J. Innovative Eng. Nat. Sci., vol. 5, no. 2, pp. 588-605, 2025.
http://doi.org/10.61112/jiens.1618552

Original Paper

Journal of Innovative Engineering and Natural Science
(Yenilikçi Mühendislik ve Doğa Bilimleri Dergisi)
https://dergipark.org.tr/en/pub/jiens

Comparative performance analysis of path planning algorithms for mecanum wheeled mobile robots using ROS-Gazebo

Dilara Galeli[a], Bilge Kartal Cetin[b] and Kamil Cetin[a,c,*]

[a]*Department of Electrical and Electronics Engineering, Faculty of Engineering and Architecture, Izmir Katip Celebi University, Izmir, 35620, Turkiye.*
[b]*Department of Electrical and Electronics Engineering, Faculty of Engineering, Ege University, Izmir, 35100, Turkiye.*
[c]*Akıllı Fabrika Sistemleri Uygulama ve Araştırma Merkezi, Izmir Katip Celebi University, Izmir, 35620, Turkiye.*

ARTICLE INFO

Article history:
Received 12 January 2025
Received in revised form 10 March 2025
Accepted 27 March 2025

Available online

Keywords:
Mobile robot
LiDAR
ROS
Gazebo
Path planning
Autonomous navigation
Performance metrics

ABSTRACT

Effective path planning is crucial for mobile robots to navigate safely and efficiently in various environments. The main purpose of this study is to compare and investigate the performances of three important path planning algorithms, namely A*, Dijkstra and Rapidly exploring Random Trees (RRT), for a mobile autonomous robot with LiDAR sensors and mechanical wheels. The mobile robot can move in multiple directions with its mecanum wheels and can precisely avoid obstacles with the help of LiDAR sensors and decision-making mechanisms (path planning algorithms). Various scenarios with cluttered and open areas were used in a simulated ROS Gazebo environment to evaluate the effectiveness of each algorithm. The performance metrics among the algorithms were analyzed with respect to path length, time to reach the target, velocity command frequency and CPU utilization. In terms of travel time performance criterion, A* performed approximately 35% better than Dijkstra and 85% better than RRT. In terms of the path length traveled, A* reached the target in approximately 11% shorter path length than Dijkstra and 17% shorter path length than RRT. In terms of the number of velocity commands processed, A* outperformed Dijkstra by approximately 36% and RRT by 38%. In terms of CPU utilization performance criterion, RRT performed approximately 10% better than A* and 74% better than Dijkstra. As a result, significant information was obtained about the strengths and weaknesses of each algorithm in selecting the most appropriate path planning strategies for mobile autonomous robots.

I. INTRODUCTION

With the rapid proliferation of mobile autonomous robots in various industries in recent years, efficient autonomous navigation has become a critical necessity to enable robots to operate effectively in complex and dynamic environments [1] and [2]. This topic poses serious challenges such as mapping unknown environments, positioning robots in these environments, planning optimal paths towards the desired destination, and avoiding obstacles in real time. The most important path planning algorithms developed such as Dijkstra developed by E. W. Dijkstra in 1959 [3], A* developed by P. E. Hart in 1968 [4], and Rapidly-exploring Random Tree (RRT) developed by S. M. LaValle in 1998 [5] play a significant role in addressing these challenges while enabling robots to successfully perform their tasks.

Today, with the new advancements in the field of path planning and navigation of mobile autonomous systems, these algorithms (A*, Dijkstra and RRT) are still widely preferred for optimal route generation. Various studies have investigated the performance of these widespread algorithms, compared their performance and proposed hybrid approaches to improve the overall efficiency of the path planning process. [6] introduced the Hybrid RRT-A* algorithm by combining RRT algorithm's exploration capability and A* algorithm's optimal path generation to improve convergence and computational efficiency within the ROS framework. Similarly, [7] and [8] have

Corresponding author. Tel.: +90-232-270-5500; e-mail: kamil.cetin@ikcu.edu.tr proposed hybrid approaches such as RRT-Dijkstra, demonstrating the improvements in execution time and path optimization, and therefore supporting that hybrid approaches can propose significant improvements compared to classic ones. Apart from proposing hybrid approaches, several studies compared the algorithms and emphasized the trade-offs. In this context, [9] and [10] examined and compared A and RRT algorithms and emphasized A* algorithm's advantage in creating shorter paths and RRT's advantage in thorough exploration. In SLAM-related experiments, [11] examined the impact of various types of geometric shapes on mapping performance using TurtleBot3 robot in Gazebo, demonstrated that the object geometry can significantly affect the SLAM performance. [12] assessed the performance of A* and Dijkstra algorithms in the ROS framework and demonstrated that the Dijkstra algorithm surpasses A* by creating shorter and more continuous routes. Additionally, [13] and [14] provided comprehensive overviews of widespread path-planning methods by emphasizing the trade-offs between execution time and path efficiency across various path-planning methodologies. [15] proposed an improved A* algorithm within ROS framework, which outperformed the classic A* algorithm in both simulation environment and real-world experiments by generating shorter paths in a shorter time than the classic A* algorithm. In addition, [16] examined the performance of A* and RRT algorithms using TurtleBot3 robot within static and dynamic environments and indicated that A* is more suitable for static environments whereas RRT is more suitable for dynamic environments in terms of route precision.

The aforementioned studies generally emphasize the advantages and disadvantages of A*, Dijkstra and RRT algorithms, providing a valuable source for robotics applications. However, these and many similar studies in the literature contain some research gaps. In [6, 7, 8, 9, 10, 13], only theoretical studies have been conducted and there is no practical simulation study with mobile robots. [11] only used SLAM navigation and did not integrate these path planning algorithms on the mobile robot. [12] implemented a mobile robot application using only A* and Dijkstra algorithms and compared them only in terms of path length. [14] has implemented all of these algorithms in ROS and Rviz simulation environment, but there is no realistic mobile robot study with measurement-based feedback. In addition, in terms of performance, only path length and duration were examined in a more limited way. [15] implemented only A* algorithm on a LiDAR based mobile robot in ROS and Rviz simulation environment, but did not make any analysis and performance comparison in terms of path length and time compared to other algorithms. [16] implemented simulation applications of all of these algorithms in the Gazebo environment of the TurtleBot mobile robot, but did not make any performance comparisons.

Robot Operating System (ROS) is an open-source framework developed in 2007 for robotics research and development [17, 18]. It supports multiple programming languages and enables flexible communication between nodes through a master node and topics [19]. ROS's rapid growth is driven by its open-source nature and collaborative environment, allowing researchers to contribute to and benefit from shared resources. Gazebo is a ROS-based 3D simulation tool that allows robotic systems, including sensors and motors, to be designed and tested in a virtual environment [20]. It uses "world" models, created with the Build editor, where robots interact with physical elements like terrain and obstacles defined in Simulation Description Format (SDF) files. By integrating with ROS, users can simulate and control robots in these worlds, allowing for realistic testing and validation before real-world deployment. RViz is a powerful visualization tool in ROS that enables real-time monitoring of robot status, sensor data, and navigation information, essential for mapping and path planning [17]. During simulation in Gazebo, RViz helps visualize data such as laser scans, point clouds, and odometry to ensure accurate robot behavior and performance. According to the research gaps in the above-mentioned studies and in the literature in general, this study aims to examine the performance of these algorithms by implementing a mobile robot in a more realistic simulation environment and to present a more comprehensive comparison result.

SLAM (Simultaneous Localization and Mapping) is a technique widely used by robotics researchers and developers to create a map of an unknown environment. Using odometric data such as laser scans from LiDAR (Light Detection and Ranging) sensor, a map of an unknown environment can be created, and the robot can be positioned on the map simultaneously [21]. Among different SLAM techniques, in this paper we have chosen to use the Gmapping algorithm for LiDAR-based 2D mapping. The ROS navigation stack is a directory that allows the interaction of the packages to make a mobile robot navigate around its environment autonomously. It simply stitches together most of the basic competencies of a robot: self-location, path planning, obstacle avoidance, and motion control. When a 2D navigation goal (2DNavGoal) is set for a robot in RViz with the ROS navigation stack, the robot can reach the desired location autonomously. This work aims to perform a performance test comparison of the A*, Dijkstra, and RRT path planning algorithms on a mobile autonomous robot with LiDAR sensors and mecanum wheels designed in Gazebo within an indoor 'world' environment that is mapped using the SLAM Gmapping method within the ROS framework.

II. PATH PLANNING ALGORITHMS

2.1. A algorithm*

Hart in [19] demonstrated the best-in-class property of a class of search strategies by explaining how heuristic information from the problem domain can be incorporated into the formal mathematical theory of graph search. A* algorithm has been determined as a heuristic search algorithm type that uses a combination of heuristic and exact cost to discover the most suitable route between a starting point and the goal [22]. It compares the cost values of its expanded nodes and selects the most cost-promising node for further exploration up to the point when the target node has been reached. By always expanding the node with the lowest cost, a path is formed which has a minimum total cost [23]. The evaluation function is expressed as

$$f(n) = g(n) + h(n) \tag{1}$$

with g(n) as a known cost from the start node to the node 'n' while h(n) represents a heuristic estimation (Manhattan, Euclidean or Chebyshev) of the cost from the current node to the goal. A* prioritizes nodes to explore by integrating g(n) and h(n).

In the A* algorithm, the heuristic function h(n) is influential in its performance. The Euclidean distance heuristic calculates the straight-line distance between two points. This is perfect for when one can move diagonally, as in continuous or grid-based environments with movement based on 'free'. The Euclidean distance heuristic function is determined by the square root of the sum of squares of differences between the Cartesian coordinates of the points [24]. It is given by the equation below [25]

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2}$$

where (x_1, y_1) is the current position and (x_2, y_2) is the target position in Cartesian space.

The distance measured along the grid-based axes when the movement is constricted to vertical and horizontal paths is known as the entire Manhattan distance heuristic. Mathematically, it is a summation of the absolute differences of the positions in Cartesian space [24], given in the equation below [25]

$$h(n) = |x_2 - x_1| + |y_2 - y_1| \tag{3}$$

In this study, we used Euclidean and Manhattan distance heuristics for performance comparisons.

2.2. Dijkstra's algorithm

Dijkstra's algorithm developed by Dijkstra in [3] is a search-based shortest-path algorithm, which is used to find the shortest path between a starting node (which is also called the 'source node') and all other nodes in a weighted graph [26]. Its working principle is iteratively selecting the node with the shortest known distance from the starting node, exploring all its neighbors and updating their tentative distances if a shorter path is found. Dijkstra algorithm can find the shortest path with an optimal solution, but it requires more time to complete the searching process [27].

The difference between the Dijkstra and A* algorithm is that the A* algorithm uses a heuristic function to prioritize nodes that are likely to lead to a better path whereas Dijkstra's algorithm explores all possible paths without prioritizing them [28].

2.3. RRT algorithm

RRT, first developed in [5], is probabilistic planner that excels at addressing challenging path planning problems in high-dimensional spaces. RRT is a sampling-based path planning algorithm, which is based on incremental sampling [29]. It is a probabilistic algorithm that builds a tree by incrementally extending branches towards random samples in the configuration space.

Sampling-based algorithms such as RRT have the advantage of being able to find a feasible motion plan relatively quickly even in complex environments. However, even though the RRT algorithm quickly finds a feasible solution, it may not always provide the most optimal path [29].

III. SIMULATION SETUP

The experiments are conducted using the simulation environment of the Robot Operating System (ROS Noetic), with Gazebo 11.11.0. This framework operates under the Ubuntu 20.04 (64-bit) operating system. For the visualization of the SLAM and path-planning processes, RViz is utilized. For the data collection process, terminal windows are utilized for managing rosbag recordings. Rosbag allows data capture and analysis for debugging and testing by recording and playing back the messages that are exchanged between ROS nodes and storing the message data in a file. A custom Python script is implemented to start recording rosbag files when the robot starts moving and automatically stop the recording when the robot reaches the target and stops. This approach assures that only the relevant data during the robot's motion is captured. From the recorded rosbag files, travel times along routes and number of messages sent to cmd_vel topic are directly obtained. In addition, the recorded rosbag files are later analyzed and Python scripts are created to obtain the performance metrics such as the total traveled distance and average CPU utilization. These scripts process /odom topic data to calculate the traveled distance and monitor CPU usage during navigation by detecting when navigation starts, starting CPU logging and stopping it when the goal is reached. In the simulation study, each algorithm has been run under the same conditions, environment, mobile robot, target points and parameters given in Table 1 and Table 2 in order to make a fair performance comparison between the algorithms.

Table 1. Local planner configuration

Parameter	**Value**
max_vel_x (maximum forward linear velocity)	0.4 m/s
max_vel_x_backwards (maximum reverse linear velocity)	0.2 m/s
max_vel_theta (maximum angular velocity)	0.3 rad/s
acc_lim_x (linear acceleration limit)	0.5 m/s^2
acc_lim_theta (angular acceleration limit)	0.5 rad/s^2

3.1. Robot model

In this study, a four-wheeled mecanum mobile robot with a LiDAR sensor is used as shown in Figure 1. Robot descriptions are provided in Xacro (XML Macro) format, which defines links, joints, mass, size and inertia values of the robot as given in Table 2. For defining the kinematics of the mobile robot, differential drive controller Gazebo plugin 'diff_drive_controller' is used. For mapping process and navigation, a LiDAR sensor was specifically designed with the dimensions, and an operational range, as defined in Table 3, that is suitable for effective mapping and navigation and mounted on top of the robot.

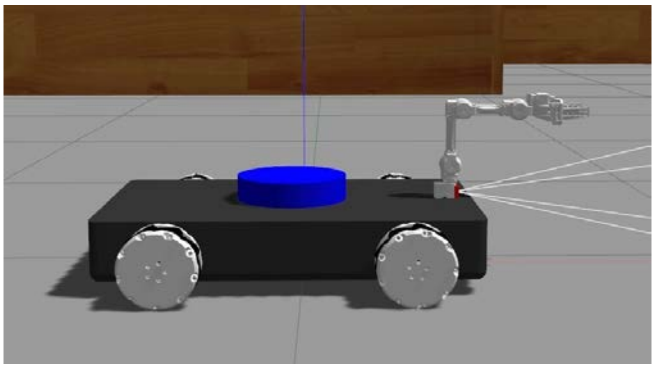

Figure 1. Mecanum wheeled mobile robot model in Gazebo

Table 2. Mecanum wheeled mobile robot properties

Parameter	**Value**
Wheel torque	30 Nm
Wheel diameter	0.13 m
Wheel acceleration	5 rad/s^2
Mass	31.0431 kg

Table 3. Specifications of the LiDAR sensor designed in Gazebo

Dimensions *(m)*		**Range** *(m)*		**Field of View** *(°)*		**Scanning Parameters**		**Noise**		
Radius	**Height**	**Min**	**Max**	**Min**	**Max**	**Samples**	**Resolution** *(°)*	**Type**	**Mean** *(m)*	**StdDev** *(m)*
0.12	0.07	0.3	12	-180	180	360	1	Gaussian	0	0.02

The field of view (FOV) of the LiDAR sensor describes the angular range of the area it can scan. In this study, the LiDAR’s FOV, as given in Table 1, ranges from -180 to 180, which indicates that it can scan the entire horizontal plane in a full 360 degrees circle. In addition, LiDAR takes 360 samples per scan, with each sample corresponding to a 1 degree of interval. In other words, LiDAR collects one data point for each degree of rotation.

3.2. Gazebo world

A special Gazebo world (as shown in Figure 2) is created using the Build editor. The environment is enclosed by outer walls and within these walls, there are additional smaller walls and three tables. Two of the tables, identified by dark yellow color, are smaller and have a single leg while the third table, which is larger and gray, is supported with four legs. These tables and the smaller walls were added to the world to increase the complexity of the robot navigation process and to facilitate the testing of planning algorithm behaviors.

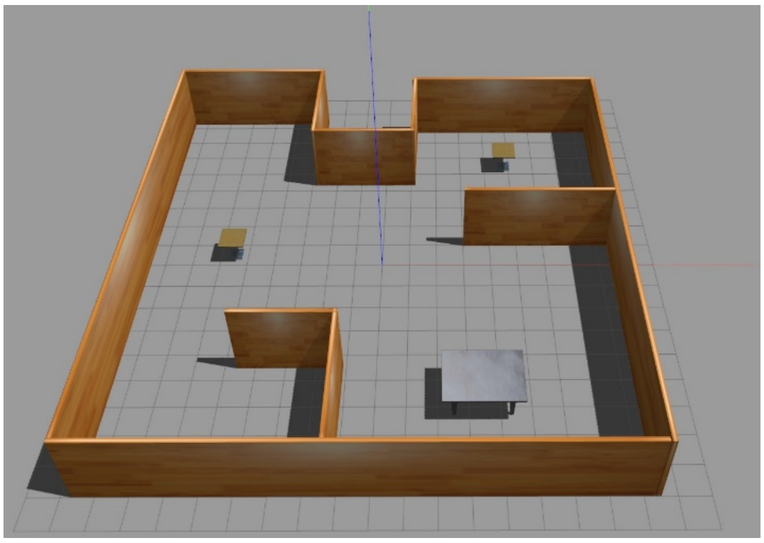

Figure 2. Custom-designed Gazebo world

3.3. SLAM and generated world map

For the navigation process of the mobile robot, a map of the Gazebo world needs to be created. In this study, the world map is generated and provided to the navigation system using the SLAM Gmapping algorithm.

In the SLAM algorithm’s flowchart (as shown its architecture in Figure 3), the process starts by detecting whether the robot is in motion. If movement is detected, odometry data and laser data are gathered. These two inputs are processed by the SLAM algorithm for estimating the robot’s position while correcting errors in odometry. The SLAM algorithm updates the map of the environment simultaneously, and the map is managed by the map server.

This continuous loop allows the robot to maintain accurate localization and mapping, which is essential for autonomous navigation. If no motion is detected, the system stops updates until the robot resumes motion.

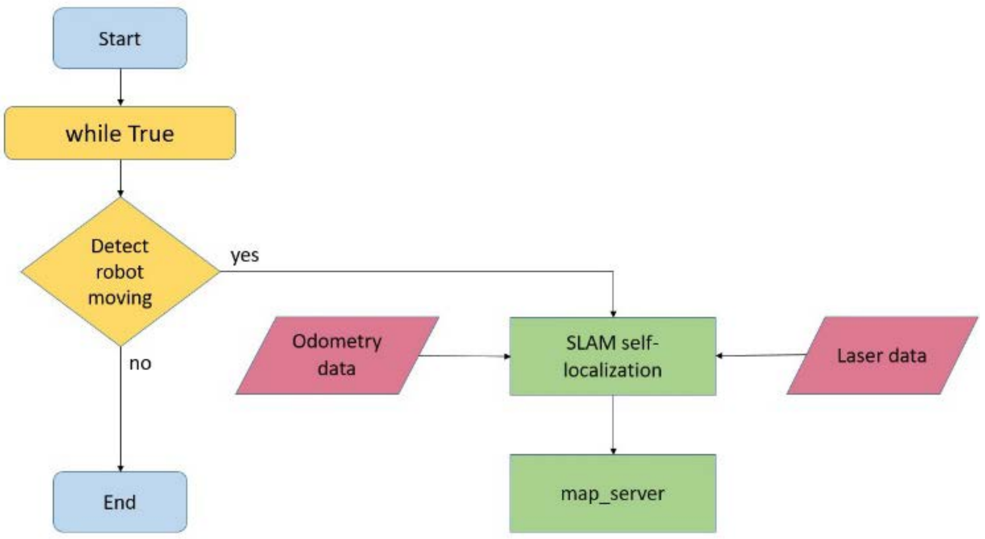

Figure 3. SLAM localization flowchart

There are several SLAM algorithms in ROS, such as HectorSLAM, Gmapping, KartoSLAM, CoreSLAM and LagoSLAM [30]. In this study, Gmapping is used because of its efficiency and its capability for 2D occupancy grid mapping. Gmapping is a SLAM algorithm, which is based on Rao-Blackwellized particle filter (RBPF) that creates grid maps from laser data. It considers the latest observation from sensors and the robot's movement to generate the proposal distribution [31].

Figure 4 shows the generated map of the custom-designed Gazebo environment in RViz after the SLAM Gmapping algorithm process. The map accurately depicts the characteristics of the environment, such as walls, tables and open spaces.

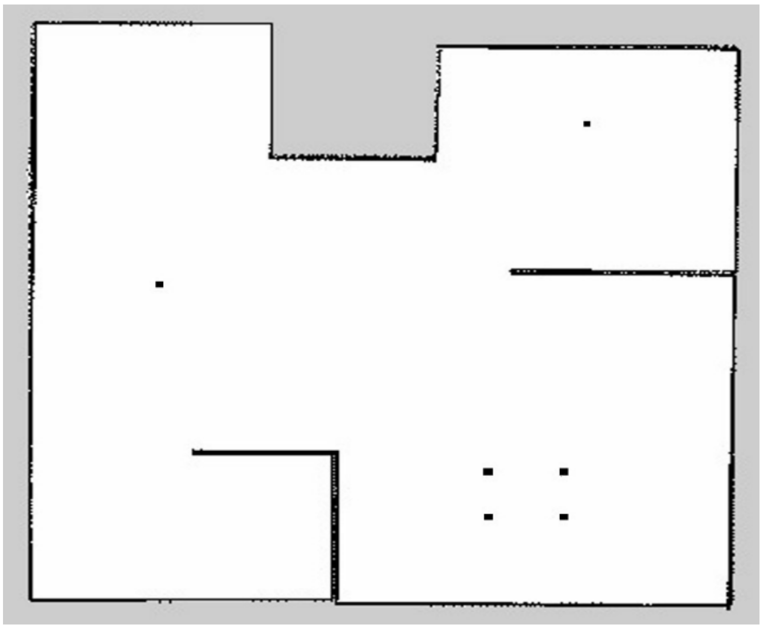

Figure 4. Generated world map

3.4. Selection of target points

As shown in Figure 5, target points in RViz are specifically selected for observing the mobile robot navigation effectively. These five points are strategically chosen to monitor potential collisions with the environmental features in the Gazebo world, such as tables and walls. By selecting the target points in critical locations (middle, top left, bottom left, top right, bottom right), the aim is to assess the robot's obstacle avoidance capabilities with the help of path planning algorithms.

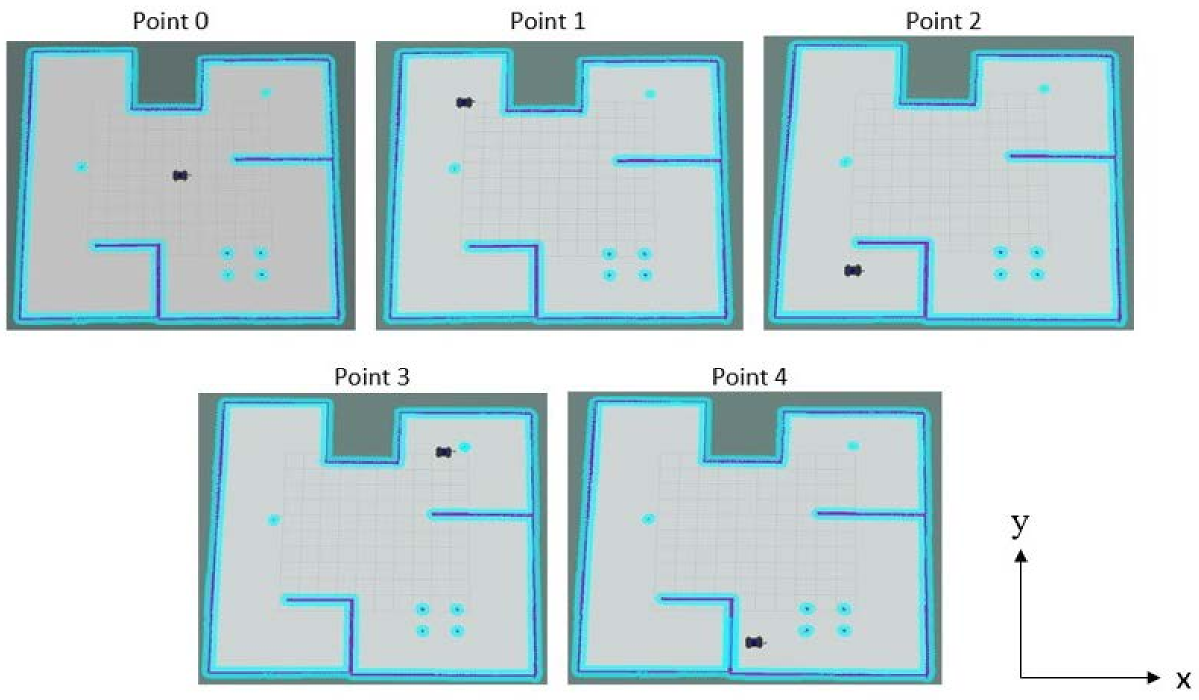

Figure 5. Target points from 0 to 4

3.5. ROS navigation stack

The ROS Navigation Stack (as shown in Figure 6) is a set of software tools designed for self-localization, path planning, obstacle avoidance and motion control [32]. It processes all the sensor information and generates a safe velocity command for differential drive plugin 'diff_drive_controller' in Gazebo. The generated velocity commands are published to the 'cmd_vel' topic. For localization of robots on map during navigation, AMCL (Adaptive Monte Carlo Localization) and 'move_base' packages are utilized.

AMCL is a probabilistic localization system, used for robots operating in 2D. It employs the adaptive Monte Carlo localization method, which utilizes a particle filter to monitor the robot's position relative to a known map [33]. AMCL estimates the robot's pose by means of the two steps: filtering and resampling. In AMCL, lasers, laser scans, and transformations are inputs for creating pose estimates outputs. By using AMCL, the robot will be able to understand precisely where it is located within the map that was made.

The 'move_base' package in ROS works as an interface, integrating the path planning and obstacle-avoiding processes, enabling a robot to move from a starting point to the desired one while navigating across obstacles in the environment. According to [34], it integrates a global and a local path planners and obstacle avoiders into a single package. The global path planner inside the 'move_base' can also use various algorithms such as A*, Dijkstra and RRT. For local path planning, TEB (Time Elastic Band) and DWA (Dynamic Window Approach) planners can be used. Some of the most important parameters of the 'move_base' package are presented in the following configuration file and may be altered to change the behavior of the robot.

- ‘base_global_planner’ and ‘base_local_planner’: These parameters specify the types of global and local path planners to be used in navigation process.
- ‘global_costmap’ and ‘local_costmap’: These are the parameters that define the recovery behavior for 'move_base'. Recovery behaviors instruct the robot to try and reroute its path when faced with obstacles.
- ‘recovery_behaviors’: This parameter defines the recovery strategies for 'move_base' to allow the robot to redirect itself along specified recovery behaviors when it faces obstacles [34].
- ‘goal_tolerance’: This parameter defines the acceptable distance to the goal position.
- ‘goal_reached_threshold’: This parameter defines the distance at which the robot is considered to have reached the target.

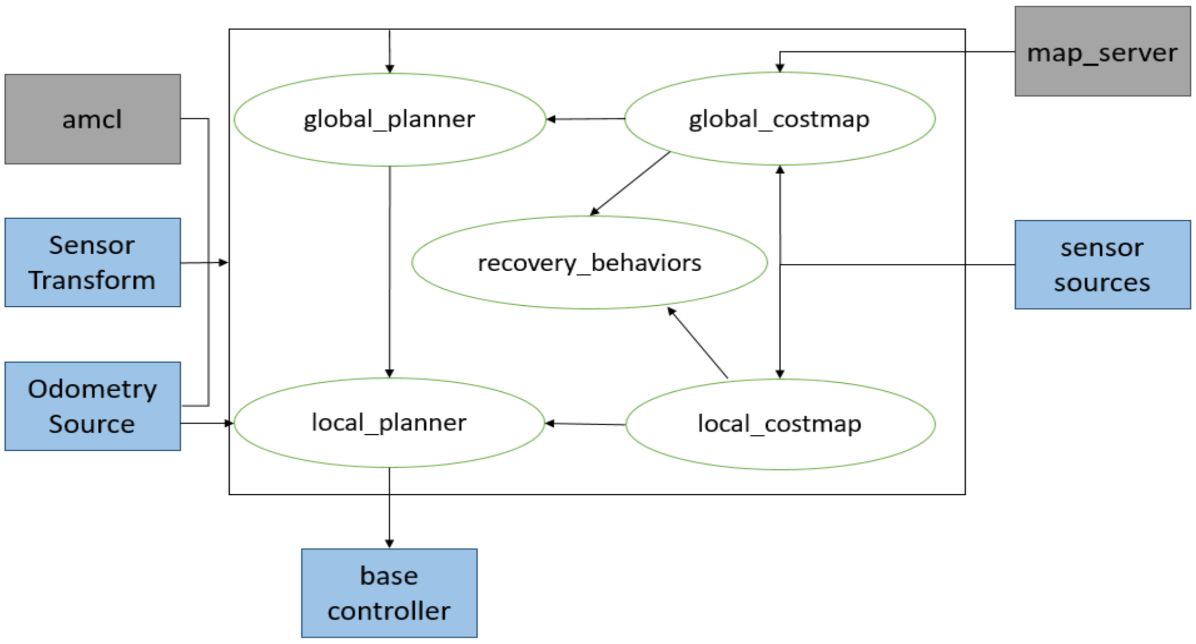

Figure 6. The architecture of ROS navigation stack

The ROS ‘rqt_graph’ command is used to visualize the communication structure, which shows how nodes interact with each other through published or subscribed topics or services. In these graphs, each node is represented as a box, and the topics are represented as arrows connecting the nodes. The ‘rqt_graph’ schematic of this study is given in Figure 7.

The important components in the ‘rqt_graph’ schematic can be explained as below:

- /map_server: This node is a part of the map_server package, which loads and publishes static 2D occupancy grid maps to the /map topic.
- /map: This topic publishes the static occupancy grid map of the environment.
- /cmd_vel: This topic publishes linear and angular velocity commands to control the robot's movements.
- /odom: This topic provides odometry data, which contains the robot's position and velocity.
- /scan: This topic publishes laser scan data obtained from the LiDAR sensor for obstacle detection, which is crucial for the mapping process.

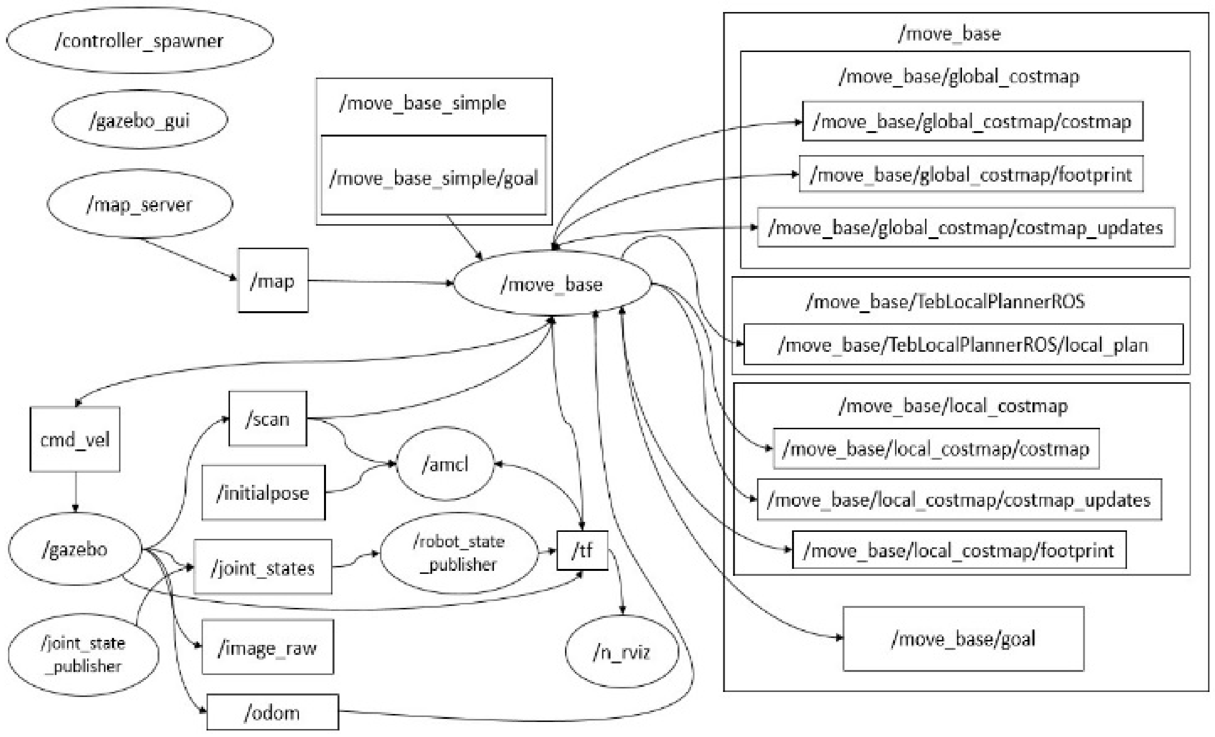
/controller_spawner
/gazebo_gui
/map_server
/move_base_simple
/move_base_simple/goal
/map
/move_base
/move_base
/move_base/global_costmap
/move_base/global_costmap/costmap
/move_base/global_costmap/footprint
/move_base/global_costmap/costmap_updates
/move_base/TebLocalPlannerROS
/move_base/TebLocalPlannerROS/local_plan
/move_base/local_costmap
/move_base/local_costmap/costmap
/move_base/local_costmap/costmap_updates
/move_base/local_costmap/footprint
/move_base/goal
cmd_vel
/scan
/initialpose
/amcl
/gazebo
/joint_states
/robot_state_publisher
/tf
/joint_state_publisher
/image_raw
/n_rviz
/odom

Figure 7. ROS graph of the simulation setup

IV. RESULTS

The performance results of A*, Dijkstra and RRT path planning algorithms, which were tested on a LiDAR-based and mecanum wheeled mobile robot in a specially designed Gazebo environment within the ROS framework, were analyzed. Manhattan distance heuristics and Euclidean distance heuristics were used separately to analyze the A* algorithm.

For an example of the path plans generated by the algorithms, the robot's trajectories on Route 3-4 are shown in Figures 8-11.

Figure 8 shows the path plan, depicted in black line, generated by A* (Euclidean) algorithm on Route 3-4.

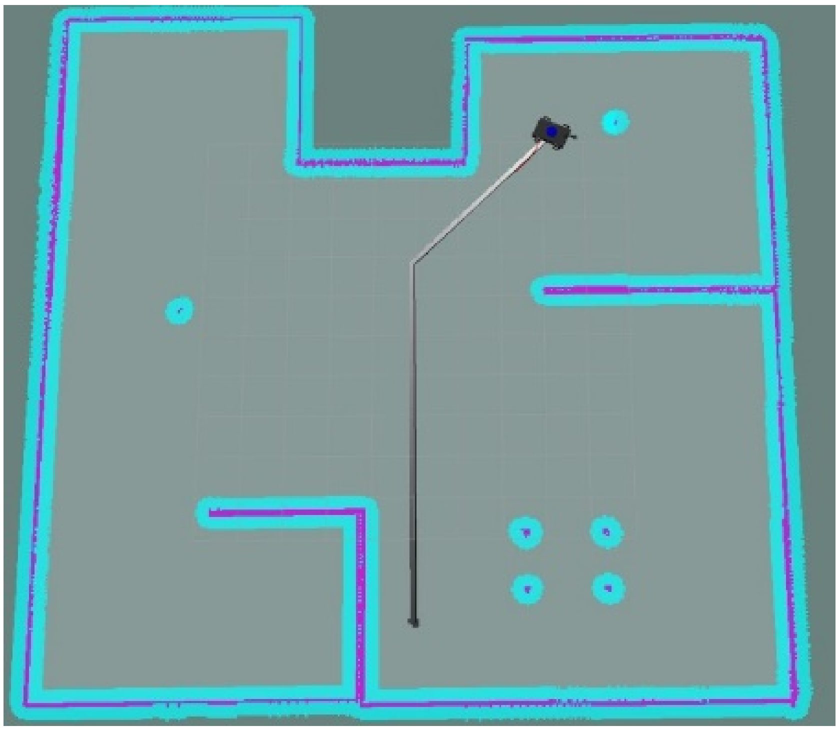

Figure 8. A* (Euclidean) algorithm’s path plan on Route 3-4

Figure 9 shows the path plan, depicted in black line, generated by A* (Manhattan) algorithm on Route 3-4.

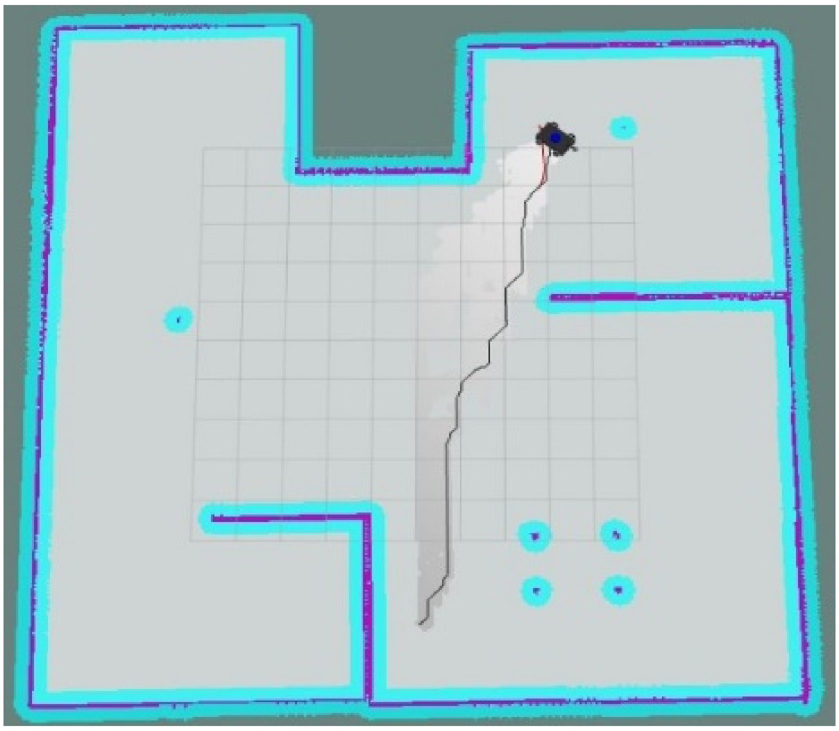

Figure 9. A* (Manhattan) algorithm's path plan on Route 3-4

Figure 10 shows the path plan, depicted in black line, generated by Dijkstra’s algorithm on Route 3-4.

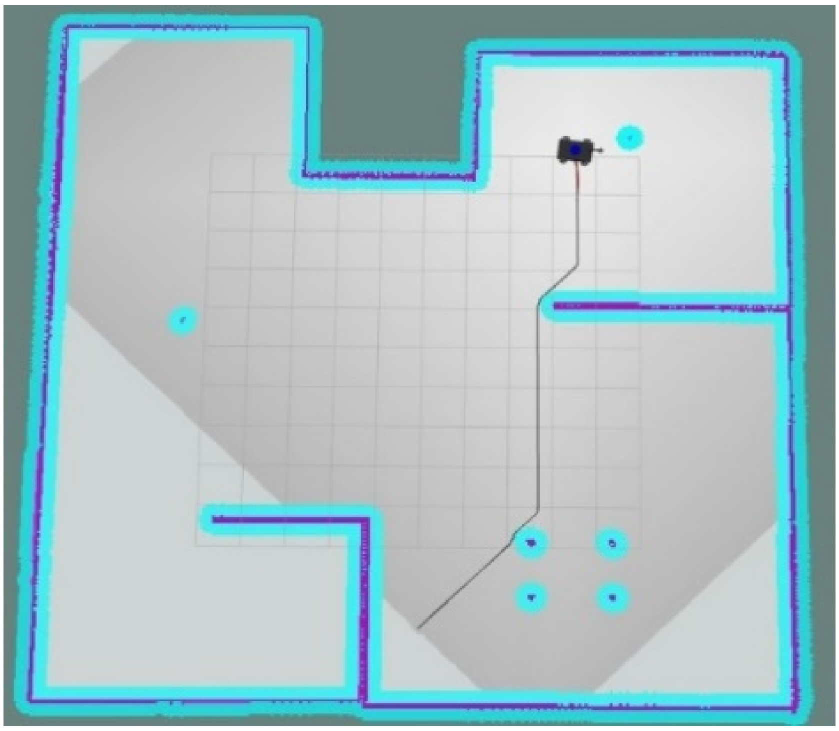

Figure 10. Dijkstra's algorithm's path plan on Route 3-4

Figure 11 shows the path plan, depicted in black line, generated by RRT algorithm on Route 3-4. In addition, the green, tree-like lines show the random trees produced by the RRT algorithm, since it is a sampling-based algorithm.

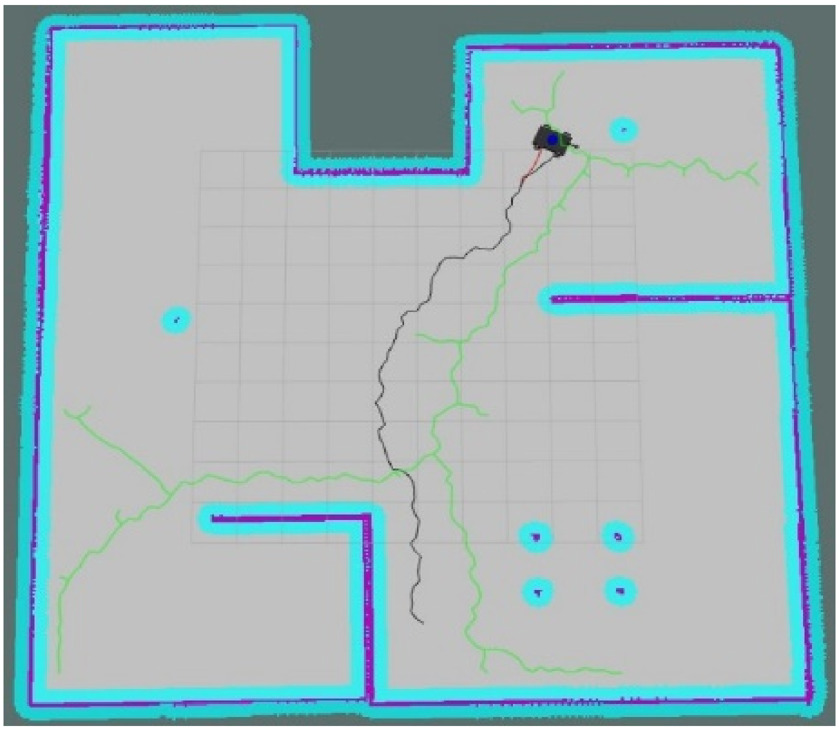

Figure 11. RRT algorithm's path plan on Route 3-4

Table 4 gives the positions in meters of the five target points in Cartesian space X-Y-Z, as shown in Figure 5. It should be noted that the mobile robot moves only in X-Y surface coordinates, and the Z coordinate is always fixed at 0.09 m.

Table 4. Coordinates of the target points in Cartesian space

Point	**X-Coordinate** *(m)*	**Y-Coordinate** *(m)*	**Z-Coordinate** *(m)*
0	0	0	0.09
1	-5	5	0.09
2	-5	-6	0.09
3	3.5	5	0.09
4	0	-7	0.09

In Tables 4-8, Route i to $i+1$ (i = 0, 1, 2, 3, 4) indicates that the robot's trajectory starts from point i and finishes at point $i+1$. Table 5 shows the performance analysis of the algorithms based on the travel time parameter.

Table 5. Travel times along routes

Route	**A* (Euclidean)** *(s)*	**A* (Manhattan)** *(s)*	**Dijkstra** *(s)*	**RRT** *(s)*
0-1	25.2	28.8	23.4	39.9
1-2	40.3	43.9	66	52.6
2-3	39.4	39.3	39.7	100
3-4	44.3	35.7	77	46.4
4-0	22.2	22.1	23.9	75
Total	171.4	169.8	230	313.9

From the observation of Table 3, in terms of total travel times, A* algorithms (both Euclidean and Manhattan) are faster than Dijkstra and RRT. A*(Manhattan) is the most efficient overall with the shortest duration 169.8 seconds, closely followed by A*(Euclidean) at 171.4 seconds. The duration of Dijkstra and RRT algorithms are significantly longer, with Dijkstra taking 230 seconds in total (approximately 35% longer than A* Manhattan and 34% longer than A* Euclidean) and RRT taking 313.9 seconds in total (approximately 85% longer than A* Manhattan, 83% longer than A* Euclidean and 36% longer than Dijkstra).

Table 6 shows the performance analysis of the algorithms based on the total traveled distance to reach the target points. From the observation of Table 6, in terms of total traveled distances, A* algorithms (both Euclidean and Manhattan) surpass Dijkstra and RRT by creating shorter paths towards the goals. The created path length with A* (Manhattan) is the most effective with the shortest path length 62.765 meters, closely followed by A* (Euclidean) with 63.637 meters. The created path lengths with Dijkstra and RRT are significantly longer, with Dijkstra taking 70.855 meters in total (approximately 13% longer than A* Manhattan and approximately 11% longer than A* Euclidean) and RRT taking 74.227 meters in total (approximately 18% longer than A* Manhattan, 17% longer than A* Euclidean and 5% longer than Dijkstra).

Table 6. Path lengths along routes

Route	**A* (Euclidean)** *(m)*	**A* (Manhattan)** *(m)*	**Dijkstra** *(m)*	**RRT** *(m)*
0-1	9.157	9.374	8.572	8.998
1-2	13.926	14.512	17.601	15.703
2-3	16.462	16.296	16.279	22.215
3-4	15.971	14.423	20.144	16.332
4-0	8.121	8.160	8.259	10.979
Total	63.637	62.765	70.855	74.227

Table 7 shows the performance analysis of the algorithms based on the number of messages sent to ‘cmd_vel’ topic during navigation along routes. From the observation of Table 7, in terms of the total number of messages sent to ‘cmd_vel’ topic, A* algorithm (both Euclidean and Manhattan variants) surpasses Dijkstra and RRT by sending fewer messages. A* (Manhattan) is the most efficient overall with the fewest number of messages 1699, followed by A* (Euclidean) with 1717. In Dijkstra and RRT algorithms, the sent messages are more, with Dijkstra sending 2317 (approximately 36% more than A* Manhattan and 35% more than A* Euclidean) and RRT sending 2340 messages in total (approximately 38% more than A* Manhattan, 36% more than A* Euclidean and 1% more than Dijkstra).

Table 7. Number of messages sent to ‘cmd_vel’ topic

Route	**A* (Euclidean)**	**A* (Manhattan)**	**Dijkstra**	**RRT**
0-1	252	289	234	272
1-2	403	439	666	474
2-3	395	393	398	729
3-4	444	357	779	427
4-0	223	221	240	438
Total	1717	1699	2317	2340

Table 8 shows the performance analysis of algorithms based on the CPU utilization averages during the navigation process along routes. From the observation of Table 8, in terms of average CPU utilization during navigation process, RRT outperforms both A* variants and Dijkstra algorithm by having lower average CPU usage. RRT is the most effective overall with average CPU usage of 28.69%, followed by A* variants with 30.93% (Euclidean) and 31.89% (Manhattan). Dijkstra algorithm’s average CPU usage is dramatically higher with having average of 47.94% (approximately 67% higher than RRT, 55% higher than A* Euclidean and 50% higher than A* Manhattan).

Table 8. Average CPU utilization during navigation

Route	**A* (Euclidean)** *(%)*	**A* (Manhattan)** *(%)*	**Dijkstra** *(%)*	**RRT** *(%)*
0-1	46.19	51.14	48.10	30.41
1-2	26.99	26.74	47.95	28.09
2-3	26.77	27.08	47.65	27.65
3-4	27.35	27.37	48.21	28.67
4-0	27.33	27.14	47.80	28.63
Average	30.93	31.89	47.94	28.69

V. DISCUSSION

In this paper, for optimal efficiency in terms of the travel times along routes and generated path lengths, A*(Manhattan) is shown to be the best choice with the shortest duration (169.8 seconds) and shortest total path length (62.765 m) according to Tables 5-6, followed by A* (Euclidean). This observation could be attributed to A* algorithm’s heuristic-based search process and the accuracy of the heuristics of its variants in this study. In general, A* (Manhattan) is generally faster in grid-based environments where only vertical and horizontal movements are allowed. Dijkstra’s algorithm is shown to be significantly slower and generates longer paths than both A* variants, likely because it does not use any heuristics and explores all possible paths, which may have led to longer travel times and longer distances. On the other hand, RRT algorithm is typically more suitable for complex and dynamic environments where grid-based search algorithms such as A* and Dijkstra may struggle. However, in this study, since the environment is not very complicated and there are no dynamic objects, RRT is shown to be much slower and generates longer paths compared to all other algorithms. This indicates that it can be inefficient for simple path finding tasks.

For optimal efficiency in terms of the number of messages sent to ‘cmd_vel’ topic, A* (Manhattan) is the best choice by sending the fewest number of messages (1699) to the ‘cmd_vel’ topic during navigation according to Table 7, followed by A* (Euclidean). Dijkstra algorithm, on the other hand, while guaranteeing an optimal path, results in more computational cost with a significantly higher number of messages, which is making it less efficient in terms of communication load. Also, the RRT algorithm tends to have a high message count; this could mean that more frequent adjustments are needed in navigation processes due to its approach to path-planning in dynamic or complex spaces.

For the best average CPU utilization along the navigation process, based on Table 8, RRT algorithm shows the minimum average CPU utilization. It is then followed by both variants of A*, Euclidean and Manhattan. This will indicate that among the rest of the comparative algorithms, RRT algorithm has more computational efficiency in terms of CPU utilization than both variants of A* and Dijkstra's algorithm. This can be explained by the fact that RRT does search without the use of a grid, hence avoiding some of the heavy computation common in grid-based path-finding algorithms like A* and Dijkstra. The highest average CPU utilization is contributed by Dijkstra, probably because it is an exhaustive search algorithm and does not make use of heuristics. A* Euclidean and Manhattan variants represent a similar CPU utilization pattern but with slight variation. Its CPU utilizations are higher compared to RRT and lower than Dijkstra, since it uses a heuristic function that guides its search, therefore, making A* much more computationally efficient in the resource utilizations compared to Dijkstra.

Although our findings provide an overview of the efficiency of various pathfinding algorithms individually, there are also studies that can be conducted to reveal how the algorithms can be optimized for real-world use. For instance, blending hybrid methods that utilize the best traits of grid-based and non-grid-based algorithms can enhance efficiency as well as flexibility in detailed, dynamic environments. Moreover, the application of novel technologies, such as machine learning to dynamically change heuristics or multi-agent systems to discover paths in a collaborative way, could overcome the said weaknesses of algorithms like RRT in less complex environments. These new approaches might offer more flexible and scalable algorithms, thereby rendering them more suitable to realistic robotic pathfinding issues where real-time adaptability, computational efficiency, and communication cost are imperative concerns.

VI. CONCLUSIONS

In this paper, performance analysis for three path planning algorithms, namely A*, Dijkstra, and RRT, was conducted for a LiDAR-based mecanum wheeled mobile robot in the ROS-based Gazebo simulation environment. By evaluating these algorithms with respect to the performance parameters such as path length, travel time, number of velocity commands sent and CPU utilization, the study provides significant results regarding their performance for specific robotic navigation tasks. In conclusion, the comparison of A*, Dijkstra, and RRT algorithms in a simulated ROS Gazebo environment provided essential insights into their performances on respective basis for path planning in mecanum wheeled mobile robot. A* performed with better travel time, optimized path length, and velocity command rate, making it the optimal choice to minimize travel time and command processing optimization. However, RRT was more efficient compared to A* and Dijkstra in terms of CPU, suggesting that it might be more optimally utilized where there are stricter computational constraints. Dijkstra, despite offering a consistent solution, had relatively lower performance across all areas compared to the other two algorithms. In general, the study highlights compromises between different path planning methods and the necessity of selecting the appropriate algorithm based on some performance requirements and system constraints. The experiment was only done on simulated data, a limitation in that the results might not be entirely true under real conditions. Still, it is anticipated that the performance outcomes of these algorithms will be in a close proportionate ratio with the ones obtained in real experiments, even though they might differ because of the discrepancy between simulated and real environments. Future work can extend this analysis to include physical mobile robots, real-world environments, dynamic obstacles, and hybrid algorithm approaches to further improve the adaptability and efficiency of robotic navigation systems.

ACKNOWLEDGMENT

This project is supported by TÜBİTAK with project No. 1139B412302608, 2023. The authors thank TÜBİTAK for their supports.

REFERENCES

1. Levine S, Shah D (2022) Learning robotic navigation from experience: principles, methods and recent results. Phil Trans R Soc B 378:20210447. https://doi.org/10.1098/rstb.2021.0447
2. Sahoo SK, Choudhury BB (2023) A review of methodologies for path planning and optimization of mobile robots. Journal of process management and new technologies 11(1-2):122-140. https://doi.org/10.5937/jpmnt11-45039
3. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische Mathematik 1:269–271. https://doi.org/10.1007/BF01386390
4. Hart PE, Nilsson NJ, Raphael B (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4(2):100–7. https://doi: 10.1109/TSSC.1968.300136
5. LaValle SM (1998) Rapidly-exploring random trees: A new tool for path planning. Technical Report Computer Science Department, Iowa State University.
6. Al-Ansarry, Suhaib, Al-Darraji S (2021) Hybrid RRT-A*: An Improved Path Planning Method for an Autonomous Mobile Robots. Iraqi Journal for Electrical & Electronic Engineering 17(1):1-10. https://doi.org/10.37917/ijeee.17.1.13
7. Dirik M, Kocamaz F (2020) Rrt-dijkstra: An improved path planning algorithm for mobile robots. Journal of Soft Computing and Artificial Intelligence 1(2):69-77.
8. Chen R, Hu J, Xu W (2022) An RRT-Dijkstra-Based Path Planning Strategy for Autonomous Vehicles. Applied Sciences 12(23):11982. https://doi.org/10.3390/app122311982
9. Zammit C, Van Kampen EJ (2022) Comparison between A* and RRT algorithms for 3D UAV path planning. Unmanned Systems 10(02):129-146. https://doi.org/10.1142/S2301385022500078
10. Cai Q (2024) A Comparison Between A* and RRT Algorithm in Path Planning for Mobile Robot. Highlights in Science, Engineering and Technology 97:282-287. https://doi.org/10.54097/2stv5y97
11. Aydemir H, Tekerek M, Gök M (2021) Examining of the Effect of Geometric Objects on SLAM Performance Using ROS and Gazebo. El-Cezeri 8(3):1441-1454. https://doi.org/10.31202/ecjse.943364
12. Zhou L, Zhu C, Su X (2022) Slam algorithm and navigation for indoor mobile robot based on ros. IEEE International Conference on Software Engineering and Artificial Intelligence pp. 230-236. Xiamen, China, June 10-12. https://doi.org/10.1109/SEAI55746.2022.9832313

13. Karur K, Sharma N, Dharmatti C, Siegel JE (2021) A survey of path planning algorithms for mobile robots. Vehicles 3(3):448-468. https://doi.org/10.3390/vehicles3030027
14. Gök M, Akçam ÖŞ, Tekerek M (2022) Performance Analysis of Search Algorithms for Path Planning, KSU Journal of Engineering Sciences 26(2):379-394. https://doi.org/10.17780/ksujes.1171461
15. Yao J (2023) Path planning algorithm of indoor mobile robot based on ROS system. IEEE International Conference on Image Processing and Computer Applications pp. 523-529. Changchun, China, August 11-13. https://doi.org/10.1109/ICIPCA59209.2023.10257966
16. Tamang MT, Maheriya D, Sharif MS, Sutharssan T (2024) Autonomous Navigation for TurtleBot3 Robots in Gazebo Simulation Environment. International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies pp. 568-574. Sakhir, Bahrain, November 17-19. https://doi.org/10.1109/3ict64318.2024.10824668
17. Megalingam RK, Rajendraprasad A, Manoharan SK (2020) Comparison of planned path and travelled path using ROS navigation stack. International Conference for Emerging Technology pp. 1-6. Belgaum, India, June 5-7. https://doi.org/10.1109/INCET49848.2020.9154132
18. Breiling B, Dieber B, Schartner P (2017) Secure communication for the robot operating system. IEEE international systems conference pp. 1-6. Montreal, QC, Canada, April 24-27. https://doi.org/10.1109/SYSCON.2017.7934755
19. Quigley M, Conley K, Gerkey B et al (2009) ROS: an open-source Robot Operating System. 3-3.2, p. 5. ICRA workshop on open source software, Kobe, Japan, May 12-17.
20. Rivera ZB, De Simone MC, Guida D (2019). Unmanned ground vehicle modelling in Gazebo/ROS-based environments. Machines, 7(2), 42. https://doi.org/10.3390/machines7020042
21. Qiao J, Guo J, Li Y (2024) Simultaneous localization and mapping (SLAM)-based robot localization and navigation algorithm. Appl Water Sci 14, 151. https://doi.org/10.1007/s13201-024-02183-6
22. Duchoň F, Babinec A, Kajan M, Beňo P, Florek M, Fico T, Jurišica L (2014) Path planning with modified a star algorithm for a mobile robot. Procedia engineering, 96:59-69. https://doi.org/10.1016/j.proeng.2014.12.098
23. Zhang L, Li Y (2021) Mobile robot path planning algorithm based on improved a star. Journal of Physics: Conference Series 1848-1, p.012013, Sanya, China, Jan. 29-31. https://doi.org/10.1088/1742-6596/1848/1/012013
24. Nayak S, Bhat, M, Reddy NS, Rao BA (2022) Study of distance metrics on k-nearest neighbor algorithm for star categorization. Journal of Physics: Conference Series v.2161-1, pp.012004, Manipal, India, Oct. 28-30. https://doi.org/10.1088/1742-6596/2161/1/012004
25. Bernardo GT, Vogás LM, Rodrigues SD, Lopes TG et al (2022) A-star based algorithm applied to target search and rescue by a uav swarm. Latin American Robotics Symposium, Brazilian Symposium on Robotics, and Workshop on Robotics in Education, pp.49-54. São Bernardo do Campo, Brazil, Oct. 18-21. https://doi.org/10.1109/LARS/SBR/WRE56824.2022.9996054
26. Javaid A (2013) Understanding Dijkstra's algorithm. SSRN 2340905. https://doi.org/10.2139/ssrn.2340905
27. Candra A, Budiman MA, Hartanto K (2020) Dijkstra's and a-star in finding the shortest path: A tutorial. International Conference on Data Science, Artificial Intelligence, and Business Analytics pp.28-32, Medan, Indonesia, July 16-17. https://doi.org/10.1109/DATABIA50434.2020.9190342
28. Rachmawati D, Gustin L (2020) Analysis of Dijkstra's algorithm and A* algorithm in shortest path problem. Journal of Physics: Conference Series, Medan 1566-1, pp.012061, Indonesia, Nov. 26-27, 2019. https://doi.org/10.1088/1742-6596/1566/1/012061
29. Karaman S, Walter MR, Perez A, Frazzoli E, Teller S (2011) Anytime motion planning using the RRT. IEEE international conference on robotics and automation pp.1478-1483, Shanghai, China, May 9-13. https://doi.org/10.1109/ICRA.2011.5980479
30. Duchoň F, Hažík J, Rodina J et al (2019) Verification of slam methods implemented in ros. Journal of Multidisciplinary Engineering Science and Technology, 6(9):2458-9403. https://doi.org/10.22223/tr.2016-1/2011
31. Rojas-Fernández M, Mújica-Vargas D, Matuz-Cruz M, López-Borreguero D (2018) Performance comparison of 2D SLAM techniques available in ROS using a differential drive robot. International Conference on Electronics, Communications and Computers pp.50-58, Cholula, Mexico, Feb. 21-23. https://doi.org/10.1109/CONIELECOMP.2018.8327175
32. Walenta R, Schellekens T, Ferrein A, Schiffer S (2017) A decentralised system approach for controlling AGVs with ROS. IEEE AFRICON pp.1436-1441, Cape Town, South Africa, Sept. 18-20. https://doi.org/10.1109/AFRCON.2017.8095693
33. Thale SP, Prabhu MM, Thakur PV, Kadam P (2020) ROS based SLAM implementation for Autonomous navigation using Turtlebot. ITM Web of conferences v.32 p.01011, Navi Mumbai, India, June 27-28. https://doi.org/10.1051/itmconf/20203201011

34. Chen SP, Peng CY, Huang GS, Lai CC, Chen CC, Yen MH (2023). Comparison of 2D and 3D LiDARs Trajectories and AMCL Positioning in ROS-Based move_base Navigation. IEEE International Conference on Omni-layer Intelligent Systems pp. 1-6, Berlin, Germany, July 23-25. https://doi.org/10.1109/COINS57856.2023.10189271