An optimization of path planning A* for static uniform grid based on pruning algorithms: Experimental experience

Mohammed Hammoud, Sergey Lupin

***Abstract* — Path-finding in uniform-cost grid environments is a popular task in different applications, like and video games, and robotics. In this project, several classical algorithms are presented and their work is explained, such as A*, Dijkstra, and Wave-front algorithm. A novel search strategy called Jump Point Search which uses pruning to decrease the discovered space is also presented. Jump Point Search is a designed optimal and fast algorithm for grids with no memory overhead. Moreover, Jump Point Search improvement will be discussed together with JPS+. We will use a benchmark to evaluate each of mentioned algorithms using different criteria, such as operation time, the number of visited nodes, and path length. Our environment will be a 2D uniform static grid. The aim of this article is to investigate the performance of several grid-based path-finding algorithms. We find that JPS produces the same path length as A* but with dramatically decreasing in time.**

Keywords— Artificial Intelligence, path planning algorithms, Pruning algorithms, Robotics, Game development.

I. Introduction

Path planning is finding a visible collision-free path that will lead the agent from the initial to the target configuration. It defines the next action to do from the current state. The algorithm selects the best next state to move to from all potential states. This decision is made according to some function of criteria, usually defined using one of the distance measures, such as the shortest Euclidean distance to the target state.

Zero or multiple paths can be existing between certain initial and target states connecting the states. There are usually several possible paths (that is, paths that do not encounter obstacles). Criteria used in the evaluation of planning algorithms:

- Completeness: the ability to always find the solution if it exists and correctly report an error when there is none.
- Cost Optimization: is there a solution with the lowest path cost among all solutions?
- Time Complexity: the time required to calculate the path if it exists.
- Space Complexity: it defines the required memory to find the solution.
- Path Length: According to these criteria, the main goal is to get the shortest possible path. This length can be calculated with different formulas (Euclidean, Manhattan, etc.).
- Path Smoothness(PS): the path should be free from sharp turns, in other words, the goal of smoothness is to have a straight path as much as possible, and this will help minimize power consumption since turns on a straight path use much less memory than a curvy path. The smoothness of the path can be calculated using the following equation(1):

$$PS = \sum_i (180 - \phi_i) \quad (1)$$

- Detach from obstacles: the calculated path must be kept as far away from obstacles as possible.

In this work, we compared the classical algorithm (A*) with the Jump Point Search algorithm (JPS). In the next sections, we will briefly cover the classical algorithms, and in detail the JPS.

II. Literature review

A. Dijkistra

Dijkstra's Algorithm calculates all shortest paths from a given initial node to all other nodes in a fully connected graph. This algorithm requires information about the relative distance between all nodes Distances must not be negative.

B. A*

This is a search for the best first [9], which uses an evaluation function, refer to equation (2):

$$f(n) = g(n) + h(n) \quad (1)$$

where g(n) is the cost of the path from the start state to node n, and h(n) is the heuristic which estimates cost of the shortest path from n to the target state, so we have f(n) equals the estimated the cost of the best path that continues from n to the target. This approach allows the algorithm to distinguish between more or less promising nodes and, as a result, find a solution more efficiently. This heuristic function can be the Euclidean distance or the Manhattan distance (sum of vertical and horizontal displacements) from the current node to the target node. Algorithm A* is complete because it finds the optimal path if it exists and if the heuristic is feasible (optimistic). Its disadvantage is high memory usage. If all costs to reach the goal are set to zero, operation A* is equal to Dijkstra's algorithm (h(n) = 0).

C. Jump Point Search

JPS is an optimized search algorithm for improving optimal search Aa* by expanding special nodes, called jump points [3] from the grid based on some rules. JPS excludes symmetric paths from a uniform 8-connected grid and provides one path between the destination and the target.

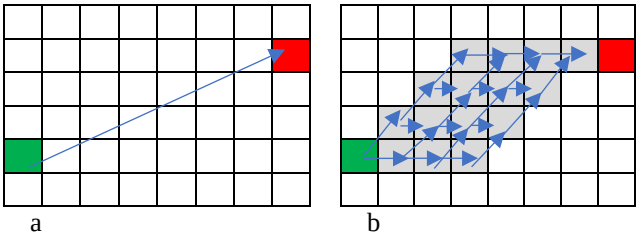


Fig. 1 Symmetries paths from start to goal in grid environment

Fig. 1 represents many equivalent best paths through this rectangular area. These tracks are symmetrical (costs the same).

A* adds nearest neighbor nodes to the set of what to explore next. JPS identifies situations where path symmetry is present and ignores certain nodes as we expand our search, as shown in Fig. 2.

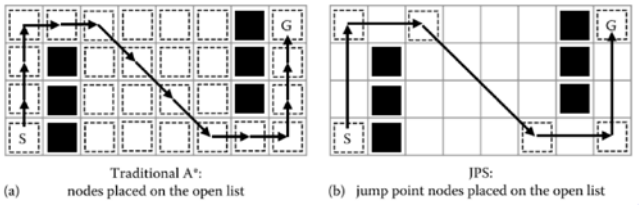


Fig. 2 Difference between A* and JPS

JPS can recognize symmetric paths and ignore all but one, while A* adds all neighbors to the open list, as in the classic A* algorithm. JPS suggests moving right and keeps moving in that direction until we find a node y with at least one more neighbor without dominance, as shown in Fig. 3.

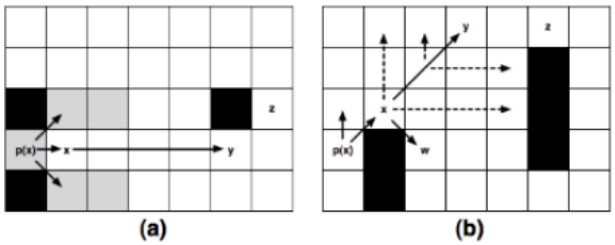


Fig. 3 JPS rules

1) Work principle of JPS

The main idea is to minimize the immediate neighbors around a node. The goal is to prove the presence of an optimal path between the node's parent and each neighbor, which doesn't pass through the current node. Jump points are intermediate points on the map through which a minimum of one optimal path must be passed.

2) Pruning rules

For a given node x accessed through a parent node p(x), suppose π' = (p, ..., n) - path without x, π = (p, x, n) - path with x. We truncate some neighbor's n of x, if one of two rules is satisfied:

- In the case of horizontal and vertical movement, there is a path shorter than one pass through current node x as shown in equation (3)

$$\text{length}(\pi') \leq \text{length}(\pi) \quad (2)$$

- π' has a diagonal move earlier than π, and equation (4) is satisfied.

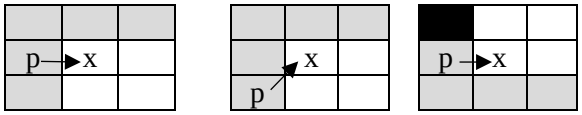
$$\text{length}(\pi') = \text{length}(\pi) \quad (3)$$

3) Jump points

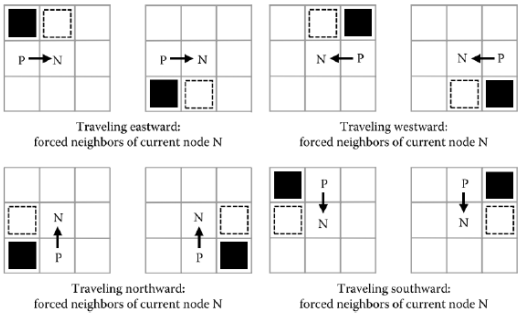
JPS determines node x as the jump point on the following rules:

1) x is a start or a target node.
2) x has a minimum of one forced neighbor.
3) The search direction from p(x) to x is diagonal and the other directions (vertical and horizontal) met the first two rules Fig. 3b.

There are two sets of rules: pruning rules and jumping rules. Looking to Fig. 4b, node x is currently being expanded. The direction of movement from the parent of node x is either horizontal, vertical, or diagonal.



(a) straight, diagonal, and forced neighbor



(b) The eight forced neighbor cases [2].

Fig. 4 Work principles of JPS

Since there is an optimal path from p(x) to the gray marked neighbors of x without passing through x, we can prune the gray-marked neighbors. The remaining neighbors after pruning is called natural neighbors of x (marked in white). The existence of obstacles signals that it is important to consider a set of size k, where ($0 \leqslant k \leqslant 2$). These nodes represent forced neighbors of the current node that only occur when moving in a forward or cardinal direction (north, south, west, or east). Forced neighbors signal that the normal pruning strategy will fail and that the current node must consider additional nodes.

Now we will explain the work of the JPS algorithm in the case illustrated in Fig. 5.

1) Starting with a single node in the open queue, expand vertically and find nothing (Fig. 5a).
2) Expanding horizontally, we find a node with a forced neighbor (highlighted in purple). Add this node to the open set (Fig. 5b).
3) Expand diagonally, finding nothing because we run into the edge of the map (Fig. 5c).
4) Explore the next best one (from the open node). Since we were moving horizontally, when we reached this node, we continue to jump horizontally (Fig. 5d).
5) Since there is a forced neighbor, it is necessary to expand in this direction. Following the rules of diagonal jumps, we move diagonally, then look both vertically and horizontally (Fig. 5e).

6) Found nothing, we again move diagonally (Fig. 5f).
7) This time, after expanding horizontally (nowhere to go) and vertically, we see the target node. It is no less interesting than finding a node with a forced neighbor, so we add this node to the open set (Fig. 5g).
8) Expanding the last open node, reaching the goal (Fig. 5a).
9) By skipping the last iteration of the algorithm – adding the target itself to the open set only to recognize it as such - we found a better way (Fig. 5i).

D. *Jump Point Search improvement*

JPS spends most of its time searching the grid for successors instead of manipulating nodes from lists. JPS has been enhanced using various techniques such as block-based symmetry breaking and improvement of cleaning rules. The majority of jump points are target independent. The improved JPS version uses a new preprocessing that calculates and stores jump points of nodes on the map.

Such block-based operations allow applying the pruning rules of JPS to multiple nodes simultaneously. This will allow the grid to be scanned faster which increases the overall pathfinding performance. To achieve this, the grid must be encoded as a matrix of bits. Each bit corresponds to a unique location and implies the nodes that can be traversed.

In JPS we will scan horizontally then vertically then diagonally (just one node at a time), in the order shown with the red numbers. While new JPS scan several nodes at the same time.

JPS performs searching iteratively along a given row or column, and it stops in case finding a forced neighbor in an adjacent row or detecting a dead-end or the target node in the current row. The stopping condition can be validated using an operation on B ↑, BN, and B ↓. This algorithm is shown below (applied at the same time on the example shown in Fig. 7, where w = 8).

1) Get B_N by reading w nodes from N.
   B_N = [0, 0, 0, 0, 0, 1, 0, 0], Dead node exists at:
   b_N = ffs(B_N) − 1 = 4
   where ffs stands for find the first set
2) Get B ↑ by reading w nodes, from the previous row for node N
   B ↑= [0, 0, 0, 0, 0, 0, 0, 0]
3) Get B ↓ (B_d)by reading w nodes, from the previous row for node N
   B ↓= [0, 0, 1, 1, 0, 0, 0, 0]
4) Detect dead node exists in position i of byte B in case B_i = 0 and B_{i+1} = 1. In other words, this can be done, as shown in equation (5):
   $$b_N = \text{find first set}(B_N) - 1 \quad (4)$$
   b_N= 5 − 1 = 4
5) Detect forced neighbors, B_i is a forced neighbor if B_i is traversable and B_{i-1} is not(obstacles). This is done using equation (6):
   $$\text{forced(B)} = (\text{B} \ll 1)\ \&!\ \text{B} \quad (5)$$
6) Find stop byte B_s, using equation (7).
   $$B_s = \text{forced}(B_y) | \text{forced}(B_d) | B_N \quad (6)$$
   B_s = [0, 0, 0, 0, 1, 1, 0, 0]

B_s= 0 → The search has to stop. Since B_N If bs = 0, jump w−1 nodes and repeat, else search has to stop. If b_s is less than b_N, then there is a jump point at B_N [b_s− 1] else we hit a dead node. At position b_s= ffs (B_N)

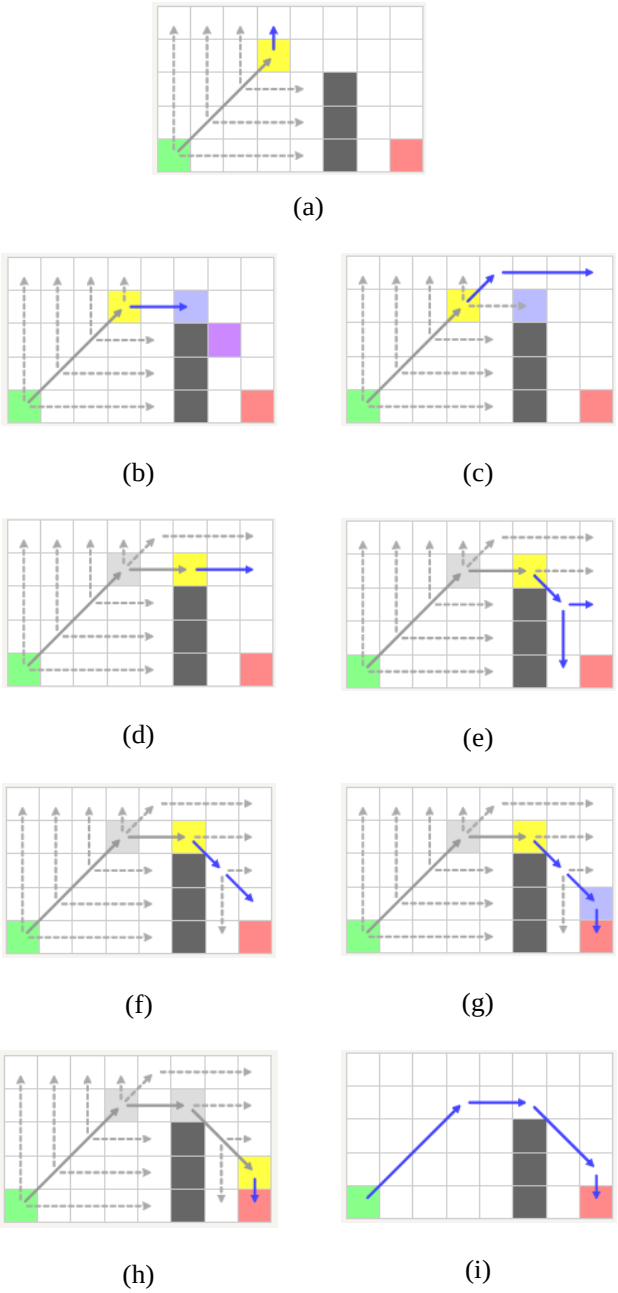


Fig. 5 JPS example workflow [4]

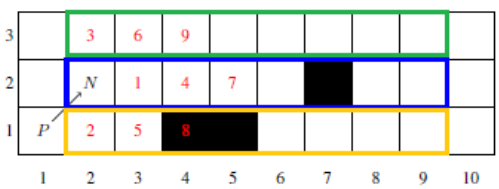


Fig. 6 JPS (red numbers) and Improved JPS [5]

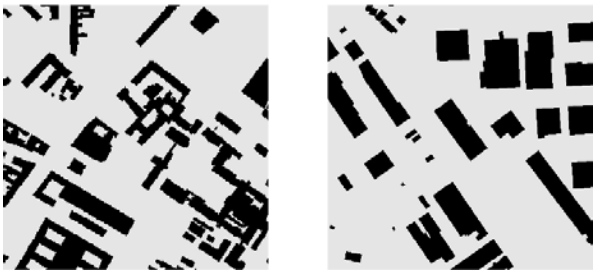
JPS+ has disadvantages represented by preprocessing step (offline algorithm). The time complexity of JPS is quadratic while it has linear space requirements w.r.t the number of

nodes in the grid. The disadvantage of the prepossessing step requires re-computing the map changes and introducing a substantive memory overhead.

JPS+ is a speed algorithm since it searches for jump point successors in constant time rather than scanning the grid for jump points. In most cases, JPS+ requires scanning a small part of the map and in this case, time dramatically increases if the map is obstacle free. In this case, one diagonal jump may lead to all nodes being scanned. Additionally, re-computing the path via block-based symmetry is performed very fast.

IV. Methods and Methodology

We used the benchmark [8] as shown in Fig. 8 to evaluate each of the mentioned algorithms against various criteria such as running time, number of nodes visited, and path length.



(a) Berlin_0_256 map (b) Boston_2_512 map

Fig. 7 Samples from the dataset used to test and evaluate algorithms

V. Results

For all maps in the dataset [10], the results are illustrated below. We have tested both algorithms (A*, JPS) on different maps from this dataset. Some results samples for some maps are shown in Table. 1 and Table. 2. Each table corresponds to a different map size: 256 × 256 to 512 × 512.

As shown in Tables, JPS finds a path faster than A*. JPS also produces a path almost similar to A*. From another point, we have tested the algorithms on all datasets, which are shown in Fig 9-14.

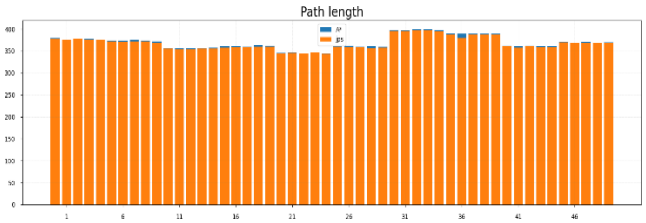


Fig. 8 Path length comparison between A* (blue) and JPS (orange) for the entire dataset for a map size of 256 x 256

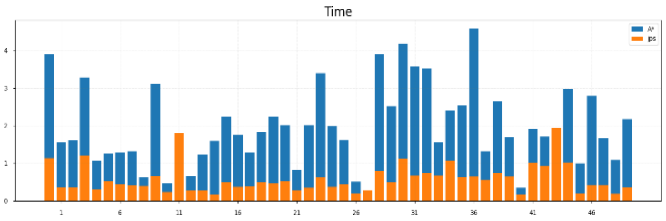


Fig. 9 Time comparison between A* (blue) and JPS for map size 256 x 256

Fig. 10 Path length comparison between A* (blue) and JPS (orange) for the entire dataset for a map size of 512 x 512

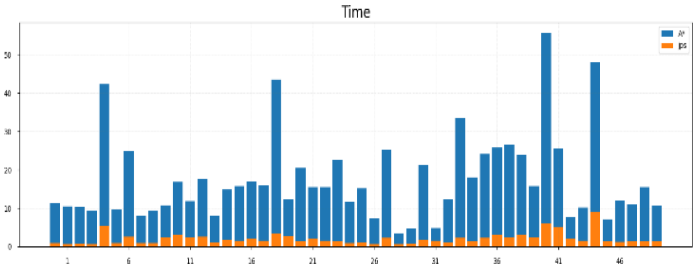


Fig. 11 Time comparison between A* (blue) and JPS for map size 512 x 512

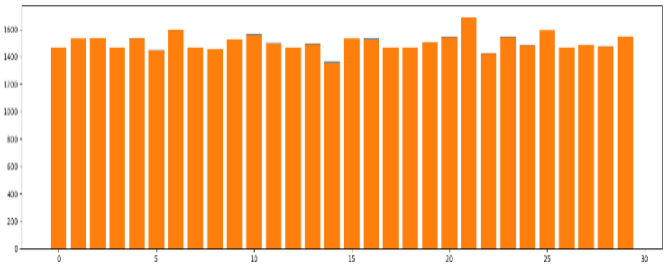


Fig. 12 Path length comparison between A* (blue) and JPS (orange) for the entire dataset for a map size of 1024 x 1024

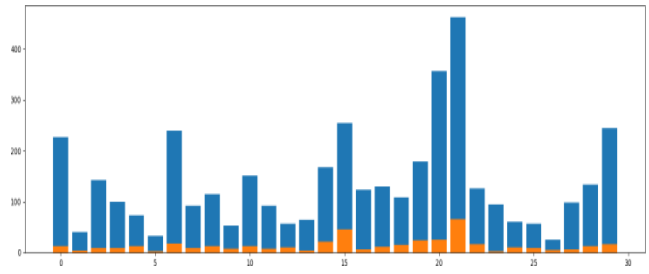


Fig. 13 Time comparison between A* (blue) and JPS for map size 1024 x 1024

Table. I Algorithms Comparison for 256 * 256 map

Map 256 x 256	Start		Goal		Benchmark (length)	Algorithm			
						A*		JPS	
	x	y	x	y		length	Time (sec)	length	Time (sec)
Boston_0-256	352	0	0	407	752.3595	752.3595	11.3571	751.7737	0.9301
	268	10	24	482	752.0530	752.0530	10.4115	751.4672	0.6380
Paris_0-256	495	503	24	85	722.0509	722.0509	16.7910	720.2935	2.9828
	509	48	12	495	720.8570	720.8570	14.8685	717.3423	1.7216
London_0-256	214	26	51	199	828.5311	828.5311	12.3035	825.6022	2.7474
	244	40	507	502	828.1930	828.1930	16.9394	828.1930	2.0201
Moscow_0-256	5	499	510	20	729.1829	729.1829	15.0875	728.5971	1.0276
	67	499	502	6	731.8692	731.8692	4.7098	729.5260	167.0000

Table. II Algorithms Comparison for 512 x 512 map

Map 512 * 512	Start		Goal		Benchmark (length)	Algorithm			
						A*		JPS	
	x	y	x	y		length	Time(sec)	length	Time
Boston_0-512	4	227	181	7	379.1564	379.1564	3.8948	377.3991	1.1406
	125	1	26	233	376.4113	376.4113	1.0783	375.2397	0.2997
Paris_0-512	242	243	6	18	390.3036	390.3036	2.5419	387.3747	0.6335
	239	253	7	10	389.4752	389.4752	1.6995	387.1320	0.6477
London_0-512	153	13	49	254	397.6001	397.6001	4.1897	395.8427	1.1271
	31	108	132	25	397.8305	397.8305	2.4166	394.3158	1.0736
Moscow_0-512	247	4	7	247	361.0559	361.0559	1.6264	359.8843	0.4414
	20	241	246	0	360.0854	360.0854	2.5279	357.1564	0.4946

Fig. 15 shows a visual representation of the path generated by JPS.

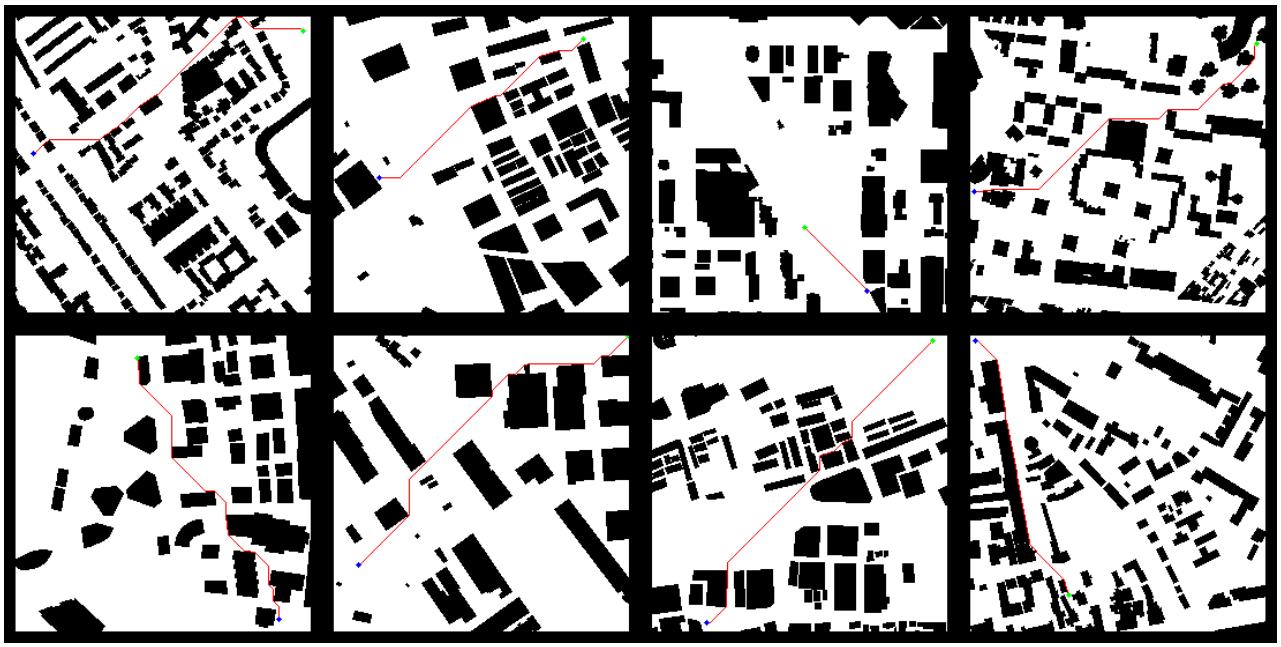


Fig. 14 Visual representation of the shortest path between two points based on JPS

CONCLUSION

In this work, we compared A* with a pruning algorithm jump point search. Using the jumping approach, JPS is able to quickly move over the map with no need to add nodes to the open list. This reduces the operations number and nodes' number in the list. JPS is optimal and eliminates nodes with no memory overhead. A disadvantage of JPS is to ignore the possibility from any angle so that JPS solutions always have a gap with the actual paths. This will be a possible working direction that will be implemented on enhanced JPS for better performance.

REFERENCES

[1] C. Y. Lee, "An algorithm for path connections and its applications, "IEEE Transactions on Electronic Computers, vol. EC-10, no. 3, pp.346–365, Sep. 1961.

[2] D. Harabor and A. Grastien, “Online graph pruning for pathfinding on grid maps,” in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 25, no. 1, 2011, pp. 1114–1119.
[3] “The JPS pathfinding system,” in International Symposium on Combinatorial Search, vol. 3, no. 1, 2012.
[4] https://zerowidth.com/2013/ a-visual-explanation-of-jump-point-search.html.
[5] D. Harabor and A. Grastien, “Improving jump point search,” in Proceedings of the International Conference on Automated Planning and Scheduling, vol. 24, 2014, pp. 128–135.
[6] S. Rabin, Game AI Pro 360: guide to movement and pathfinding CRC Press, 2019.
[7] Pygame, https://youtu.be/NmM4pv8uQwI.
[8] “Moving AI lab: 2d maps and benchmark problems,” https://www.movingai.com/benchmarks/random/index.html.
[9] B. Stout, “Smart moves: Intelligent pathfinding,” Game developer magazine, vol. 10, pp. 28–35, 1996.
[10] “Moving AI lab: 2d maps and benchmark problems, city street,” https://www.movingai.com/benchmarks/street/index.html.

Paper received 25 June 2023.
Mohammed Hammoud, PhD student, National Research University of Electronic Technology (MIET) (e-mail: hammoudmsh93@gmail.com);
Sergey Lupin, Professor, National Research University of Electronic Technology (MIET), (e-mail: lupin@miee.ru).