JUMP POINT SEARCH ALGORITHM PERFORMANCE ANALYSIS

Дусеев В.Р., Рудь М.Н., Мальчуков А.Н.
Научный руководитель: Мальчуков А.Н., к.т.н. доцент
Томский политехнический университет, 634050, Россия, г. Томск, пр. Ленина, 30
E-mail: vagiz.d@gmail.com

Jump Point Search (JPS) [1] is a unique online symmetry-breaking algorithm, which speeds up path finding on uniform-cost grid maps by "jumping over" many locations that would otherwise need to be explicitly considered. JPS is faster and more powerful than RSR: it can consistently speed up A* search by over an order of magnitude and more. Unlike other similar algorithms JPS requires no preprocessing and has no memory overheads. Further, it is easily combined with most existing speedup techniques — including abstraction and memory heuristics.

The Algorithm

This section and the next describe the mechanical details and algorithmic properties of Jump Point Search. A set of figures is provided in order to explain the algorithm`s behavior.

JPS [1] can be described in terms of two simple pruning rules which are applied recursively during search: one rule is specific to straight steps, the other for diagonal steps. The key intuition in both cases is to prune the set of immediate neighbours around a node by trying to prove that an optimal path (symmetric or otherwise) exists from the parent of the current node to each neighbour and that path does not involve visiting the current node. Figure 1 outlines the basic idea.

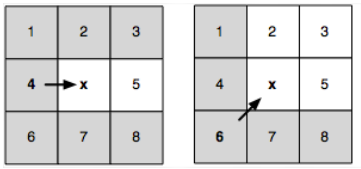

Figure 1. Neighbour Pruning

Node x is currently being expanded. The arrow indicates the direction of travel from its parent, either straight or diagonal. In both cases we can immediately prune all grey neighbours as these can be reached optimally from the parent of x without ever going through node x.

We will refer to the set of nodes that remain after pruning as the natural neighbours of the current node. These are marked white in Figure 1. Ideally, we only ever want to consider the set of natural neighbours during expansion. However, in some cases, the presence of obstacles may mean that we need to also consider a small set of up to k additional nodes ($0 \leq k \leq 2$). We say that these nodes are forced neighbours of the current node. Figure 2 gives an overview of this idea.

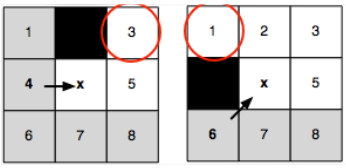

Figure 2. Forced Neighbours

Node x is currently being expanded. The arrow indicates the direction of travel from its parent, either straight or diagonal. Notice that when x is adjacent to an obstacle the highlighted neighbours cannot be pruned; any alternative optimal path, from the parent of x to each of these nodes, is blocked [2].

We apply these pruning rules during search as follows: instead of generating each natural and forced neighbour we instead recursively prune the set of neighbours around each such node. Intuitively, the objective is to eliminate symmetries by recursively "jumping over" all nodes which can be reached optimally by a path that does not visit the current node. We stop the recursion when we hit an obstacle or when we find a so-called jump point successor. Jump points are interesting because they have neighbours that cannot be reached by an alternative symmetric path: the optimal path must go through the current node.

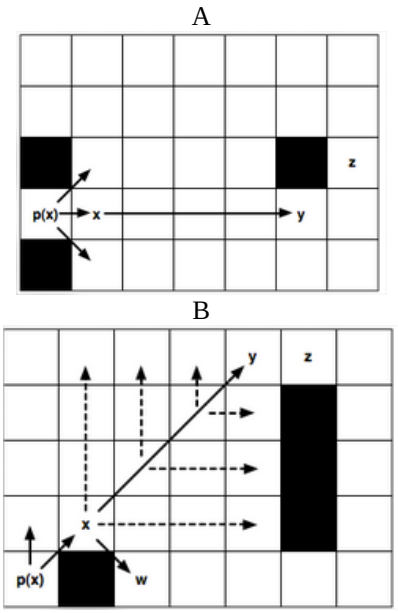

Figure 3. (A) jumping straight; (B) jumping diagonally

The details of the recursive pruning algorithm are reasonably straightforward: to ensure optimality we need only assign an ordering to how we process natural neighbours (straight steps before diagonal). I will not attempt to outline it further here; the full details are in the paper and my aim is only to provide

a flavour for the work. Figures 3 gives two examples of the pruning algorithm in action. In the first case we identify a jump point by recursing straight; in the second case we identify a jump point by recursing diagonally.

Node x is currently being expanded. p(x) is its parent.

(A) We recursively apply the straight pruning rule and identify y as a jump point successor of x. This node is interesting because it has a neighbour z that cannot be reached optimally except by a path that visits x then y. The intermediate nodes are never explicitly generated or even evaluated.

(B) We recursively apply the diagonal pruning rule and identify y as a jump point successor of x. Notice that before each diagonal step we first recurse straight (dashed lines). Only if both straight recursions fail to identify a jump point do we step diagonally again. Node w, which is simply a forced neighbour of x, is generated as normal.

Properties and Performance

Jump Point Search is nice for a number of reasons:

- It is optimal.
- It involves no pre-processing.
- It requires no extra-memory overheads.
- It can consistently speed up A* search by over 10 times; making it not only competitive with, but often better than, approximate techniques such as HPA* [3].

Properties 1-3 are interesting theoretical results, and rather surprising, but I will not address them further here. My main objective in this article is simply provide a flavour for the work; for a full discussion, including proofs, please see the original paper [1]. Property 4 is perhaps of broadest practical interest so I will give a short summary of our findings below.

We evaluated JPS on four map sets taken from Nathan Sturtevant's freely available pathfinding library, Hierarchical Open Graph. Two of the benchmarks are realistic, originating from popular BioWare video games Baldur's Gate II: Shadows of Amn and Dragon Age: Origins. The other two Adaptive Depth and Rooms are synthetic though the former could be described as semi-realistic. In each case we measured the relative speedup of A* + JPS vs. A* alone.

Briefly: JPS can speed up A* by a factor of between 3-15 times (Adaptive Depth), 2-30 times (Baldur's Gate), 3-26 times (Dragon Age) and 3-16 times (Rooms). In each case the lower figure represents average performance for short pathfinding problems and the higher figure for long pathfinding problems (i.e. the longer the optimal path to be found, the more benefit is derived from applying JPS).

What makes these results even more compelling is that in 3 of the 4 benchmarks A* + JPS was able to consistently outperform the well known HPA* algorithm [3]. This is remarkable as A* + JPS is always performing optimal search while HPA* is only performing approximate search. On the remaining benchmark, Dragon Age, we found there was very little to differentiate the performance of the two algorithms.

Caveat emptor: It is important to highlight at this stage that A* + JPS only achieves these kids of speedups because each benchmark problem set contains a large number of symmetric path segments (usually manifested in the form of large open areas on the map). In such cases JPS can exploit the symmetry and ignore large parts of the search space. This means A* both generates and expands a much smaller number of nodes and consequently reaches the goal much sooner. When there is very little symmetry to exploit however we expect that our gains will be more modest.

Final Thoughts

The explicit identification and elimination of symmetries in pathfinding domains is an idea that until now has received little attention in the academic literature. Approaches for dealing with symmetry, such as Jump Point Search, provide us with powerful new tools for reducing the size of explicit regular search spaces. By eliminating symmetry we speed up not just A* but entire classes of similar pathfinding algorithms. Also, consider: JPS is entirely orthogonal to almost every other speedup technique applicable to grid maps. Thus, there is no reason why we couldn't combine it, or other similar methods, with hierarchical pathfinding approaches, memory heuristics or even other optimality-preserving state-space reduction techniques. That means the results presented thus far are only the tip of the iceberg in terms of performant grid-based pathfinding methods.

Another exciting aspect of this work is the possibilities it opens for further research. For example: could we pre-process the map and go even faster? Or: are there analogous jumping rules that one could develop for weighted grids? What about other domains? Could we apply the lessons learned thus far to help solve other interesting search problems? The answers to the first two questions already appear to be positive; the third is something we want to explore in the near future.

References

1. D. Harabor and A. Grastien. Online Graph Pruning for Pathfinding on Grid Maps. In National Conference on Artificial Intelligence (AAAI), 2011.
2. D. Harabor, A. Botea, and P. Kilby. Path Symmetries in Uniform-cost Grid Maps. In Symposium on Abstraction Reformulation and Approximation (SARA), 2011.
3. A. Botea, M. Müller, and J. Schaeffer. Near Optimal Hierarchical Path-finding. In Journal of Game Development (Issue 1, Volume 1), 2004.