Comparison of JPS+ and JPS

Reza Mashayekhi
Course instructor: Prof. Nathan Sturtevant
Report fo Single Agent Search course at UoA

Abstract

Pathfinding on a grid with uniform-cost edges is being used in video games and robotics. Many algorithms like A* exists for optimal pathfinding. Jump Point Search (JPS) is an optimal algorithm that has been shown to perform well by doing pruning on A* and having jump points as successors. Also, JPS+, a state-of-the-art algorithm, has improved the performance of JPS by using offline preprocessing and an online manipulating of a block of bits at the same time. In this report, the performance of JPS and JPS+ has been compared and interpreted.

Introduction

Pathfinding is finding a path from the start to the goal. Pathfinding on a grid is popular in robotics and video game development. Classical algorithms like A* generate all combinations of the paths of the same length during the search, so there is symmetry in these algorithms, which is time-consuming. However, JPS, by having a canonical ordering (diagonal moves have higher priority than cardinal moves) and choosing only one path among all paths with the same length, avoids symmetry. Therefore, JPS spends more time on generating successors rather than manipulating the open list. However, JPS has an underlying time-consuming operation which is exploring the grid during the search for jump points, but since it reduces the open list size, in practice, it has been shown to perform better.

JPS+ with three enhancements to JPS has improved the performance: (i) a block-based search that reads more cells at the same time rather than reading only one cell. (ii) a preprocessing part that computes jump points for each cell before the search. (iii) An improved pruning rule prunes some of the jump points.

Related Work

Many algorithms exist in pathfinding since it's applicable in different areas like road networking, video games, and robotics.

TRANSIT (Bast, Funke, and Matijevic 2006) and Contraction Hierarchies (Geisberger et al. 2008) are both using preprocessing for computing the nodes with the most shortest paths passing through them and then doing the online search using the preprocessed information. These algorithms are useful in road networks which have nodes with very high reach, so they act like a highway in roads. However, they haven't been performing well on grids.

Swamps is another algorithm which in preprocessing part, detects the parts of the map that aren't a part of the shortest path unless the start/goal is there. By pruning many nodes, it reduces the search space.

SUB (Uras, Koenig, and Hernandez 2013) is a state-of-the-art algorithm that creates a large subgoal graph by putting the subgoals at the corner of obstacles and connecting the ones that are easily reachable and necessary for finding the shortest path.

However, in video game companies, since it's not necessary to find an optimal path, it is usual to use abstraction and refinement, which is a suboptimal algorithm. In the preprocessing part, it creates an abstraction of the map, then in the online search, it produces the path in the abstract map, then refines it, which results in finding the actual path.

JPS

Algorithm 1: A*(State start, State goal)

```
open=[];
add start to open;
closed=[];
while open is not empty do
    extract the state in open with the minimum f cost
      called n;
    add n to closed
    if (n is goal): return the goal
    successors=successors(n);
    for each successor in n called s:
    ...if s was in closed: skip
    ...else if s was in open with higher f cost: update s
    ...else add s to open
end
return not found the goal
```

JPS uses the A* algorithm with a few pruning rules, which reduces the size of the open list while generating and ex-

ploring the map more. First, we should know the canonical ordering. Since there are many shortest paths with the same length, we can consider only one of them at first. So, in all paths with the same length, we prioritize the one that has its diagonal moves before the cardinal ones. Figure 1 shows the priority of the same length possible paths.

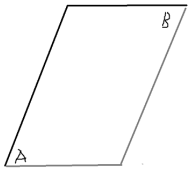

Figure 1: Canonical ordering: The black path from A to B has higher priority than the gray path from A to B because all the diagonals are before the cardinals.

Jump Points: in a horizontal move from p to x, when there is an obstacle in the up (or down) neighbor of p state, but after the move, there isn't an obstacle in the up(or respectively down) neighbor of the x, we call x a jump point as shown in figure 2.

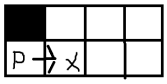

Figure 2: X is a jump point in a move from P to X

We have jump points for the vertical moves too, but there isn't any jump point for diagonal moves.

The algorithm of the JPS is based upon jump points and canonical ordering. JPS is an A* algorithm with successors being these jump points. So during the search, most of the time is spent on finding these jump points. For finding the jump points for a state, the algorithm does a canonical ordering on the whole map, and it searches along each path until it reaches a jump point or the goal or a dead-end (obstacle or end of the map) in the path it was searching. Doing the search with respect to canonical ordering in a map would look like figure3.

Figure 3: Finding jump points for a state by doing search with respect to canonical ordering. Picture from movingai.com

Jump points store the direction they should search for when they are being expanded. For example, if a jump point is reached by an upward vertical move and it has no obstacle on its right, it should search in the North, North East, and East directions when it's being expanded. The direction it should search is the move that was being reached by + the side that it has no obstacle + the diagonal move combination of the two previous moves as shown in figure 4.

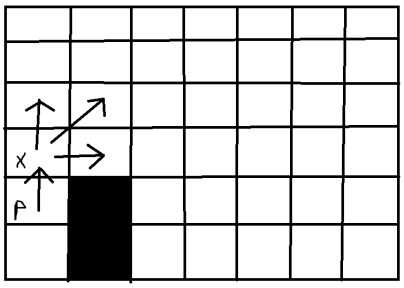

Figure 4: X is a jump point reached from P with an upward move. Since it has been reached by an upward move and its right neighbor has no obstacle, it should search in North which is the direction it was reached by + the East direction because it has no obstacle on the right side + the diagonal North East direction which is the combination of the two previous moves

Algorithm 2: Succesors(State s)

```
successors=[];
based on s direction, do the search from s in
  canonical ordering until it reaches a jump point, the
  goal, or dead end on each path:
...if it reached a jump point or the goal, add the jump
  point/goal to the successors;
return successors;
```

The time complexity is O$[b^d]$ like the A* algorithm where b is the branching factor and d is the length of the optimal path to the goal.

JPS+

JPS+ has three improvements: (i)Block-based Symmetry Breaking, (ii) Preprocessing, and (iii) Improved Pruning Rules. We explain each in more detail.

(i) Block-based Symmetry Breaking

The grid is encoded into bits where each bit represents each cell. 1 means that cell has an obstacle, and 0 for no obstacles. Since we are looking for jump points in the cardinal moves and bits are stored as bytes, we can read 8 bits together and therfore, speed up the process of reading and writing to memory instead of looking at each cell separately like what has been done in the JPS. We also store a rotation of the map for traversing the grid vertically too for detecting jump points in vertical moves.

(ii) Preprocessing

In this part, we find the intermediate jump points for each cell offline and store them. So in online search, instead of looking at the map for jump points which is time-consuming, we already know where the intermediate jump points are. Therefore even in our search for jump points, we jump instead of traversing cell by cell.

Intermediate Jump points: Each cell has eight intermediate jump points, one for each of 8 directions. For detecting the intermediate jump point in diagonal directions, we traverse diagonally from the cell, and if there is a jump point in the direction of the diagonal move on the same row or column, we stop and call it the intermediate jump point. If there wasn't any jump point and we reached an obstacle, we store it again, but we call it sterile. These sterile intermediate jump points are useful for finding the goal and are not added to the open list during the search.

In cardinal directions, if there was a jump point on the same row(or column, depending on the direction), we call it intermediate jump point, and if there wasn't any jump point and we reached the obstacle, we store it as intermediate jump point and call it sterile. These intermediate jump points are shown in figure 5.

For detecting the goal in online search, we compare the goal's row and column to the cell and its intermediate jump point to see whether there is an intersection or not. If there was an intersection, we traverse from the goal cardinally to see whether it's reachable or not, like in figure 6.

(iii) Improved Pruning Rules

The jump points here are exactly the jump points introduced in JPS in this project. The JPS paper version of jump points is the jump points introduced in preprocessing part except for the sterile ones. There are many intermediate jump points in the paper version of JPS for a single jump point in improved pruning. These intermediate jump points can be generated during the search and be added to the open list, while the improved pruning only adds one single jump point for all those intermediate jump points, which speeds up the A* algorithm used in JPS.

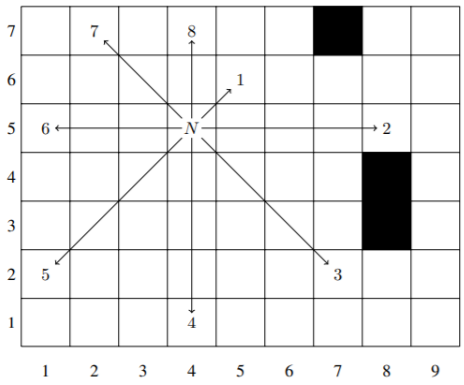

Figure 5: All 8 intermediate jump point for cell N is shown above. 4, 5 ,6, 7, and 8 are sterile.

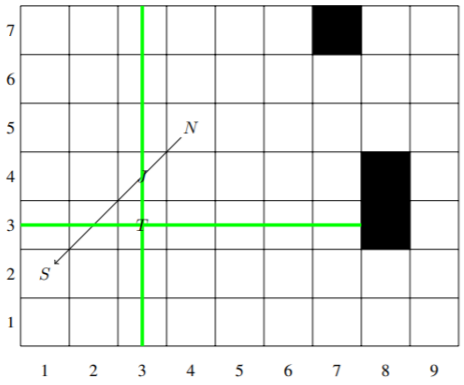

Figure 6: It is detecting the goal during the online search by looking for the intersection of the goal with the intermediate jump point and the cell which is being traversed. N is the cell being traversed, S is one of its intermediate jump points, and T is the goal.

Results

Both of these algorithms have been implemented and been tested on Dragon Ages 2 maps. The interesting point about the result is the maximum time in JPS which is much higher than JPS+. As shown in figure 7, JPS has the maximum value of 134336 microseconds while JPS+ has the maximum value of 27441 microseconds which is a huge difference. Also, JPS+ has an average of 353 microseconds while JPS has an average of 423 which is a great improvement. Since in video games, there are many queries for each agent, time is really important and even 70 microsecond on average improvement can be useful.

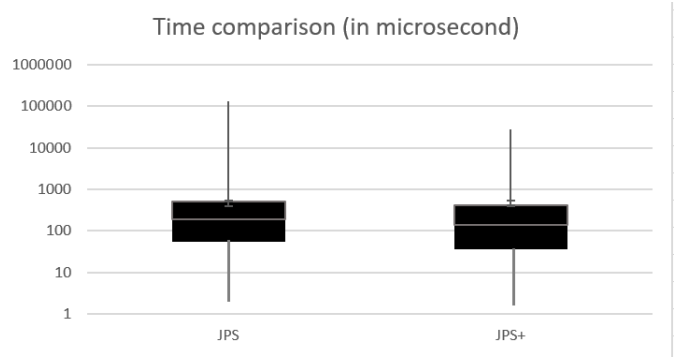

Figure 7: Time comparison of JPS and JPS+

Also in this implementation block based symmetry hasn't been implemented, so the performance of JPS+ could be even better.

challenges

Aside from many small bugs, the main challenge was combining the improved pruning with the preprocessing part. I also used the canonical heuristic instead of the arctile, which is more accurate. Breaking the tie to state with higher g cost improved performance. Also, two times I was forced to write the successor function again (which was really time-consuming) because I was forgetting to add directions or wasn't doing the improved pruning.

References

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

Harabor, D., and Grastien, 2014. Improving Jump Point Search. In *Proceedings of the International Conference on Automated Planning and Scheduling*

Sturtevant, N. R.; 2D benchmark of Dragon Age II. movingai.com

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

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

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

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