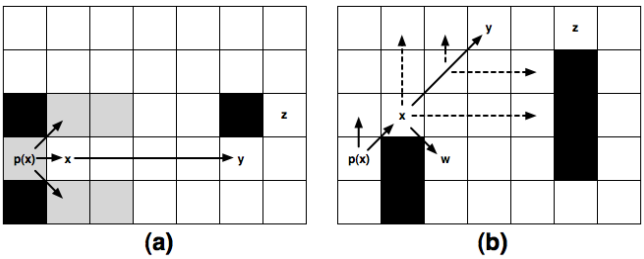

Figure 1: Examples of straight (a) and diagonal (b) jump points. Dashed lines indicate a sequence of interim node evaluations that reached a dead end. Strong lines indicate eventual successor nodes.

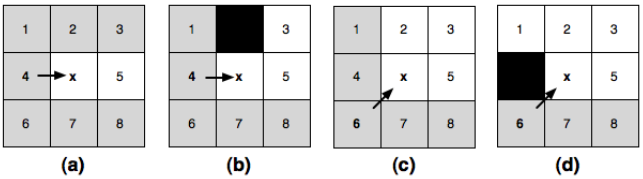

Figure 2: We show several cases where a node x is reached from its parent $p(x)$ by either a straight or diagonal move. When x is expanded we can prune from consideration all nodes marked grey.

we propose to simply step to the right and continue moving in this direction until we encounter a node such as y; which has at least one other non-dominated neighbour (here z). If we find a node such as y (a jump point) we generate it as a successor of x and assign it a g-value (or cost-so-far) of $g(y) = g(x) + dist(x, y)$. Alternatively, if we reach an obstacle we conclude that further search in this direction is fruitless and generate nothing.

In the remainder of this section we will develop a macro-step operator which speeds up node expansion by identifying jump point successors in the case of both straight and diagonal moves. First it will be necessary to define a series of pruning rules to determine whether a node should be generated or skipped. This will allow us to make precise the notion of a jump point and give a detailed description of the jump points algorithm. Then, we prove that the process of "jumping" nodes, such as x in Figure 1(a), has no effect on the optimality of search.

Neighbour Pruning Rules

In this section we develop rules for pruning the set of nodes immediately adjacent to some node x from the grid. The objective is to identify from each set of such neighbours, i.e. $neighbours(x)$, any nodes n that do not need to be evaluated in order to reach the goal optimally. We achieve this by comparing the length of two paths: π, which begins with node $p(x)$ visits x and ends with n and another path π' which also begins at node $p(x)$ and ends with n but does not mention x. Additionally, each node mentioned by either π or π' must belong to $neighbours(x)$.

There are two cases to consider, depending on whether the transition to x from its parent $p(x)$ involves a straight move

or a diagonal move. Note that if x is the start node $p(x)$ is null and nothing is pruned.

Straight Moves: We prune any node $n \in neighbours(x)$ which satisfies the following dominance constraint:

$$len(\ \langle p(x), \ldots, n\rangle \setminus x\ ) \leq len(\ \langle p(x), x, n\rangle\ ) \qquad (1)$$

Figure 2(a) shows an example. Here $p(x) = 4$ and we prune all neighbours except $n = 5$.

Diagonal Moves: This case is similar to the pruning rules we developed for straight moves; the only difference is that the path which excludes x must be strictly dominant:

$$len(\ \langle p(x), \ldots, n\rangle \setminus x\ ) < len(\ \langle p(x), x, n\rangle\ ) \qquad (2)$$

Figure 2(c) shows an example. Here $p(x) = 6$ and we prune all neighbours except $n = 2$, $n = 3$ and $n = 5$.

Assuming $neighbours(x)$ contains no obstacles, we will refer to the nodes that remain after the application of straight or diagonal pruning (as appropriate) as the *natural neighbours* of x. These correspond to the non-gray nodes in Figures 2(a) and 2(c). When $neighbours(x)$ contains an obstacle, we may not be able to prune all non-natural neighbours. If this occurs we say that the evaluation of each such neighbour is *forced*.

Definition 1. *A node $n \in neighbours(x)$ is* forced *if:*

1. n is not a natural neighbour of x

2. $len(\ \langle p(x), x, n\rangle\ ) < len(\ \langle p(x), \ldots, n\rangle \setminus x\ )$

In Figure 2(b) we show an example of a straight move where the evaluation of $n = 3$ is forced. Figure 2(d) shows an similar example involving a diagonal move; here the evaluation of $n = 1$ is forced.

Algorithmic Description

We begin by making precise the concept of a jump point.

Definition 2. *Node y is the* jump point *from node x, heading in direction $\vec{d}$, if y minimizes the value k such that $y = x + k\vec{d}$ and one of the following conditions holds:*

1. *Node y is the goal node.*
2. *Node y has at least one neighbour whose evaluation is forced according to Definition 1.*
3. *$\vec{d}$ is a diagonal move and there exists a node $z = y + k_i\vec{d}_i$ which lies $k_i \in \mathbf{N}$ steps in direction $\vec{d}_i \in \{\vec{d}_1, \vec{d}_2\}$ such that z is a jump point from y by condition 1 or condition 2.*

Figure 1(b) shows an example of a jump point which is identified by way of condition 3. Here we start at x and travel diagonally until encountering node y. From y, node z can be reached with $k_i = 2$ horizontal moves. Thus z is a jump point successor of y (by condition 2) and this in turn identifies y as a jump point successor of x

The process by which individual jump point successors are identified is given in Algorithm 1. We start with the pruned set of neighbours immediately adjacent to the current node x (line 2). Then, instead of adding each neighbour n to the set of successors for x, we try to "jump" to a node that is further away but which lies in the same relative direction to x as n (lines 3:5). For example, if the edge (x, n) constitutes a straight move travelling *right* from x, we look for a jump point among the nodes immediately to the right of x. If we find such a node, we add it to the set of successors instead of n. If we fail to find a jump point, we add nothing. The process continues until the set of neighbours is exhausted and we return the set of successors for x (line 6).

In order to identify individual jump point successors we will apply Algorithm 2. It requires an initial node x, a direction of travel $\vec{d}$, and the identities of the start node s and the goal node g. In rough overview, the algorithm attempts to establish whether x has any jump point successors by stepping in the direction $\vec{d}$ (line 1) and testing if the node n at that location satisfies Definition 2. When this is the case, n is designated a jump point and returned (lines 5, 7 and 11). When n is not a jump point the algorithm recurses and steps again in direction $\vec{d}$ but this time n is the new initial node (line 12). The recursion terminates when an obstacle is encountered and no further steps can be taken (line 3). Note that before each diagonal step the algorithm must first fail to detect any straight jump points (lines 9:11). This check corresponds to the third condition of Definition 2 and is essential for preserving optimality.

Algorithm 1 Identify Successors

Require: x: current node, s: start, g: goal

1: $successors(x) \leftarrow \emptyset$
2: $neighbours(x) \leftarrow prune(x, neighbours(x))$
3: **for all** $n \in neighbours(x)$ **do**
4: $\quad n \leftarrow jump(x, direction(x, n), s, g)$
5: $\quad$ add n to $successors(x)$
6: **return** $successors(x)$

Algorithm 2 Function $jump$

Require: x: initial node, $\vec{d}$: direction, s: start, g: goal

1: $n \leftarrow step(x, \vec{d})$
2: **if** n is an obstacle or is outside the grid **then**
3: $\quad$ **return** *null*
4: **if** $n = g$ **then**
5: $\quad$ **return** n
6: **if** $\exists\, n' \in neighbours(n)$ s.t. n' is forced **then**
7: $\quad$ **return** n
8: **if** $\vec{d}$ is diagonal **then**
9: $\quad$ **for all** $i \in \{1, 2\}$ **do**
10: $\quad\quad$ **if** $jump(n, \vec{d}_i, s, g)$ is not *null* **then**
11: $\quad\quad\quad$ **return** n
12: **return** $jump(n, \vec{d}, s, g)$

Optimality

In this section we prove that for each optimal length path in a grid map there exists an equivalent length path which can be found by only expanding jump point nodes during search (Theorem 1). Our result is derived by identifying for each optimal path a symmetric alternative which we split into contiguous segments. We then prove that each *turning point* along this path is also a jump point.

Definition 3. *A* turning point *is any node n_i along a path where the direction of travel from the previous node n_{i-1} to n_i is different to the direction of travel from n_i to the subsequent node n_{i+1}.*

Figure 3 depicts the three possible kinds of turning points which we may encounter on an optimal path. A diagonal-to-diagonal turning point at node n_k (Figure 3(a)) involves a diagonal step from its parent n_{k-1} followed by a second diagonal step, this time in a different direction, from n_k to its successor n_{k+1}. Similarly, a straight-to-diagonal (or diagonal-to-straight) turning point involves a straight (diagonal) step from n_{k-1} to reach n_k followed by a diagonal (straight) step to reach its successor n_{k+1} (Figure 3(b) and 3(c) respectively). Other types of turning points, such as straight-to-straight, are trivially suboptimal and not considered here (they are pruned by the rules we developed earlier; see again Figure 2).

We are now ready to develop an equivalence relation between jump points and the turning points that appear along certain optimal length symmetric paths which we term *diagonal-first*.

Definition 4. *A path π is diagonal-first if it contains no straight-to-diagonal turning point $\langle n_{k-1}, n_k, n_{k+1}\rangle$ which could be replaced by a diagonal-to-straight turning point $\langle n_{k-1}, n'_k, n_{k+1}\rangle$ s.t. the length of π remains unchanged.*

Given an arbitrary optimal length path π, we can always derive a symmetric diagonal-first path π' by applying Algorithm 3 to π. Note that this is merely as a conceptual device.

Lemma 1. *Each turning point along an optimal diagonal-first path π' is also a jump point.*

Proof. Let n_k be an arbitrary turning point node along π'. We will consider three cases, each one corresponding to one of the three possible kinds of optimal turning points illustrated in Figure 3.

Diagonal-to-Diagonal: Since π' is optimal, there must be an obstacle adjacent to both n_k and n_{k-1} which is forcing a detour. We know this because if there were no obstacle we would have $dist(n_{k-1}, n_{k+1}) < dist(n_{k-1}, n_k) + dist(n_k, n_{k+1})$ which contradicts the fact that π' is optimal. We conclude that n_{k+1} is a forced neighbour of n_k. This is sufficient to satisfy the second condition of Definition 1, making n_k a jump point.

Algorithm 3 Compute Diagonal First Path

Require: π: an arbitrary optimal length path

1: **select** an adjacent pair of edges appearing along π where (n_{k-1}, n_k) is a straight move and (n_k, n_{k+1}) is a diagonal move.
2: **replace** (n_{n-1}, n_k) and (n_k, n_{k+1}) with two new edges: (n_{k-1}, n'_k), which is a diagonal move and (n'_k, n_{k+1}) which is a straight move. The operation is successful if (n_{k-1}, n'_k) and (n'_k, n_{k+1}) are both valid moves; i.e. node n'_k is not an obstacle.
3: **repeat** lines 1 and 2, selecting and replacing adjacent edges, until no further changes can be made to π.
4: **return** π

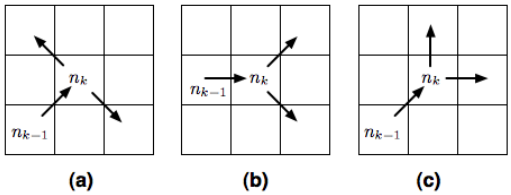

Figure 3: Types of optimal turning points. (a) Diagonal-to-Diagonal (b) Straight-to-Diagonal (c) Diagonal-to-Straight.

Straight-to-Diagonal: In this case there must be an obstacle adjacent to n_k. If this were not true n_k could be replaced by a Diagonal-to-Straight turning point which contradicts the fact that π' is diagonal-first. Since π' is guaranteed to be diagonal-first we derive the fact that n_{k+1} is a forced neighbour of n_k. This satisfies the second condition of Definition 1 and we conclude n_k is a jump point.

Diagonal-to-Straight: There are two possibilities in this case, depending on whether the goal is reachable by a series of straight steps from n_k or whether π' has additional turning points. If the goal is reachable by straight steps then n_k has a jump point successor which satisfies the third condition of Definition 1 and we conclude n_k is also a jump point. If n_k is followed by another turning point, n_l, then that turning point must be Straight-to-Diagonal and, by the argument for that case, also a jump point. We again conclude that n_k has a jump point successor which satisfies the third condition of Definition 1. Thus, n_k is also a jump point. □

Theorem 1. *Searching with jump point pruning always returns an optimal solution.*

Proof. Let π be an arbitrarily chosen optimal path between two nodes on a grid and π' a diagonal-first symmetric equivalent which is derived by applying Algorithm 3 to π. We will show that every turning point mentioned by π' is expanded optimally when searching with jump point pruning. We argue as follows:

Divide π' into a series of adjacent segments s.t. $\pi' = \pi'_0 + \pi'_1 + \ldots + \pi'_n$. Each $\pi'_i = \langle n_0, n_1, \ldots, n_{k-1}, n_k\rangle$ is a subpath along which all moves involve travelling in the same direction (e.g. only "up" or "down" etc). Notice that with the exception of the start and goal, every node at the beginning and end of a segment is also a turning point.

Since each π'_i consists only of moves in a single direction (straight or diagonal) we can use Algorithm 2 to jump from $n_0 \in \pi'_i$, the node at beginning of each segment to $n_k \in \pi'_i$, the node at the end, without necessarily stopping to expand every node in between. Intermediate expansions may occur but the fact that we reach n_k optimally from n_0 is guaranteed. It remains to show only that both n_0 and n_k are identified as jump points and thus necessarily expanded. By Lemma 1 each turning point along π' is also a jump point, so every turning point node must be expanded during search. Only the start and goal remain. The start node is necessarily expanded at the beginning of each search while the goal node is a jump point by definition. Thus both are expanded. □

Experimental Setup

We evaluate the performance of jump point pruning on four benchmarks taken from the freely available pathfinding library Hierarchical Open Graph (HOG, `http://www.googlecode.com/p/hog2`):

- **Adaptive Depth** is a set of 12 maps of size 100×100 in which approximately $\frac{1}{3}$ of each map is divided into rectangular rooms of varying size and a large open area interspersed with large randomly placed obstacles. For this benchmark we randomly generated 100 valid problems per map for a total of 1200 instances.
- **Baldur's Gate** is a set of 120 maps taken from BioWare's popular roleplaying game *Baldur's Gate II: Shadows of Amn*; it appears regularly as a standard benchmark in the literature (Björnsson and Halldórsson 2006; Harabor and Botea 2010; Pochter et al. 2010). We use the variaton due to Nathan Sturtevant where all maps have been scaled to size 512×512 to more accurately represent modern pathfinding environments. Maps and all instances are available from `http://movingai.com`
- **Dragon Age** is another realistic benchmark; this time taken from BioWare's recent roleplaying game *Dragon Age: Origins*. It consists of 156 maps ranging in size from 30×21 to 1104×1260. For this benchmark we used a large set of randomly generated instances, again due to Nathan Sturtevant and available from `http://movingai.com`.
- **Rooms** is a set of 300 maps of size 256×256 which are divided into symmetric rows of small rectangular areas (7×7), connected by randomly placed entrances. This benchmark has previously appeared in (Pochter et al. 2010). For this benchmark we randomly generated 100 valid problems per map for a total of 30000 instances.

Our test machine is a 2.93GHz Intel Core 2 Duo processor with 4GB RAM running OSX 10.6.4.

Results

To evaluate jump points we use a generic implementation of A* which we adapted to facilitate online neighbour pruning and jump point identification. We discuss performance in terms of *speedup*: i.e. relative improvement to the time taken to solve a given problem, and the number of nodes expanded, when searching with and without graph pruning applied to the grid. Using this metric a search time speedup of 2.0 is twice as fast while a node expansion speedup of 2.0 indicates half the number of nodes were expanded. In each case higher is better. Figure 4 shows average search time speedups across each of our four benchmarks. Table 1 shows average node expansion speedups; the best results for each column are in bold.

Comparison with Swamps: We begin by comparing jump points to Swamps (Pochter et al. 2010): an optimality preserving pruning technique for speeding up pathfinding. We used the authors' source code, and their implementation of A*, and ran all experiments using their recommended running parameters: a swamp seed radius of 6 and "no change limit" of 2.

	A. Depth	B. Gate	D.Age	Rooms
Jump Points	**20.37**	**215.36**	**35.95**	**13.41**
Swamps	1.89	2.44	2.99	4.70
HPA*	4.14	9.37	9.63	5.11

Table 1: Average A* node expansion speedup.

As per Figure 4, jump point pruning shows a convincing improvement to average search times across all benchmarks. The largest differences are observed on Baldur's Gate and Dragon Age where searching with jump points reaches the goal between 25-30 times sooner while searching with Swamps attains only a 3-5 times speedup. Similar trends are observed when looking at Table 1, where the improvement to the total number of nodes expanded is even more pronounced.

Based on these results we conclude that while Swamps are effective for identifying areas of the map not relevant to reaching the goal, those areas which remain still require significant effort to search. Jump points use a much stronger yet orthogonal strategy to prove that many nodes expanded by Swamps can be ignored. As the two ideas appear complementary we posit that they could be easily combined: first, apply a Swamps-based decomposition to prune areas not relevant to the current search. Then, use jump points to search the remaining portions of the map.

Comparison with HPA*: Next, we compare jump points pruning to the HPA* algorithm (Botea, Müller, and Schaeffer 2004). Though sub-optimal, HPA* is very fast and widely applied in video games. To evaluate HPA* we measure the total cost of insertion and hierarchical search. We did not refine any abstract paths, assuming instead that a database of pre-computed intra-cluster paths was available. Such a configuration represents the fastest generic implementation of HPA* but requires additional memory overheads. While searching we used a single-level abstraction hierarchy and a fixed cluster size of 10. These settings are recommended by the original authors who note that larger clusters and more levels are often of little benefit.

As per Figure 4, jump points pruning is shown to be highly competitive with HPA*. On Adaptive Depth, Baldur's Gate and Rooms, jump points have a clear advantage – improving on HPA* search times by several factors in some cases. On the Dragon Age benchmark jump points have a small advantage for problems of length < 500 but for longer instances there is very little between the two algorithms. Table 1 provides further insight: although searching with jump points expands significantly fewer nodes than HPA*, each such operation takes longer.

We conclude that jump point pruning is a competitive substitute for HPA* and, for a wide variety of problems, can help to find solutions significantly faster. HPA* can still be advantageous, particularly if a memory overhead is acceptable and optimality is not important, but only if the cost of inserting the start and goal nodes into the abstract graph (and possibly refinement, if a path database is not available), can be sufficiently amortized over the time required to find an abstract path. One direction for further work could be to use

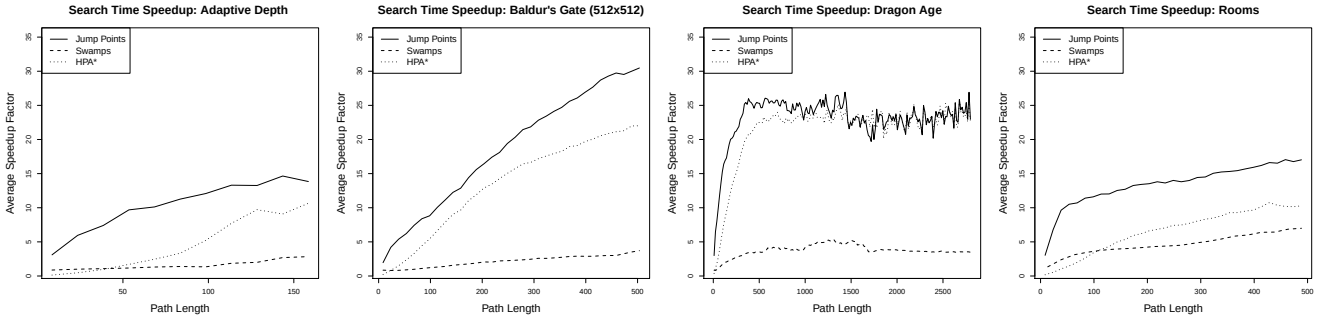

Figure 4: Average A* search time speedup on our each of our four benchmarks.

jump point pruning to speed up HPA*: for example during insertion and refinement.

Conclusion

We introduce a new online node pruning strategy for speeding up pathfinding on undirected uniform-cost grid maps. Our algorithm identifies and selectively expands only certain nodes from a grid map which we call *jump points*. Moving between jump points involves only travelling in a fixed direction, either straight or diagonal. We prove that intermediate nodes on a path between two jump points never need to be expanded and "jumping" over them does not affect the optimality of search.

Our method is unique in the pathfinding literature in that it has very few disadvantages: it is simple, yet highly effective; it preserves optimality, yet requires no extra memory; it is extremely fast, yet requires no preprocessing. Further, it is largely orthogonal to and easily combined with competing speedup techniques from the literature. We are unaware of any other algorithm which has all these features.

The new algorithm is highly competitive with related works from the literature. When compared to Swamps (Pochter et al. 2010), a recent state-of-the-art optimality preserving pruning technique, we find that jump points are up to an order of magnitude faster. We also show that jump point pruning is competitive with, and in many instances clearly faster than, HPA*; a popular though sub-optimal pathfinding technique often employed in performance sensitive applications such as video games.

One interesting direction for further work is to extend jump points to other types of grids, such as hexagons or texes (Yap 2002). We propose to achieve this by developing a series of pruning rules analogous to those given for square grids. As the branching factor on these domains is lower than square grids, we posit that jump points could be even more effective than observed in the current paper. Another interesting direction is combining jump points with other speedup techniques: e.g. Swamps or HPA*.

Acknowledgements

We would like to thank Adi Botea, Philip Kilby and Patrik Haslum for their encouragement, support and many helpful suggestions during the development of this work. We also thank Nir Pochter for kindly providing us with source code for the Swamps algorithm.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

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.

Davis, I. L. 2000. Warp speed: Path planning for Star Trek Armada. In *AAAI Spring Symposium (AIIDE)*, 18–21.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 956–961.

Fukunaga, A. S. 2008. Integrating symmetry, dominance, and bound-and-bound in a multiple knapsack solver. In *CPAIOR*, 82–96.

Gent, I. P., and Smith, B. M. 2000. Symmetry breaking in constraint programming. In *ECAI*, 599–603.

Harabor, D., and Botea, A. 2010. Breaking path symmetries in 4-connected grid maps. In *AIIDE*, 33–38.

Lee, J.-Y., and Yu, W. 2009. A coarse-to-fine approach for fast path finding for mobile robots. In *IROS*, 5414 –5419.

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

Sturtevant, N. R.; Bulitko, V.; and Björnsson, Y. 2010. On learning in agent-centered search. In *AAMAS*, 333–340.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.

Sun, X.; Yeoh, W.; Chen, P.-A.; and Koenig, S. 2009. Simple optimization techniques for a*-based search. In *AAMAS (2)*, 931–936.

Wang, K.-H. C., and Botea, A. 2009. Tractable multi-agent path planning on grid maps. In *IJCAI*, 1870–1875.

Yap, P. 2002. Grid-based path-finding. In *Canadian AI*, 44–55.