OPTIMIZATION PATH PLANNING FOR MOBILE ROBOT

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Mechanical Engineering, Master's thesis

2025

Abdul Wahid Zaki

Examiners: Associate professor Huapeng Wu, D.Sc. (Tech.)

Associate professor Ming Li, D.Sc. (Tech.)

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Energy Systems

Mechanical Engineering

Abdul Wahid Zaki

Optimization path planning for mobile robot

Master’s thesis

2025

48 pages, 20 figures, 2 tables and 2 appendices

Examiners: Associate Professor Huapeng Wu, D.Sc. (Tech.) and Associate Professor Ming Li, D.Sc. (Tech.)

Keywords: Hybrid algorithm, faster path planning, pathfinding time, algorithm, Multi-Landmark Bidirectional A* algorithm.

Effective and reliable path planning is essential regarding the autonomous mobile robots that work in complicated areas. This thesis proposes a faster path planning algorithm named Multi-Landmark Bidirectional A* (MLB A*) algorithm in order to achieve faster path-finding time, better navigation work in highly dynamic and partially known surroundings. Conventional A* and Bidirectional A* algorithms are useful though these algorithms have weaknesses in both computational performance and adaptability to large scale or high-density maps. The suggested MLB A* approach uses strategically positioned landmarks to exhibit the bidirectional search procedure and this will shorten the space in the search and lessen the path-finding time. The algorithm has great advantage, as it starts a forward as well as a backward search from the starting and goal positions and sets them toward intermediate landmark nodes which are nodes on the path of the goal nodes, making the search fast and efficient. In both simulated and real-world settings, experimental results show that MLB A* is more efficient in terms of faster path-finding time and success rate as compared to the standard A* and the traditional Bidirectional A* algorithms in terms of obstacle distribution complexity. This strategy is prospective in real-time robotic positioning in automation of warehouses, products delivery or service robots.

ACKNOWLEDGEMENTS

I would like to convey my sincere gratefulness to my LUT University supervisor, Associate Professor Huapeng Wu and Associate Professor Ming Li for their support, endless guidance and their wise feedback during the thesis.

Finally, thank my family and friends for always supporting me.

Abdul Wahid Zaki

Lappeenranta 19.11.2025

ABBREVIATIONS

Abbreviation/Word	Meaning
A*	A star algorithm
MLB A*	Multi-Landmark bidirectional A* algorithm
RRT	Rapidly-exploring Random Tree algorithm
D*	Dynamic Star algorithm
ACO	Ant Colony Optimization algorithm
SA	Simulated Annealing algorithm
GA	Genetic algorithm
PSO	Particle Swarm Optimization algorithm
ANN	Artificial Neural Networks algorithm
PRM	Probabilities Roadmap algorithm
DWA	Dynamic Window Approach algorithm
ASL-DWA	A* Leading Dynamic Window Approach
GCSM	Grid cell selection method
IEACO	Intelligently Enhanced Ant Colony Optimization algorithm

Table of contents

Figures

Tables

1 Introduction

The autonomous mobile robotics technology is an essential element nowadays due to its pivotal role in increasing the operational safety, effectiveness, minimizing the number of system errors and reduces its effect on the environment. Combining a highly accurate environmental perception, efficient path planning and intelligent decision- making, robots will be able to move independently, which allows them to be a fundamental part of intelligent operations systems of the future. This technology is stood firmed on four features. To begin with, the environmental perception offers the possibility to interpret and assess the surrounding environment (Baek et al., 2021, p. 1). Second, the action strategies are characterized by the behavioural decision-making (Garrido et al., 2022, p. 100349). Thirdly, path planning is used to identify the optimal paths (Zhang et al., 2018, p. 1). Last but not the least, motion control is required to support accuracy in the execution of movement (Song et al., 2023, p. 2).

Planning the path direction of a robot is the process of identifying an optimal path to be taken by a robot from its origin point to a destination mark, avoiding any obstacles in the environment also ensuring the safety and efficiency of the robot. Conventional algorithms for path planning such as RRT*, A*, Dijkstra, Bidirectional A*, Ant Colony Optimization, D* and Artificial Fish Swarm have been proved effective to measure the shortest path in known and static environment. Meanwhile in dynamic environment, local path planning algorithms are needed to incorporate with these global algorithms for ensuring real time path navigation (Liu et al., 2023, pp. 821–822).

A mobile robot implements three crucial constituents for performing the tasks effectively: localization, mapping and path planning. Localization is the process of determining the position of a robot with respect to the immediate environment, and mapping is the creation of an organized structure in the environment to create navigation for moving of the robot. Finally, path planning develops the most effective and the shortest path to achieve a certain task (Fethi et al., 2018, pp. 1–2). The path planning can be divided broadly into two parts, namely global path planning and local path planning. Global path planning indicates the act of building a whole path direction between the Starting point and the goal point with the objective by making use of the accessible map data and knowledge regarding the environment (Wang et al., 2020, p. 1). In contrast to the global planning, the local path planning is controlled dynamically, with the short-term trajectories adjusted to the real-time environmental conditions (Sánchez-Ibáñez et al., 2021, pp. 23). The management of moving obstacles, collision prevention, and curve smoothing are also implemented to make navigation safe and efficient.

Numerous researches have been devoted to the creation of various path planning algorithms may follow in both static and dynamic environment. Improved methods based on heuristics are typically used in applying the global path planning, such as the A* algorithm (Xingdong Wang et al., 2022, p. 1320), D* algorithm (Stentz, 1994, p. 3311), Dijkstra algorithm (Hartomo et al., 2019, p. 2) and its derivatives. Nevertheless, as much as these methodologies are effective, they are not usually effective in offering the best solutions in complicated situations. In order to solve these problems, methods which apply artificial intelligence-based techniques known as meta- heuristic algorithms have been launched. These are such techniques as Artificial Neural Networks (Yu et al., 2020, pp. 12), Genetic Algorithms (Yanrong Hu et al., 2004, p. 4350), Ant Colony Optimization (Fu et al., 2025, p. 2), Particle Swarm Optimization (Song et al., 2021, p. 1), and Simulated Annealing that would provide more flexibility and possible adjustments to the intricate tasks of path planning.

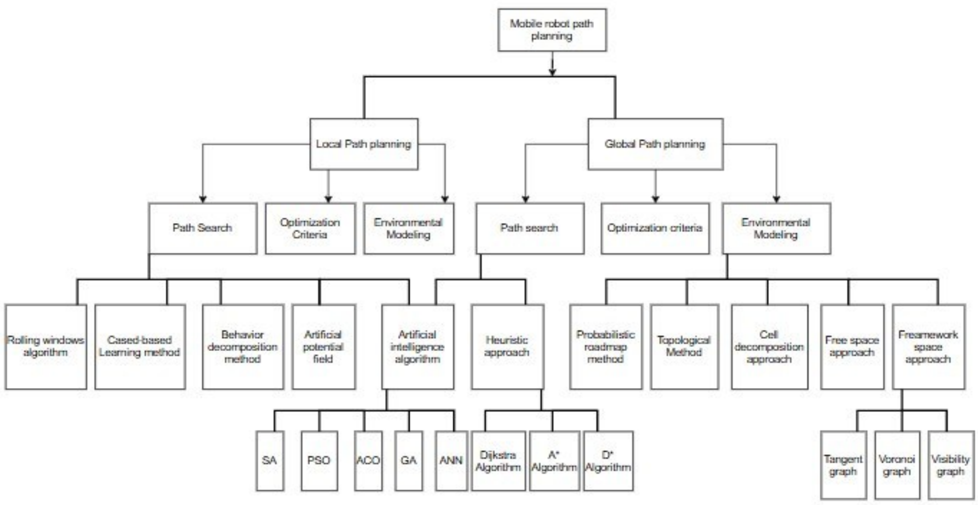

Figure 1. Classification of Algorithm.

This thesis introduces a new hybrid approach of existing Bidirectional A* algorithm for faster pathfinding named Multi-landmark Bidirectional A* algorithm that develops the optimal path in faster time among the rest of the algorithms. This study compares the performance of different algorithms in terms of computational pathfinding time. The implementations and testing of the algorithms are done in Visual Studio Code using Python scripts. A Pygame grid-based map is designed to evaluate performance and this has an identified starting point and an destination mark; some obstacles, in the form of walls and objects, have irregular shapes. In this architecture, a certain degree of immediacy in the setting is brought into play. Each algorithm is expected to result in the finding of a path between the origin point and the destination mark and both the travelled distance and pathfinding time are documented. Finally, the result generates the graphical path in the simulated environment.

There are five sections of the thesis. The first part offers a general introduction that presents the readers with an overview of the problem, the motivation behind the research, its key objectives, and contributions. The second chapter is dedicated to the literature review and the associated studies of the matter. In the third section, the approach such as algorithm is described that is applied in this thesis and methods of comparison. The fourth section is the focus of the results and discussion of the work. In the last part; it symbolizes the conclusions regarding the work and recommendations on the further progress.

1.1 Problem definition

Mobile robot pathfinding problem refers to the determination of the most efficient path between an origin point and a specific destination point within the surroundings and avoiding obstacles and following the mobility limitations of a robot. Such issue is also essential in dynamic or cluttered environment such as the warehouses when the robot has to move in real-time and minimizing pathfinding time, energy, or distance, assuring safety and avoiding any collisions. Such factors as terrain, turning radius, robot kinematics, and traffic of other agents are usually taken into account for the optimization. The problem can be solved with several algorithms, one of them being the heuristic-based algorithm such as Dijkstra, A*, Bidirectional A*, the sampling-based algorithm such as RRT* and the meta-heuristic algorithm such as Genetic Algorithms and Particle Swarm Optimization, all having a trade-off between computation time and path optimality. It is the difficulty in finding a path that is both practicable and shortened under the dynamic nature of the environment and constraint in computational ability (Lavalle, 2006, p. 27).

1.1.1 Problem of different algorithms

The direct search algorithms have a major role in the autonomous robot navigation field. The algorithm in Dijkstra is able to calculate the shortest path without depending to heuristic information. A* algorithm is a common algorithm applicable in determining the shortest path with a heuristic direction, but in the complicated setting, its search space may be extensive, and hence it is computationally costly (Luo et al., 2020, pp. 147829-147830). The overall performance of this heuristic function also strictly depends on its selection. Several modified variations of A* algorithms have been developed in order to enhance the efficiency. Weighted A* functions faster as heuristic estimates are the priority over actual costs, so it can be applied to real- time navigation, but with this trade-off it may diminish the optimality of the paths (Liu et al., 2022, p. 2). D* Lite is created to operate in dynamic environments but it re-computes only the required portions of the path rather than recalculating all the path components, making it more efficient at the expense of complexity (Alsulaiman et al., 2011, pp. 173174). Theta* uses line-of-sight checks to enable diagonal motions, but it needs a mildly more extensive computation (Daniel et al., 2010, p. 570). The Probabilities Roadmap (PRM) is another algorithm that is effective in a high-dimensional space, as it constructs a network of sampled points. Nonetheless, dense sampling is required to obtain high-quality paths, which rises the planning time and additional smoothing steps (Gang et al., 2015, pp. 148–149). Another sampling-based approach that is also very efficient in the identification of the shortest paths is called RRT* algorithm which is very fast in finding the shortest path and is at the same time inefficient in reaching the optimal solution (Xuewu Wang et al., 2022, p. 686). In the meantime, the Ant Colony Optimization (ACO) algorithm replicates the foraging behavior of ants, as it assists robots to solve the problems in complicated environments. Nevertheless, conservative ACO computes slowly and frequently result in undesirable patterns (Sharan et al., 2025, pp. 10–11).

1.2 Motivation

The research motivation for the faster path-finding algorithm of mobile robots is emerged due to the fact that it is necessary to increase their efficiency in relation to path-finding time that is flexible into a complex and dynamic environment. With the growing demand of autonomous robots in industries, logistics and transportation, path planning with the minimum possible time is crucial in ensuring the smooth movement, avoiding collisions with obstacles and minimizing the duration of traveling and energy expenditure. Conventional algorithms such as Dijkstra, A*, Bidirectional A*, RRT and Ant Colony Optimization have become useful, but these algorithms have been challenged with their limitations like slow processing, high path-finding time, generation of redundant nodes, local minima traps and lack of scalability in practical environment (Sharan et al., 2025, p. 15). These constraints enlighten the need of creating upgraded and hybrid algorithms, which integrate the advantages of various methods, adaptive heuristic functionality as well as global and local planning methods. This also paves the way of achieving a real-time, high-precision, robust and faster path-finding algorithm that can capable of managing uncertain circumstances, enduring high-performance and operational reliability.

The knowledge about the application of various algorithms incorporated simultaneously to determine the faster path planning in mobile robot is important to consider the robustness, efficiency and adaptability in distinct environments. By integrating global planners like A* or Dijksra algorithm (optimal for static maps) with local reactive process such as Dynamic Window Approach (DWA) or Potential Fields (effective to navigate dynamic obstacles), real-time navigation can be improved without averting local minima (Katona et al., 2024, p. 28). RRT* or PRM are sampling-based methods which are most effective in high-dimensional spaces. On the contrary, multi-objective measures like path length, path-finding time and energy efficiency are optimized through heuristic or evolution based (e.g. Genetic Algorithms) algorithms. This hybrid approach creates a balance in path-finding time, optimality and fault tolerance which assures the reliability in complicated situations (Shanmugaraja et al., 2025, p. 16).

1.3 Objectives

In this research work, the focal point of path planning is the creation of a hybrid algorithm by the combination of Multi-Landmark and Bidirectional A* (shortly named as MLBA*) algorithm to find the path faster for mobile robot. The main objective of this work is to pave the way of creating a new hybrid MLB A* algorithm which will be faster in terms of path finding time compared to other conventional algorithms and evaluate the algorithm performance utilizing into complex warehouse environments.

1.4 Contributions

This research focuses to generate an optimal hybrid algorithm named Multi-Landmark Bidirectional A* algorithm to find the path faster (reducing path finding time) for mobile robot, smoothing the path and also compare the path created through MLB A* to the other conventional algorithm like A*, Dijkstra, and Bidirectional A* algorithm generated paths.

- MLB A* algorithm finds the path faster than other algorithms.
- It compares the path finding time through MLB A* algorithm to that of other algorithms and this path maintains a distance from any obstacles to avoid the collision.
- This thesis main proposal is to find the shortest path faster for the mobile robot and finally to create a smooth path in a warehouse environment.

2 Literature Review

The literature review represents different types of traditional algorithms including Dijkstra algorithm, A* algorithm, Bidirectional A* algorithm, Ant colony optimization algorithm, the limitations of these algorithms and their solution that made before.

2.1 Dijkstra Algorithm

Dijkstra algorithm, named after the Dutch scientist Edsger W. Dijkstra, is a classic graph search-based algorithm that tracks the corresponding shortest path distance from the starting node to all of the surrounding nodes. At first this algorithm chooses the node with the shortest distance from the starting node and then consequently path distances surrounding that node is recorded. This would be continued till all the nodes are incorporated and the algorithm determines the shortest path from starting node to all of the other nodes. (Singal et al., 2014, pp. 12–13) In table 1, several studies carried out to overcome the drawbacks of Dijkstra algorithm through modifying this algorithm are given.

Table 1. The drawbacks of Dijkstra algorithm through modifying this algorithm

Drawbacks	**Improvement in Dijkstra algorithm**	**Improvement method**	**Reference**
Lower optimal path planning and higher processing time	Equipped with an ultrasonic sensor as an obstacle detector	Whenever an obstacle was detected in a node, the path would be updated immediately neglecting that particular node	(Alshammrei et al., 2022, pp. 5949–5952)
Lower efficiency against static and dynamic obstacles	Incorporating Canny edge detection and a dynamic point setting algorithm	Complex navigation to obstacles was achieved and optimized path planning in real time	(Yang, 2024, p. 2)
Lower accuracy of path optimization	Delaunay triangulation for mapping the surface optimization	Through that strategy, 2D developable surface channel would create	(Luo et al., 2020, pp. 147836–147837)

		and determine the optimal route	
Higher path length and turning angle	Combined with eight angle search method in lieu of the traditional four angle search method	Eight directions could be attained from the starting node and thus distance to the end node would be reduced significantly	(Sun et al., 2021, pp. 3–5)

2.2 A* Algorithm

A* algorithm is well known for its vital role in wide range of contexts such as Position finding in robotics (Duchon et al., 2014a, p. 61), network routing (Sun et al., 2024, pp. 1–2), AI and machine learning (Hoang Linh et al., 2025, p. 10), advanced logistics system (M. Yu et al., 2023, p. 15).

A* algorithm is considered as the most common type path finding algorithm with some advantageous features such as compatibility with wide range of contexts, excellent optimality through determining the shortest path for a specific goal, fast real time response, incorporation of customized instruction by changing cost and heuristic function.

A* algorithm, an extension version of Dijkstra's algorithm, is such a graph search-based planning algorithm that applies heuristic function to figure out the shortest path relied on cost estimation. Unlike the Dijkstra's algorithm, it does not favour only the positions near at the starting node but also implies the heuristic information to favour the nearest positions at the ending node.(Tang et al., 2025, pp. 3–4) The difference of path selection between Dijkstra and A* algorithm is represented in figure 2.

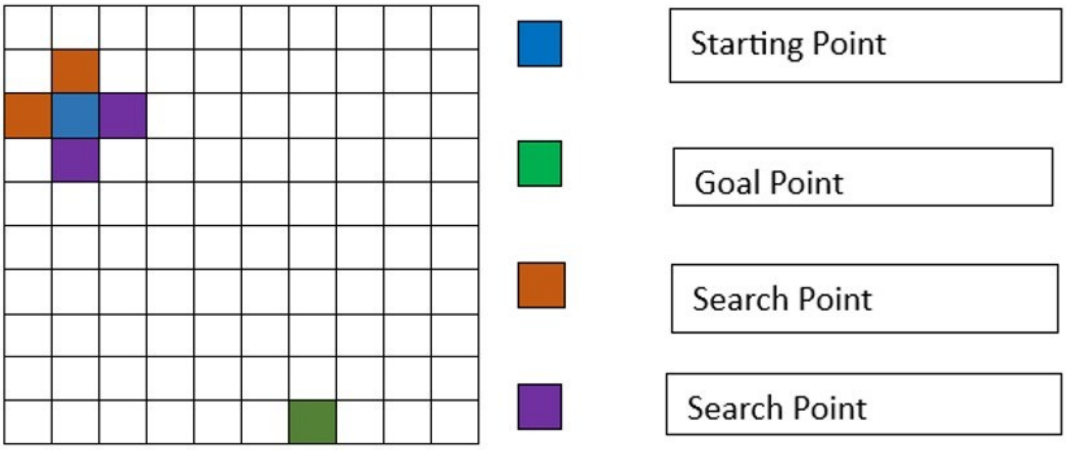

Figure 2. The path searching through Dijkstra and A* algorithm

The former one explores all reachable positions (indicated by purple and brown squares in fig 2) while the latter one prospects only the particular positions (indicated by purple squares in fig 2) considering the goal point (indicated by green).

However determination of the most optimized path length and higher success rate are the major challenges for this type of path planning algorithm. Fu et al. (2018) experimented an advanced A* algorithm consisted of a pre-processing and post processing stage to achieve a superior optimized path length. During the pre-processing stage, a local path was created between the current and end node before each path finding while in the post processing stage an optimized path length was achieved through removing intermediate nodes. (Fu et al., 2018, pp. 19–20)

Liu et al. (2022) proposed an improved A* algorithm, named A* Leading Dynamic Window Approach (ASL-DWA) for serving two pivotal roles to counter the drawbacks of A* algorithm. A hybrid heuristic function was introduced to eradicate the large number of path finding nodes. Moreover a modified global path yaw angle was designed considering the behaviour between real time position and global path position to become more efficient towards unknown obstacles. (H. Liu et al., 2022, p. 99499) Zhang et al. (2021) designed a novel A* algorithm that was very effective against static and dynamic obstacles in terms of path length and path processing time. The obstacles were identified as static and dynamic obstacles utilising GoogleNet and then ray tracing algorithm was adapted for avoiding these. Moreover, this study was infused with evaluation function improvement, adaptive step size adjustment and path planning with quadratic B-spline smoothing that resulted in better outcomes than traditional A* algorithm. (Zhang et al., 2021, p. 420)

2.3 Bidirectional A* Algorithm

Bidirectional A* algorithm, an extended variant of A* algorithm, is a heuristic search based algorithm that performs simultaneously forward (towards the ending node) and backward search (towards the starting node) which continues till the intersection of the two paths produced from the searches. It outperforms the traditional A* algorithm in regards to searching speed and time, memory capacity, search efficiency, number of path nodes.

Though this algorithm has better efficiency relative to the unidirectional search algorithms, several drawbacks such as higher memory consumption, higher failure rate of convergence in case of narrower paths, known location of the end node in advance have turned down this algorithm unsuitable for complex environments. (Pedamkar, 2023)

He et al. (He et al., 2023, p. 1518) proposed an improved bidirectional A* path planning algorithm that exhibited excellent processing time and path optimization through incorporating dynamic weights and corner optimization. Zhang (Zhang, 2024, pp. 5–6) suggested that bidirectional A* algorithm would exhibit better performance in terms of search space and search efficiency in comparison with A* algorithm for wider problem space known end nodes position. Li et al. (Li et al., 2024, pp. 3–4) reported a variable neighbourhood search strategy based bidirectional A* algorithm to overcome the challenges of higher number replaced with 8-11-13 neighbour search, adjusting the heuristic function with a novel distance function, incorporating pruning and cubic spline interpolation – these were the key innovations feasible for static environments incorporated with an improved evaluation function with a view to reducing the number of nodes, path length and navigation time against traditional bidirectional A* algorithm. In another study, a bidirectional virtual search algorithm associated with Grid cell selection method (GCSM) was developed that exhibited superior outcomes in terms of key performance indexes such as computational time, path optimization, path efficiency, success rate, real time adaptation, lower memory consumption and space expansion with compared to traditional algorithms in static and dynamic both environments. (Kumar et al., 2025, p. 2)

2.4 Ant Colony Optimization Algorithm

Ant colony optimization (ACO), proposed by Marco Dorigo and colleagues, was developed by the colonial survival characteristic of ants through which ants determine the shortest path between nest and feeding source. During food sourcing, the ants initially spread out in a random manner from their nests and spill out a chemical pheromone along with their movement. They are inclined to select the path that is spotted with high concentration of pheromone. Whenever an ant finds a food source, it immediately assesses the amount and quality of food which in turn results in the concentration level of pheromone trail (Blum, 2005, pp. 355–356). A detail representation of this algorithm is given in figure.

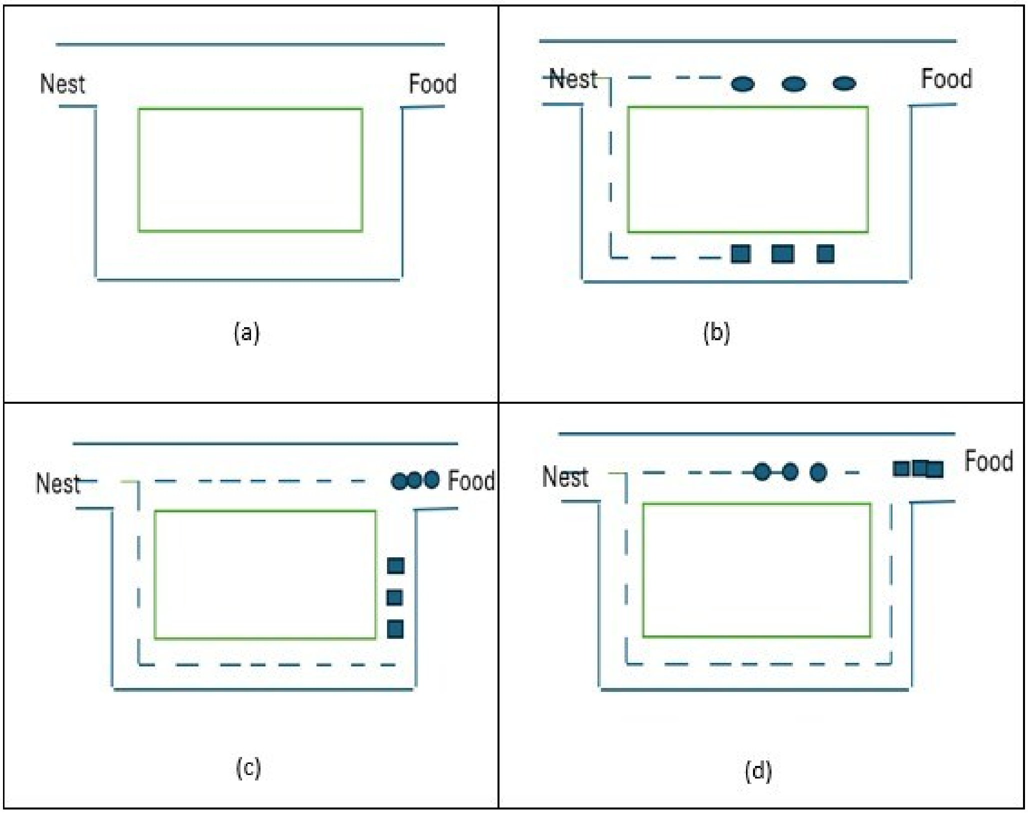

Figure 3. (a) No sign of pheromone because all ants are inside the nest. (b) Half of ants choose short path and remaining of ants choose long path (c) Since the ants who chosen the short path have reached the food storage earlier, the probability of r

Ant colony optimization is characterized with impressive adaptability even in dynamic environments, robust global search ability, parallel computing capability, inherent feedback system, heuristic searching mechanisms, indirect communication between the agents- all of these features make this algorithm appropriate for complex navigation purposes. (Xu et al., 2023, p. 6; Zhou et al., 2024)

Despite the fact that this algorithm system possesses several advantages in the path planning of mobile robots, still it is lacked behind due to some shortcomings such as easy stagnation, high search and processing time, local optimization. (Akka & Khaber, 2018) Several researchers have implemented some improvements to overcome these challenges.

Liu et al. (Liu et al., 2016, p. 5838) implemented a novel combined system of pheromone diffusion and geometric optimization to improve the processing time. Path planning guidance was reinforced and number of ants characterized with incomplete paths was reduced which consequently resulted in lower processing time. Li et al. (Li et al., 2025, p. 5) developed an intelligently enhanced ant colony optimization algorithm (IEACO) incorporated with six novel approaches, showed outstanding path planning performance against traditional ACO algorithm in terms of convergence speed, path length, planning stability, computational efficiency, energy consumption. Incorporation of pheromone distribution, introducing ε-greedy strategy, adjusting exponential strategy, establishing heuristic function considering turning angle and path distance, incorporating a compatible pheromone update mechanism and a path evaluation framework – all of these key changes turned that algorithm a potential approach towards real world dynamic applications.

Wu et al.(Wu et al., 2023, pp. 2–3) reported an advanced ant colony optimization through incorporating some innovations such as incorporation of triangle inequality idea, a pseudo-random state transfer strategy, the advanced model of partitioned pheromone and the backtracking mechanism. This improved ACO outperformed several variants of ACO, A* and Dijkstra algorithm in terms of various performance indexes such as faster and smoother processing, shorter path length with the lowest standard deviation.

3 Method

The Multi-Landmark Bidirectional A* (MLB A*) algorithm is a hybrid pathfinding algorithm which is designed to identify the shortest paths on a two-dimensional occupancy grid while considering both distance and terrain difficulty. The MLBA* algorithm is selected because it extends the classical A* approach by integrating heuristic landmark estimates and bidirectional search, both of which significantly improve efficiency in large or complex navigation spaces. Moreover, the path smoothing makes sure that the path obtained will be suitable to use in real robotic applications. It depicts the grid in form of a graph, uses strategically selected landmarks to come up with effective distance estimates and does concurrent searches both at the start and the goal point to minimize the time of computation. Furthermore, it also includes simplification of the paths to generate routes that are feasible to use in real-world scenarios including navigation of mobile robots. This MLB A* algorithm is introduced in the method section in a simple appearance with only minimum mathematical complexity thus making it easily available to other people working in the field of navigation and robotics.

Multi-Landmark Bidirectional A* algorithm is used to find the faster path and it does so by:

- Making the map a graph of the interconnected points.
- Estimation of distances with the help of special points (landmarks).
- Seeking the path from the start and goal points simultaneously.
- Making the generated path straight so as to make it feasible.

MLBA* algorithm is a path-finding algorithm that uses the traditional A* algorithm which discovers paths in a grid where the squares on a grid have a cost (e.g. low cost to easy terrain, high cost to hard terrain or very high cost to obstacles). This formulation is consistent with modern path planning research, which emphasizes terrain-awareness to achieve practical robot navigation. (Chen et al., 2023)

The Multi-Landmark Bidirectional A* algorithm improves on A* algorithm by:

- Landmark Estimates: Using the map corners and the centre to guess how far it is to the goal.

- Two-Way Search: Searching from both the start and goal, meeting in the middle to save time.
- Terrain Focus: Choosing paths that avoid hard or uneven terrain.
- Path Smoothing: Making the path direct and robot friendly.

3.1 Main Concepts of MLBA* algorithm

The steps of MLBA* algorithm is represented below:

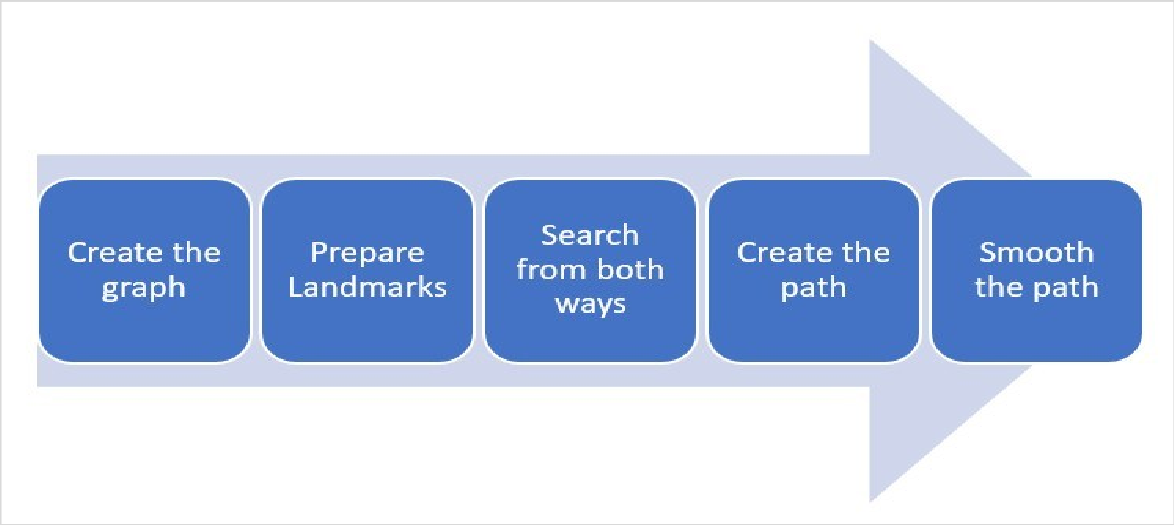

Figure 4. MLBA* algorithm steps

3.1.1 The Grid as a Graph

The map is represented as a grid where each square is assigned a cost value (e.g., 0 for easy terrain, 100 for difficult terrain, and 255 for walls). Each walkable square with a cost less than 253 is treated as a point in the graph, and each point is connected to its eight neighbours (up, down, left, right, and diagonals). The cost of moving between two points is determined by three factors: the distance between them (1 for straight moves and approximately 1.4 for diagonals), the average terrain cost of the two squares, and the difference in terrain costs between the two positions. Mathematically, the movement cost from point A to point B is given as:

- Cost = Distance × (1 + (Average Cost of A and B)/25 + 5 × | Cost of B - Cost of A |)

Which ensures that the selected path remains short while avoiding areas of high or uneven terrain difficulty.

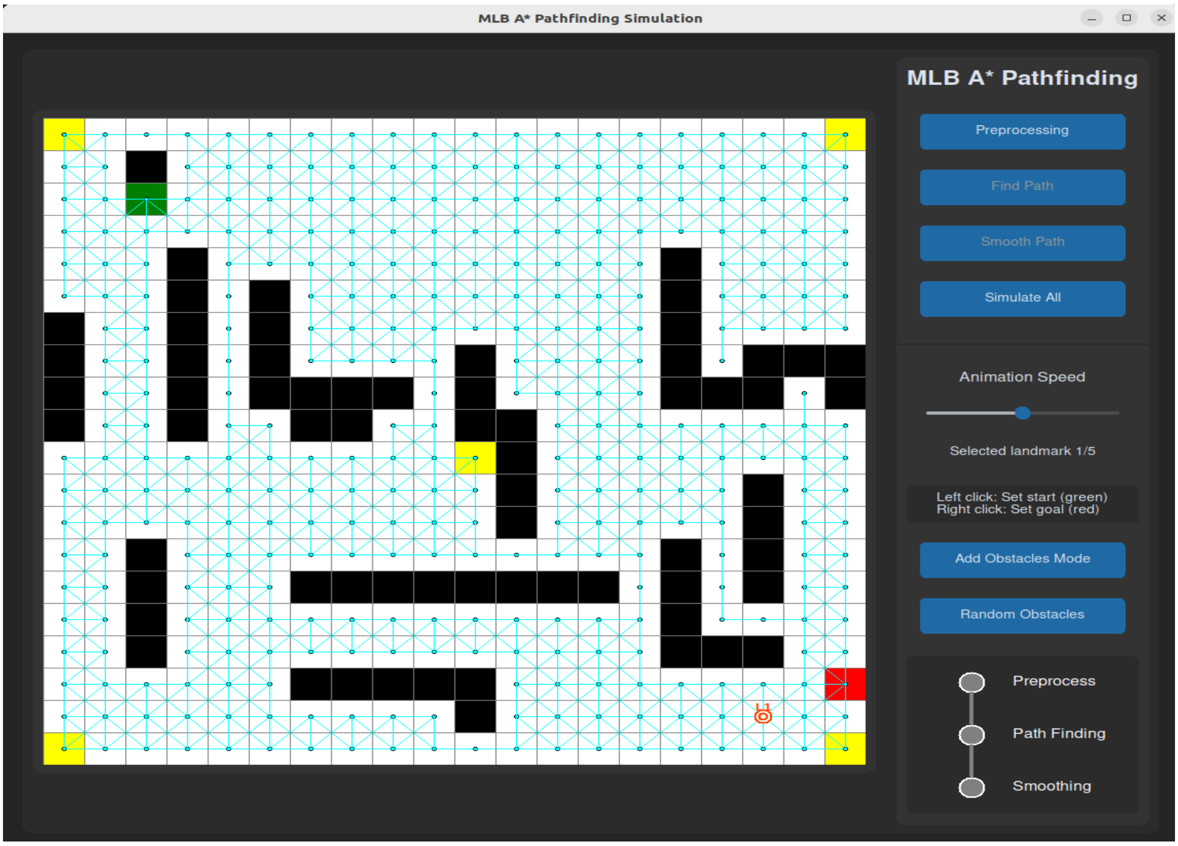

Figure 5. Creating the Graph

3.1.2 Estimating Distances with Landmarks

To generate faster path planning, the Multi-Landmark Bidirectional A* algorithm estimates the distance from any given point to the goal using two complementary heuristics. The first is the straight-line estimate, which calculates the direct Euclidean distance between the current point and the goal. The straight line to the goal:

$$\text{Distance} = \sqrt{(xgoal - x)^2 + (ygoal - y)^2}$$

The second is the landmark estimate, where five main landmarks (usually the four corners and the center of the grid) are predetermined and the shortest possible route of each of the landmarks to every other point is precalculated. The heuristic is then set as the maximum absolute difference between the distance of a point and a landmark and the distance between the goal and also the identical landmark. The landmark estimate is given below:

$$\text{Landmark Estimate} = \text{Largest} \mid \text{Distance to Landmark} - \text{Distance to Goal} \mid$$

The itinerary algorithm elects the larger of the two estimates and modifies it with the addition of the terrain cost which results in the final heuristic function as:

$$\text{Final Estimate} = \text{Largest of (Straight-Line, Landmark)} \times (1 + \text{Terrain Cost}/50)$$

This hybrid heuristic search helps to find the feasible paths and avoids overestimation of the actual distance.

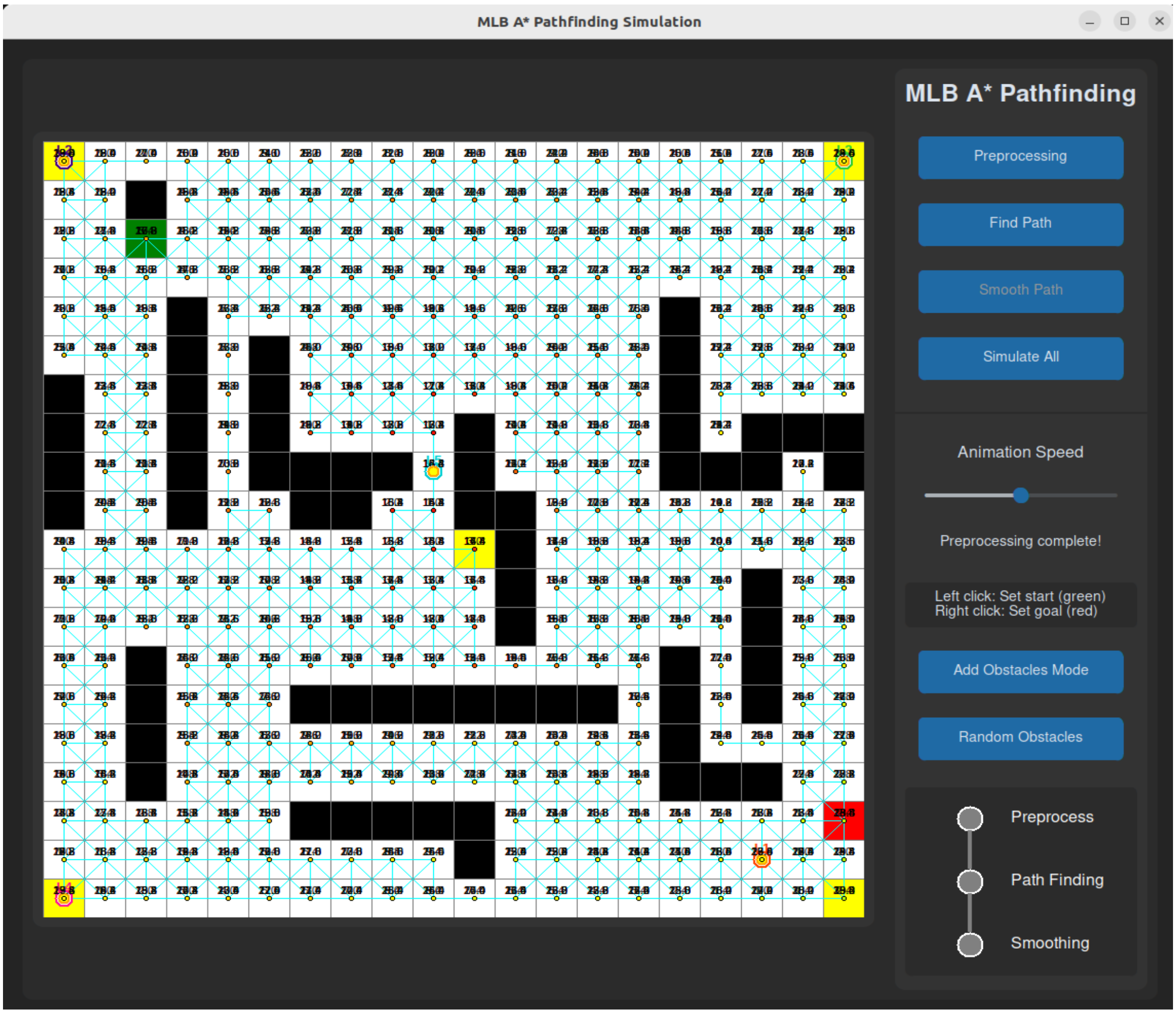

Figure 6. Prepare Landmarks.

3.1.3 Searching from Start and goal points simultaneously

Multi-Landmark Bidirectional A* algorithm is a bidirectional search algorithm, which expands simultaneously in both directions starting at the start and ending up at the goal and vice versa. Once the two searches intersect at an intermediate node, the algorithm incorporates the partial paths to create the complete path. This bidirectional method is more efficient as the overall search space is minimized as each search will only search a part of the map and not the whole environment. The determination of the total cost at the meeting point is the addition of the cost from the start, cost from the goal and a small modification of the terrain differences to ensure accuracy and to guarantee a good computational efficiency.

- Total Cost = Cost from Start + Cost from Goal + Small Terrain Cost

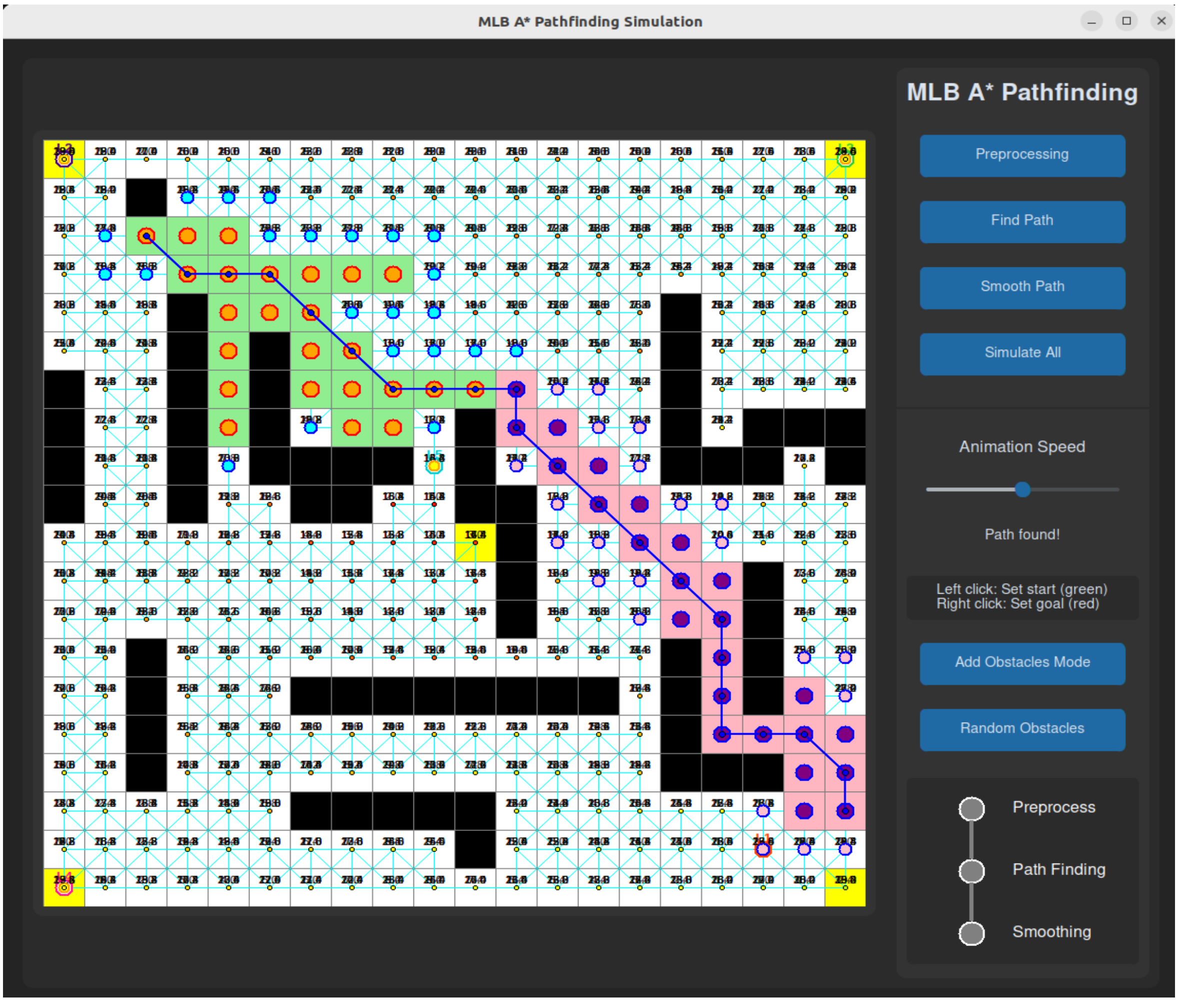

Figure 7. Search from both ways and prepare the path.

3.1.4 Smoothing the Path

On the first trajectory originated using the algorithm can include redundant turns because of the grid construction; therefore, a smoothing procedure is used to achieve the best path. This process analyses three consecutive points (e.g. A, B and C) and verifies the fact whether a direct line between point A and point C is free from hindrance and is linked with less traversal costs. When the direct line is valid, the point B is eliminated and thus the path becomes simplified and when the line is not valid, point B is not eliminated. Line segment is acceptable when there is a very low highest cost as well as the average cost (e.g. below zero under proper situation). This smoothing process will reduce the unnecessary stops in the path, which will lead to a short path, more natural path and easy to navigate by the robot.

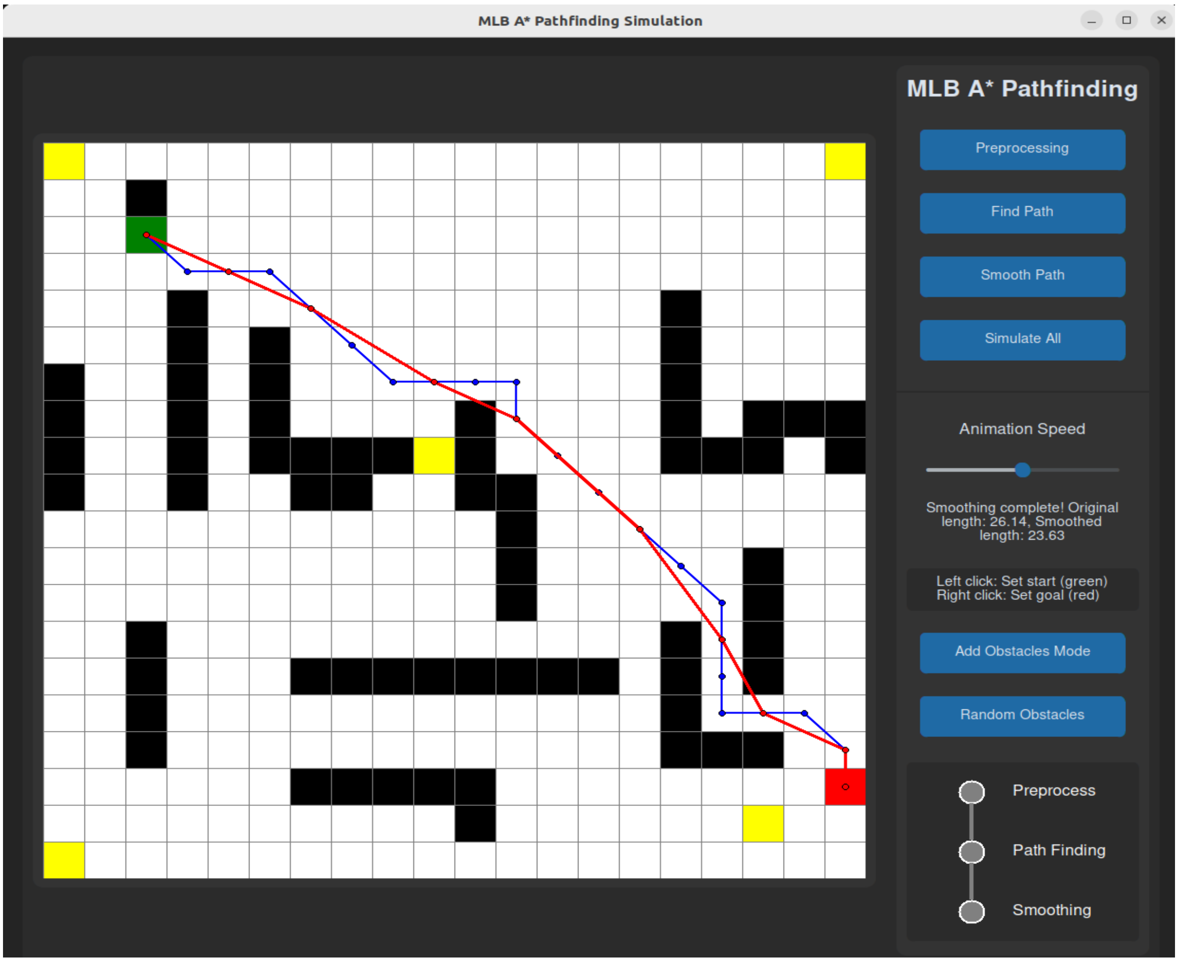

Figure 8. Smoothing the path

3.2 Mathematical Analysis

As a measurement of the computational efficiency of the Multi-Landmark Bidirectional A* (MLBA*) algorithm, it was mathematically analysed and the complexity of major steps which are landmark setup, bidirectional searching, and path smoothing were estimated.

Grid Size Consideration:

A grid of H rows by W columns contains HW points (squares) with approximately 8HW connections.

Landmark Setup:

During the initial process, five landmarks are chosen to calculate the distance to every grid point. This pre-processing stage makes sure that several heuristic estimates would be useful in the search process. The calculating time required to do this step may be written in the form:

- $\text{Time} \approx 5 \times HW \times \log(HW)$

This stage is computationally costly, but it is executed once and after that the result is reused to do numerous searches.

Searching Phase:

During the search part, the algorithm uses the bidirectional search method with the start and the goal node as starting point. Approximately $\sqrt{HW}$ nodes are searched each in every search, and the total time of search is:

- $\text{Time} \approx 2 \times \sqrt{HW} \times \log(HW)$

This method saves a lot of time on the searching process as opposed to A*.

Path Smoothing:

After the valid path has been acquired, it will undergo a process of smoothing to eliminate extra unnecessary turns and produce a more realistic path that will be followed by the robot. When the path comprised of P nodes, the smoothing process takes:

- $\text{Time} \approx P \times \sqrt{HW}$

The slowest part of the process is the landmark setup though it produces the future searches faster. Bidirectional search is time-saving as opposed to searching the entire grid of HW steps.

3.3 Warehouse environment

The given warehouse is designed in order to make sure that the space of storage and the efficiency of the operation are well balanced. The facility has an average width of 30 meters and a length of 50 meters, which offers a functional floor space of 1500 square meters.

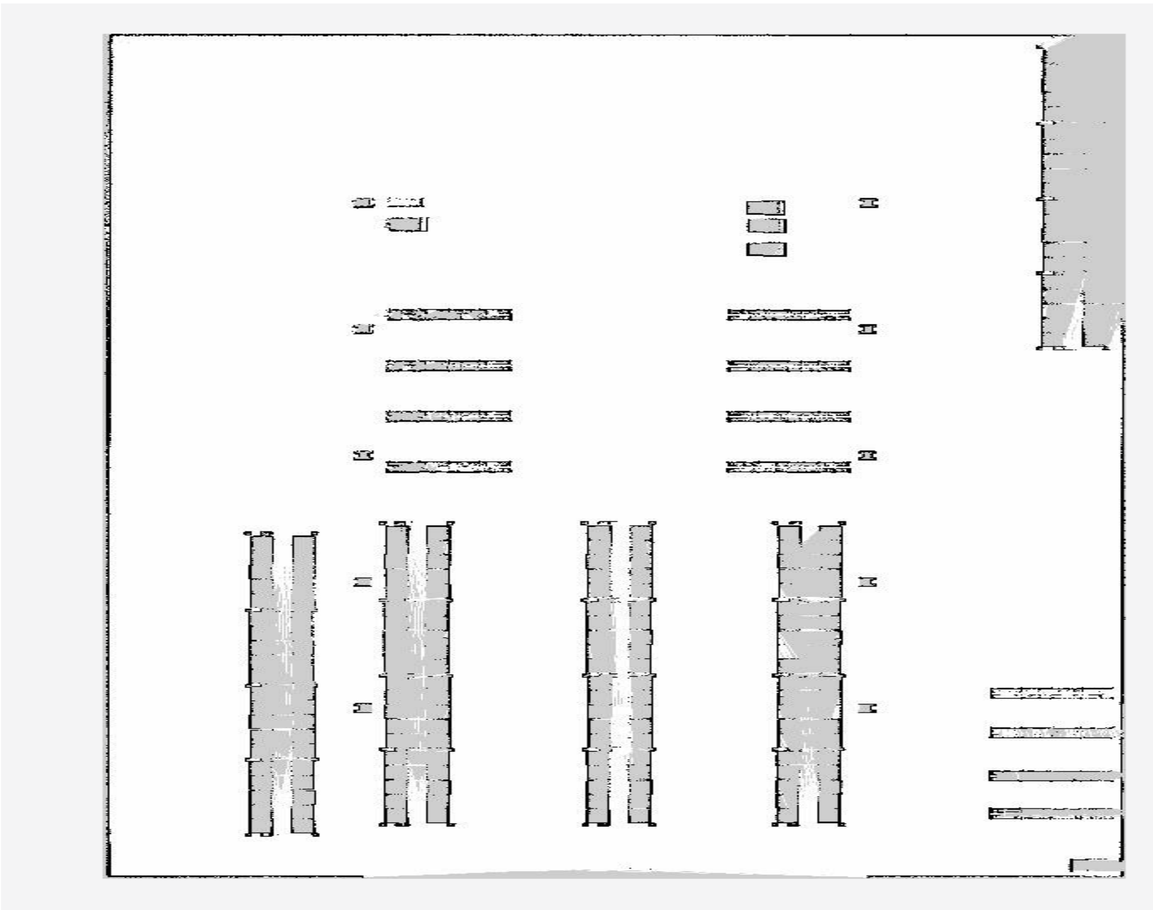

Figure 9. Warehouse Layout with Large and Small Shelves

This space plan can suit various types of storage installations. They are large pallet racks, smaller shelving units, and particular storage areas of objects. The arrangement of these components is in some order, so that handling of the material can be systematic and at the same time easily accessible to the manual operators as well as machine equipment like forklifts.

There are three main configurations of the storage infrastructure. The large shelves have an approximate width of 2.14 meters and length of 17.74 meters, and in one unit has a total of 38.02 square meters area, hence fit one pallet goods of bulk items of high capacity where high loads are required. The small shelves have dimensions such that 3.90 meters in width and 0.68 meters in length with an area of 2.66 square meters. These units are planned to be fitted into the warehouse environment so that they can store medium-sized or regularly needed products and thus efficiently robots can pick the orders. Moreover, the object storage areas have a width of 1.27 meters, a length of 0.88 meters, and a total area of 1.12 square meters and are used as the point of staging small packages, equipment, or temporary goods handling.

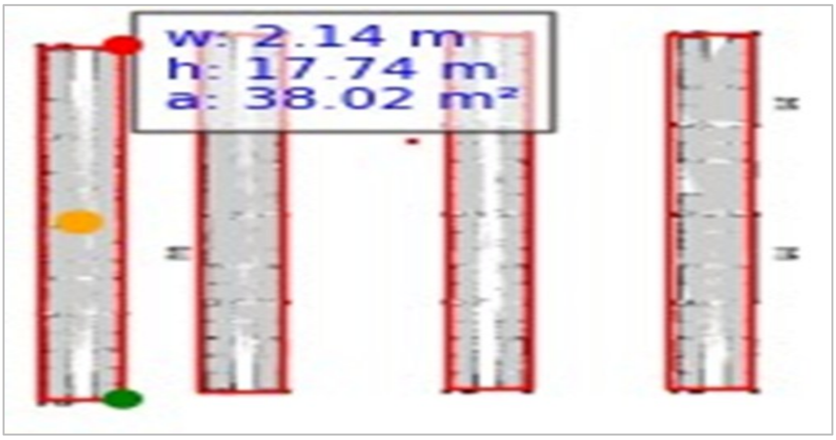

Figure 10. Large Shelf Dimension

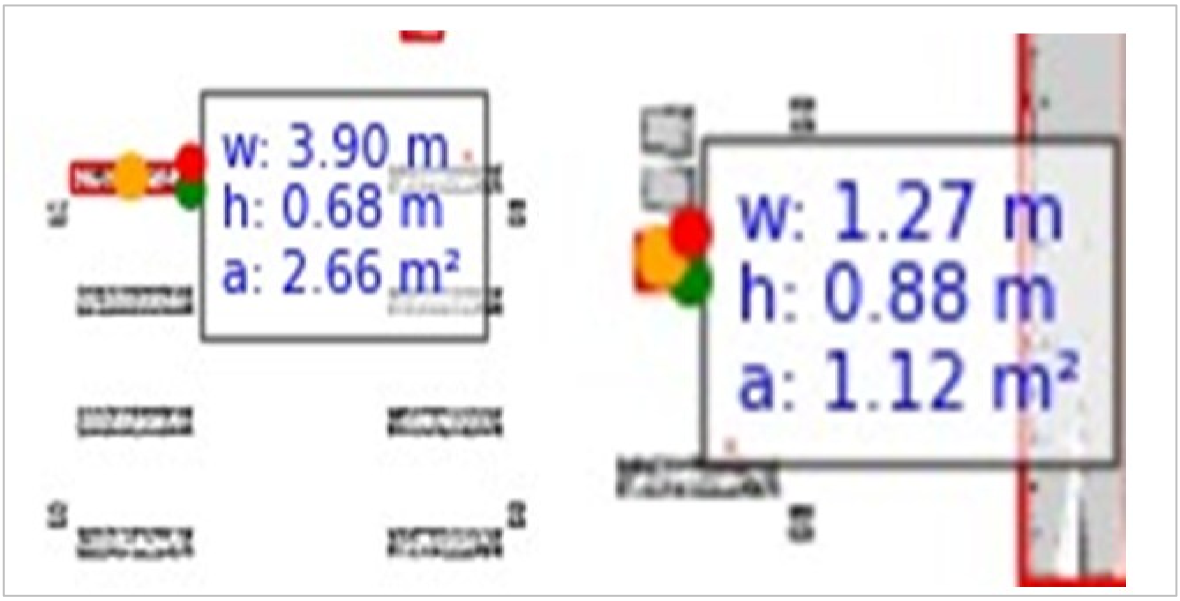

Figure 11. Small shelf (Left) and Object (Right) dimensions.

The warehouse space is structured as the systematizing of shelves into parallel rows with a large distance in between them so that the robots could move freely. This layout improves the efficiency of work processes because the movement of robots and vehicles becomes easier and operational capacity is minimized during peak working days. Vertical storage is fully used with the integration of high pallet racks, and the smaller racks and areas of objects lead to flexibility in operations and fast availability. At all, these design features create a warehouse environment that is conducive to high-density storage, faster picking and material handling that results in an improved logistical performance and a general productivity of operations.

4 Results and Discussion

In this study, the performance of Multi-Landmark Bidirectional A* algorithm for the fastest path planning is compared to that of other algorithms such as A* algorithm, Dijkstra algorithm and Bidirectional A* algorithm. The pathfinding times among various algorithms are also evaluated in this research experiment. Pathfinding time is the metric that define the performance of the path planning between the MLB A* algorithm and other traditional algorithms.

4.1 Multi Landmark Bidirectional A* (MLB A*) Algorithm

The Multi-Landmark Bidirectional A* (MLB A*) algorithm was able to produce the shortest path in faster time within the warehouse setting between the starting point (blue) and the goal point (green), avoiding the obstacles and storage shelves (Figure 12). The distance between robot and shelves is marked by the yellow line near the shelves so as to avoid collision between the robot and shelves. The resulting path, marked in red, is fairly straightforward with a few deviations indicating efficiency of the algorithm in going through structured spaces. The algorithm had a pathfinding time of 1.187 seconds in terms of performance. This finding features the power of the MLB A* method in the sense that the multiple landmarks method improves accuracy of heuristics whereas the bidirectional search method lessens the pathfinding time which results into a faster and more economical solution.

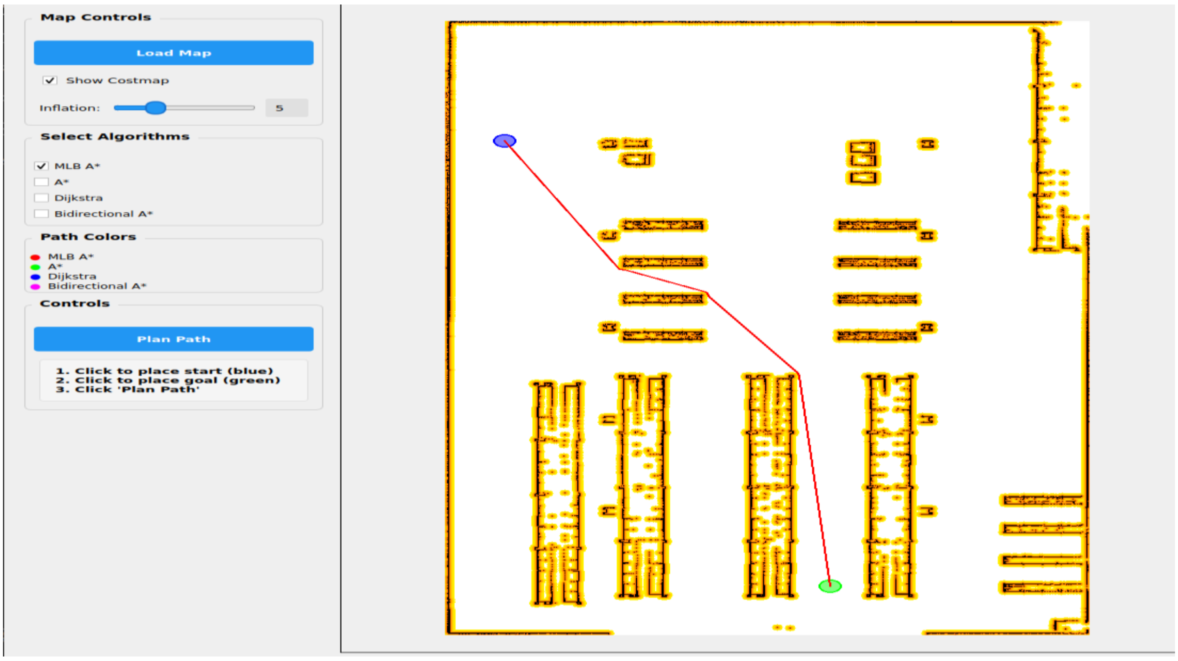

Figure 12. Path Planning through MLB A* Algorithm.

4.2 MLB A* and A* Algorithms

One of the path planning algorithms that have seen the greatest acceptance in robotics has been the A* algorithm, because of its simplicity, configurability and capability to ensure the smallest path based on an admissible heuristic. Although it is certain to compute a traveling path that is feasible, with large grid maps A * is computationally expensive and means that the computational pathfinding time is high as well. Nevertheless, A* has this disadvantage, but it is still effective to generate collision-free paths, and flexibility encourages the use of other criteria of costs, including energy or safety. (Duchon et al., 2014b)

Figure 13 indicated that using the A* algorithm and the Multi-Landmark Bidirectional A* (MLB A*) algorithm, a path to the goal (in green), developed by the pre-defined start point (in blue) within the warehouse setting, was planned out. The created path demonstrates that A* (green path) algorithm manages to avoid obstacles and shelves but creates a more prolonged non-direct path with several turns. Conversely, MLB A* (red line) algorithm finds a shorter path within less path-finding time which has less divergence. This is a demonstration of the superiority of MLB A* because when various landmarks are incorporated, heuristic precision gets enhanced and bi- directional search method expedites the convergence. As such, MLB A* provides a more effective solution in terms of path-finding time, thus its suitable to achieve a better solution in structured warehouse space than the traditional A* variant.

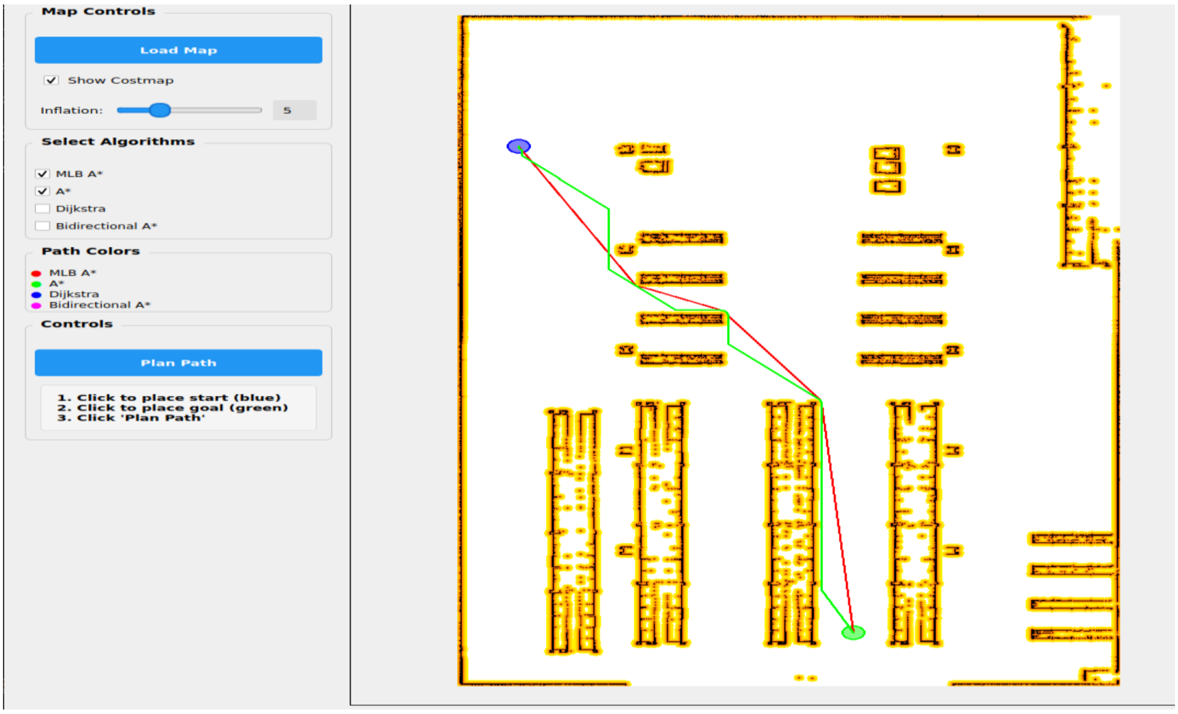

Figure 13. Path Planning through MLB A* and A* Algorithms

4.3 MLB A* and Dijkstra Algorithms

Both Dijkstra algorithm and MLB A* algorithm were used in the warehouse setting to generate a path between the specific start and goal points. Dijkstra (blue line), which is depicted in the figure, has been able to overcome obstacles but it creates a path that is longer and less competent since it has had several turns and diversions. This makes Dijkstra algorithm computationally expensive since it takes a lot of nodes to be searched without a heuristic guiding it, especially on large or complicated environments. This results in high path-finding time and low adaptability to the real-time use. (Julius Fusic et al., 2018) On the other side, the MLB A* algorithm (red line) resulted more straightforward and efficient line, which avoided many unnecessary deviations and yet managed to avoid obstacles. This result demonstrates the benefit of MLB A that takes the advantage of multiple landmarks to improve heuristic estimation and uses bidirectional search to reduce pathfinding time significantly. The MLB A* algorithm is thus more appropriate in the real-time movement within a structured warehouse since it is faster than Dijkstra algorithm both in pathfinding time and performances.

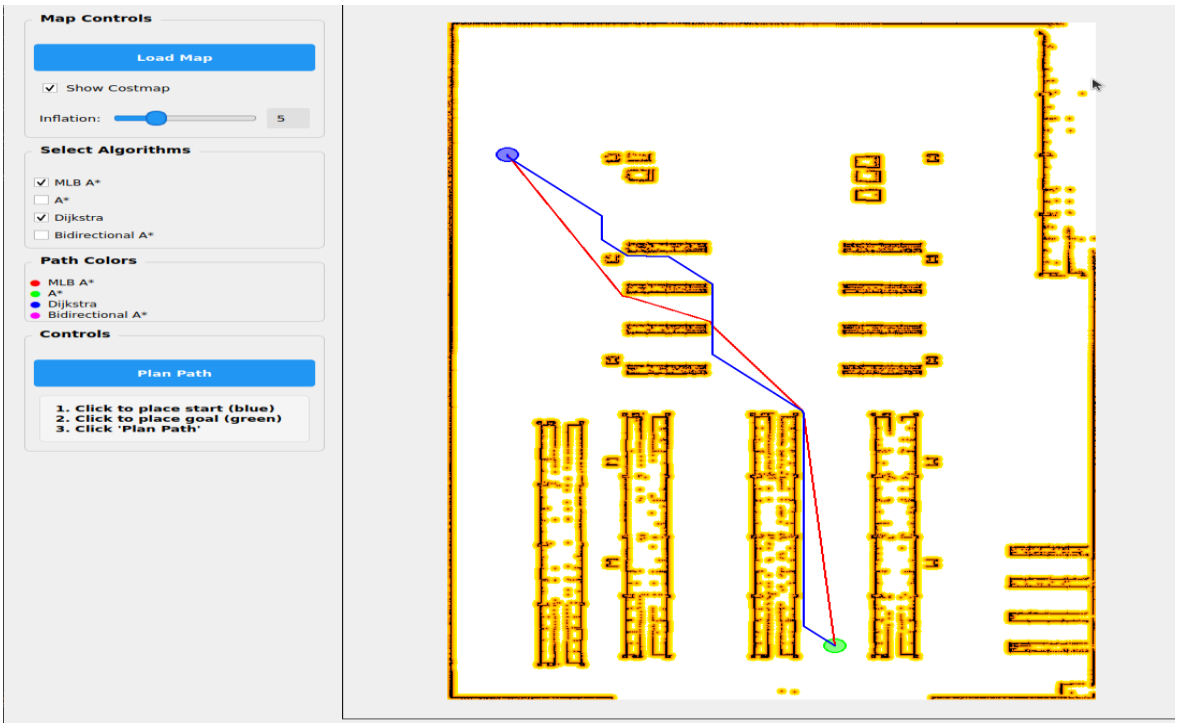

Figure 14. Path Planning through MLB A* and Dijkstra Algorithms

4.4 MLB A* and Bidirectional A* Algorithm

The Bidirectional A* algorithm is much more beneficial than the traditional A* approach as discussed in that search frontiers are broadened both in the initial node and the end node, which lessens the pathfinding time and enhances the computational capabilities. This Bidirectional method especially in conjunction with an increased search field (e.g. 16 directions) is effective in reducing redundant nodes, minimizing the length of the path, and reducing the number of inflection points, which reduces paths to smoother, more efficient versions. The algorithm is particularly beneficial in large scale or complicated environments in which unidirectional A* is computationally demanding. (Liu et al., 2023)

The performance of both Bidirectional A* algorithm and MLB A* algorithm was evaluated in the figure 15 for planning the path between the start and the goal point. As presented in the figure, the Bidirectional A* algorithm provides a feasible and collision-free path (pink path) but also advances numerous angular turns along the path around shelves and obstacles.

This is determined by the fact that it expands both at the start and end points simultaneously, decreasing the pathfinding time however there is the possibility that it will end up with longer and less straight path in structured spaces. The requirement to maintain two simultaneous searches also makes the implementation more complex and the memory management system more concerning. Also, unless extended features like smoothing capabilities are in place, the generated paths can still have redundant curves or sharp edges, and thus be less useful in practice to real-world mobile robots. Further, bidirectional search being fast reduces computations but in a very irregular or asymmetric environment where the two edges of the search do not meet effectively, it can decrease performance. (Liu et al., 2023)

Comparatively, the MLB A* (red line) generates a shorter path with less deviations, showing that it is more efficient in terms of pathfinding time and driving management. The developed performance of the MLB A* can be explained by the fact that several landmarks are used to estimate the heuristic which makes the search more efficient and the Bidirectional strategy which can drastically decrease the pathfinding time. All in all, MLB A* is more efficient than Bidirectional A* in terms of the quality of the paths obtained and thematic performance, which is why it is more likely to succeed in such tasks as navigation in complex warehouses in real-time.

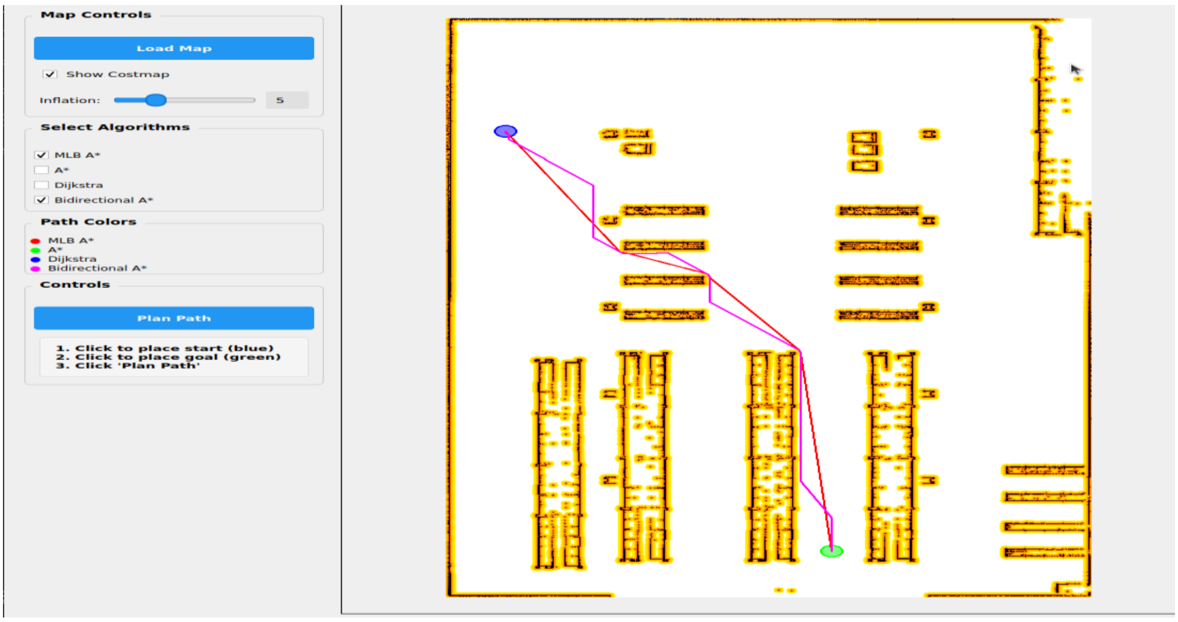

Figure 15. Path Planning through MLB A* and Bidirectional A* Algorithms.

4.5 Compare the Performance

4.5.1 Test 1

A test scenario is used in the warehouse environment to test the performance of four global path planning algorithms A*, Dijkstra, Bidirectional A*, and MLB A*. The performance metrics involved the pathfinding time which summarized the result in the table.

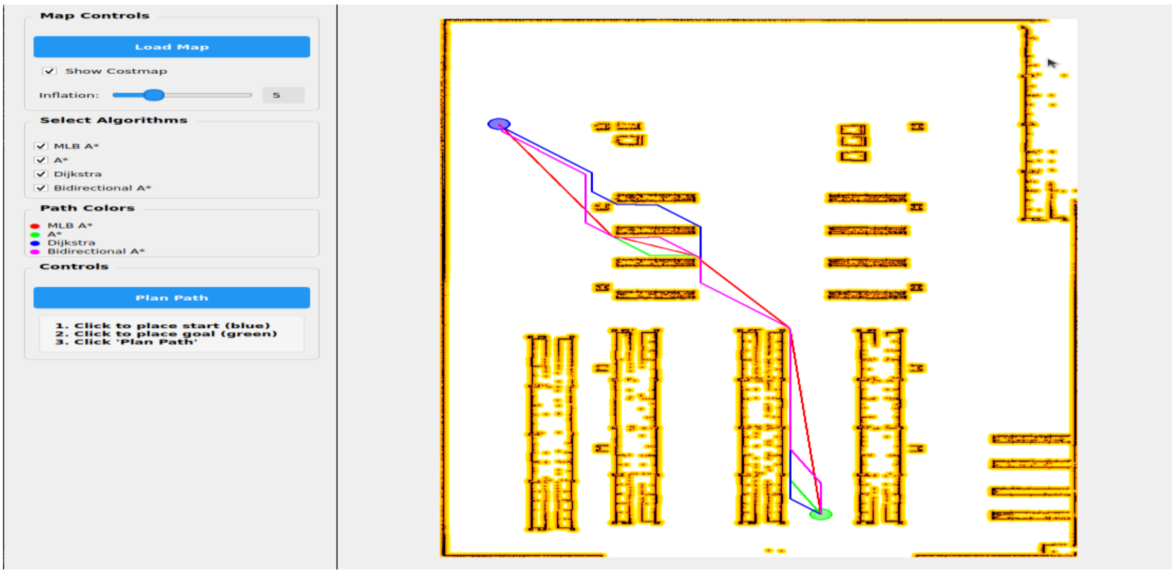

Figure 16. Compare the performance (Test 1)

The results of various algorithms are demonstrated on the table below.

Table 2. Algorithm Performance

	Algorithm	**Time (s)**
1	MLB A*	1.187
2	A*	2.217
3	Dijkstra	1.571
4	Bidirectional A*	1.637

A feasible path was produced by the A* algorithm, but it took the highest pathfinding time of 2.217 seconds, indicating it is more likely to search explore a wider search space than make completely heuristic improvements. The path-finding time of the path found by the Dijkstra algorithm was 1.571 seconds that is lower than the pathfinding time of A* algorithm, which is indicative of its exhaustive node exploration approach. The Bidirectional

A* algorithm was more efficient since it expanded in both the start and goal points at the same time, resulting in a shorter pathfinding time of 1.637 seconds compared to the standard A* algorithm and also little larger than the Dijkstra algorithm. The MLB A* algorithm performed the best though because it took 1.187 seconds to find the path. This path-finding time is the smallest time among the other algorithms. The outcome shows that multiple landmarks in heuristic estimation and a Bidirectional search expansion are beneficial due to its lower pathfinding time.

In general, the comparison makes a clear indication that the MLB A* is the best among the other traditional algorithms, resulting in the most faster pathfinding time. Although A* and Dijkstra algorithms both ensure the proper solution, they are not efficient enough in real-time situations. Bidirectional A* algorithm enhances the performance reasonably, but the combination of multi-landmark heuristic in MLB A* algorithm yields the faster path planning strategy concerning the structured warehouse surroundings.

4.5.2 Test 2

In figure 17, the Dijkstra algorithm created the path that had the highest path-finding time of 1.607 seconds which depicts the drawbacks of its exhaustive exploration search. Later, the A* algorithm builds that path with same starting and goal points with a less path-finding time of 1.574 seconds than the Dijkstra algorithm. The Bidirectional A* algorithm had a little bit better performance, producing a path with a shorter path-finding time of 1.085 seconds because of its bidirectional expansion strategy. The MLB A* algorithm was again used to obtain the best result by generating a path within the shortest path-finding time of 0.249 seconds.

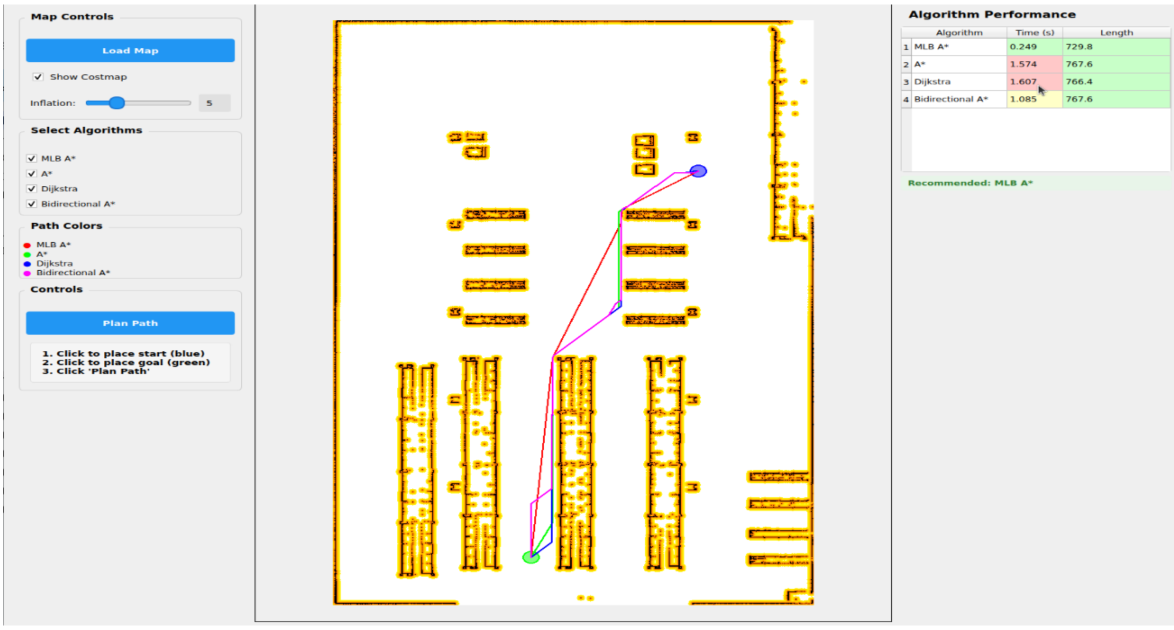

Figure 17. Compare the performance (Test 2)

As in this comparison, it is clarified that the MLB A* algorithm is performing better than the other conventional algorithms as it is planning the past faster. Whereas A* and Dijkstra algorithms determine valid paths, they have greater path-finding time making them less satisfactory in real-time navigation. Bidirectional A* is more efficient, though it still cannot be as efficient as the multi- landmark heuristic is in MLB A*. Thereby, MLB A* is more robust and efficient, making it the best choice to use in real time for the navigation of robots in the organized warehouse.

4.5.3 Test 3

In the test position 3, the A* algorithm was able to create a path with path-finding time of 1.097 seconds, whereas the Dijkstra algorithm created that path with smaller path-finding time of 1.317 seconds which is expected given the fact that the algorithm is exhaustive in search. The Bidirectional A* algorithm was more efficient as it achieves a less path-finding time of 1.061 seconds taking advantage of bidirectional expansion from both the start and end points. The MLB A* algorithm also performed better this time and created the path within the smallest path-finding time of 0.143 seconds which is significantly faster compared to all other traditional algorithms.

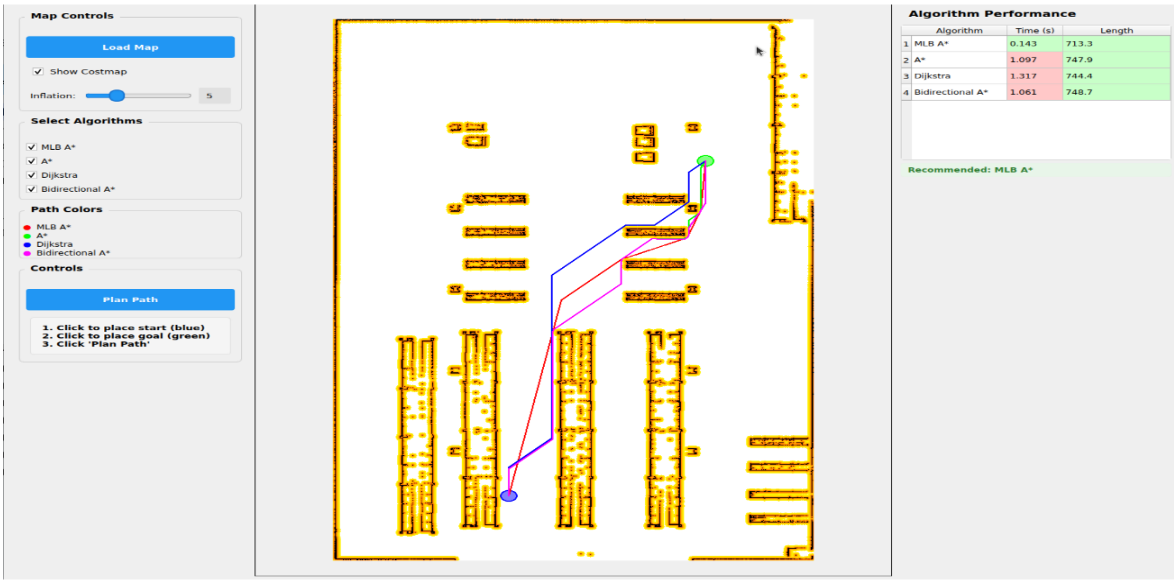

Figure 18. Compare the performance (Test 3)

4.6 Compare the Performance in Other Environments

4.6.1 Environment 2

In figure 19, all four of the above algorithms were able to produce accurate paths between the start and destination point and avoid obstacles in the above warehouse environment. The performance measures demonstrate the fair contrast of efficiency and path planning. The path-finding time of A* algorithm was 0.217 seconds, although the path-finding time of Dijkstra algorithm was 0.107 seconds that was slightly shorter than A* algorithm because it used exhaustive node expansion. The Bidirectional A* algorithm generated the same path with path-finding time of 0.211 seconds which is less than A* algorithm and more than Dijkstra algorithm, proving that it is efficient in larger surroundings and rather less preference in smaller surroundings. The Multi-Landmark Bidirectional A* algorithm had the most desirable performance with the shortest path-finding time of 0.070 seconds and surpassed all other algorithms.

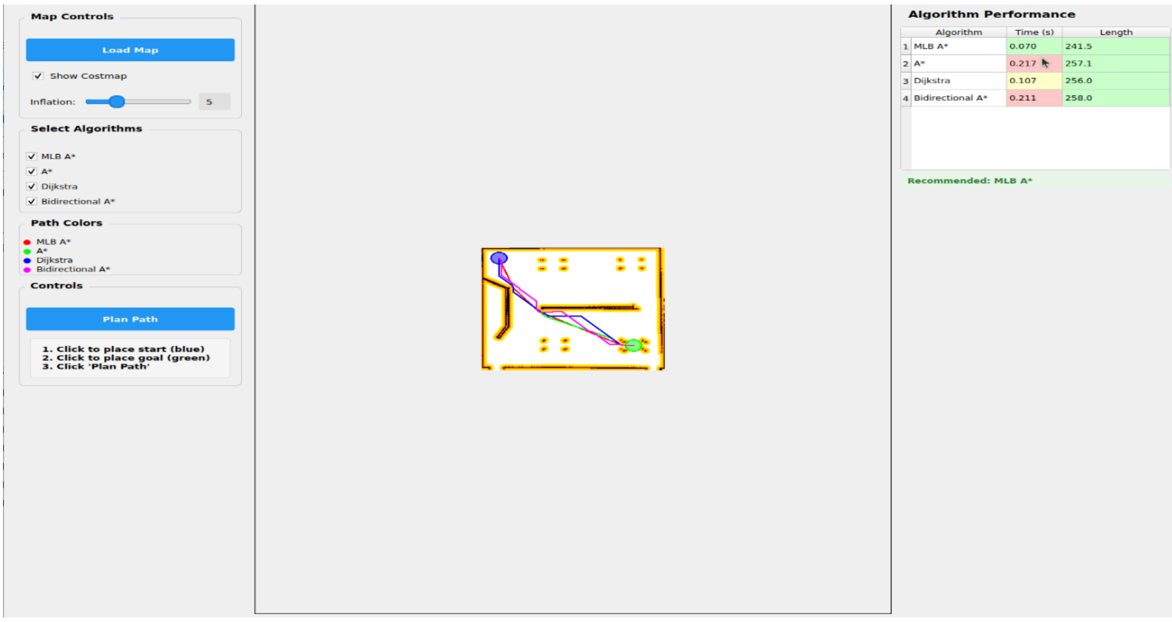

Figure 19. Compare the performance (Environment 2)

4.6.2 Environment 3

In Figure 20, all the four algorithms managed to find the proper paths connecting the start point and the goal point in the above environment, avoiding the narrow passages and multiple turns. The path-finding time of A* algorithm was 0.431 seconds, although the Dijkstra algorithm build that path with the shorter path-finding time of 0.185 seconds than A* algorithm, which is reflective of exhaustive search algorithm. Results from the Bidirectional A* algorithm produced that path with the path-finding time of 0.470 seconds was greater, proving the fact that bidirectional search minimizes search depth, but it is not efficient in the environment that contains very narrow corridors. MLB A* algorithm resulted the shortest path-finding time of 0.165 seconds, which were typically more efficient and optimized the paths.

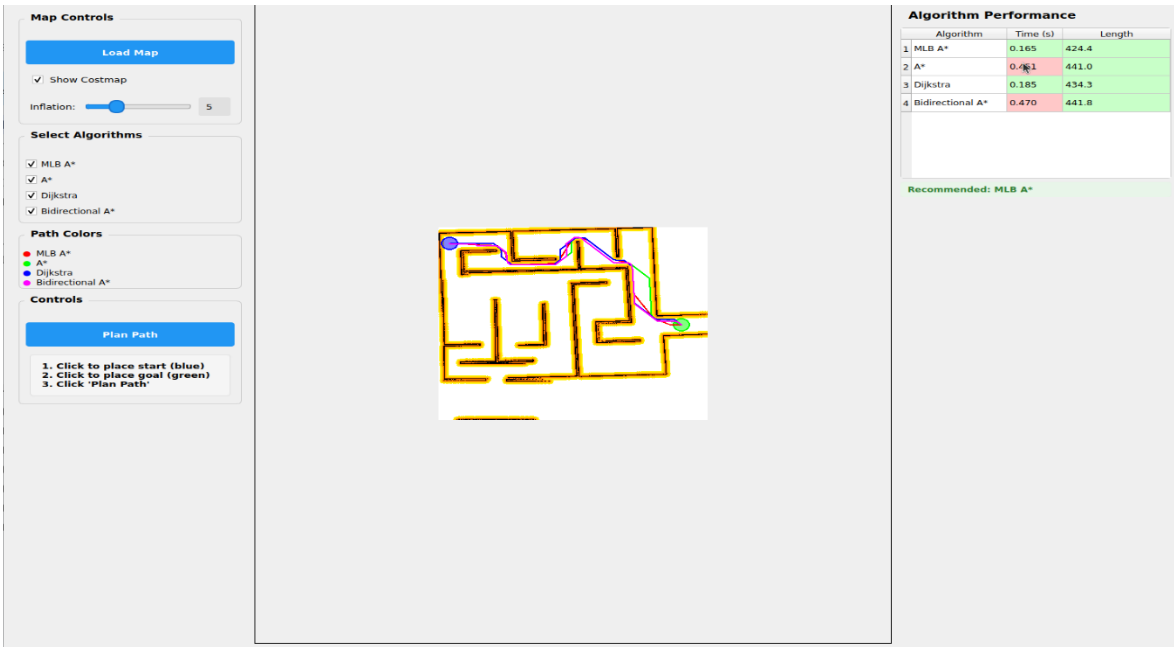

Figure 20. Compare the performance (Environment 2)

4.7 Discussion

The performance of the MLB A* algorithm in the warehouse setting is very clear as evidenced in the experimental results of the warehouse environment that the new algorithm performs quite well when compared to the traditional path planning algorithms like the A, Dijkstra, and Bidirectional A. In every case, MLB A* has always generated shorter and smoother paths with lower computation time. Unlike A* and Dijkstra, which could frequently explore more search space and yield longer paths, MLB A* could find the most direct path, with the aid of a series of landmarks to enhance the quality of heuristic information and using a bidirectional search approach to minimize the number of explored nodes. This not only reduced the path-finding time but also effective in complex structured maps like warehouses where the algorithm needed to make decisions fast to move the mobile robots.

The strengths of MLB A* are summarized in such a way that they reliably determine the shortest and most workable route by use of correct heuristic estimates that are respectively based on landmarks. Thereafter, its bidirectionality decreases the depth of search and makes it quicker to converge as opposed to unidirectional methods. Lastly, the algorithm is able to adjust changes in the terrain, prevent needless route deviations and risky areas, and post-processing route softening features are used to generate routes favorable to be used by robots.

Such features render MLB A* very applicable to practical uses, especially in warehouses and logistics when path efficiency and computing speed are of equal significance too.

However, the algorithm has certain constraints as well. Landmark distance calculations and storage calculated in the preprocessing step take extra time and memory (particularly in large scale grid maps). Furthermore, the techniques used to smooth the path after generation can sometimes retain unnecessary turns rather than entirely simplifying the path. The scalability of MLB A* is still remained under a challenging situation in very large grids due to the fact that the size of memory needed to store heuristic information of a number of landmarks increases vastly.

In general, the overall outcome suggests that A*, Dijkstra and Bidirectional A* can still be applied to particular situations; however, the MLB A* algorithm is more appropriate in terms of the balance between faster path planning and computational efficiency and thus it depicts itself as a potential option in the autonomous navigation conditions in a structured environment, such as warehouses.

5 Conclusion

This study develops Multi-Landmark Bidirectional A* (MLB A*) algorithm that is an effective and smart method to find paths in an intricate environment. Multi-landmark heuristics and bidirectional search are incorporated together to achieve the minimized path-finding time and smooth paths which are essential to deliver produce faster path planning involving robotics applications, among other uses. It is a useful tool in research and development of navigation as its steps are clear. MLB A* algorithm is also the most appropriate when it comes to real-time navigation in areas characterized by high density of obstacles and limited passageways. This algorithm is applicable in large warehouse industry to discover the faster path and enhance the performance of industrial works. In future, MLB A* algorithm will be improved to adapt the dynamic environment and multiple robots in the warehouse. MLB A* algorithm will be integrated with warehouse environment through ROS2.

References

Akka, K. & Khaber, F. 2018. Mobile robot path planning using an improved ant colony optimization. International Journal of Advanced Robotic Systems,15 (3).

Al-Mutib, K., Alsulaiman, M., Emaduddin, M., Ramdane, H. & Mattar, E. 2011. D* lite based real-time multi-agent path planning in dynamic environments. 3rd International Conference on Computational Intelligence, Modelling and Simulation. Langkawi, Malaysia. 20-22.9.2014. IEEE. P. 170–174.

Alshammrei, S., Boubaker, S. & Kolsi, L. 2022. Improved Dijkstra Algorithm for Mobile Robot Path Planning and Obstacle Avoidance. Computers, Materials and Continua, 72 (3). Pp. 5939–5954.

Baek, M., Mun, J., Kim, W., Choi, D., Yim, J. & Lee, S. 2021. Driving environment perception based on the fusion of vehicular wireless communications and automotive remote sensors. Sensors, 21 (5). Pp. 1–29.

Blum, C. 2005. Ant colony optimization: Introduction and recent trends. Physics of Life Reviews, 2 (4). Pp. 353–373.

Chen, J., Li, M., Su, Y., Li, W. & Lin, Y. 2023. Direction constraints adaptive extended bidirectional A* algorithm based on random two-dimensional map environments. Robotics and Autonomous Systems, 165.

Daniel, K., Nash, A., Koenig, S. and Felner, A., 2010. Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research, 39. Pp. 533–579.

Duchoň, F., Babinec, A., Kajan, M., Beňo, P., Florek, M., Fico, T. and Jurišica, L., 2014. Path planning with modified A* algorithm for a mobile robot. Procedia Engineering, 96. Pp. 59–69.

Fethi, D., Nemra, A., Louadj, K. & Hamerlain, M. 2018. Simultaneous localization, mapping, and path planning for unmanned vehicle using optimal control. Advances in Mechanical Engineering, 10 (1).

Fu, B., Chen, L., Zhou, Y., Zheng, D., Wei, Z., Dai, J. & Pan, H. 2018 An improved A* algorithm for the industrial robot path planning with high success rate and short length. Robotics and Autonomous Systems, 106. Pp. 26–37.

Fu, B., Chen, Y., Quan, Y., Zhou, X. & Li, C. 2025. Bidirectional artificial potential field-based ant colony optimization for robot path planning. Robotics and Autonomous Systems, 183.

Gang, L. and Wang, J., 2015. PRM path planning optimization algorithm research. WSEAS Transactions on Systems and Control, 10. Pp. 148–153.

Garrido, F. and Resende, P., 2022. Review of decision-making and planning approaches in automated driving. IEEE Access, 10. Pp. 100348–100366.

Hartomo, K., Ismanto, B., Nugraha, A., Yulianto, S. and Laksono, B., 2019. Searching the shortest route to distribute disaster's logistical assistance using Dijkstra method. Journal of Physics: Conference Series, 1402(7). Pp. 077014.

He, P., Xu, Z., Long, X., Hou, K., and Xiang, Y. 2023. Path Planning of Mobile Robot Based on Improved A-Star Bidirectional Search Algorithm. 2023 IEEE International Conference on Unmanned Systems (ICUS 2023). Hefei, China. 13–15 October 2023. IEEE. P. 1517-1522.

Hoang Linh, T., Khang, N.H.D., Dinh Chuong, H. & Duc Tam, H. 2025. Combining deep learning algorithm and a-star algorithm to increase the accuracy of tracking lost gamma source. Applied Radiation and Isotopes, 225.

Hu, Y., and Yang, S.X. 2004. A Knowledge Based Genetic Algorithm for Path Planning of a Mobile Robot. IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. New Orleans, LA, USA. 26 April–01 May 2004. IEEE. P. 4350-4355.

Fusic, S.J., Ramkumar, P., and Hariharan, K. 2018. Path Planning of Robot Using Modified Dijkstra Algorithm. 2018 National Power Engineering Conference (NPEC). Madurai, India. 09–10 March 2018. IEEE. P. 1-5.

Katona, K., Neamah, H.A. & Korondi, P. 2024. Obstacle Avoidance and Path Planning Methods for Autonomous Navigation of Mobile Robot. Sensors, 24 (11).

Kumar, M.D.Y. & Rajchandar, K. 2025. Bi-directional virtual search algorithm for efficient and collision-free path planning in autonomous robots navigating static and dynamic environments. Ain Shams Engineering Journal, 16 (9).

Lai, X., Li, J.H. & Chambers, J. 2021. Enhanced Center Constraint Weighted A* Algorithm for Path Planning of Petrochemical Inspection Robot. Journal of Intelligent and Robotic Systems: Theory and Applications, 102 (4).

LaValle, S.M. 2006. Planning Algorithms. Cambridge: Cambridge University Press. 844 p.

Li, P., Li, Y. & Dai, X. 2024. VNS-BA*: An Improved Bidirectional A* Path Planning Algorithm Based on Variable Neighborhood Search. Sensors, 24 (21).

Li, P., Wei, L. and Wu, D., 2025. An Intelligently Enhanced Ant Colony Optimization Algorithm for Global Path Planning of Mobile Robots in Engineering Applications. Sensors, 25(5).

Haoxin, L. and Yonghui, Z., 2022. ASL-DWA: An Improved A-star Algorithm for Indoor Cleaning Robots. IEEE Access, 10. Pp. 99498–99515.

Liu, J., Liu, Q., Zhang, Y., Zhang, Y., and Wu, J. 2023. A Path Planning Method for Mobile Robots Based on Improved Bidirectional A* Algorithm. 2023 3rd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI). Wuhan, China. 15–17 December 2023. IEEE. P. 821-824.

Liu, J., Yang, J., Liu, H., Tian, X. & Gao, M. 2017. An improved ant colony algorithm for robot path planning. Soft Computing. 21 (19). Pp. 5829–5839.

Liu, L., Wang, B. & Xu, H. 2022. Research on Path-Planning Algorithm Integrating Optimization A-Star Algorithm and Artificial Potential Field Method. Electronics, 11 (22).

Luo, M., Hou, X. and Yang, J., 2020. Surface optimal path planning using an extended Dijkstra algorithm. IEEE Access, 8. Pp. 147827–147838.

Sánchez-Ibáñez, J.R., Pérez-Del-pulgar, C.J. & García-Cerezo, A. 2021. Path planning for autonomous mobile robots: A review. Sensors. 21 (23).

Shanmugaraja, M., Thangamuthu, M. & Ganesan, S. 2025. Hybrid Path Planning Algorithm for Autonomous Mobile Robots: A Comprehensive Review. Journal of Sensor and Actuator Networks, 14 (5).

Sharan, S., Ngoc Minh, A.T., Dominguez-Jimenez, J.J., and Nauth, P. 2025. Comparison of Standard Ant Colony Optimization, Dijkstra and A* Algorithms for Path Planning of Robot in an Indoor Environment. 9th International Conference on Mechanical Engineering and Robotics Research (ICMERR 2025). Barcelona, Spain. 15–17.01.2025. IEEE. P. 10-16.

Singal, P. and Chhillar, R.S., 2014. Dijkstra shortest path algorithm using global position system. International Journal of Computer Applications, 101(6). Pp. 12–18.

Song, B., Wang, Z. & Zou, L. 2021. An improved PSO algorithm for smooth path planning of mobile robots using continuous high-degree Bezier curve. Applied Soft Computing, 100.

Song, X., Gao, H., Ding, T., Gu, Y., Liu, J. & Tian, K. 2023. A Review of the Motion Planning and Control Methods for Automated Vehicles. Sensors, 23 (13).

Stentz, A. 1994. Optimal and Efficient Path Planning for Partially-Known Environments. Proceedings of the 1994 IEEE International Conference on Robotics and Automation. San Diego, CA, USA. 08–13 May 1994. IEEE. P. 3310-3317.

Sun, X., Shen, Y., and Zhang, H. 2024. A Stable Satellite Routing Algorithm Based on A-Star Algorithm in Dynamic Networks. The 3rd International Conference on Electronic Information and Communication Engineering. Guangzhou, China. 12–14.01.2024. IOP Publishing. Journal of Physics: Conference Series, Volume 2807.

Sun, Y., Fang, M., and Su, Y. 2020. AGV Path Planning Based on Improved Dijkstra Algorithm. 3rd International Conference on Modeling, Simulation and Optimization Technologies and Applications (MSOTA 2020). Beijing, China. 22–23.11.2020. IOP Publishing. Journal of Physics: Conference Series, Volume 1746.

Tang, Y., Zakaria, M.A. & Younas, M. 2025. Path Planning Trends for Autonomous Mobile Robot Navigation: A Review. Sensors, 25 (4).

Wang, N. and Xu, H., 2020. Dynamics-constrained global-local hybrid path planning of an autonomous surface vehicle. IEEE Transactions on Vehicular Technology, 69(7). Pp. 6928–6942.

Wang, X., Wei, J., Zhou, X., Xia, Z. & Gu, X. 2022. AEB-RRT*: an adaptive extension bidirectional RRT* algorithm. Autonomous Robots, 46 (6). Pp. 685–704.

Wang, X., Zhang, H., Liu, S., Wang, J., Wang, Y. & Shangguan, D. 2022. Path planning of scenic spots based on improved A* algorithm. Scientific Reports, 12 (1).

Wu, S., Li, Q. and Wei, W., 2023. Application of ant colony optimization algorithm based on triangle inequality principle and partition method strategy in robot path planning. Axioms, 12(6).

Xu, Y., Li, Q., Xu, X., Yang, J. & Chen, Y. 2023. Research Progress of Nature-Inspired Metaheuristic Algorithms in Mobile Robot Path Planning. Electronics, 12 (15).

Yang, Z. 2024. Research on Path Planning Algorithm of Mobile Robot Based on Dijkstra. 2nd International Conference on Mechatronic Automation and Electrical Engineering (ICMAEE 2024). Hybrid Conference, Nanjing, China. 22–24 November 2024. IET. P. 222-227.

Yu, J., Su, Y. & Liao, Y. 2020. The Path Planning of Mobile Robot by Neural Networks and Hierarchical Reinforcement Learning. Frontiers in Neurorobotics, 14.

Yu, M., Luo, Q., Wang, H. & Lai, Y. 2023. Electric Logistics Vehicle Path Planning Based on the Fusion of the Improved A-Star Algorithm and Dynamic Window Approach. World Electric Vehicle Journal, 14 (8).

Zhang, H.Y., Lin, W.M. & Chen, A.X. 2018. Path planning for the mobile robot: A review. Symmetry, 10 (10).

Zhang, L., Zhang, Y., Zeng, M. & Li, Y. 2021. Robot navigation based on improved A* algorithm in dynamic environment. Assembly Automation, 41 (4). Pp. 419–430.

Zhang, X. (2024). Enhancing a star algorithm for robot path planning. Applied and Computational Engineering, 34 (1). Pp. 1–6.

Zhou, L., Jin, S., Wang, J., Zhang, H., Shi, M. & Zhou, H.P. (2024). 3D positioning of Camellia oleifera fruit-grabbing points for robotic harvesting. Biosystems Engineering, 246. Pp. 110–121.

Appendices

Appendix 1. Multi-Landmark Bidirectional A* Algorithm

Step 1: Graph Construction and Pre-processing

Input: costmap C, dimensions (height, width)

Output: Graph G(V, E), landmark distances D_L

```
V ← ∅, E ← ∅
for each cell (x, y) in C do
    if C[y, x] < 253 then
        V ← V ∪ {(x, y)}
    end if
end for
movements ← {(0,1), (1,0), (0,-1), (-1,0), (1,1), (-1,1), (1,-1), (-1,1)}
for each node v(x, y) ∈ V do
    for each (dx, dy) ∈ movements do
        u ← (x + dx, y + dy)
        if u ∈ V and C[u.y, u.x] < 254 then
            d ← √(dx² + dy²)
            cost_gradient ← |C[u.y, u.x] - C[v.y, v.x]|
            avg_cost ← (C[v.y, v.x] + C[u.y, u.x]) / 2
            w ← d × (1.0 + avg_cost/25.0 + 5 × cost_gradient)
            E ← E ∪ {(v, u, w)}
        end if
    end for
end for
```

```
 L ← {corners, center} ∩ V
 for each landmark ℓ ∈ L do
    D_L[ℓ] ← Dijkstra(G, ℓ)
 end for
 return G(V, E), D_L
```

Step 2: Multi-Landmark Heuristic

Input: node v, target t, landmarks L, distances D_L, costmap C

Output: heuristic value h

```
  h_landmark ← 0
  for each landmark ℓ ∈ L do
      if v ∈ D_L[ℓ] and t ∈ D_L[ℓ] then
          estimate ← |D_L[ℓ][v] - D_L[ℓ][t]|
         h_landmark ← max(h_landmark, estimate)
      end if
  end for

  h_euclidean ← √((t.x - v.x)² + (t.y - v.y)²)

 cost_factor ← 1.0 + 5.0 × (C[v.y, v.x] / 255.0)
 h ← max(h_landmark, h_euclidean) × cost_factor

 return h
```

Step 3: MLB A* Path Planning

Input: Graph G(V, E), start s, goal g, landmarks L, distances D_L

Output: path P from s to g

```
 Q_f ← priority queue with (0, s)
 Q_b ← priority queue with (0, g)
 g_f[s] ← 0, g_b[g] ← 0
 π_f[s] ← null, π_b[g] ← null
 μ ← ∞, v_m ← null

 while Q_f ≠ ∅ and Q_b ≠ ∅ and μ = ∞ do
   // Forward search expansion
   (f, v) ← Q_f.pop()
   if v ∈ g_b then
     cost ← g_f[v] + g_b[v]
     if cost < μ then
       μ ← cost, v_m ← v
     end if
   end if

   for each neighbor u of v with weight w do
     new_cost ← g_f[v] + w
     if u ∉ g_f or new_cost < g_f[u] then
       g_f[u] ← new_cost
       π_f[u] ← v
```

```
            f_value ← new_cost + MultiLandmarkHeuristic(u, g, L, D_L)
            Q_f.push((f_value, u))
        end if
    end for

    // Backward search expansion
    (f, v) ← Q_b.pop()
    if v ∈ g_f then
        cost ← g_f[v] + g_b[v]
        if cost < μ then
            μ ← cost, v_m ← v
        end if
    end if

    for each neighbor u of v with weight w do
        new_cost ← g_b[v] + w
        if u ∉ g_b or new_cost < g_b[u] then
            g_b[u] ← new_cost
            π_b[u] ← v
            f_value ← new_cost + MultiLandmarkHeuristic(u, s, L, D_L)
            Q_b.push((f_value, u))
        end if
    end for
end while
```

```
if v_m = null then
    return ∅
end if

// Path reconstruction
P ← []
v ← v_m
while v ≠ null do
    P.prepend(v)
    v ← π_f[v]
end while

v ← π_b[v_m]
while v ≠ null do
    P.append(v)
    v ← π_b[v]
end while

return PathSmoothing(P)
```

Step 4: Path Smoothing

Input: path P

Output: smoothed path P'

```
P' ← [P[0]]
```

```
  for i ← 1 to |P| - 2 do
     p1 ← P'[last]
     p2 ← P[i]
     p3 ← P[i + 1]
     if not LineOfSight(p1, p3) then
        P'.append(p2)
     end if
  end for
 P'.append(P[|P| - 1])
 return P'
```

Step 5: Line-of-Sight Check

Input: points p1, p3, costmap C, threshold θ

Output: boolean

```
  (x, y) ← (p1.x, p1.y)
  dx ← |p3.x - p1.x|, dy ← |p3.y - p1.y|
  sx ← sign(p3.x - p1.x), sy ← sign(p3.y - p1.y)
  err ← dx - dy

  while true do
     if C[y, x] ≥ 254 or C[y, x] > θ then
        return false
     end if
```

```
    if x = p3.x and y = p3.y then
        break
    end if

    e2 ← 2 × err
    if e2 > -dy then
        err ← err - dy
        x ← x + sx
    end if
    if e2 < dx then
        err ← err + dx
        y ← y + sy
    end if
end while
return true
```

Appendix 2 All codes

MLB_planner

```
import heapq

import math

import numpy as np

from typing import Dict, List, Tuple, Optional

from dataclasses import dataclass

from PyQt5.QtCore import QPoint

from base_planner import BasePlanner, MapMetaData


class MLBPathPlanner(BasePlanner):

    def __init__(self, costmap: np.ndarray, metadata: MapMetaData):

        super().__init__(costmap, metadata)

        self.graph = {}

        self.coords = {}

        self.landmark_distances = {}

        self.node_lookup = {}

        self.next_node_id = 0


        # Build graph and precompute distances

        self._build_graph()

        self._compute_landmark_distances()


    def _build_graph(self):
```

```python
"""Convert costmap to graph with proper cost consideration"""
height, width = self.costmap.shape
node_id = 0

for y in range(height):
    for x in range(width):
        if self.costmap[y, x] < 253:
            self.node_lookup[(x, y)] = node_id
            self.coords[node_id] = (x, y)
            self.graph[node_id] = []
            node_id += 1

# Connect nodes with costs
movements = [(0,1), (1,0), (0,-1), (-1,0),
            (1,1), (-1,1), (1,-1), (-1,-1)]

for node_id, pos in self.coords.items():
    x, y = pos
    for dx, dy in movements:
        nx, ny = x + dx, y + dy
        if (nx, ny) in self.node_lookup:
            if self.costmap[ny, nx] < 254:
                neighbor_id = self.node_lookup[(nx, ny)]
                dist = math.sqrt(dx*dx + dy*dy)
```

```
                cost_current = self.costmap[y, x]
                cost_next = self.costmap[ny, nx]
                cost_gradient = abs(cost_next - cost_current) * 1.0
                avg_cost = (cost_current + cost_next)
                weight = dist * (1.0 + avg_cost/25.0 + 5 * cost_gradient)
                self.graph[node_id].append((neighbor_id, weight))

    def _compute_landmark_distances(self):
        """Precompute distances from all nodes to landmarks"""
        corners = [(0,0), (0,self.costmap.shape[1]-1),
                   (self.costmap.shape[0]-1,0),
                   (self.costmap.shape[0]-1,self.costmap.shape[1]-1)]

        landmarks = []
        for y, x in corners:
            if (x, y) in self.node_lookup:
                landmarks.append(self.node_lookup[(x, y)])

        center = (self.costmap.shape[0]//2, self.costmap.shape[1]//2)
        if center in self.node_lookup:
            landmarks.append(self.node_lookup[center])

        for landmark in landmarks:
            self.landmark_distances[landmark] = self._dijkstra(landmark)
```

```
def _dijkstra(self, start: int) -> dict:
    """Compute shortest path distances from start to all nodes"""
    distances = {start: 0}
    queue = [(0, start)]

    while queue:
        dist, current = heapq.heappop(queue)

        if current in distances and dist > distances[current]:
            continue

        for neighbor, weight in self.graph[current]:
            distance = dist + weight

            if neighbor not in distances or distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))

    return distances

def plan_path(self, start: QPoint, goal: QPoint) -> list:
    """Main MLB A* pathfinding method"""
    start_node = self.node_lookup.get((start.x(), start.y()))
```

```
goal_node = self.node_lookup.get((goal.x(), goal.y()))

if start_node is None or goal_node is None:
    return []

def get_heuristic(node: int, target: int) -> float:
    landmark_h = 0
    for landmark, distances in self.landmark_distances.items():
        if node in distances and target in distances:
            estimate = abs(distances[node] - distances[target])
            landmark_h = max(landmark_h, estimate)

    x1, y1 = self.coords[node]
    x2, y2 = self.coords[target]
    euclidean_h = math.sqrt((x2-x1)**2 + (y2-y1)**2)

    cost_factor = 1.0 + (self.costmap[y1, x1]/255.0) * 5.0
    return max(landmark_h, euclidean_h) * cost_factor

forward_queue = [(0, start_node)]
backward_queue = [(0, goal_node)]

forward_cost = {start_node: 0}
backward_cost = {goal_node: 0}
```

```
forward_came_from = {start_node: None}
backward_came_from = {goal_node: None}

best_cost = float('inf')
meeting_point = None

while forward_queue and backward_queue and best_cost == float('inf'):
    _, current = heapq.heappop(forward_queue)

    if current in backward_cost:
        x, y = self.coords[current]
        total_cost = forward_cost[current] + backward_cost[current] + self.costmap[y, x]/255.0
        if total_cost < best_cost:
            best_cost = total_cost
            meeting_point = current

    for neighbor, weight in self.graph[current]:
        new_cost = forward_cost[current] + weight

        if neighbor not in forward_cost or new_cost < forward_cost[neighbor]:
            forward_cost[neighbor] = new_cost
            forward_came_from[neighbor] = current
```

```
                priority = new_cost + get_heuristic(neighbor, goal_node)
                heapq.heappush(forward_queue, (priority, neighbor))

        if backward_queue:
            _, current = heapq.heappop(backward_queue)

            if current in forward_cost:
                x, y = self.coords[current]
                total_cost = forward_cost[current] + backward_cost[current] + self.costmap[y,
x]/255.0
                if total_cost < best_cost:
                    best_cost = total_cost
                    meeting_point = current

            for neighbor, weight in self.graph[current]:
                new_cost = backward_cost[current] + weight

                if neighbor not in backward_cost or new_cost < backward_cost[neighbor]:
                    backward_cost[neighbor] = new_cost
                    backward_came_from[neighbor] = current
                    priority = new_cost + get_heuristic(neighbor, start_node)
                    heapq.heappush(backward_queue, (priority, neighbor))

    if meeting_point is None:
```

```
    return []

path = []
current = meeting_point
while current is not None:
    x, y = self.coords[current]
    path.append(QPoint(x, y))
    current = forward_came_from[current]
path.reverse()

current = backward_came_from[meeting_point]
while current is not None:
    x, y = self.coords[current]
    path.append(QPoint(x, y))
    current = backward_came_from[current]

smoothed_path = [path[0]]
for i in range(1, len(path) - 1):
    p1 = smoothed_path[-1]
    p2 = path[i]
    p3 = path[i+1]

    if self._is_valid_connection(p1, p3):
        continue
```

```
            else:
                smoothed_path.append(p2)
        smoothed_path.append(path[-1])

        return smoothed_path

    def _is_valid_connection(self, p1: QPoint, p3: QPoint) -> bool:
        """Check if a direct connection between two points is valid and low-cost."""
        x1, y1 = p1.x(), p1.y()
        x3, y3 = p3.x(), p3.y()

        dx = abs(x3 - x1)
        dy = abs(y3 - y1)
        x, y = x1, y1
        sx = 1 if x1 < x3 else -1
        sy = 1 if y1 < y3 else -1
        err = dx - dy

        max_cost = 0
        COST_THRESHOLD = 0

        total_cost = 0
        points_checked = 0
```

```
while True:
    if not (0 <= x < self.costmap.shape[1] and 0 <= y < self.costmap.shape[0]):
        return False

    current_cost = self.costmap[y, x]
    if current_cost >= 254:
        return False

    max_cost = max(max_cost, current_cost)
    total_cost += current_cost
    points_checked += 1

    if max_cost > COST_THRESHOLD:
        return False
    if (total_cost / points_checked) > (COST_THRESHOLD / 2):
        return False

    if x == x3 and y == y3:
        break

    e2 = 2 * err
    if e2 > -dy:
        err -= dy
        x += sx
```

```
            if e2 < dx:
                err += dx
                y += sy

        return True
```

Dijkstra Planner

```
from base_planner import BasePlanner
import heapq
import math
from PyQt5.QtCore import QPoint

class DijkstraPlanner(BasePlanner):
    def __init__(self, costmap, metadata):
        super().__init__(costmap, metadata)
        self.graph = {}
        self.coords = {}
        self.node_lookup = {}
        self._build_graph()

    def _build_graph(self):
        """Convert costmap to graph representation with costs"""
        height, width = self.costmap.shape
```

```
node_id = 0

# Create nodes for traversable cells
for y in range(height):
    for x in range(width):
        if self.costmap[y, x] < 253:  # Non-obstacle cells
            self.node_lookup[(x, y)] = node_id
            self.coords[node_id] = (x, y)
            self.graph[node_id] = []
            node_id += 1

# Connect nodes with weighted edges
movements = [(0,1), (1,0), (0,-1), (-1,0),
            (1,1), (-1,1), (1,-1), (-1,-1)]

for node_id, pos in self.coords.items():
    x, y = pos
    current_cost = self.costmap[y, x]

    for dx, dy in movements:
        nx, ny = x + dx, y + dy
        if (nx, ny) in self.node_lookup:
            if self.costmap[ny, nx] < 254:
                neighbor_id = self.node_lookup[(nx, ny)]
```

```
                # Calculate distance
                dist = math.sqrt(dx*dx + dy*dy)

                # Include costmap values in weight calculation
                neighbor_cost = self.costmap[ny, nx]
                avg_cost = (current_cost + neighbor_cost) / 2



                weight = dist * (1 + avg_cost/100.0)


                self.graph[node_id].append((neighbor_id, weight))

    def plan_path(self, start: QPoint, goal: QPoint) -> list:
        """Dijkstra's algorithm on graph representation"""
        # Get node IDs for start and goal positions
        start_node = self.node_lookup.get((start.x(), start.y()))
        goal_node = self.node_lookup.get((goal.x(), goal.y()))

        if start_node is None or goal_node is None:
            return []

        # Initialize Dijkstra data structures
        distances = {start_node: 0}
        came_from = {start_node: None}
```

```
queue = [(0, start_node)]
visited = set()

while queue:
    current_dist, current = heapq.heappop(queue)

    if current == goal_node:
        break

    if current in visited:
        continue

    visited.add(current)

    # Explore neighbors in graph
    for neighbor, weight in self.graph[current]:
        if neighbor in visited:
            continue

        distance = current_dist + weight

        if neighbor not in distances or distance < distances[neighbor]:
            distances[neighbor] = distance
            came_from[neighbor] = current
```

```
                    heapq.heappush(queue, (distance, neighbor))

        # Reconstruct path
        if goal_node not in came_from:
            return []

        path = []
        current = goal_node
        while current is not None:
            x, y = self.coords[current]
            path.append(QPoint(x, y))
            current = came_from[current]
        path.reverse()

        return path
```

A* Planner

```
from base_planner import BasePlanner
import heapq
import math
from PyQt5.QtCore import QPoint

class AStarPlanner(BasePlanner):
    def euclidean_heuristic(self, current, goal):
```

```
        dx = current[1] - goal[1]
        dy = current[0] - goal[0]
        # Make heuristic more conservative with costs
        base_distance = math.sqrt(dx*dx + dy*dy)
        cost_factor = 1.0 + (self.costmap[current] / 255.0) * 15.0  # Increased cost weight
        return base_distance * cost_factor

    def plan_path(self, start: QPoint, goal: QPoint) -> list:
        # Convert QPoint to tuple coordinates (y, x)
        start_pos = (start.y(), start.x())
        goal_pos = (goal.y(), goal.x())

        # Initialize data structures
        frontier = []  # Priority queue
        heapq.heappush(frontier, (0, start_pos))
        came_from = {start_pos: None}
        g_score = {start_pos: 0}  # Cost from start to current node
        f_score = {start_pos: self.euclidean_heuristic(start_pos, goal_pos)}
        closed_set = set()

        while frontier:
            current_f, current = heapq.heappop(frontier)

            if current == goal_pos:
```

```
    # Path found, reconstruct it
    path = []
    while current is not None:
        path.append(QPoint(current[1], current[0]))
        current = came_from[current]
    path.reverse()
    return path

closed_set.add(current)

# Check all neighbors (8-connected grid)
for dy, dx in [(0,1), (1,0), (0,-1), (-1,0),
              (1,1), (-1,1), (1,-1), (-1,-1)]:
    neighbor = (current[0] + dy, current[1] + dx)

    # Skip if out of bounds
    if not (0 <= neighbor[0] < self.costmap.shape[0] and
           0 <= neighbor[1] < self.costmap.shape[1]):
        continue

    # Much stricter cost threshold
    if (neighbor in closed_set or
        self.costmap[neighbor] >= 100):  # Lower threshold to better avoid costs
        continue
```

```
                # Calculate movement cost with stronger penalties
                movement_cost = math.sqrt(2) if dx != 0 and dy != 0 else 1.0
                cost_at_cell = self.costmap[neighbor]

                # Exponential cost penalty
                cost_multiplier = math.exp(cost_at_cell / 50.0)  # Exponential growth with cost
                tentative_g = g_score[current] + movement_cost * cost_multiplier

                if neighbor not in g_score or tentative_g < g_score[neighbor]:
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g
                    f_score = tentative_g + self.euclidean_heuristic(neighbor, goal_pos)
                    heapq.heappush(frontier, (f_score, neighbor))

        return []  # No path found
```

Bidirectional A* planner

```
from base_planner import BasePlanner
import heapq
import math
from PyQt5.QtCore import QPoint

class BidirectionalAStarPlanner(BasePlanner):
```

```
def euclidean_heuristic(self, p1, p2):
    """Calculate Euclidean distance with cost consideration"""
    dx = p1[1] - p2[1]
    dy = p1[0] - p2[0]
    base_cost = math.sqrt(dx*dx + dy*dy)
    # Add cost consideration to heuristic
    terrain_factor = 1.0 + (self.costmap[p1] / 255.0) * 3.0
    return base_cost * terrain_factor

def get_neighbor_cost(self, current, next_pos, movement_cost):
    """Calculate cost between nodes considering terrain"""
    # Base movement cost
    base_cost = movement_cost

    # Calculate terrain costs
    current_cost = self.costmap[current]
    next_cost = self.costmap[next_pos]

    # Cost gradient penalty
    cost_gradient = abs(next_cost - current_cost) / 255.0
    terrain_cost = next_cost / 255.0

    # Combined cost calculation
    return base_cost * (1.0 + terrain_cost * 5.0 + cost_gradient * 3.0)
```

```
def plan_path(self, start: QPoint, goal: QPoint) -> list:
    start_pos = (start.y(), start.x())
    goal_pos = (goal.y(), goal.x())

    # Initialize data structures
    start_queue = [(0, start_pos)]
    goal_queue = [(0, goal_pos)]

    start_came_from = {start_pos: None}
    goal_came_from = {goal_pos: None}

    start_cost = {start_pos: 0}
    goal_cost = {goal_pos: 0}

    start_closed = set()
    goal_closed = set()

    best_path_cost = float('inf')
    meeting_point = None

    # Pre-compute movements and costs
    movements = [(0,1), (1,0), (0,-1), (-1,0),
                 (1,1), (-1,1), (1,-1), (-1,-1)]
```

```
base_costs = [1.0, 1.0, 1.0, 1.0,
             math.sqrt(2), math.sqrt(2), math.sqrt(2), math.sqrt(2)]

while start_queue and goal_queue and best_path_cost == float('inf'):
    # Forward search
    if start_queue:
        _, current = heapq.heappop(start_queue)

        if current in goal_cost:
            total_cost = start_cost[current] + goal_cost[current]
            if total_cost < best_path_cost:
                best_path_cost = total_cost
                meeting_point = current

        if current in start_closed:
            continue
        start_closed.add(current)

        for (dy, dx), base_cost in zip(movements, base_costs):
            next_pos = (current[0] + dy, current[1] + dx)

            if not (0 <= next_pos[0] < self.costmap.shape[0] and
                    0 <= next_pos[1] < self.costmap.shape[1]):
                continue
```

```
                # Skip high-cost and obstacle cells
                if self.costmap[next_pos] >= 200:  # Lower threshold
                    continue

                if next_pos in start_closed:
                    continue

                new_cost = start_cost[current] + self.get_neighbor_cost(current, next_pos,
base_cost)

                if next_pos not in start_cost or new_cost < start_cost[next_pos]:
                    start_cost[next_pos] = new_cost
                    start_came_from[next_pos] = current
                    priority = new_cost + self.euclidean_heuristic(next_pos, goal_pos)
                    heapq.heappush(start_queue, (priority, next_pos))

        # Backward search with similar improvements
        if goal_queue:
            _, current = heapq.heappop(goal_queue)

            if current in start_cost:
                total_cost = start_cost[current] + goal_cost[current]
                if total_cost < best_path_cost:
```

```
                best_path_cost = total_cost
                meeting_point = current

        if current in goal_closed:
            continue
        goal_closed.add(current)

        for (dy, dx), base_cost in zip(movements, base_costs):
            next_pos = (current[0] + dy, current[1] + dx)

            if not (0 <= next_pos[0] < self.costmap.shape[0] and
                    0 <= next_pos[1] < self.costmap.shape[1]):
                continue

            if self.costmap[next_pos] >= 200:
                continue

            if next_pos in goal_closed:
                continue

            new_cost = goal_cost[current] + self.get_neighbor_cost(current, next_pos, base_cost)

            if next_pos not in goal_cost or new_cost < goal_cost[next_pos]:
```

```
                goal_cost[next_pos] = new_cost
                goal_came_from[next_pos] = current
                priority = new_cost + self.euclidean_heuristic(next_pos, start_pos)
                heapq.heappush(goal_queue, (priority, next_pos))

    if meeting_point is None:
        return []

    # Reconstruct path
    path = []
    current = meeting_point
    while current is not None:
        path.append(QPoint(current[1], current[0]))
        current = start_came_from[current]
    path.reverse()

    current = goal_came_from[meeting_point]
    while current is not None:
        path.append(QPoint(current[1], current[0]))
        current = goal_came_from[current]

    return path
```

Base Planner

```python
import numpy as np
from dataclasses import dataclass
from PyQt5.QtCore import QPoint
import math


@dataclass
class MapMetaData:
    resolution: float = 0.05
    width: int = 0
    height: int = 0
    origin_x: float = 0.0
    origin_y: float = 0.0


class BasePlanner:
    def __init__(self, costmap: np.ndarray, metadata: MapMetaData):
        self.costmap = costmap
        self.metadata = metadata

    def plan_path(self, start: QPoint, goal: QPoint) -> list:
        """Base method to be implemented by specific planners"""
        raise NotImplementedError

    def get_path_cost(self, path: list) -> float:
        """Calculate total path cost considering costmap"""
```

```
        if not path:
            return float('inf')

        total_cost = 0
        for i in range(len(path) - 1):
            p1, p2 = path[i], path[i + 1]
            segment_cost = self.costmap[p2.y(), p2.x()]
            dist = math.sqrt((p2.x() - p1.x())**2 + (p2.y() - p1.y())**2)
            total_cost += dist * (1 + segment_cost/255.0)
        return total_cost
```

Path Sim

```
import sys
import numpy as np
import math
import time
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout,
                             QHBoxLayout, QPushButton, QLabel, QFileDialog,
                             QCheckBox, QSlider, QGroupBox, QTableWidget,
                             QTableWidgetItem, QProgressDialog)
from PyQt5.QtCore import Qt, QPoint, QThread, pyqtSignal
from PyQt5.QtGui import QPainter, QColor, QPixmap, QPen
import yaml
from PIL import Image
import os
```

```python
from scipy.ndimage import distance_transform_edt
from base_planner import MapMetaData
from mlb_planner import MLBPathPlanner
from astar_planner import AStarPlanner
from dijkstra_planner import DijkstraPlanner
from bidirectional_astar_planner import BidirectionalAStarPlanner


class MapLoader(QThread):
    finished = pyqtSignal(bool, object)
    progress_update = pyqtSignal(str, int)


    def __init__(self, file_name):
        super().__init__()
        self.file_name = file_name


    def run(self):
        try:
            self.progress_update.emit("Loading YAML configuration...", 20)
            QThread.msleep(100)


            with open(self.file_name, 'r') as f:
                map_info = yaml.safe_load(f)


            self.progress_update.emit("Processing map file...", 40)
```

```
            QThread.msleep(100)

            yaml_dir = os.path.dirname(os.path.abspath(self.file_name))
            image_path = os.path.join(yaml_dir, map_info['image'])

            if not os.path.exists(image_path):
                self.finished.emit(False, "Map image not found")
                return

            self.progress_update.emit("Loading map image...", 60)
            QThread.msleep(100)

            map_data = np.array(Image.open(image_path))

            self.progress_update.emit("Processing map data...", 80)
            QThread.msleep(100)

            self.finished.emit(True, map_data)

        except Exception as e:
            self.finished.emit(False, str(e))

class SimplePathfindingGUI(QMainWindow):
    def __init__(self):
```

```
super().__init__()
self.setWindowTitle("Simple Pathfinding GUI")

screen = QApplication.primaryScreen().geometry()
window_height = int(screen.height() * 0.9)
self.setGeometry(100, 0, 1300, window_height)

self.start_pos = None
self.goal_pos = None
self.placing_start = True
self.map_data = None
self.costmap = None
self.show_costmap = False
self.inflation_radius = 5
self.planners = {}
self.current_planner = None
self.path = []
self.planning_enabled = False
self.algorithm_paths = {}
self.algorithm_colors = {
    'MLB A*': QColor(255, 0, 0),
    'A*': QColor(0, 255, 0),
    'Dijkstra': QColor(0, 0, 255),
    'Bidirectional A*': QColor(255, 0, 255)
```

```
        }

        self.init_ui()

    def init_ui(self):
        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        main_layout = QHBoxLayout(main_widget)

        left_panel = QWidget()
        left_panel.setFixedWidth(300)
        left_layout = QVBoxLayout(left_panel)
        left_layout.setSpacing(10)
        left_layout.setContentsMargins(10, 15, 10, 15)

        group_box_style = """
            QGroupBox {
                font-weight: bold;
                font-size: 14px;
                padding-top: 20px;
                margin-top: 10px;
                border: 1px solid #cccccc;
                border-radius: 6px;
            }
```

```
    QGroupBox::title {
        subcontrol-origin: margin;
        left: 7px;
        padding: 0 5px 0 5px;
    }
"""

common_widget_style = """
    QLabel {
        font-size: 13px;
        min-height: 25px;
    }
    QPushButton {
        font-size: 13px;
        padding: 8px 15px;
        min-height: 35px;
        border-radius: 4px;
        background-color: #2196F3;
        color: white;
        border: none;
    }
    QPushButton:hover {
        background-color: #1976D2;
    }
```

```
    QPushButton:pressed {
        background-color: #0D47A1;
    }
    QCheckBox {
        font-size: 13px;
        padding: 8px;
        spacing: 8px;
    }
    QSlider {
        min-height: 25px;
    }
"""

map_controls = QGroupBox("Map Controls")
map_controls.setStyleSheet(group_box_style)
map_controls_layout = QVBoxLayout()
map_controls_layout.setSpacing(8)
map_controls_layout.setContentsMargins(8, 15, 8, 8)

load_map_btn = QPushButton("Load Map")
load_map_btn.setStyleSheet("""
    QPushButton {
        font-size: 13px;
        font-weight: bold;
```

```
        padding: 5px 15px;
        min-height: 30px;
        border-radius: 4px;
        background-color: #2196F3;
        color: white;
        border: none;
    }
    QPushButton:hover {
        background-color: #1976D2;
    }
    QPushButton:pressed {
        background-color: #0D47A1;
    }
""")
load_map_btn.clicked.connect(self.load_map)
map_controls_layout.addWidget(load_map_btn)

self.costmap_checkbox = QCheckBox("Show Costmap")
self.costmap_checkbox.setStyleSheet(common_widget_style)
self.costmap_checkbox.stateChanged.connect(self.toggle_costmap)
map_controls_layout.addWidget(self.costmap_checkbox)

slider_widget = QWidget()
slider_layout = QHBoxLayout(slider_widget)
```

```
slider_layout.setContentsMargins(5, 5, 5, 5)
slider_layout.setSpacing(10)

slider_label = QLabel("Inflation:")
slider_label.setStyleSheet("""
    QLabel {
        font-size: 13px;
        min-width: 60px;
    }
""")

self.radius_slider = QSlider(Qt.Horizontal)
self.radius_slider.setMinimum(0)
self.radius_slider.setMaximum(20)
self.radius_slider.setValue(self.inflation_radius)
self.radius_slider.setStyleSheet("""
    QSlider {
        min-height: 30px;
    }
    QSlider::groove:horizontal {
        height: 6px;
        background: #e0e0e0;
        border: 1px solid #999999;
        border-radius: 3px;
```

```
        margin: 0px;
    }
    QSlider::handle:horizontal {
        background: #2196F3;
        border: 2px solid #1976D2;
        width: 16px;
        height: 16px;
        margin: -8px 0;
        border-radius: 9px;
    }
    QSlider::handle:horizontal:hover {
        background: #1976D2;
        border: 2px solid #0D47A1;
    }
    QSlider::sub-page:horizontal {
        background: #2196F3;
        border: 1px solid #1976D2;
        border-radius: 3px;
    }
""")

self.radius_value_label = QLabel(f"{self.inflation_radius}")
self.radius_value_label.setStyleSheet("""
    QLabel {
```

```
        font-size: 13px;
        min-width: 30px;
        padding: 2px 5px;
        background: #e0e0e0;
        border-radius: 3px;
    }
""")

self.radius_slider.valueChanged.connect(self.update_inflation_radius)


slider_layout.addWidget(slider_label)
slider_layout.addWidget(self.radius_slider)
slider_layout.addWidget(self.radius_value_label)
map_controls_layout.addWidget(slider_widget)

map_controls.setLayout(map_controls_layout)
left_layout.addWidget(map_controls)


algo_group = QGroupBox("Select Algorithms")
algo_group.setStyleSheet(group_box_style)
algo_layout = QVBoxLayout()
algo_layout.setSpacing(5)
algo_layout.setContentsMargins(8, 15, 8, 8)
```

```
self.algo_checkboxes = {}
for algo_name in ['MLB A*', 'A*', 'Dijkstra', 'Bidirectional A*']:
    checkbox = QCheckBox(algo_name)
    if algo_name == 'MLB A*':
        checkbox.setChecked(True)
    self.algo_checkboxes[algo_name] = checkbox
    algo_layout.addWidget(checkbox)

algo_group.setLayout(algo_layout)
left_layout.addWidget(algo_group)

legend_group = QGroupBox("Path Colors")
legend_group.setStyleSheet(group_box_style)
legend_layout = QVBoxLayout()
legend_layout.setSpacing(1)
legend_layout.setContentsMargins(2, 2, 2, 2)

for algo_name, color in self.algorithm_colors.items():
    label_widget = QWidget()
    label_layout = QHBoxLayout(label_widget)
    label_layout.setContentsMargins(2, 0, 2, 0)

    color_dot = QLabel("●")
```

```
            color_dot.setStyleSheet(f"color: rgb({color.red()}, {color.green()},
{color.blue()})")
            name_label = QLabel(algo_name)
            name_label.setStyleSheet("color: black")

            label_layout.addWidget(color_dot)
            label_layout.addWidget(name_label)
            label_layout.addStretch()
            legend_layout.addWidget(label_widget)

        legend_group.setLayout(legend_layout)
        left_layout.addWidget(legend_group)

        controls_group = QGroupBox("Controls")
        controls_group.setStyleSheet(group_box_style)
        controls_layout = QVBoxLayout()
        controls_layout.setSpacing(8)
        controls_layout.setContentsMargins(8, 15, 8, 8)

        self.plan_path_btn = QPushButton("Plan Path")
        self.plan_path_btn.setStyleSheet("""
            QPushButton {
                font-size: 13px;
                font-weight: bold;
```

```
            padding: 5px 15px;
            min-height: 30px;
            border-radius: 4px;
            background-color: #2196F3;
            color: white;
            border: none;
        }
        QPushButton:hover {
            background-color: #1976D2;
        }
        QPushButton:pressed {
            background-color: #0D47A1;
        }
        QPushButton:disabled {
            background-color: #BDBDBD;
        }
    """)
    self.plan_path_btn.clicked.connect(self.trigger_path_planning)
    self.plan_path_btn.setEnabled(False)
    controls_layout.addWidget(self.plan_path_btn)

    instructions = QLabel("1. Click to place start (blue)\n2. Click to place goal (green)\n3.
Click 'Plan Path'")
    instructions.setWordWrap(True)
```

```
instructions.setStyleSheet("""
    QLabel {
        font-size: 13px;
        line-height: 1.6;
        padding: 10px;
        background-color: #f5f5f5;
        border-radius: 4px;
        border: 1px solid #e0e0e0;
        margin: 5px;
        font-weight: 500;
    }
""")
controls_layout.addWidget(instructions)

controls_group.setLayout(controls_layout)
left_layout.addWidget(controls_group)

left_layout.addStretch()

center_panel = QWidget()
center_layout = QVBoxLayout(center_panel)
center_layout.setContentsMargins(0, 0, 0, 0)

self.map_label = QLabel()
```

```
self.map_label.setMinimumSize(800, 600)
self.map_label.setAlignment(Qt.AlignCenter)
self.map_label.setStyleSheet("border: 1px solid black;")
center_layout.addWidget(self.map_label)

right_panel = QWidget()
right_panel.setFixedWidth(350)
right_layout = QVBoxLayout(right_panel)
right_layout.setContentsMargins(5, 5, 5, 5)

results_title = QLabel("Algorithm Performance")
results_title.setStyleSheet("""
    font-weight: bold;
    font-size: 12pt;
    padding: 5px;
""")
right_layout.addWidget(results_title)

self.results_table = QTableWidget()
self.results_table.setColumnCount(3)
self.results_table.setHorizontalHeaderLabels(["Algorithm", "Time (s)", "Length"])
self.results_table.setMinimumHeight(200)
self.results_table.setMaximumHeight(250)
self.results_table.horizontalHeader().setStretchLastSection(True)
```

```
self.results_table.setEditTriggers(QTableWidget.NoEditTriggers)
self.results_table.setStyleSheet("""
    QTableWidget {
        border: 1px solid #ccc;
        border-radius: 4px;
        padding: 0px;
    }
    QTableWidget::item {
        padding: 4px;
    }
""")

self.results_table.setColumnWidth(0, 120)
self.results_table.setColumnWidth(1, 70)
self.results_table.setColumnWidth(2, 70)

right_layout.addWidget(self.results_table)

self.recommendation_label = QLabel()
self.recommendation_label.setStyleSheet("color: green; font-weight: bold;")
right_layout.addWidget(self.recommendation_label)
right_layout.addStretch()

main_layout.addWidget(left_panel)
```

```
        main_layout.addWidget(center_panel, 1)
        main_layout.addWidget(right_panel)

    def generate_costmap(self):
        if self.map_data is None:
            return

        try:
            binary_map = (self.map_data < 128).astype(np.uint8)
            distance_map = distance_transform_edt(1 - binary_map)
            self.costmap = np.zeros_like(distance_map)
            self.costmap[binary_map == 1] = 254

            inflation_cells = (distance_map <= self.inflation_radius) & (binary_map != 1)
            distances = distance_map[inflation_cells]

            if len(distances) > 0:
                costs = 253 * np.exp(-distances / (self.inflation_radius / 3))
                self.costmap[inflation_cells] = costs.clip(1, 253)

            metadata = MapMetaData(
                resolution=0.05,
                width=self.costmap.shape[1],
                height=self.costmap.shape[0]
```

```
            )

            self.planners = {
                'MLB A*': MLBPathPlanner(self.costmap, metadata),
                'A*': AStarPlanner(self.costmap, metadata),
                'Dijkstra': DijkstraPlanner(self.costmap, metadata),
                'Bidirectional A*': BidirectionalAStarPlanner(self.costmap, metadata)
            }
            self.current_planner = self.planners['MLB A*']

        except Exception as e:
            print(f"Error generating costmap: {str(e)}")

    def toggle_costmap(self, state):
        self.show_costmap = bool(state)
        self.display_map()

    def load_map(self):
        file_name, _ = QFileDialog.getOpenFileName(self, "Open Map File", "",
                                                   "YAML files (*.yaml);;All Files (*)")
        if file_name:
            try:
                self.progress = QProgressDialog("Initializing...", None, 0, 100, self)
                self.progress.setWindowModality(Qt.ApplicationModal)
```

```
        self.progress.setWindowTitle("Loading Map")
        self.progress.setCancelButton(None)
        self.progress.setMinimumDuration(0)
        self.progress.setAutoClose(False)
        self.progress.setWindowFlags(Qt.Dialog | Qt.FramelessWindowHint |
Qt.WindowStaysOnTopHint)
        self.progress.setStyleSheet("""
          QProgressDialog {
            background-color: white;
            border: 2px solid #2196F3;
            border-radius: 8px;
            padding: 20px;
            min-width: 300px;
          }
          QLabel {
            font-size: 14px;
            color: #333;
            margin: 10px;
            qproperty-alignment: AlignCenter;
          }
          QProgressBar {
            border: 1px solid #2196F3;
            border-radius: 5eldompx;
            text-align: center;
```

```
            height: 20px;
            background-color: white;
        }
        QProgressBar::chunk {
            background-color: #2196F3;
            border-radius: 4px;
        }
    """)

    self.progress.show()
    QApplication.processEvents()

    self.loader = MapLoader(file_name)
    self.loader.progress_update.connect(self._update_progress)
    self.loader.finished.connect(
        lambda success, result: self.handle_map_load_complete(success, result)
    )
    self.loader.start()

except Exception as e:
    if hasattr(self, 'progress'):
        self.progress.close()
    print(f"Error in load_map: {str(e)}")
```

```
def _update_progress(self, text, value):
    if hasattr(self, 'progress'):
        self.progress.setLabelText(text)
        self.progress.setValue(value)
        QApplication.processEvents()

def handle_map_load_complete(self, success, result):
    try:
        if success:
            self.map_data = result
            self.progress.setLabelText("Generating costmap...")
            self.progress.setValue(90)
            QApplication.processEvents()

            self.generate_costmap()
            self.progress.setLabelText("Finalizing...")
            self.progress.setValue(95)
            QApplication.processEvents()

            self.display_map()
            self.progress.setValue(100)

        else:
            print(f"Error loading map: {result}")
```

```
        except Exception as e:
            print(f"Error in handle_map_load_complete: {str(e)}")
        finally:
            if hasattr(self, 'progress'):
                self.progress.close()
                delattr(self, 'progress')

    def display_map(self):
        if self.map_data is None:
            return

        height, width = self.map_data.shape[:2]
        pixmap = QPixmap(width, height)
        pixmap.fill(Qt.white)

        painter = QPainter(pixmap)

        if self.show_costmap and self.costmap is not None:
            for y in range(height):
                for x in range(width):
                    cost = self.costmap[y, x]
                    if cost > 0:
                        if cost >= 254:
```

```
                painter.fillRect(x, y, 1, 1, QColor(0, 0, 0))
            else:
                intensity = int((cost / 253) * 255)
                painter.fillRect(x, y, 1, 1,
                           QColor(255, 255 - intensity, 0))
else:
    for y in range(height):
        for x in range(width):
            if self.map_data[y, x] < 128:
                painter.fillRect(x, y, 1, 1, QColor(0, 0, 0))

if self.start_pos:
    painter.setPen(QPen(Qt.blue, 2))
    painter.setBrush(QColor(0, 0, 255, 127))
    painter.drawEllipse(self.start_pos, 10, 10)

if self.goal_pos:
    painter.setPen(QPen(Qt.green, 2))
    painter.setBrush(QColor(0, 255, 0, 127))
    painter.drawEllipse(self.goal_pos, 10, 10)

for algo_name, path in self.algorithm_paths.items():
    if path:
        color = self.algorithm_colors[algo_name]
```

```
            painter.setPen(QPen(color, 2))
            for i in range(len(path) - 1):
                painter.drawLine(path[i], path[i + 1])

        painter.end()
        self.map_label.setPixmap(pixmap)

    def mousePressEvent(self, event):
        if self.map_data is None:
            return

        pos = self.map_label.mapFrom(self, event.pos())

        map_height, map_width = self.map_data.shape[:2]
        label_width = self.map_label.width()
        label_height = self.map_label.height()

        x_margin = (label_width - map_width) // 2
        y_margin = (label_height - map_height) // 2

        adjusted_pos = QPoint(pos.x() - x_margin, pos.y() - y_margin)

        if not self.is_valid_position(adjusted_pos):
            return
```

```
    if self.placing_start:
        self.start_pos = adjusted_pos
        self.placing_start = False
    else:
        self.goal_pos = adjusted_pos
        self.placing_start = True

    self.plan_path_btn.setEnabled(self.start_pos is not None and
                                  self.goal_pos is not None)

    self.display_map()

def is_valid_position(self, pos):
    if not self.map_data is None:
        x, y = pos.x(), pos.y()
        if 0 <= x < self.map_data.shape[1] and 0 <= y < self.map_data.shape[0]:
            return self.map_data[y, x] >= 128
    return False

def update_inflation_radius(self, value):
    self.inflation_radius = value
    self.radius_value_label.setText(str(value))
    if self.map_data is not None:
```

```
        self.generate_costmap()
        if self.show_costmap:
            self.display_map()

    def trigger_path_planning(self):
        if not (self.start_pos and self.goal_pos and self.costmap is not None):
            return

        start_cost = self.costmap[self.start_pos.y(), self.start_pos.x()]
        goal_cost = self.costmap[self.goal_pos.y(), self.goal_pos.x()]

        if start_cost >= 254 or goal_cost >= 254:
            print("Error: Start or goal position is in obstacle area!")
            return

        self.algorithm_paths.clear()

        performance_metrics = {}

        try:
            for algo_name, checkbox in self.algo_checkboxes.items():
                if checkbox.isChecked():
                    planner = self.planners[algo_name]
```

```
start_time = time.time()
path = planner.plan_path(self.start_pos, self.goal_pos)
computation_time = time.time() - start_time

if path:
    self.algorithm_paths[algo_name] = path

    total_cost = 0
    path_length = 0
    max_cost_encountered = 0
    cells_explored = len(path)

    for i in range(len(path) - 1):
        p1, p2 = path[i], path[i + 1]
        segment_cost = self.costmap[p2.y(), p2.x()]
        max_cost_encountered = max(max_cost_encountered, segment_cost)

        dist = math.sqrt((p2.x() - p1.x())**2 + (p2.y() - p1.y())**2)
        path_length += dist

        total_cost += dist * (1 + segment_cost/255.0)

    performance_metrics[algo_name] = {
        'total_cost': total_cost,
```

```
                    'path_length': path_length,
                    'computation_time': computation_time,
                    'max_cost': max_cost_encountered,
                    'cells_explored': cells_explored
                }
            else:
                print(f"{algo_name}: No valid path found")

        if performance_metrics:
            self._generate_comparison_report(performance_metrics)

        self.display_map()

    except Exception as e:
        print(f"Error during path planning: {str(e)}")

def _generate_comparison_report(self, metrics):
    self.results_table.setRowCount(0)

    min_time = min(m['computation_time'] for m in metrics.values())
    max_time = max(m['computation_time'] for m in metrics.values())
    min_length = min(m['path_length'] for m in metrics.values())
    max_length = max(m['path_length'] for m in metrics.values())
```

```
good_color = QColor(200, 255, 200)
medium_color = QColor(255, 255, 200)
poor_color = QColor(255, 200, 200)

for algo_name, metric in metrics.items():
    row = self.results_table.rowCount()
    self.results_table.insertRow(row)

    name_item = QTableWidgetItem(algo_name)
    self.results_table.setItem(row, 0, name_item)

    time_val = metric['computation_time']
    time_item = QTableWidgetItem(f"{time_val:.3f}")

    if time_val <= min_time * 1.2:
        time_item.setBackground(good_color)
    elif time_val >= max_time * 0.8:
        time_item.setBackground(poor_color)
    else:
        time_item.setBackground(medium_color)

    self.results_table.setItem(row, 1, time_item)

    length_val = metric['path_length']
```

```
    length_item = QTableWidgetItem(f"{length_val:.1f}")

    if length_val <= min_length * 1.1:
        length_item.setBackground(good_color)
    elif length_val >= max_length * 0.9:
        length_item.setBackground(poor_color)
    else:
        length_item.setBackground(medium_color)

    self.results_table.setItem(row, 2, length_item)

best_overall = min(metrics.items(), key=lambda x: (
    x[1]['computation_time'] * 0.5 +
    x[1]['path_length'] * 0.5
))[0]

self.recommendation_label.setText(f"Recommended: {best_overall}")
self.recommendation_label.setStyleSheet("""
    QLabel {
        color: #2e7d32;
        font-weight: bold;
        font-size: 12px;
        padding: 5px;
        background-color: #e8f5e9;
```

```
            border-radius: 4px;
        }
    """)

    print("\n=== Detailed Algorithm Performance ===")
    print("----------------------------------------")
    print(f"{'Algorithm':<15} {'Path Cost':<12} {'Length':<10} {'Time (s)':<10} {'Max Cost':<10}")
    print("-" * 60)

    for algo_name, metric in metrics.items():
        print(f"{algo_name:<15} "
              f"{metric['total_cost']:>9.2f} "
              f"{metric['path_length']:>9.2f} "
              f"{metric['computation_time']:>9.3f} "
              f"{metric['max_cost']:>9.1f}")

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = SimplePathfindingGUI()
    window.show()
    sys.exit(app.exec_())
```