Volume 5 Issue 2 (September 2025)
DOI 10.24191/jaeds.v5i2.94
eISSN 2805-5756

Journal of Applied Engineering Design & Simulation

Performance Comparison of A and Dijkstra Algorithms with Bézier Curve in 2D Grid and OpenStreetMap Scenarios

Zakariah Yusuf[1,2], Sufian Mohamad[1,*] Wan Suhaifiza Wan Ibrahim[1]
[1]Electrical Engineering Studies, Universiti Teknologi MARA, Cawangan Johor, Kampus Pasir Gudang, 81750 Masai, Johor, Malaysia.
[2]Process Instrumentation and Control (PiCON) Research Initiative Group, Universiti Teknologi MARA, Shah Alam 40450 Selangor, Malaysia.
[*]corresponding author: zakariahyusuf@uitm.edu.my

ABSTRACT

This paper presents a comparative study of the A* and Dijkstra algorithms for path planning in both 2D grid maps and real-world OpenStreetMap (OSM) environments. The evaluation focused on three key performance metrics: computational efficiency, path smoothness, and the number of turns. Both algorithms were tested under varying obstacle densities, and Bézier curve smoothing was applied to enhance path quality. In 2D grid maps, A* consistently generated smoother paths with fewer turns, especially in complex environments. Its heuristic-based search allowed it to expand fewer nodes, resulting in faster computation times compared to Dijkstra. On the other hand, Dijkstra's algorithm, though robust and optimal, exhibited longer runtimes and produced paths with more turns due to its exhaustive search approach. In the OSM-based scenarios, both algorithms yielded paths of identical length. However, A* significantly outperformed Dijkstra in terms of runtime across most test cases, further demonstrating its computational advantage. These findings validate A*'s practical advantage of real-time applications where both efficiency and path quality are crucial. While Dijkstra remains a reliable benchmark, A* offers a balanced trade-off between speed and path quality, making it more suitable for real-world path planning applications in both structured and unstructured environments.

Keywords: Path Planning; A* Algorithm; Dijkstra Algorithm; 2D Map; OpenStreetMap; Bézier Curve Smoothing

Nomenclature

S	Position on the Bézier curve
t	Curve parameter
P_0	Start point of the curve
P_1	First control point
P_2	Second control point
P_3	End point of the curve
x_i	x-pisition
yi	y-pisition

Abbreviations

*A**	A-Star Algorithm
2D	Two Dimensional
OSM	Open Street Map
GIS	Geographical Information Systems

1.0 INTRODUCTION

The revolution in autonomous vehicles (AV) and autonomous mobile robotic (AMR) technology has significantly impacted the transportation industry influencing various sectors and driving advancements in logistics, public transportation, last mile delivery etc. [1], [2], [3]. This technology employs a combination of many modules to make it successful such as navigation, control system, mapping and localization, decision maker and others. Path planning is an important component in autonomous navigation systems [4]. Among the most popular and first global planning algorithms is the Dijkstra's algorithm, named after its creator Edsger Dijkstra [5]. The algorithm is widely used in various applications, including network routing, drone, geographical information systems (GIS), AV, and robotics [6], [7], [8]. Dijkstra's algorithm searches all possible paths from the start node to the goal, ensuring that the shortest path is discovered. However, unlike the other path planning

Received on 28.02.2025
Accepted on 23.06.2025
Published on 26.09.2025

algorithm, it does not use a heuristic function to prioritize nodes. Instead, it expands nodes based solely on the cumulative cost from the start node. This approach ensures the shortest path; however, it can be less efficient in large or complex environments and it requires more computational time [9].

The A* algorithm is a widely used pathfinding technique known for its balance between efficiency and accuracy [10]. It works by combining elements of Dijkstra's algorithm with a heuristic function, which helps the algorithm estimate the cost of reaching the goal from any given node. This heuristic method is often calculated using the Manhattan or Euclidean distance that guides A* towards the target more efficiently than Dijkstra's algorithm, which does not use such an estimate. A* operates by maintaining two key lists which are the open list and the closed list. The open list holds nodes that are yet to be explored, while the closed list contains nodes that have already been evaluated. At each step, the algorithm selects the node from the open list with the lowest estimated cost, explores its neighbouring nodes, and updates their costs based on this information. The process repeats until the goal is reached, or there are no more nodes left in the open list. Once the goal is found, A* reconstructs the optimal path by tracing backward from the goal to the starting point [11].

Recent advancements in path planning algorithms, such as D* Lite and Rapidly exploring Random Trees (RRT)[12], have expanded the possibilities for dynamic and real-time pathfinding in complex and high-dimensional environments. D* [13]. For instance, it is well-suited for environments where frequent changes require incremental updates to the path, while RRT is particularly effective in motion planning for robotics and autonomous vehicles. Despite these developments, the A* and Dijkstra algorithms remain fundamental and widely used for static pathfinding problems due to their robustness, simplicity, and effectiveness in known environments. This study focuses on these algorithms, providing a detailed analysis of their performance in solving pathfinding tasks in various scenarios.

In this work the 2D map is used to compare the performance of the algorithms. 2D grid maps are commonly used for path planning especially for the indoor mobile robot [14], [15]. These maps represent the environment as a grid of cells, where each cell can be traversable or non-traversable. The grid-based representation simplifies the path planning process by discretizing the environment into manageable units. In a 2D grid map, the A* and Dijkstra's algorithms can be applied to find paths from a starting cell to a goal cell. The algorithms consider the connectivity between neighbouring cells and account for obstacles by marking certain cells as prohibited route. The efficiency and accuracy of the path planning process depend on the resolution of the grid map and the complexity of the environment.

OpenStreetMap (OSM) is a collaborative mapping platform that provides detailed geographical data for various locations worldwide [16]. OSM data can be used for path planning in real-world environments, offering a more realistic and dynamic representation compared to grid maps. OSM includes information about roads, buildings, landmarks, and other features, making it valuable for AVs navigating urban areas.

Path planning with OSM involves converting the map data into a graph representation, where nodes correspond to intersections or waypoints and edges represent road segments. The A* and Dijkstra's algorithms can then be applied to this graph to find optimal routes. This approach enables AVs to navigate complex urban environments, accounting for real-time traffic conditions, road closures, and other dynamic factors.

The remainder of this paper is structured as follows: Section 2 of this paper provides a detailed overview of the methodologies used for implementing the A* and Dijkstra's algorithms in 2D and OpenStreetMap including the theoretical foundations and pseudocode representations. Section 3 discusses the performance evaluation of the algorithms. Section 4 presents the simulation setup, describing the 2D grid map and OpenStreetMap environments used for testing. Section 5 discusses the results of the experiments, comparing the performance of the two algorithms based on the performance metrics. Finally, Section 6 concludes the paper by summarizing the key insights and suggesting directions for future research.

2.0 METHODOLOGY

The general methodology of this work starts with the development of the testing platform. The 2D grid map is widely used to represent the environment of the world. The starting and goal point are pre-determined so the algorithm can be deployed. The next step is to run the algorithm for both methods. The cubic spline algorithm is applied for path smoothing. Both results are analysed on the same 2D grid in terms of path length, number of turns and computational time. The same procedure is applied to the OpenStreetMap path finding method. However, in OpenStreetMap the evaluation only considers the path length and the time only. Figure 1 shows the general methodology implementation flow chart.

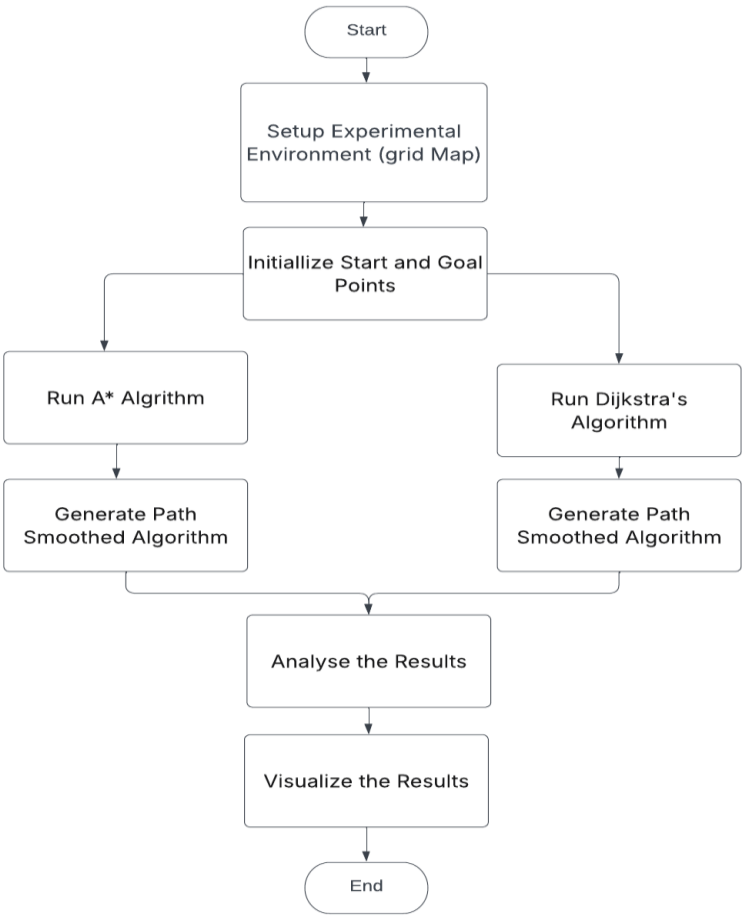

Figure 1. Methodology Flow Chart

2.1 Dijkstra algorithm

Dijkstra's algorithm operates by systematically exploring the graph, maintaining a distance map dist that tracks the shortest known distances from the starting node to all other nodes. The pseudocode for the Dijkstra algorithm is given by:

```
Input:
- A graph M = (N, E), where N is the set of nodes and E is the set of edges
- A start node s ∈ N
- A goal node g ∈ N

Output:
- The shortest path from s to g, or a failure message if no path exists

1. For each node n ∈ N, set Dist[n] ← ∞
2. Set Dist[s] ← 0
3. Initialize cameFrom ← {} (an empty map to trace the optimal path)
4. Initialize a priority queue PQ, ordered by Dist[n], containing all nodes in N
5. while PQ ≠ Ø do
6.     n_min ← node in PQ with the smallest Dist[n]
7.     if n_min = g then
8.         return reconstruct_path(cameFrom, n_min)
9.     for each neighbor n_adj ∈ Adj(n_min) do
10.        if Dist[n_min] + w(n_min, n_adj) < Dist[n_adj] then
11.            Dist[n_adj] ← Dist[n_min] + w(n_min, n_adj)
12.            cameFrom[n_adj] ← n_min
13.            Update PQ to reflect the new priority of n_adj
14. return “No path found”
```

The algorithm begins by initializing the *Dist* array, which keeps track of the shortest known distance from the start node s to every other node n in the graph. Initially, all nodes are assigned a distance of infinity (∞) except for the start node, which has a zero distance. The *cameFrom* map is initialized as empty and is used to store the predecessor of each node, allowing the reconstruction of the shortest path once the goal node g is reached. A priority queue (PQ) is used to efficiently manage the nodes based on their *Dist* values, ensuring that the node with the smallest distance is explored first. As the algorithm progresses, the Dist of each neighboring node is updated, and the node is added to the priority queue if it has not been explored or if a shorter path to it is found.

The core of the algorithm works by iteratively selecting the node n_min with the smallest $Dist(n)$ from the priority queue, which ensures that the closest unexplored node is always processed. If n_min is equal to the goal node g, the algorithm terminates, and the *reconstruct_path* function is called to backtrack through the *cameFrom* map and return the shortest path from the start to the goal. For each adjacent neighbor n_adj of the current node, the algorithm checks if the distance to the neighbor through the current node is shorter than the previously known distance. If so, it updates the *Dist* value for *n_adj* and sets the current node as its predecessor in the *cameFrom* map. The priority queue is then updated to reflect the new *Dist* value for *n_adj*. If the goal node is unreachable, the algorithm returns a failure message, indicating no path exists.

2.2 A* algorithm

The pseudocode and implementation provided below outline how the A* algorithm explores the search space, keeping track of the shortest path while considering both the actual and estimated costs. The path reconstruction function is responsible for tracing the final path once the goal node is reached.:

```
Input:
- Graph M = (N, E), where N is the set of nodes and E is the set of edges
- Start node s ∈ N
- Goal node g ∈ N
Output:
- The shortest path from s to g, or a failure message if no path is found

1. Initialize the open set O ← {s}
2. Initialize a map cameFrom to reconstruct the final path
3. For each node n ∈ N, set g(n) ← ∞
4. Set g(s) ← ∅
5. For each node n ∈ N, set f(n) ← ∞
6. Set f(s) ← h(s, g), where h(·) is the heuristic function
7. while O ≠ ∅ do
8.     n_current ← argmin_n ∈ O f(n)
9.     if n_current = g then
10.         return ReconstructPath (cameFrom, g)
11.     Remove n_current from O
12.     for each neighbor n_neighbor ∈ Neighbors(n_current) do
13.         t ← g(n_current) + dist (n_current, n_neighbor)
14.         if t < g(n_neighbor) then
15.             cameFrom[n_neighbor] ← n_current
16.             g(n_neighbor) ← t
17.             f(n_neighbor) ← g(n_neighbor) + h(n_neighbor, g)
18.             if n_neighbor ∉ O then
19.                 Add n_neighbor to O
20. end while
21. return "No path found"
```

In the A* algorithm, the open set (O) is used to manage the nodes yet to be explored, containing the start node initially. The algorithm evaluates nodes based on their total cost estimated $f(n)$ value, which is a combination of two key components: the cost to reach the node ($g(n)$) and an estimated heuristic $h(n, g)$, which predicts the remaining cost to reach the goal. The algorithm explores the node with the lowest $f(n)$ value, ensuring an efficient search toward the goal. The *cameFrom* map tracks the predecessors of each node, enabling path reconstruction once the goal is reached. As nodes are evaluated, their $g(n)$ values are updated to reflect the cost of the shortest known path, and the $f(n)$ values are recalculated to prioritize nodes that are closer to the goal.

The algorithm iteratively explores the most promising nodes based on the heuristic estimate, and for each neighbouring node, the tentative cost (t) is calculated as the sum of the cost to reach the current node and the cost to move to the neighboring node. If this (t) is smaller than the previously known cost for the neighboring node, it updates the $g(n)$ and $f(n)$ values accordingly, adding the neighbor to the open set if it has not been explored. This process continues until the goal node is reached, at which point the *ReconstructPath* function is employed to backtrack from the goal to the start node, thus providing the optimal path. If no path is found, the algorithm returns a failure message, indicating that no viable path exists between the start and goal nodes.

The reconstruct path function is responsible for tracing back the optimal path from the goal node to the start node using the path_trace map, which stores each node's predecessor during the search process. Starting from the goal, it iteratively follows the recorded parent nodes until the start node is reached, thereby constructing the full path in reverse order. This function used in this work for retrieving the result after algorithms A* or Dijkstra has completed their exploration. The pseudocode of the of the reconstruct path is given by:

```
1. Initialize totalPath ← [current]
2. while current ∈ path_trace do
3.    current ← path_trace[current]
4.    Prepend current to totalPath
5. return totalPath
```

2.3 Path smoothing algorithm

In this work, Bézier curve [17] method is used to create smooth curves that pass through or near a series of waypoints generated by algorithms A* and Dijkstra. Bézier curves are widely recognized in the literature as an effective and computationally efficient method for path smoothing in robotic and autonomous vehicle navigation [18]. Their parametric nature enables the generation of smooth, continuous trajectories that are well-suited for systems with limited turning capabilities and nonholonomic constraints.

In this work, Bézier curve smoothing is applied directly to the waypoints generated by the initial path planning algorithm, ensuring continuity and reducing sharp turns that could hinder real-time control. The method's simplicity and reliability make it a practical choice, especially in scenarios where real-time performance and path feasibility are prioritized over complex smoothing techniques. The Bézier curve is represented in equation 1.

$$S(t) = (1-t)^3 P_0 + 3(1-t)^2 2tP_1 + 3(1-t)t^2 P_2 + t^3 P_3 \quad (1)$$

Where t is a parameter that varies from 0 to 1. P_0, P_1, P_2 *and* P_3 are the (x, y) control points.

2.4 Performance matric

Three performances matric are used in the evaluation of the algorithm which are path length generated by the algorithm and the equations of the path length evaluation are given by:

$$Path\_Len = \sum_{i}^{N-1} \sqrt{(x_{i-1} - x_i)^2 + (y_{i-1} - y_i)^2} \quad (2)$$

Where (x_i, y_i) are coordinates of *i-th* point in the path and coordinates of the (x_{i-1}, y_{i-1}) *i-1 th* point in the path, while the $\sqrt{(x_{i-1} - x_i)^2 + (y_{i-1} - y_i)^2}$ is the Euclidean distance.

The next performance is the number of turns generated by the algorithm. The matric is crucial in 2D map because

it impacts various aspects of path quality, including smoothness, navigation ease, comfort, safety, control complexity, and efficiency. Evaluating paths based on the number of turns helps in designing more practical routes for autonomous systems, especially in robotics.

The last performance measured in this work is the computational time of both algorithms where the measurement starts from the deployment until the optimal path is generated.

3.0 SIMULATION SETUP

The experiment of this work involves two types of environments which are the 2D grid map and OpenStreetMap. 2D grid map is utilized to represent the navigable environment in our pathfinding study. The grid divides the space into discrete cells, each marked as either free or occupied. This representation allows us to apply pathfinding algorithms to navigate through the grid efficiently. 2D grid environments of sizes 30×30 and 50×50 are utilized to evaluate the performance of pathfinding algorithms under varying spatial complexities. The obstacle density is systematically varied, with 30% and 40% of the grid cells occupied by obstacles simulating the scenarios. The map, obstacle generation and path planning development are implemented using Python 3.10, where NumPy library employs random placement and clustering techniques to create diverse environmental configurations. The PC used in this work is Dell Latitude 7280, 16 GB RAM, running using Windows 11 64 bit.

We employ the path planner algorithm to find paths from a start point to various destination points, considering the grid's cell states. The grid's resolution affects the map's detail and the path finding performance. For accurate pathfinding and realistic navigation, we implement path smoothing techniques to convert the discrete paths into smoother trajectories. Visualization of the grid map and the computed paths is performed to analyse the pathfinding results, with colour coding used to differentiate between free and occupied cells where black (“0”) means the obstacle while white (“1”) is the free route. Figure 2 presents the 2D grid map.

In this study, we used OpenStreetMap data to extend the application of our pathfinding algorithms. OSM provides a rich dataset of geographical information, including street networks and points of interest, which is crucial for realistic path planning in urban environments. The Overpass API is used to query specific map features and retrieve data relevant to our study area. The data are processed using the OSMnx python library, which converts the map data into a graph structure suitable for applying pathfinding algorithms such as A* and Dijkstra.

The processed data are then analysed to determine optimal paths based on various criteria, including distance and number of turns. Visualization tools are employed to present the paths and analyse their performance. Fig. 3 shows the example of OpenStreetMap.

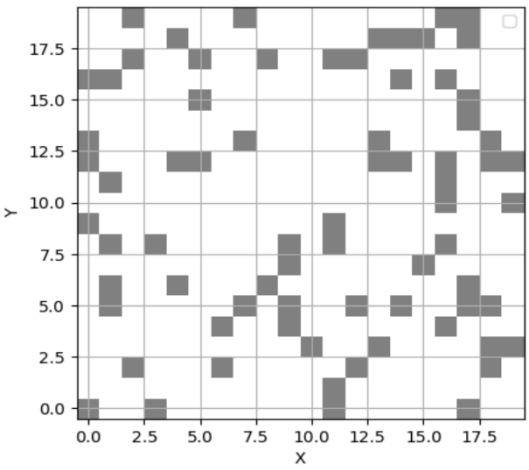

Figure 2. 2D Grid Map

Figure 3. OpenStreetMap

3.0 RESULTS AND DISCUSSION

The results of our pathfinding algorithms are evaluated using both synthetic (2D grid map) and real-world datasets (OSM map). The experiments on the 2D grid map revealed that the A* algorithm significantly has similar performance to the Dijkstra algorithm. The results tabulated in Table 1 demonstrate that both Dijkstra's algorithm and A* perform similarly in terms of path length and number of turns, but A* outperforms Dijkstra in computation time. Both algorithms produce paths of identical length (53 units) and the same number of turns (21), which is expected since A* uses an admissible heuristic, ensuring optimality comparable to Dijkstra. However, A* achieves this result more efficiently, requiring only 0.003 seconds compared to Dijkstra's 0.005 seconds. This

improvement in computation time is a direct consequence of A*'s heuristic-guided search, which prioritizes nodes likely to lead to the goal, reducing the number of nodes explored compared to Dijkstra's exhaustive outward expansion. In contrast, Dijkstra explores all possible nodes equally, leading to higher computational overhead. These findings highlight A*'s advantage in scenarios where the goal is known and a suitable heuristic can be applied, as it achieves the same optimal solution with reduced processing time. These results are illustrated in Fig. 4 for the Dijkstra and 5 for the A* algorithm. This result is generated at 30 x 30 map size with 30 % obstacle intensity. Table 1 presents the summary of the path planner performance at 30 % obstacle intensity.

Table 1: 2D Grid 30% Obstacle

	Dijkstra	A*
Path Length	53	53
No. of Turns	21	21
Time (s)	0.005	0.003

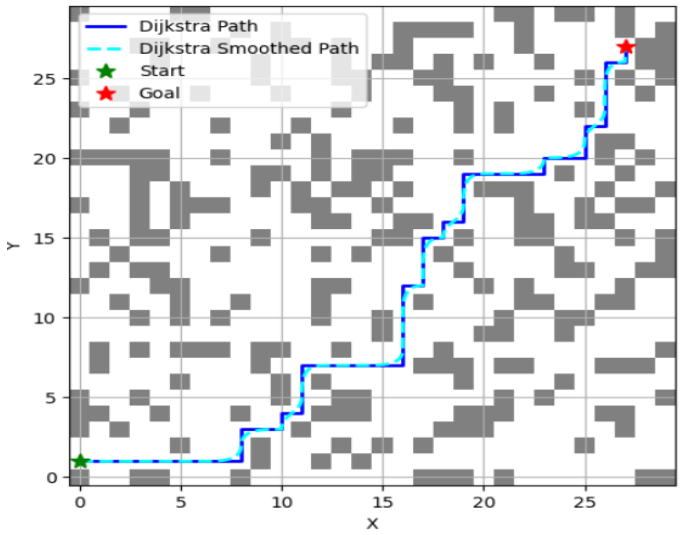

Figure 4. Dijkstra performance at 30% obstacles intensity

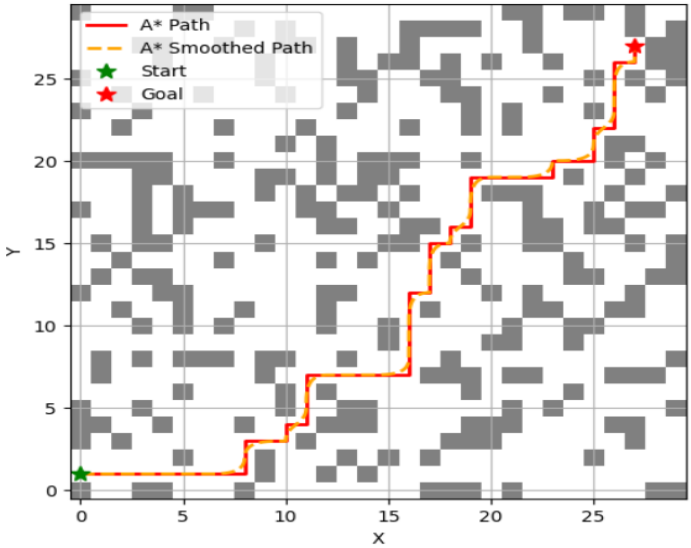

Figure 5. A* performance at 30% obstacles intensity

Table 2 presents a performance comparison of Dijkstra's algorithm and A* in a 2D grid environment with 40% obstacle density. Both algorithms generate paths of identical length (118 units), confirming their ability to find the shortest path. However, A* demonstrates a clear advantage in terms of smoothness and computational efficiency. The number of turns produced by A* (48) is significantly lower than that of Dijkstra (55), indicating that A* generates smoother trajectories with fewer directional changes. This improvement in smoothness is particularly important for applications requiring efficient navigation, such as robotics or autonomous systems, where excessive turns can increase mechanical strain and energy consumption. Additionally, A* outperforms Dijkstra in computation time, requiring only 0.0478 seconds compared to Dijkstra's 0.06320 seconds. This reduction in computation time highlights A*'s heuristic-guided search strategy, which avoids unnecessary node exploration by focusing on promising paths toward the goal. Overall, the results demonstrate that A* is more efficient and practical than Dijkstra in environments with higher obstacle densities, offering comparable path lengths while improving smoothness and reducing processing time. Navigation in this kind of environment can improve the energy efficacy of any mobile platform, for example, mobile robot.

Table 2: 2D Grid 40% Obstacle

	Dijkstra	A*
Path Length	118	118
No. of Turns	55	48
Time (s)	0.06320	0.0478

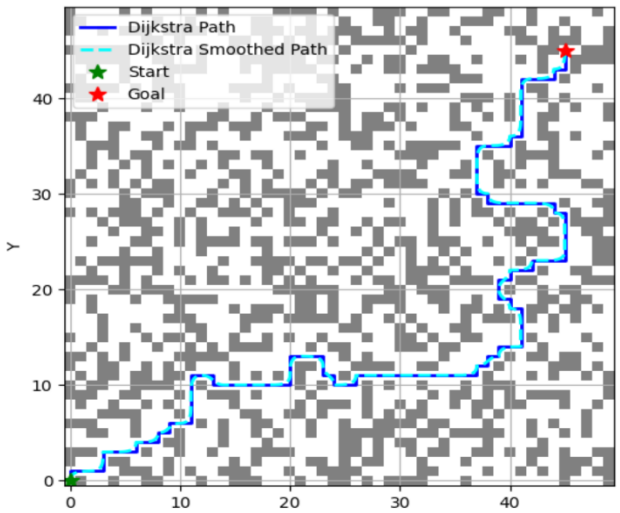

Figure 6. Dijkstra performance at 40% obstacles intensity

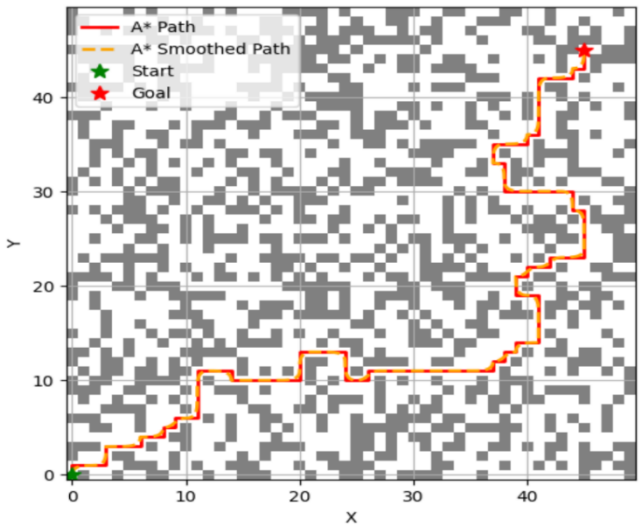

Figure 7. A* performance at 40% obstacles intensity

On the application of OpenStreetMap data for UiTM Pasir Gudang, Johor area, the A* algorithm continued to show superior performance, with smoother and more direct paths as shown in Fig. 8 and quantified in Table III. This performance is attributed to the heuristic guidance employed by A*, which enhances navigation efficiency. In contrast, while Dijkstra's algorithm provided valid paths, it was less efficient in complex route networks. Overall, the findings confirm that A* is a more effective path-finding solution for both grid maps and real-world scenarios.

The table compares Dijkstra's algorithm against the A* to find the shortest path cost and the number of nodes visited when we have starting node and destination. In all cases, both algorithms yielded identical path lengths, demonstrating that they successfully found the optimal path. For Destination 1, the path length was 602.03 meters, while for Destination 2 and Destination 3, the path lengths were 739.55 meters and 805.19 meters, respectively.

However, in terms of computational time, there is a slight difference. Dijkstra's algorithm took 0.0478 seconds for Destination 1, whereas A* took a bit more than it in terms of time i.e., 0.0632 while going from source node to target destination Node. In contrast, A* outperformed Dijkstra's algorithm for Destination 2, with the time of 0.0316 seconds versus 0.0469 seconds, while destination 3 had the time of 0.0476 seconds compared to Dijkstra's 0.0549 seconds. It can be seen that the path length of destinations 2 and 3 are significantly higher than destination 1.

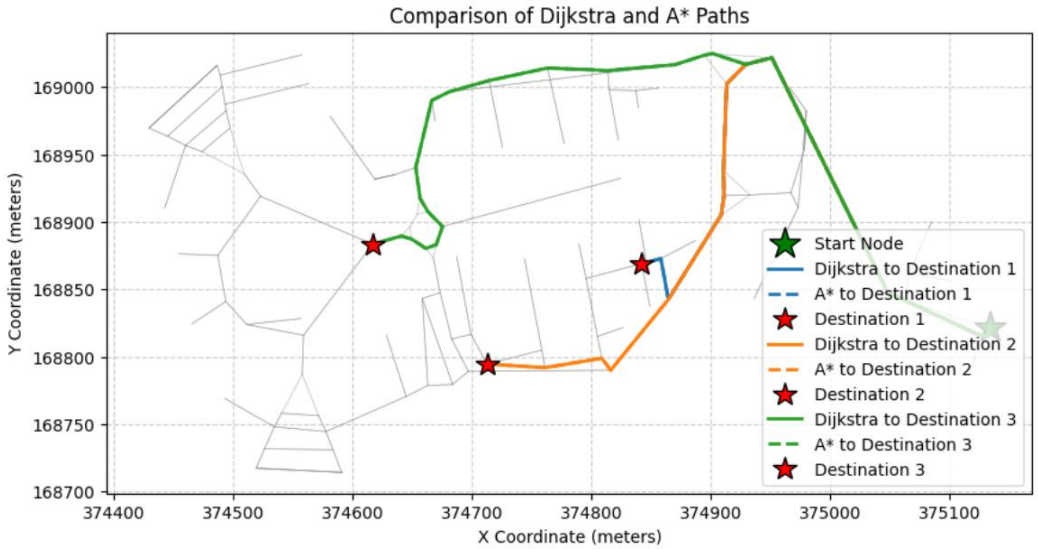

Fig. 8: Result on Openstreet Map

Table 3: Openstreet Map testing performance

Destination	Criteria	Dijkstra	A*
Destination 1	Path Length	602.03 meters	602.03 meters
	Time	0.0478 sec	0.0632 sec
Destination 2	Path Length	739.55 meters	739.55 meters
	Time	0.0469 sec	0.0316 sec
Destination 3	Path Length	805.19 meters	805.19 meters
	Time	0.0549 sec	0.0476 sec

4.0 CONCLUSION

This paper presents the comparison application of the Dijkstra and A* algorithms in terms of their performance. Both algorithms found the same optimal path in terms of length, as they are designed to meet this objective, but A* consistently outperforms Dijkstra in terms of speed. This efficiency arises from A*'s use of heuristics, allowing it to prioritize more promising nodes and explore fewer paths compared to Dijkstra's exhaustive approach. The differences in time are more noticeable as the grid size and complexity increase, making A* more suitable for larger environments where computational resources and time efficiency are crucial.

In the OpenStreetMap experiment with real-world road networks (UiTM Pasir Gudang, Johor), the performance differences between Dijkstra and A* are similarly reflected. Both algorithms yielded identical path lengths, but A* exhibited faster processing times for most destinations. While both are capable of handling complex, large-scale maps, the A* algorithm's heuristic-based approach makes it a better choice for real-time path finding tasks, especially when working with dynamic environments or where quick responses are needed. Overall, the experiment highlights A*'s advantage in terms of computational efficiency without compromising accuracy in both grid-based and real-world map scenarios.

However, obviously the limitation of the A* algorithm is dependent on grid resolution for pathfinding. A lower grid resolution can result in an inaccurate heuristic, leading to suboptimal paths, while a higher resolution increases computational costs. Balancing resolution and efficiency are crucial. Another limitation is A*'s scalability in large OSM networks. As the network size grows, the number of nodes and edges increases, causing higher memory usage and longer computation times. This can make A* computationally expensive, especially for real-time or large-scale applications, requiring optimizations like pre-computation or advanced heuristics to improve performance.

ACKNOWLEDGEMENT

The authors would like to thank the management of Electrical Engineering Studies, Universiti Teknologi MARA, Pasir Gudang Campus for full support of this research.

AUTHORS CONTRIBUTION

The authors confirm contribution to the paper as follows: **study conception and design:** Zakariah Y; **data collection:** Sufian M; **analysis and interpretation of results:** Wan Shuhafiza WI; **draft manuscript preparation:** Zakariah Y. All authors reviewed the results and approved the final version of the manuscript.

DECLARATION OF COMPETING OF INTEREST

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

REFERENCES

[1] M. Sadaf *et al.*, "Connected and Automated Vehicles: Infrastructure, Applications, Security, Critical Challenges, and Future Aspects," 2023. doi: 10.3390/technologies11050117.

[2] Enoch Oluwademilade Sodiya, Uchenna Joseph Umoga, Olukunle Oladipupo Amoo, and Akoh Atadoga, "AI-driven warehouse automation: A comprehensive review of systems," *GSC Advanced Research and Reviews*, vol. 18, no. 2, 2024, doi: 10.30574/gscarr.2024.18.2.0063.

[3] V. Engesser, E. Rombaut, L. Vanhaverbeke, and P. Lebeau, "Autonomous Delivery Solutions for Last-Mile Logistics Operations: A Literature Review and Research Agenda", *Sustainability,* 15(3)*,* 2023. doi: 10.3390/su15032774.

[4] J. R. Sánchez-Ibáñez, C. J. Pérez-Del-pulgar, and A. García-Cerezo, "Path planning for autonomous mobile robots: A review," *Sensors*, 21(23), 2021. doi: 10.3390/s21237898.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer Math (Heidelb)*, vol. 1, no. 1, 1959, doi: 10.1007/BF01386390.

[6] S. K. Sahoo and B. B. Choudhury, "A Review of Methodologies for Path Planning and Optimization of Mobile Robots," *Journal of Process Management and New Technologies*, vol. 11, no. 1–2, 2023, doi: 10.5937/jpmnt11-45039.

[7] A. Fitro, O. S. Bachri, A. I. Sulistio Purnomo, and I. Frendianata, "Shortest path finding in geographical information systems using node combination and dijkstra algorithm," *International Journal of Mechanical Engineering and Technology*, vol. 9, no. 2, 2018.

[8] S. Ergün, S. Z. A. Gök, T. Aydoğan, and G. W. Weber, "Performance analysis of a cooperative flow game algorithm in ad hoc networks and a comparison to Dijkstra's algorithm," *Journal of Industrial and Management Optimization*, vol. 15, no. 3, 2019, doi: 10.3934/jimo.2018086.

[9] M. Luo, X. Hou, and J. Yang, "Surface Optimal Path Planning Using an Extended Dijkstra Algorithm," *IEEE Access*, vol. 8, 2020, doi: 10.1109/ACCESS.2020.3015976.

[10] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah, and E. Gunawan, "A Systematic Literature Review of A*Pathfinding," in *Procedia Computer Science*, 2021. doi: 10.1016/j.procs.2021.01.034.

[11] H. Zhang, Y. Tao, and W. Zhu, "Global Path Planning of Unmanned Surface Vehicle Based on Improved A-Star Algorithm," *Sensors*, vol. 23, no. 14, 2023, doi: 10.3390/s23146647.

[12] H. Wang, X. Zhou, J. Li, Z. Yang, and L. Cao, "Improved RRT* Algorithm for Disinfecting Robot Path Planning," *Sensors*, vol. 24, no. 5, 2024, doi: 10.3390/s24051520.

[13] S. Mondal and B. Chen, "Development of Autonomous Vehicle Motion Planning and Control Algorithms with D* Planner and Model Predictive Control," in *Lecture Notes in Networks and Systems*, 2024. doi: 10.1007/978-3-031-47718-8_52.

[14] M. Rivai, D. Hutabarat, and Z. M. Jauhar Nafis, "2D mapping using omni-directional mobile robot equipped with LiDAR," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 18, no. 3, 2020, doi: 10.12928/TELKOMNIKA.v18i3.14872.

[15] A. J. Barreto-Cubero, A. Gómez-Espinosa, J. A. Escobedo Cabello, E. Cuan-Urquizo, and S. R. Cruz-Ramírez, "Sensor data fusion for a mobile robot using neural networks," *Sensors*, vol. 22, no. 1, 2022, doi: 10.3390/s22010305.

[16] W. Li, "Synthesizing Virtual World Palace Scenes on OpenStreetMap," in *Proceedings - 2023 7th International Conference on Computer, Software and Modeling, ICCSM 2023*, 2023. doi: 10.1109/ICCSM60247.2023.00021.

[17] V. Bulut, "Path planning for autonomous ground vehicles based on quintic trigonometric Bézier curve: Path planning based on quintic trigonometric Bézier curve," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 43, no. 2, 2021, doi: 10.1007/s40430-021-02826-8.

[18] S. Blažic, G. Klancar, M. B. Loknar, and I. Škrjanc, "Warehouse Path Planning Using Low-order Bézier Curves with Minimum-Time Optimization," in *IFAC-PapersOnLine*, 2023. doi: 10.1016/j.ifacol.2023.10.578.