United International Journal for Research & Technology
Volume 04, Issue 08, 2023 | Open Access | ISSN: 2582-6832

Modifying JPS Algorithm Using Navmesh Data Structure Applied in 3D Using Unity

Raymond Carlos S. Medina[1], Hannah Shane B. Gittabao[2], and Vivien A. Agustin[3]
[1,2]Student, Pamantasan ng Lungsod ng Maynila
[3]Adviser, Pamantasan ng Lungsod ng Maynila

***Abstract*—** The JPS (Jump Point Search) algorithm is well-known for its efficiency and optimality in grid-based systems. The algorithm works well with uniform-cost grids, but JPS can be slow for large environments due to the large search space in 3D. The objective of this study is to overcome JPS's limitations in managing non-rectangular obstacles, varying obstacle types, and larger maps. Utilizing the NavMesh data structure, the research methodology entails effectively applying the JPS algorithm to non-grid maps and 3D environments. Techniques are developed to deal with variations in altitude, diverse obstacles, and maximize memory usage. The performance of Unity's NavMesh is tested by comparing computation time to the A* algorithm. The integration of JPS and NavMesh has the potential to enhance the speed, obstacle recognition, and scalability of computer graphics applications, thereby benefiting game development and virtual simulations.

***Keywords*—** 3D– 3-Dimensional, JPS – Jump Point Search Algorithm, NavMesh - Navigation Mesh, Pathfinding, Unity.

INTRODUCTION

Robotics and video games face pathfinding in grid environments with consistent expense. Current hierarchical pathfinding methods are fast and have low memory, but they often yield wasteful pathways. Grid maps are focal points in robotics, video games, and grid maps due to their simplicity in representing environments. Grids are academically interesting because there are often many paths between any two spots. These roadways are usually symmetrical, differing only in movement order. [1]

Several algorithms discovered the shortest path on a uniform-cost 2D mesh. A* optimizes breadth (Dijkstra) and depth-first search. This technique has various extensions, such as D*, HPA*, and Rectangular Symmetry Reduction, which reduce the number of nodes needed to identify the optimum path. [2]

Due to its efficiency in solving large-scale search problems, the Jump Point Search (JPS) pathfinding algorithm has gained popularity. JPS is a heuristic algorithm that uses pruning criteria to narrow the search area and focus on promising search network segments.

JPS improves search algorithms in robotics, video games, and transportation planning. JPS's potential, comparability to other path-searching algorithms, and applicability to different problem areas need further research.

JPS is known for its grid-based efficiency and optimality. The JPS Algorithm uses Jump Points to selectively expand grid map nodes while bypassing intermediary nodes, reducing wasteful node exploration. [3] A* takes short steps, whereas the JPS algorithm uses "jumping" to efficiently traverse a grid by considering larger straight-line motions along horizontal, vertical, and diagonal axes. This method optimizes A* while speeding up execution.

JPS on grids needs more variables and elements to improve performance and simulate realistically. Jumping off grid points is done for several reasons: It is optimum, requires no preprocessing, has no memory overhead, and can regularly speed up A* lookups by more than 10x, making it competitive with HPA* and often superior. [1]

While it was originally used in 2D grids, ongoing studies about the algorithm being adapted to 3D exist. JPS works in 3D contexts, however, more research is needed to optimize its implementation. [4]

In both instances, however, the algorithm's primary obstacle is the size of the map and its obstacles. The JPS algorithm can be computationally expensive in scenarios with a high number of obstacles. JPS search times can be significantly lengthier than other algorithms, such as A* and Dijkstra algorithm. The authors hypothesize that this is due to the large number of jump points generated by the algorithm in such situations, which can prolong the pathfinding procedure. [1][5]

The purpose of this paper was to propose enhancements to the Jump Point Search (JPS) algorithm for computing paths efficiently in the presence of various terrain and elements and for improving its implementation in 3D. The study examined how the JPS algorithm could be modified to resolve the issues identified during the process analysis phases. The effectiveness of the proposed enhancements was compared to the A* algorithm that is currently used with NavMesh using small maps that were created with Unity.

However, there were some limitations to this study. The proposed enhancements were only evaluated in three-dimensional environments. In addition, JPS optimization for other categories of obstacles, such as moving obstacles or obstacles with sharp curves, was not addressed in the research. In addition, the study compared the proposed enhancements to a limited number of benchmark maps. The proposed enhancements were evaluated based on speed and algorithm performance, but the impact of the proposed enhancements on other parameters, such as memory utilization or scalability, was not investigated.

RELATED WORKS

The JPS algorithm sometimes generates inefficient pathways. The algorithm generates poor pathways in lengthy, narrow corridors. According to the authors, the algorithm's pruning criteria can prematurely terminate searches in tight corridors, resulting in inferior pathways. [1]

Several studies have been initiated to improve the algorithm.

A study in 2019 presented a jump point search method with safe distance (SD-JPS) for path planning to address robot collision in complex situations due to control or positioning error and robot size. A rapid jump point search method-based jump point definition and node domain matrix improve the JPS algorithm. The SD-JPS approach can calculate the robot's safe distance from the barrier and increase its movement freedom using any size node domain matrix. It calculates multiple safety distances and designs the best path faster. [6]

JPS has also been improved for optimal path-planning for various purposes in multiple studies. "Global path planning of mobile robot based on improved JPS+ algorithm" introduced Bidirectional JPS+, which simultaneously searches for a path from the start and goal nodes. In vast maps, this can speed up the search. A safety feature prevents Bidirectional JPS+ from hitting obstructions. The study compared Bidirectional JPS+ to JPS and A*. Bidirectional JPS+ outperformed the other two algorithms in speed, safety, and efficiency. [7]

A recent study in 2022 introduced APF-JPS wherein key nodes and path planning time are lowered compared to the standard JPS technique, which ranks second in overall performance, while the node usage rate climbs by 23.4%. Thus, the APF-JPS method improves path planning by reducing processing load, improving real-time performance, and increasing the robot's endurance time. [8]

They eventually proposed in their next study the 3D JPS which solves low-altitude drone path planning and autonomous obstacle avoidance problems. A virtual-target gravity field and three-dimensional Bresenham's line algorithm helped the drone avoid obstructions between its starting and final positions. This study will also address the need to optimize the dynamic-obstacle-avoidance technique of the 3D JPS algorithm and reduce calculation time to improve path quality and computational efficiency. The researchers want to improve the dynamic-obstacle-avoidance technique and minimize the computation time of the 3D JPS algorithm to increase path quality and computational efficiency. [9]

The JPS Algorithm has also been developed in 3D wherein it was called JPS-3D, a 3D-enhanced version of the JPS algorithm. 3D path symmetry breaking finds and expands jump points. Between jump locations, only straight, 2D, or 3D diagonals can be taken. [10]

JPS-NavMesh Algorithm

JPS Algorithm

In 2011, Harabor and Grastien proposed the JPS algorithm. The JPS algorithm's main goal is to improve the A* algorithm's heuristic function by applying the neighbor pruning rule and forced-neighbor judgment method to the process of discovering subsequent path nodes. The JPS algorithm significantly reduces the number of nodes in the open list that must be accessed, lowering the algorithm's time and space costs.

The following parts comprise the JPS algorithm: (1) Pruning rules, which filter out and eradicate nodes on the grid map that do not require expansion. (2) Jumping rules, which identify and evaluate the jump nodes in the grid map. [1]

The traditional JPS algorithm takes a start node and a goal node as inputs and returns a path from the start to the goal if one exists from a start node to a goal node. It manages data structures like the open set, the closed set, and the parent map. The algorithm starts by adding the start node to the open set and initializing all other required variables. The program then enters a loop that proceeds until either a path is discovered, or the open set becomes empty. In each iteration, the algorithm selects the current node the open set node with the lowest cost. If the current node is the target node, the algorithm returns the path it constructs by following the parent pointers from the target node to the start node. Otherwise, the current node is designated as visited by being added to the set of visited nodes. The algorithm identifies successors of the current node by determining forced neighbors and jump points using functions. These successors are evaluated, and if a successor is not in the open set or if its tentative cost is less than its current cost, the algorithm updates the successor's parent and cost values. This procedure is repeated until all successors have been evaluated. If there is no path detected, the algorithm will return null.[1]

NavMesh

NavMesh, also known as Navigational Mesh, is a component of Unity's navigation and path finding system. The navigation system allows you to construct characters that can travel about the game world intelligently by using navigation meshes generated automatically from the Scene geometry. Dynamic barriers allow to change the characters' path at runtime, while off-mesh linkages allow to construct specialized behaviors such as opening doors or jumping down from a cliff. NavMesh is a data structure that defines the game world's walkable surfaces and allows to identify a path from one walkable area to another. The data structure is generated automatically based on the level's geometry.[11]

To consider a navigation, mesh successful, several requirements must be met. These requirements include the automatic generation of the mesh itself. Additionally, the mesh should effectively exclude obstacles, ensuri ng that they do not hinder the pathfinding process. Another crucial requirement is the ability to generate near-optimal paths between any two points within the mesh. These paths should provide a clear and efficient route from the starting position to the desired goal position, represented as a list of points. Meeting these requirements ensures the effectiveness and usability of the navigation mesh for pathfinding purposes. [12]

JPS-Navmesh Algorithm

The provided pseudo code below implements the proposed JPS algorithm integrated into the NavMesh Data Structure:

```
Function JPS(startNode, goalNode)
    openSet = empty priority queue
    closedSet = empty Set
    gScore = map Of node To infinity
    parent = map Of node To null

    gScore[startNode] = 0
    openSet.add(startNode)

    While openSet Is Not empty
        currentNode = openSet.pop()

        If currentNode == goalNode Then
            Return constructPath(parent, currentNode)

            closedSet.add(currentNode)

            neighbors = getNeighbors(currentNode)

            foreach neighbor in neighbors
            If neighbor Then In closedSet
                Continue While

                tentativeGScore = gScore[currentNode] +
distance(currentNode, neighbor)

            If neighbor Then Not In openSet Or tentativeGScore <
gScore[neighbor]
                parent[neighbor] = currentNode
                gScore[neighbor] = tentativeGScore

                If neighbor Then Not In openSet
                    openSet.add(neighbor)

                        Return null // Path Not found

Function constructPath(parent, currentNode)
    path = empty list
    While currentNode Is Not null
        path.prepend(currentNode)
        currentNode = parent[currentNode]
          Return path
```

Figure 1. *Psuedocode of Modified JPS-NavMesh Algorithm*

The modified algorithm begins by initializing data structures such as the open set, closed set, and maps for storing node costs and parent nodes. The algorithm then enters a loop where it selects the node with the lowest cost from the open set and checks if it is the goal node. If not, the node is added to the closed set, and its neighboring nodes are explored. For each neighbor, the algorithm calculates a tentative cost from the start node and updates the parent and cost if it is lower than the existing value or the neighbor is not in the open set. This process continues until either the goal node is reached, in which case the path is constructed and returned, or the open set becomes empty, indicating an unreachable goal. The path construction involves tracing back through the parent nodes. Using the JPS algorithm, the code will, in theory, efficiently discover paths by prioritizing nodes based on their costs and backtracking through their parents.

The integration of NavMesh significantly alters the pathfinding process in a few significant ways. Instead of considering all adjacent nodes as potential successors, NavMesh is used to identify valid successor nodes. Using the connectivity and accessibility information provided by the NavMesh, this enables the algorithm to select nodes that are further away from the present node but still in a relevant direction. This ensures that successors align with the navigation mesh structure, taking into account the geometry and connectivity of non-rectangular obstacles.

Secondly, the jump function is enhanced by incorporating NavMesh, allowing for the identification of valid jump points. When attempting a jump in a particular direction, the algorithm verifies the validity of the next node using NavMesh. The algorithm recognizes that there is no legitimate jump point in that direction if the next node corresponds to a wall or obstacle. If the next node is the goal or adjacent to a wall, however, it is considered a valid jump point. NavMesh facilitates the accurate identification of these jump sites by providing precise obstacle representation and connectivity data.

Lastly, NavMesh has a significant impact on the pruning procedure. Pruning, which entails selectively evaluating nodes, is guided by the connectivity information derived from NavMesh. Nodes are pruned horizontally, vertically, and diagonally in accordance with NavMesh's recommendations. If it is determined that evaluating the nodes adjacent to a specific node would yield better results, the algorithm eliminates that node and evaluates its neighbors instead. The NavMesh functions as a guide for the pruning process, indicating which areas should be further explored and which can be skipped.

By integrating the NavMesh into the JPS algorithm, the pathfinding process becomes more adept at contemplating non-rectangular obstacles represented by the NavMesh. This allows the algorithm to make informed decisions regarding successor nodes, jump points, and pruning, resulting in more accurate and efficient pathfinding in environments with non-rectangular obstacles.

METHODOLOGY

The proposed methodology involves several key steps.

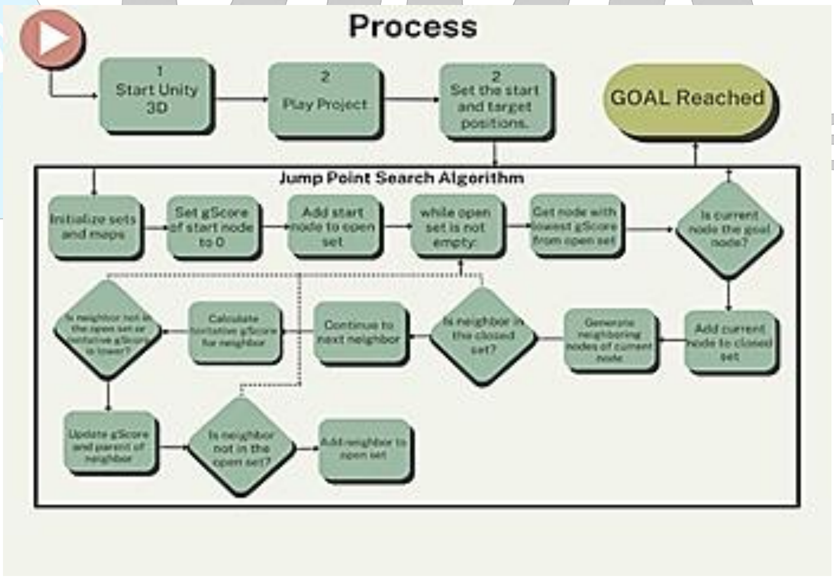
Process
1 Start Unity 3D
2 Play Project
2 Set the start and target positions.
GOAL Reached
Jump Point Search Algorithm
Initialize sets and maps
Set gScore of start node to 0
Add start node to open set
while open set is not empty:
Get node with lowest gScore from open set
Is current node the goal node?
Add current node to closed set
Generate neighboring nodes of current node
Is neighbor in the closed set?
Continue to next neighbor
Calculate tentative gScore for neighbor
Is neighbor not in the open set or tentative gScore is lower?
Update gScore and parent of neighbor
Is neighbor not in the open set?
Add neighbor to open set

Figure 2. *Conceptual Framework of the JPS-NavMesh Algorithm*

Firstly, the researchers will adapt the JPS algorithm to utilize the NavMesh data structure, leveraging its preprocessed walkable surface information and efficient triangle-based navigation queries. This integration will enable JPS to navigate complex 3D environments with irregular shapes and varying obstacles.

Next, the researchers will focus on optimizing the JPS algorithm's traversal and search techniques to take advantage of the NavMesh representation. This includes considering navigation constraints, such as obstacles and restricted areas, to ensure accurate and efficient pathfinding results.

The researchers utilized Unity as the primary application for development. Unity is a versatile game development platform with tools for creating interactive 2D and 3D experiences. By employing Unity and following this methodology, the project benefited from its tools, asset store, and user-friendly interface to create an interactive and visually appealing simulation.

Additionally, the researchers will explore techniques to enhance the ground obstacles recognizability of the JPS algorithm in the context of Unity's NavMesh.

Specifically, this includes:

1. Successor generation: Instead of considering all adjacent nodes as successors, the NavMesh is utilized to identify valid successor nodes. The NavMesh provides information about the connectivity and accessibility of polygons, enabling the algorithm to move to nodes that are further away but still in the same relative direction as the present node. This ensures that successors are selected based on the navigation mesh structure, considering the shape and connectivity of non-rectangular obstacles into account.

2. Jump function: NavMesh is used to determine valid jump coordinates, thereby enhancing the jump function. When conducting a jump in a particular direction, the algorithm verifies that the subsequent node is a valid location according to NavMesh. The algorithm determines there is no jump point in that direction if the next node is a wall (obstacle). However, if the next node is the target or adjacent to a wall, it is a valid jump point. The NavMesh assists in identifying valid jump locations by providing an accurate representation of obstacles and connectivity data.

3. Pruning: Pruning is influenced by NavMesh connectivity. Based on the NavMesh data, the algorithm prunes nodes horizontally, vertically, and obliquely. If the nodes adjacent to a specific node should be evaluated instead, the algorithm removes that node and continues with the evaluation of adjacent nodes. The NavMesh directs the pruning process by indicating which areas must be thoroughly investigated and which can be skipped.

To validate the effectiveness of the proposed methodology, the researchers will conduct a series of tests and comparisons against the most used pathfinding algorithm in Unity, which is the A*. Pathfinding time, distance, and operation time will be considered to assess the improvements achieved by the modified JPS algorithm using NavMesh.

By integrating JPS with the NavMesh feature in Unity, this research aims to provide a more efficient and scalable pathfinding solution for complex 3D environments. The outcomes of this research have the potential to benefit various applications, including game development, virtual simulations, and architectural walkthroughs, by enabling faster and more accurate pathfinding in Unity-based projects.

SIMULATION

Unity, a powerful 3D engine, and the programming language C# were utilized during the implementation and testing of the algorithms. The testing program required importing a variety of 3D models that represented the area and its challenges. The program established the start and end points for testing at random and conducted a predetermined number of tests, although the specific behaviors depended on the selected algorithm. It is essential to recognize that precise and recurrent tests may be subject to limitations due to background activities running in Windows 10. In each instance, multiple test trials were conducted in an attempt to overcome these limitations.

***Table 1.** Average Speed Results from Testing*

MAP	A* (A-B)	A* (B-A)	JPS (A-B)	JPS (B-A)
Open Area	0.29005ms	0.01983 ms	1.52283 ms	0.00616 ms
Barnyard	0.29082 ms	0.02781 ms	1.6063 ms	0.00566 ms
Enclosure	0.2818 ms	0.02462 ms	1.52092 ms	0.00594 ms
50*50 Maze	1.90611 ms	1.23518 ms	1.47018 ms	0.00506 ms
50*50 Open Area	2.09586 ms	1.75482 ms	1.5254 ms	0.00588 ms

Table 1 presents the accumulated results of the average speed of each algorithm from random point A to random point B and backward.

On the First Environment initial test, A* did better than JPS in terms of the amount of time it took to prepare the way from point A to point B. Although A* was initially speedier, JPS emerged victorious in the second test where the path had to be retraced from point B to point A.

The distinction resides in how A* and JPS approach pathfinding. A* does not retain information regarding previously processed paths. A* begins the preprocessing phase from scratch for each new pathfinding request, analyzing the complete graph or grid from the starting node to the destination node. In contrast, JPS is capable of recalling the paths it has traversed. JPS stores information about jump points and the paths it has taken once a path is discovered or a section of the map is explored. This information can be used in subsequent pathfinding queries within the same map, allowing JPS to bypass previously explored regions and avoid redundant calculations. This memory of previously investigated paths significantly improves the performance of JPS, especially when multiple paths must be computed in the same environment.

In the second environment, the barnyard, both the A* and JPS implementations took longer to finish than they did in the first environment. The A* implementation obtained the best preprocessing time in this test, with a time of 0.29082 milliseconds. On the other hand, the JPS implementation required 1.6063 milliseconds longer for preprocessing. Interestingly, identical to the initial test, JPS-Navmesh demonstrated superior performance when retracing the steps from point B to point A. JPS completed the backtracking process in 0.00566 milliseconds, substantially less than A*'s preprocessing time of 0.02781 milliseconds. These results suggest that the barnyard's scale, complexity, or layout may have impacted the overall pathfinding performance. The A* implementation was more effective at locating the initial path from point A to point B, whereas the JPS algorithm was superior at backtracking from point B to point A.

In the third environment, the goal was to find a target that was hidden in a square with only one entrance. In this scenario, the preprocessing durations for A* and JPS differed. The preprocessing time for A* was 0.2818 milliseconds, while the preprocessing time for JPS was 1.52092 milliseconds. Similarly, to the prior tests, JPS exhibited remarkable backtracking performance. It completed the backtracking procedure in a very brief 0.005944444 milliseconds, whereas A* required a slightly longer 0.024622222 milliseconds for preprocessing. It is essential to observe that the preprocessing times reported are unique to your tests and the hidden target scenario's conditions. Pathfinding performance can be affected by several variables, including map layout, complexity, and implementation specifics.

In the fourth setting, which was a big maze, JPS proved to be the best algorithm for getting through it. JPS accomplished a preprocessing time of 1.47018 milliseconds, whereas A* required 1.9061 milliseconds. Even though A* had an extended preprocessing time for the large maze, JPS performed exceptionally well in navigating it.

Similar to previous experiments, JPS demonstrated its effectiveness during the backtracking phase. It completed the backtracking procedure in an impressively quick 0.00506 milliseconds, whereas A* required 1.23518 milliseconds for preprocessing.

These results demonstrate how proficiently JPS can navigate complex environments, such as large mazes. While A* may require more time for preprocessing in such scenarios, JPS excels in both the initial pathfinding and backtracking phases due to its ability to avoid unnecessary node expansions and store path information. It should be emphasized that the preprocessing times indicated are unique to the large maze environment in which your specific tests were conducted. The effectiveness of pathfinding can vary considerably based on a variety of factors, such as the chosen implementation strategy, the complexity of the maze, and the available computational resources.

In the fifth and final environment, JPS-Navmesh once again showed how well it could move through large areas with many obstacles and different terrain. It completed preprocessing in 1.5254 milliseconds, while A* required 2.095858 milliseconds to complete the same task. Despite the extended preprocessing time that A* required for the large area, JPS navigated through it with remarkable proficiency. In addition, previous observations indicate that JPS excelled in the retracing phase once again. It completed the task in an impressively quick 0.00588 milliseconds, whereas A* required 1.7548 milliseconds longer.

These results demonstrate the capability of JPS with Navmesh to navigate challenging environments with numerous obstacles and varied terrains. Despite A*'s extended preprocessing time, JPS achieved superior performance during both the initial pathfinding and backtracking phases by skipping unnecessary node expansions and effectively storing path information.

CONCLUSION

The research results indicate that integrating the NavMesh data structure into the Jump Point Search (JPS) algorithm for navigating complex 3D environments is effective. Utilizing memory of previously explored paths, JPS with NavMesh outperforms the A* algorithm in certain scenarios, resulting in more efficient pathfinding and quicker backtracking. JPS-NavMesh's adaptability is evident in large maps with intricate structures, numerous obstacles, and diverse terrains. However, JPS-NavMesh's efficacy varies based on the environment and implementation details. Recommendations consist of investigating hybridization with other pathfinding algorithms, conducting additional research to improve the JPS algorithm in 3D, collaborating with game developers, incorporating machine learning techniques, and conducting continuous monitoring and evaluation. By adhering to these recommendations, the JPS algorithm with NavMesh integration can continue to evolve and contribute to advancements in pathfinding algorithms, which will benefit industries that rely on effective navigation systems.

REFERENCES

[1] D. Harabor and A. Grastien, “An Optimal Any-Angle Pathfinding Algorithm,” Proceedings of the International Conference on Automated Planning and Scheduling, vol. 23, pp. 308–311, Jun. 2013, doi: 10.1609/icaps.v23i1.13609.

[2] A. J. Patel, “Red Blob Games: Introduction to A*,” Introduction to the A* Algorithm. https://www.redblobgames.com/pathfinding/a-star/introduction.html

[3] S. R. Lawande, G. Jasmine, J. Anbarasi, and L. I. Izhar, “A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games,” Applied Sciences, vol. 12, no. 11, p. 5499, May 2022, doi: 10.3390/app12115499.

[4] P. Ranttila, “JPS Algorithm Adaptation and Optimization to Three-dimensional Space - UTUPub,” JPS Algorithm Adaptation and Optimization to Three-dimensional Space - UTUPub, Jun. 24, 2019. https://www.utupub.fi/handle/10024/148054

[5] D. Harabor and A. Grastien, “Online Graph Pruning for Pathfinding On Grid Maps,” Proceedings of the AAAI Conference on Artificial Intelligence, vol. 25, no. 1, pp. 1114–1119, Aug. 2011, doi: 10.1609/aaai.v25i1.7994.

[6] X. Zheng, X. Tu, and Q. Yang, “Improved JPS Algorithm Using New Jump Point for Path Planning of Mobile Robot,” 2019 IEEE International Conference on Mechatronics and Automation (ICMA), Aug. 2019, Published, doi: 10.1109/icma.2019.8816410.

[7] C. Jiang, S. Sun, J. Liu, and Z. Fang, “Global path planning of mobile robot based on improved JPS+ algorithm,” 2020 Chinese Automation Congress (CAC), Nov. 2020, Published, doi: 10.1109/cac51589.2020.9327403.

[8] Y. Luo, J. Lu, Q. Qin, and Y. Liu, “Improved JPS Path Optimization for Mobile Robots Based on Angle-Propagation Theta* Algorithm,” Algorithms, vol. 15, no. 6, p. 198, Jun. 2022, doi: 10.3390/a15060198.

[9] Y. Luo, J. Lu, Y. Zhang, Q. Qin, and Y. Liu, “3D JPS Path Optimization Algorithm and Dynamic-Obstacle Avoidance Design Based on Near-Ground Search Drone,” Applied Sciences, vol. 12, no. 14, p. 7333, Jul. 2022, doi: 10.3390/app12147333.

[10] T. K. Nobes, D. Harabor, M. Wybrow, and S. D. C. Walsh, “The JPS Pathfinding System in 3D,” Proceedings of the International Symposium on Combinatorial Search, vol. 15, no. 1, pp. 145–152, Jul. 2022, doi: 10.1609/socs.v15i1.21762.

[11] A. Y. Kapi, “A Review on Informed Search Algorithms for Video Games Pathfinding,” International Journal of Advanced Trends in Computer Science and Engineering, vol. 9, no. 3, pp. 2756–2764, Jun. 2020, doi: 10.30534/ijatcse/2020/42932020.

[12] M. Karlsson, “A navigation mesh-based pathfinding implementation in CET designer. ,” Linkoping University., 2021, Published, [Online]. Available: https://liu.diva-portal.org/smash/get/diva2:1560399/FULLTEXT01.pdf