Comparison of Path Finding Algorithms for Autonomous Navigation of Mobile Robots

Lasse Harju
Department of Energy and Mechanical Engineering
Aalto University
Espoo, Finland
lasse.harju@aalto.fi

Muhammad Anis
Department of Energy and Mechanical Engineering
Aalto University
Espoo, Finland
muhammad.1.anis@aalto.fi

Elais Salo
Department of Energy and Mechanical Engineering
Aalto University
Espoo, Finland
elias.salo@aalto.fi

Hammad Hussain
Department of Energy and Mechanical Engineering)
Aalto University
Espoo, Finland
hammad.hussain@aalto.fi

Pejman Habibiroudkenar
Department of Energy and Mechanical Engineering
Aalto University
Espoo, Finland
pejman.habibiroudkenar@aalto.fi

Petri Kuosmanen
Department of Energy and Mechanical Engineering
Aalto University
Espoo, Finland
petri.kuosmanen@aalto.fi

Panu Kiviluoma
Department of Energy and Mechanical Engineering
Aalto University
Espoo, Finland
panu.kiviluoma@aalto.fi

***Abstract*—Autonomous navigation of mobile robots relies on various path planning algorithms to determine a collision-free trajectory from a start to a goal position. The effectiveness of these algorithms is influenced by several factors, including computational efficiency, latency, power consumption, and reliability. The selection of an appropriate algorithm is application-dependent, as different scenarios prioritize these factors differently. This paper presents a comparative analysis of three widely used path planning algorithms A* (A-star), Dijkstra's algorithm, and Rapidly-exploring Random Tree (RRT) in an indoor environment. The performance of these algorithms is evaluated through both simulation and real-world experiments, with key metrics including path length, computational time, and the number of nodes explored. Based on the results, A* provides the optimal balance between path efficiency and computational power. Dijkstra's algorithm may generate more optimal paths in some cases but at the cost of significantly higher computational time. RRT, while requiring fewer nodes, suffers from suboptimal path quality and higher computational times, making it less suitable for the tested application.(Git https://version.aalto.fi/gitlab/harjul4/mirarobot)**

Keywords—ROS, A*, Dijkstra, Rapidly-exploring Random Tree, Mobile robots, SLAM

I. Introduction

As autonomous mobile robots become more popular, the need for advanced and efficient path planning methods continues to rise. Autonomous navigation consists of three key components: localization, environment mapping, and path planning [1]. To navigate successfully, a robot must first understand its environment, determine its position within it using multiple sensors such as LiDar and cameras, and then plan a collision-free path to move safely and efficiently from the starting point to the destination. Path planning is a fundamental challenge in autonomous navigation. It can be categorized into three main aspects: global navigation, local navigation, and individual behavior execution [2][6]. Global navigation relies on pre-existing knowledge, such as preloaded maps, to determine the robot's movement and path. In contrast, local navigation handles real-time uncertainties encountered during movement. For execution of individual behavior, robots need to carefully consider their surroundings, follow high-level instructions, and use path planning results to decide their next moves. The combination of global and local navigation allows the robot to adapt and execute its motion effectively in various scenarios.

For the autonomous navigation of mobile robots, it is essential that the robot possesses sufficient intelligence to determine the shortest and obstacle-free path. Path planning plays a crucial role in achieving this objective and is accomplished through various algorithms such as A* and Dijkstra. These algorithms have been widely implemented for years; however, ongoing research continues to enhance their efficiency, adaptability, and real-time performance. Despite significant advancements, there is still considerable room for improvement, particularly in handling dynamic environments and integrating with advanced sensor technologies. The implementation of these algorithms varies depending on the specific use case, environmental conditions, and the computational capabilities of the robot.

At CERN, radiation survey robots are essential tools used for mapping and monitoring radiation levels within experimental areas. Currently, these robots are manually controlled, requiring significant human intervention and careful operation, which can be time consuming and challenging. To address these limitations, our goal is to develop an autonomous navigation system for the Mira robot. By automating navigation, we aim to significantly reduce time spent on manual control. To validate the system's effectiveness, we conduct testing of the navigation system using virtual simulation and Dbot at Aalto University.

II. Methods

A. Software Framework

We used the Robot Operating System (ROS) as the core software framework to integrate the hardware and

algorithms, due to our robots already working through it. While creating the system we used the following ROS packages:

- **rviz**: Visualization tool for monitoring real time robot position and sensor data.
- **Rclpy**: Python API for writing ROS nodes.
- **Nav msgs:** Package that provides message types for robot navigation.
- **Gmapping:** A SLAM package that uses lidar data to create a 2D occupancy grid.

All path planning algorithms were implemented in Python. The algorithms were first tested in a simulated using Gazebo environment, where various scenarios, such as narrow corridors were modeled. To evaluate the performance of the path planning algorithms, we conducted a series of experiments in a simulated environment to minimize variables such as path length, computation time and nodes explored to make our algorithms more accurate. The experiments tested the robot's ability to navigate from a fixed start point to various goal positions while avoiding obstacles. After thoroughly testing the algorithm in a simulated a simulated environment we used Dbot to test the algorithms in a real-world setting. Dbot is equipped with a lidar sensor.

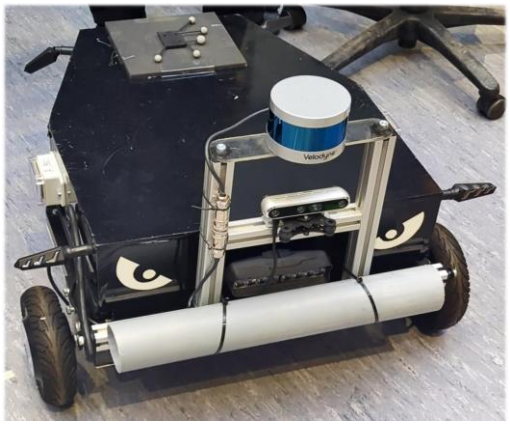

Figure 1: Dbot at Aalto

B. Mapping Technique

We used Simultaneous Localization and Mapping (SLAM) to update the robot's onboard maps using data from the LiDAR sensor. SLAM is a technique used by robots to build a map of an unknown environment while simultaneously keeping track of their own position within that map. SLAM is crucial for enabling the robot to build and maintain an accurate map of its environment. To implement SLAM, we utilized the Gmapping algorithm due to its compatibility with ROS. Gmapping helped us create 2D occupancy grid maps by integrating laser scan data and wheel odometry information. This approach allowed the robot to construct real-time occupancy maps to represent the surrounding environment with grids that indicate free space, obstacles, and unknown areas. The real-time mapping capability ensured that the robot's navigation algorithms always had up to date maps, allowing them to adjust the robot's path as needed.

C. Path Finding Techniques

We focused on the implementation and comparison widely used path palling algorithms: A* (Astar), Dijkstra and RRT (Rapidly Exploring Random Tree). These algorithms were selected due to their proven effectiveness in graph-based search methods and their ability to compute optimal paths.

Dijkstra Algorithm

Dijkstra's algorithm is a classic path-finding algorithm that guarantees the shortest path from a start node to all other nodes in a weighted graph. The core principle is to explore nodes in order of their distance from the start node, ensuring that once a node's shortest distance is determined, it will not change. Dijkstra's primary limitation lies in its inefficiency for large graphs, as it explores all directions equally, regardless of the target node's location. This lack of goal orientation can make it slower in our path planning system. [3]

```
function Dijkstra(startNode, goalNode, distanceFunction)
    nodesToExplore = {startNode}
    // Set of nodes to be evaluated
    bestPathMap = {}
     // Maps each node to its best previous node
    costFromStart = {startNode: 0}
    // Tracks the shortest distance from startNode

    while nodesToExplore is not empty:
        currentNode = node in nodesToExplore with lowest costFromStart value

        if currentNode == goalNode:
            return reconstruct_path(bestPathMap, currentNode) // Path found!

        nodesToExplore.remove(currentNode)

        for each neighborNode of currentNode:
            pathCost = costFromStart[currentNode] + distanceFunction(currentNode, neighborNode)

            if pathCost < costFromStart.get(neighborNode, Infinity):
                bestPathMap[neighborNode] = currentNode  // Update best path
                costFromStart[neighborNode] = pathCost

                if neighborNode not in nodesToExplore:
                    nodesToExplore.add(neighborNode)

    return failure  // No path found

function reconstruct_path(bestPathMap, goalNode)
    shortestPath = [goalNode]
    while goalNode in bestPathMap:
        goalNode = bestPathMap[goalNode]
        shortestPath.prepend(goalNode)  // Add previous node to the path
    return shortestPath
```

Figure 2: Pseudocode Dijkstra

A Algorithm*

A* in an informed search algorithm that extends Dijkstra's algorithm by incorporating a heuristic function to guide the node search towards the goal, therefore reducing unnecessary exploration. A*'s time complexity mirrors Dijkstra's in the worst case, but it often outperforms Dijkstra's, especially in larger graphs, due to its focused search. [4]

```
function A_Star(startNode, goalNode, heuristicFunction)
    nodesToExplore = {startNode}
    // Set of nodes to be evaluated
    bestPathMap = {}
    // Maps each node to its best previous node
    costFromStart = {startNode: 0}
     // Tracks the shortest distance from startNode
    estimatedTotalCost = {startNode: heuristicFunction(startNode)}
     // Estimated total cost (g + h)

    while nodesToExplore is not empty:
        currentNode = node in nodesToExplore with lowest estimatedTotalCost value

        if currentNode == goalNode:
            return reconstruct_path(bestPathMap, currentNode)
             // Path found!

        nodesToExplore.remove(currentNode)


        for each neighborNode of currentNode:
            pathCost = costFromStart[currentNode] + distance(currentNode, neighborNode)


            if pathCost < costFromStart.get(neighborNode, Infinity):
                bestPathMap[neighborNode] = currentNode
                // Update best path
                costFromStart[neighborNode] = pathCost
                estimatedTotalCost[neighborNode] = pathCost + heuristicFunction(neighborNode)


                if neighborNode not in nodesToExplore:
                    nodesToExplore
```

Figure 3: Pseudocode A-Star

RRT Algorithm

RRT algorithm is a widely used path planning method in robotics, especially for navigating complex and high-dimensional environments. It is particularly effective for motion planning tasks where the robot needs to find a path from a starting point to a goal while avoiding obstacles. RRT builds a tree incrementally by starting from the robot's initial position and expanding branches towards randomly sampled points in the configuration space. The core idea is to balance exploration of the space while steadily growing the tree in the direction of the goal. [5][7]

```
function RRT(startNode, goalNode, maxIterations, stepSize, obstacleChecker)
    tree = {startNode}  // Initialize tree with the start node
    pathFound = false

    for i = 1 to maxIterations:
        randomNode = generateRandomNode()

        nearestNode = findNearestNode(tree, randomNode)
        // Find the nearest node in the tree
        newNode = extendTowards(nearestNode, randomNode, stepSize)
         // Generate new node towards randomNode

        if not obstacleChecker(newNode):
            // Check if the new node is valid (no collisions)
            continue  // Skip this iteration if the new node is not valid
        tree.add(newNode)  // Add the new valid node to the tree
        if distance(newNode, goalNode) < stepSize:
            // If the goal is close enough
            tree.add(goalNode)  // Add goal node to the tree
            pathFound = true
            break  // Exit the loop once the path is found

    if pathFound:
        return reconstruct_path(tree, goalNode)  // Reconstruct and return the path from start to goal
    else:
        return failure  // No path found

function reconstruct_path(tree, goalNode)
    path = [goalNode]
    currentNode = goalNode
    while currentNode in tree:
        currentNode = findParentNode(tree, currentNode)  // Find the parent node of currentNode
        path.prepend(currentNode)  // Add the parent node to the path
    return path
```

Figure 4: Pseudocode RRT

III. Results & Analysis

The test environment consisted of three maps of CERN facility each consisting of 629x810 grids, where movement was permitted in eight directions: the four cardinal directions (up, down, left, right) and the four diagonal directions. The cost of moving to any neighboring node was set to 1. Furthermore, over 50% of the grid nodes were occupied, presenting a challenging scenario for pathfinding.

A comprehensive evaluation of each algorithm (A* (A-star), Dijkstra, and Rapidly exploring Random Tree (RRT)) was conducted to systematically assess their performance in this constrained environment. The Euclidean distance was employed as a heuristic function, making it well-suited for navigation on square grids.

Each algorithm was tested over 30 times (and with different initial and goal locations) to ensure consistent results and minimize the impact of anomalies or random errors. This repeated testing helped produce more reliable data, providing a solid foundation for comparing the performance of the algorithms in demanding conditions.

A. Performance metrics

For the comparison of performance of these algorithms, the following parameters are considered,

- **Path Length**
- **Computation Time (ms)**
- **Explored Nodes**

The algorithms were tested with multiple initial and goal positions and the results are as below:

Algorithm	*Path Length (m)*	*Computation Time (ms)*	*Nodes Explored*
*A**	*23.54*	*24.25*	*341*
Dijkstra	*23.544*	*573.31*	*341*
RRT	*26.65*	*3207.36*	*26*

Table 1. Algorithm testing results (Start X = 7.8 m, Start Y = 34.4 m , Goal X = 23.8 m , Goal Y = 17.6 m, Map: kartoslamsps1x.pgm)

Similarly, the algorithms were also tested with another map of the facility and the results for that map are as follows:

Algorithm	*Path Length (m)*	*Computation Time (ms)*	*Nodes Explored*
*A**	*27.88*	*493.01*	*461*
Dijkstra	*23.75*	*1035.63*	*413*
RRT	*35.17*	*6275.46*	*32*

Table 2. Algorithm testing results (Start X = 7.3 m, Start Y = 7.6 m , Goal X = 17.5 m , Goal Y = 25.5 m, Map: kartoslamsps2.pgm)

Similarly, the algorithms were also tested with a third map of the facility and the results for that map are as follows:

Algorithm	*Path Length (m)*	*Computation Time (ms)*	*Nodes Explored*
*A**	*21.92*	*24.18*	*355*
Dijkstra	*21.92*	*306.67*	*355*
RRT	*22.46*	*4188.14*	*23*

Table 2. Algorithm testing results (Start X = 7.3 m, Start Y = 7.6 m , Goal X = 17.5 m , Goal Y = 25.5 m, Map: kartoslamsps3.pgm)

The path created by the three algorithms are shown in the following figures:

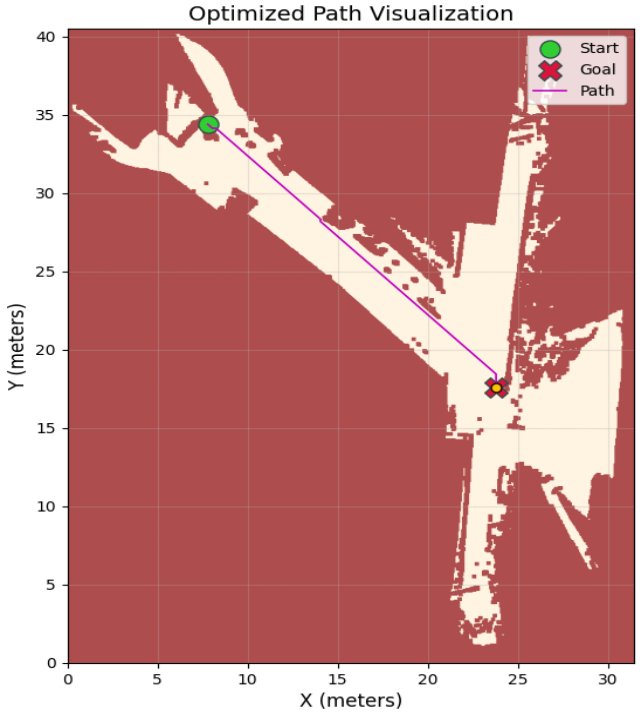

Figure 5: Path generated by Astar (Start X = 7.8 m, Start Y = 34.4 m , Goal X = 23.8 m , Goal Y = 17.6 m, Map: kartoslamsps1x.pgm)

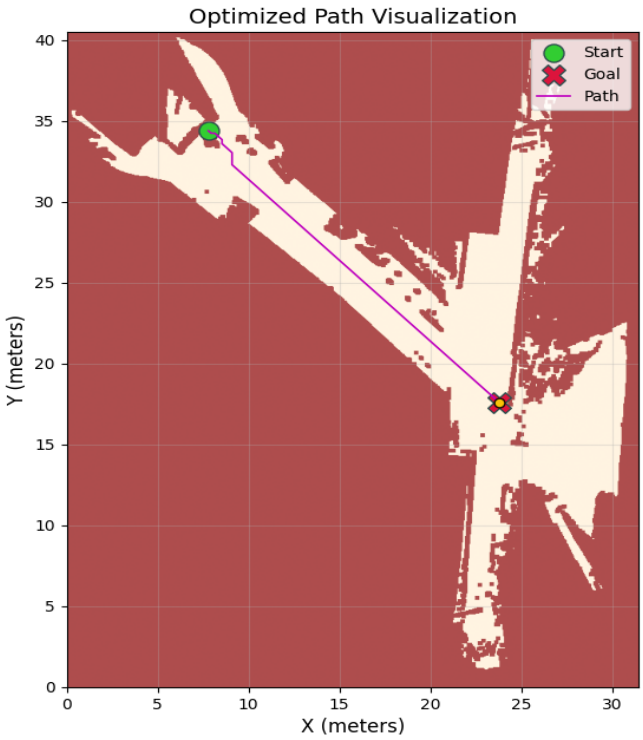

Figure 6: Path generated by Dijkstra (Start X = 7.8 m, Start Y = 34.4 m , Goal X = 23.8 m , Goal Y = 17.6 m, Map: kartoslamsps1x.pgm)

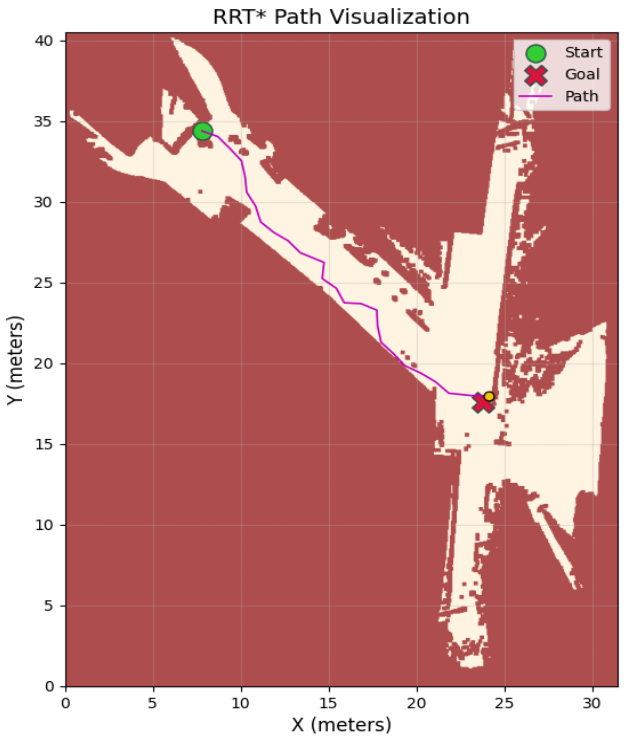

Figure 7: Path generated by RRT (Start X = 7.8 m, Start Y = 34.4 m , Goal X = 23.8 m , Goal Y = 17.6 m, Map: kartoslamsps1x.pgm)

B. Discussion

The performance of the three path-planning algorithms A* (A-star), Dijkstra's algorithm, and Rapidly exploring Random Tree (RRT) was evaluated on three different maps with distinct start and goal positions. The results highlight key differences in terms of path length, computation time, and the number of nodes explored.

In all test scenarios, A* and Dijkstra's algorithm produced nearly identical path lengths, indicating that both methods effectively find the shortest path in grid-based environments. However, Dijkstra's algorithm exhibited significantly higher computation times compared to A*, with computation times more than twice as long in all test cases. This is expected since Dijkstra's algorithm explores all possible nodes uniformly, while A* prioritizes nodes based on heuristic estimates, leading to faster convergence.

RRT, being a sampling-based method, resulted in considerably longer paths than the other two algorithms, as it does not guarantee optimality. However, its strength lies in handling high-dimensional and continuous spaces efficiently. The number of nodes explored in RRT was significantly lower than in A* and Dijkstra, which suggests that RRT's sampling approach results in sparse node exploration. However, this also leads to suboptimal paths and significantly higher computation times.

IV. Conclusion & Future Work

A. Conclusion

From the results obtained, it can be concluded that A* is the most balanced algorithm in terms of efficiency and optimality, offering fast computation while maintaining near-optimal paths. Dijkstra's algorithm, although optimal, is less efficient due to its higher computation time compared to A*. On the other hand, RRT is not particularly useful in this scenario. While it explores fewer nodes to reach the goal, it consumes significantly more time and computes longer paths than the other two algorithms.

The choice of algorithm largely depends on the application. If computation time is not a critical factor, Dijkstra's algorithm may be preferred over A*. If minimizing the number of explored nodes is a priority, regardless of computation time, RRT could be implemented. However, for applications like ours, where an optimal balance between computation time, path length, and the number of explored nodes is required, A* is the preferred algorithm.

B. Future Work

While our work has demonstrated the feasibility of autonomous navigation for the Mira robot, several areas remain for future improvement and exploration. The following directions outline potential advancements that could enhance the system's adaptability, efficiency, and real-world applicability.

1) Dynamic environtment adaptation

Currently, the navigation system primarily operates in static or semi-static environments. Future research should focus on enabling the robot to dynamically adapt to unpredictable and changing environments, such as moving obstacles, varying terrain, or human interactions. This could be achieved through real-time obstacle detection and predictive path planning using deep learning-based approaches.

2) 3D environment navigation

The present implementation operates in a 2D plane, which limits its applicability in environments with elevation changes, such as multi-level structures. Extending the navigation algorithms to function in three-dimensional spaces would significantly enhance the versatility of the system. Techniques such as 3D SLAM and point cloud-based navigation should be explored to achieve this capability.

3) Optimization of pathfinding algorithms

While A*, Dijkstra, and RRT were evaluated, future research could explore hybrid approaches that combine the strengths of multiple algorithms. Additionally, reinforcement learning-based path planning methods, such as Deep Q-Networks (DQN) and Policy Gradient methods, could be investigated to improve decision-making in complex and dynamic environments.

4) Autonomous decision-making and behaviour learning

The robot currently follows pre-programmed behaviours for navigation. Future work could integrate autonomous decision-making capabilities using artificial intelligence and machine learning techniques. This would allow the robot to learn optimal navigation strategies over time and adapt to unforeseen situations without manual intervention.

References

[1] D. Zhu, and Y. Mingzhong, "Review of path planning techniques for mobile robots [J]," *Control and Decision-making* 25.07, 2010, pp.961-967.

[2] H. Nguyen, F. Rego, J. Quintas, J.Cruz, M. Jacinto, D. Souto, A. Potes, L. Sebastio, and A. Pascoal. "A review of path following control strategies for autonomous robotic vehicles: Theory, simulations, and experiments." *Journal of Field Robotics* 40.3, 2023. pp:747-779.

[3] Parulekar, M., Padte, V., Shah, T., Shroff, K., & Shetty, R. (2013, January). Automatic vehicle navigation using Dijkstra's Algorithm. In *2013 International Conference on Advances in Technology and Engineering (ICATE)* (pp. 1-5). IEEE.

[4] MIT, "Robotic Path Planning", Available at: Robotic Path Planning - Path Planning. Accessed 23 March 2025.

[5] Garrote, L., Premebida, C., Silva, M., & Nunes, U. (2014, July). An RRT-based navigation approach for mobile robots and automated vehicles. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)* (pp. 326-331). IEEE.

[6] Yang, L., Li, P., Qian, S., Quan, H., Miao, J., Liu, M., ... & Memetimin, E. (2023). Path planning technique for mobile robots: A review. *Machines*, *11*(10), 980.

[7] Connell, D., & La, H. M. (2017, October). Dynamic path planning and replanning for mobile robots using RRT. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (pp. 1429-1434). IEEE.