Directional Arc Pruning in BLJPS Version 5

? [1]

Abstract. Improving search speed is critical for many areas, such as game AI and robotics. This paper presents substantial improvements on the Boundary Lookup Jump Point Search (BLJPS) algorithm. The paper presents increased utilization of pre-processing time and memory to increase search speed. We present an overview of the overall algorithm and details of the pruning mechanism. Performance of variants of this algorithm are evaluated.

A comparison with a node pruning algorithm is also presented. The paper shows the given approach out performs the node pruning approach in three out of four cases.

1 Introduction

Many path finding algorithms currently exist that are capable of searching grid space [2, 6, 10]. The goal of these algorithms is generally to search for an optimal path between two points on a map. Pathfinding is used within many applications such as game AI and robotics. When such applications can rely on lower search times they become more responsive and can further utilise the saved CPU overhead for their own purposes.

However, this speedup of search time generally comes at the cost of pre-processing time, memory and storage space. Such as Compressed Path Databases (CPD) [2, 3] which takes hours to preprocess maps and takes gigabytes of storage to perform extremely fast queries. However, because of the large amount of memory and storage requirements these algorithms are not well suited for mobile devices and certain applications. These requirements increase substantially when multiple maps are required and/or they consist of a high level of detail.

The swamps algorithm identifies areas that paths are unlikely to travel through, and speeds up run time searches by ignoring these areas [7, 8]. Hierarchical approaches such as [4] trade off path optimality in return for speed by pre-processing the environment into hierarchical regions and interconnecting paths.

Other algorithms such as BLJPS [10] and SubGoal [11] do very well within a smaller memory, storage and pre-processing footprint. SubGoal has been shown to be very efficient at pruning a maps nodes to improve search speed.

With this design in mind we present an arc pruning algorithm designed to exploit the direction expanding nature of BLJPS. We show how such an approach can achieve higher search speeds with a trade-off on memory and pre-processing time. Our results show higher search speeds for this approach named BLJPS5 and it has superior search speeds on three out of four map datasets relative to SubGoal Optimal (2013).

Section 3 gives a brief description of the variants of the BLJPS algorithm, and how they achieve greater search speeds relative to their predesessor. Note that each version is built on the previous version so a steady increase in search speed is found with increasing cost of memory and pre-processing time.

[1] ?, ?, email: ? @?.edu

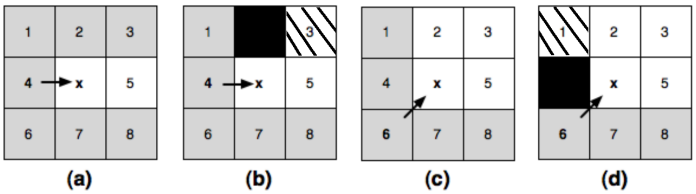

Figure 1. Forced Neighbours. White cells (Natural Neighbours). Grey cells (Culled nodes). Black cells (Blocked). Striped cells (Forced Neighbours).

2 Jump Point Search (JPS)

Jump Point Search (JPS) [6] is a path finding approach that achieves excellent speedup results without pre-processing the environment. JPS expands only certain nodes called jump points during searches, drastically reducing the cost of the search. JPS is a path finding algorithm that exploits path symmetry in uniform-cost grid maps in order to prune search paths at runtime. It returns optimal paths significantly faster than A* search [5].

JPS starts by considering all neighbouring nodes from the starting point. If JPS identifies a neighbour in a particular direction as the start of a possible path, it continues to extend that path in that direction until it is blocked by an obstacle or identifies a jump point. If a path is blocked, then all nodes along that expansion direction are pruned from further consideration.

While JPS is extending a path in a particular direction, it identifies a set of natural neighbours for a node under evaluation. A natural neighbour is defined by the direction of expansion. Expansions in a cardinal direction define their natural neighbour as the next node in the same direction (The Eastern natural neighbour is to the East). When expansion is diagonal, the set of natural neighbours includes only 3 nodes: the next node along the diagonal of the expansion, and the next vertical and horizontal nodes, also in the direction of expansion. Nodes are expanded in diagonal directions after considering the vertical and horizontal expansions until they are either blocked or a forced neighbour is found. A forced neighbour is identified when a neighbouring cell is blocked, which allows for a change of expanding direction that is not within the set of natural neighbours (see examples of Figure 1). If a forced neighbour is found then the node is identified as a jump point. This point represents the first node in a particular search direction from the original node that must consider expanding in other directions. [6].

Figures within figure 1 show how natural and forced neighbours are evaluated given the direction of travel (Diagram modified from

Table 1. Maps used in experiments. DAO maps vary in size, ranging from the smallest and largest map sizes given. Each map is associated with a varying number of paths, the average for each map set is given.

Map	Number of Maps	Map Size	Average Number of paths per map
Baldur's Gate (BG)	75	512x512	1242
Dragon Age Origins (DAO)	156	Smallest:28x22 Largest:104x1260	1022
Rooms (Rooms)	10	512x512	1928
Adaptive Depth (AD)	12	100x100	1000

[6]). Forced neighbours are striped cells, natural neighbours are clear white cells and blocked cells are black. All the grey cells represent nodes culled from search. Figure 1(a) shows that with no blocked cells then only the natural neighbour is expanded (East to 5). Figure 1(b) shows that a blocked cell to the North creates a forced neighbour at (North-East 3) in addition to the natural neighbour. Figure 1(c) demonstrates that diagonal directions have 3 natural neighbours, 1 diagonal and 2 respective cardinal (NE 3, North 2 and East 5). Figure 1(d) demonstrates that a forced node exists at (NW 1) in addition to its natural neighbours. These cases are symmetrical and can be rotated/flipped to generalize to all local scenarios.

3 Boundary Lookup Jump Point Search (BLJPS)

Improvement over Jump Point Search (JPS) [6]. Instead of iteratively searching over grid locations it pre-processes the horizontal and vertical axis into boundary lists to be accessed directly. Thus each diagonal step requires three axis look ups on the horizontal and vertical axis. Further details on BLJPS can be found at [10].

3.1 BLJPS2, BLJPS3 and BLJPS4

BLJPS2 improves speeds by storing the jump point locations along the cardinal directions. This modification reduces the number of vertical and horizontal axis lookups to 1 each per diagonal step.

BLJPS3 stores the locations of jump points along the diagonal directions. Thus it requires only a single lookup for a diagonal expansion, once again moving away from iterative stepping.

BLJPS4 creates a node corresponding to each forced neighbour on the map. Each of these nodes stores a list of nodes it is connected to in each possible direction of travel which are referred to as arcs. A search starts by adding all the relevant forced neighbours reachable from the start point to an open list. As each node is popped from the list it performs a quick check to determine if that node can move directly to the destination position, if it can be then the solution has been found. Otherwise it expands the node's arcs in the direction of travel and iterates until the open list is empty or the solution is found.

3.2 BLJPS5

This algorithm creates a backwards lookup table for incoming arcs from jump points. It also prunes arcs between nodes in a process that will be described below. When a search is initiated all the start nodes are added, like BLJPS4, but in addition all the identified end nodes are flagged. Now as the search takes place a trivial check against a node's flagged state will determine if a goal state has been reached. Thus this approach incurs a one off cost of determining the end nodes at the start of the search relative to BLJPS4 that incurs this cost at check every node expansion (thus causing it to run slower on longer more complex paths but faster on simpler paths).

BLJPS5 prunes arcs between nodes, which works in contrast to SubGoal's approach of pruning entire nodes from the graph. BLJPS5 uses two copies of the arcs for this pruning method. One acts as a read only list of all the original arcs while the second is modified during the following process. For each arc in the read only list a series of possible following arcs are identified. Where an arc comes to a node it is expanded by its natural and forced neighbour directions. If the originating node of the tested arc can connect to all the identified outgoing arcs, the arc is pruned and all the identified outgoing arcs take its place on the originating node. Thus in essence skipping the tested node and moving directly onto the next nodes during a search.

All nodes that have an arc pruned have the arc added to an incoming pruned arc list. This is used when flagging end nodes at the start of a search. Due to the pruning process an end node may have been pruned from the graph in a given direction. Therefore all directly connected end nodes are expanded in forced neighbour directions adding nodes from the incoming pruned arc list. So search can find an end-node whether it was pruned or not. Due to this process, many unnecessary nodes are skipped during search and determining if the destination has been reached is simplified to a flag check.

4 Experimental Approach

The experiments evaluate the improvements of the BLJPS algorithms and secondarily evaluates the BLJPS5 arc pruning algorithm against SubGoal. SubGoal Optimal (2013) [11] was chosen because of its quick search times and relatively small memory and pre-processing imprints using a node pruning approach. SubGoal's source was taken from [1]. The experiments of JPS [6] were replicated, using the freely distributed datasets from the Grid-Based Path Planning Competition (GPPC) [1]. The source code for these experiments is publicly available on GitHub [9]. Each map includes a set of predefined problem paths that are used to benchmark the algorithms.

The experiments were conducted using diagonal unblocked mode so SubGoal and BLJPS variants would have comparable results. This mode differs from the earlier explanation of JPS where in both cardinal directions associated with a diagonal step must be obstacle free to be legal. This unblocked mode is used in the GPPC [1] but differs from the original implementation of JPS [6].

Table 1 lists the maps used in the experiments. They were consistent in size except for the DAO maps which varied, ranging between the specified smallest and largest sizes. Each map was associated with a varying number of paths. The average number of paths for each map set is listed.

1. Adaptive Depth (AD): Relatively small and have relatively few obstacles.
2. The Baldur's Gate (BG): Large size and generally have very simple layouts with large open areas.
3. The Dragon Age Origins (DAO): Diverse in size and obstacle density.
4. The Rooms maps are large (Rooms): Extremely high density of obstacles.

The performance of BLJPS, BLJPS2, BLJPS3, BLJPS4, BLJPS5 and

Table 2. Experimental Results: BLJPS5 vs SubGoal.

Map	Algorithm	Total Search Time(ms)	Average Pre-Processing Time(ms)	Average Starting Memory(Kb)	Average Max Memory(Kb)
AD	BLJPS5	**109**	16	553	583
	SubGoal	119	**9.2**	**157**	**186**
BG	BLJPS5	**591**	**99**	2660	2703
	SubGoal	2174	178	**1347**	**1422**
DAO	BLJPS5	**6685**	**47**	3518	3668
	SubGoal	10378	142	**1741**	**1858**
Rooms	BLJPS5	8737	415	16330	17955
	SubGoal	**5596**	**42**	**2562**	**2863**

SubGoal were evaluated. On each map, all paths provided in the data set were executed giving paths of varying lengths. Optimal paths between start and end points were then generated repeatedly until each had been calculated at least 100 times or the run had taken at least 5ms, whichever took longer. This method allowed for accurate evaluation of the time taken to find a path. Taking into account finding short paths where 100 solutions were processed in too small a time window to accurately measure were repeated until 5ms passed. In contrast long paths that took an extended time to process (particularly in the case of the A* algorithm) were capped after 100 repetitions. The time taken to find a path was then calculated as the average search time over the number of iterations.

The time spent pre-processing the maps, was measured, the memory usage after completing pre-processing and the maximum memory used after completing all searches was recorded. These results are summarized in Tables 2 and 3.

5 Results and Discussion

A brief summary of the Tables 2 and 3 are as follows:

1. BLJPS5 has a faster search time than SubGoal in 3 out of 4 cases.
2. BLJPS5 is faster than BLJPS.
3. BLJPS5 occasionally pre-processes maps faster than SubGoal.
4. BLJPS5 is more memory intensive than SubGoal.

The results from Table 2 show an overall faster search speed in the maps AD, BG and DAO in favour of BLJPS5. It however does not perform as well as SubGoal on the obstacle dense map set of Rooms.

Table 3 shows BLJPS5 was substantially superior in search speed relative to BLJPS. It is worth remarking that BLJPS4 outperformed BLJPS5 on the maps AD and BG due to the relatively simple paths on these maps that didn't require the computational overhead of BLJPS5 (see section BLJPS5). On more complex maps, DAO and Rooms, the performance improvement is clearly seen.

Figure 2 shows a detailed graph of the performance of each algorithm relative to A* with respect to path lengths. The large spikes in performance for BLJPS3, 4 and 5 at shorter distances represent paths that pass around only a single corner. A function finds and returns these paths before a search takes place. The check is inexpensive and substantially speeds up searches that only have a single corner between start and end points. This performance gain was found throughout the BG testing maps, which resulted in the relatively quick search times.

BLJPS5 incurs a larger memory footprint relative to SubGoal but remains within a 20mb boundary, which is generally adequate for most applications. In respect to pre-processing time, SubGoal pre-processed faster within Rooms and AD maps, while BLJPS5 pre-processed the maps BG and DAO quicker. However, as the average pre-processing time for all cases is under 0.5 seconds it is again usable for most applications.

6 Conclusion and Further Work

The presented algorithm BLJPS5 substantially increased search speed and was able to outperform SubGoal in three out of four data sets. Future work may look to extending the BLJPS5 algorithm to further prune connections between forced neighbours. Further effort could also be made into reducing the search speed on obstacle dense maps such as the Rooms data set.

REFERENCES

[1] 'Grid-based path planning competition'. http://movingai.com/GPPC.

[2] Adi Botea, 'Ultra-fast optimal pathfinding without runtime search.', in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, (2011).

[3] Adi Botea, 'Fast, optimal pathfinding with compressed path databases.', in *Symposium on Combinatorial Search (SOCS)*, (2012).

[4] Adi Botea, Martin Müller, and Jonathan Schaeffer, 'Near optimal hierarchical path-finding', *Journal of Game Development*, **1**(1), 7–28, (2004).

[5] Xiao Cui and Hao Shi, 'A*-based pathfinding in modern computer games', *International Journal of Computer Science and Network Security*, **11**(1), 125–130, (2011).

[6] Daniel Damir Harabor and Alban Grastien, 'Online graph pruning for pathfinding on grid maps.', in *Association for the Advancement of Artificial Intelligence (AAAI)*, (2011).

[7] Nir Pochter, Aviv Zohar, and Jeffrey S Rosenschein, 'Using swamps to improve optimal pathfinding', in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pp. 1163–1164. International Foundation for Autonomous Agents and Multiagent Systems, (2009).

[8] Nir Pochter, Aviv Zohar, Jeffrey S Rosenschein, and Ariel Felner, 'Search space reduction using swamp hierarchies', in *Third Annual Symposium on Combinatorial Search*, (2010).

[9] Jason Traish, 'Bljps source code'. https://github.com/narsue/BLJPS5, (2015).

[10] Jason Traish, James Tulip, and Wayne Moore, 'Optimization using boundary lookup jump point search', *IEEE Transactions on Computational Intelligence and AI in Games*, **PP**(99), 1–1, (2015).

[11] Tansel Uras, Sven Koenig, and Carlos Hernández, 'Subgoal graphs for optimal pathfinding in eight-neighbor grids.', in *International Conference on Automated Planning and Scheduling (ICAPS)*, (2013).

Table 3. Experimental Results: BLJPS variations. Bold represents the best result.

Map	Algorithm	Total Search Time(ms)	Average Pre-Processing Time(ms)	Average Starting Memory(Kb)	Average Max Memory(Kb)
AD	BLJPS	546	**0.25**	**8**	**36**
	BLJPS2	331	0.67	20	56
	BLJPS3	225	7.3	174	209
	BLJPS4	**79**	12	280	316
	BLJPS5	109	16	553	583
BG	BLJPS	8270	**3.4**	**60**	**110**
	BLJPS2	4371	4.6	127	177
	BLJPS3	2030	51	556	610
	BLJPS4	**569**	90	1766	1800
	BLJPS5	591	99	2660	2703
DAO	BLJPS	78743	**1.8**	**49**	**124**
	BLJPS2	51927	2.9	119	193
	BLJPS3	41753	21	665	745
	BLJPS4	9839	34	2304	2342
	BLJPS5	**6685**	47	3518	3668
Rooms	BLJPS	105053	**5.6**	**365**	**572**
	BLJPS2	64421	38	605.2	807
	BLJPS3	61036	194	7466	7684
	BLJPS4	10335	314	11105	11219
	BLJPS5	**8737**	415	16330	17955

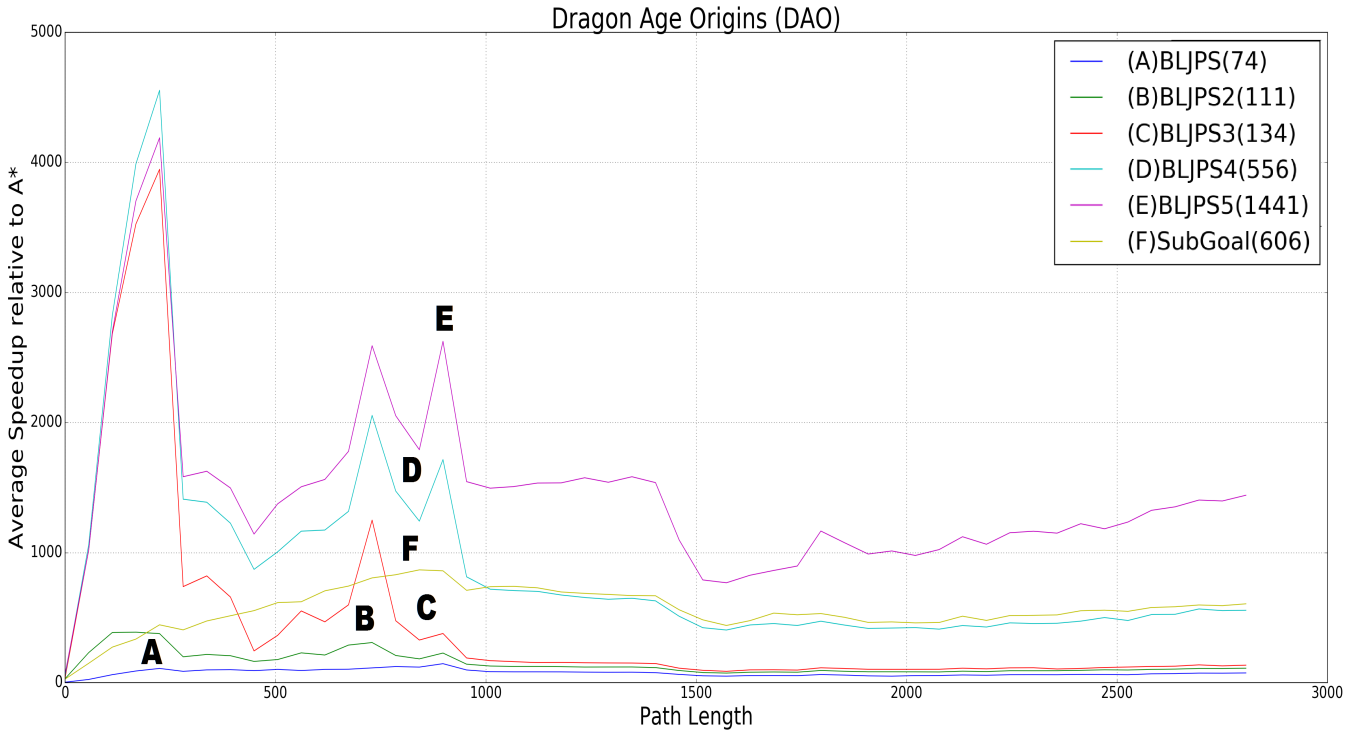

Figure 2. DAO Experiment: Average speedup relative to A* against path length.