An Analysis on Real-time Data based Path Planning Strategies for Wheeled Mobile Robots

1st Iulian Vornicu
Department of Automatic Control and Applied Informatics
Faculty of Automatic Control and Computer Engineering
Iași, Romania
iulian.vornicu@student.tuiasi.ro

***Abstract*—This paper investigates advanced path planning strategies for mobile robot navigation, with a focus on integrating LiDAR data acquisition with variants of the Rapidly-exploring Random Tree (RRT) algorithm, as well as insights into modern approaches such as genetic algorithms. Beginning with a overview of current path planning methodologies, encompassing recent trends, the paper defines the problem statement. Each algorithm employed, including RRT variants and genetic algorithms, is described, In addition to the tweaks made specifically for the robot in use. Simulation results are presented, featuring evaluations on both synthetic maps and real-world scenarios to check the solution correctness and processing time/resource consumption. The conclusions drawn underscore the efficacy of these strategies in mobile robot navigation, emphasizing their considerable potential for real-world application.**

***Index Terms*—Wheeltec S300, automatic control, path planning, trajectory control, sensor fusion**

I. Introduction

Path-planning for mobile robots is a crucial field in robotics, encompassing various strategies and algorithms to enable a robot to navigate from a starting configuration state to a destination while avoiding obstacles. In the context of dynamic obstacles this problem takes another complexity because the decision speed of algorithm is crucially. This area of study is integral to many applications, including autonomous vehicles (flying, on-road, underwater), robotic surgery, and warehouse automation. To better understand this subject, it is essential to first explore the fundamental aspects and then discuss more contemporary approaches to this problem. The practical component of this paper was conducted using two types of mobile robots: an omni-directional platform, Wheeltec S300, and an Ackerman model autonomous car, NxP MR-B3RB. The experimentation with the first vehicle, Wheeltec S300, played a crucial role in the development of this paper, while the second vehicle, NxP MR-B3RB, was utilized in the NXP Cup competition. The primary goal of this competition was to devise an algorithm that uses visual inputs [1] to facilitate autonomous navigation along a preset course. Both experiences significantly advanced the personal knowledge of ROS (Robot Operating System) and refined methods for determining paths for mobile robots. Although each vehicle presented unique challenges, the overarching focus remained on path planning, underscoring its essential role in the functionality of mobile robots. This leads us to a broader discussion on the categorization of this algorithms, where modern approaches increasingly leverage artificial intelligence to enhance efficacy and adaptability [2].

A. Graph-Based Algorithms

These algorithms are among the earliest capable of performing such tasks. Included in this list are algorithms derived from Dijkstra's and A*. They function by mapping the environment in a grid-like fashion, where paths are searched within the predefined nodes and connections [3], with associated costs that are determined based on the specific requirements of the implementation, with the aim of identifying the path that minimizes the overall cost.

B. Sampling-Based Algorithms

These algorithms were developed to address the shortcomings of graph-based algorithms, which can struggle in environments where obstacles move or change. In such dynamic settings, static graphs can quickly become outdated, requiring constant updates. Additionally, graph-based methods often falter in higher-dimensional spaces, which might include not just spatial dimensions but also dimensions related to constraints specific to different vehicle types. For example, the Ackerman steering vehicle (NxP MR-B3RB), with its unique turning constraints (in this instance, with a maximum steering angle of -60/60 degrees), operates within a higher-dimensional constraint space compared to an omnidirectional vehicle (Wheeltec S300) that can reach a designated point and then rotate to align with the target orientation. Sampling-based algorithms are more adaptable and capable of generating multiple solutions across a wide variety of situations, making them well-suited to handle the complexity and dynamism of such environments. A major player in this category of algorithms is the Rapidly-exploring Random Trees (RRT), along with several of its variations [4]. The base variant operates by randomly generating points in the space and connecting these points to the nearest existing tree node. The randomness of RRT helps it quickly cover uniformly the search space, making it highly useful for path planning in environments with many obstacles or specific constraints. This method continuously expands until it reaches the target or covers a maximum number of nodes/iterations, offering a practical approach to navigating challenging terrains.The flexibility of this solution lies in the tree's ability to rewire itself when obstacles move, creating a new valid path. A more balanced approach is the Probabilistic Roadmaps (PRM). Unlike RRT, PRM initially focuses on creating a "roadmap" of randomly placed nodes across the exploration space. It then connects these nodes with edges, ensuring that potential paths don't collide with obstacles. This creates a network of potential routes. PRM shines in scenarios where the same environment is traversed multiple times. Since the roadmap is computed just once, it can be reused for multiple path queries. This makes PRM a great choice for situations where planning time is crucial and the environment doesn't change much.

C. Artificial intelligence based approaches

With the advent of artificial intelligence (AI) techniques, the field of robotic path planning has witnessed significant advancements in recent years. This introduction provides a brief overview of some modern approaches in AI for path planning, showcasing the innovative methods driving the evolution of mobile robotics.

1) Reinforcement Learning-Based Approaches: this approach utilizes deep reinforcement learning techniques [5] to enable mobile robots to learn optimal paths through trial and error, often in complex and dynamic environments.
2) Deep Learning-Based Approaches: deep learning methods are employed to extract complex features from sensor data and maps, enabling mobile robots to plan paths efficiently and navigate autonomously in diverse environments [6].
3) Evolutionary algorithms [7] [8] are utilized to optimize path planning for mobile robots by iteratively evolving solutions through selection, crossover, and mutation, offering adaptability and scalability in various scenarios.

Each of these methods brings distinct advantages and drawbacks to mobile robot path planning. For instance, reinforcement learning enables robots to autonomously adapt their paths to dynamic environments, yet it demands significant computational resources and may pose safety concerns during learning. Deep learning empowers robots to navigate effectively by extracting intricate spatial representations from data, but it relies heavily on large labeled datasets and may lack interpretability. Meanwhile, evolutionary algorithms offer global optimization capabilities and maintain diversity in solution spaces, yet they require substantial computational resources and may converge slowly to optimal solutions.

II. Problem definition

This paper primarily addresses the evaluation, adaptation, and optimization of multiple algorithms to assess their performance and accuracy in both simulated and real-world scenarios. Specifically, the research concentrates on refining these algorithms for diverse environments, utilizing a mobile platform (referred to as Wheeltec S300, as outlined in the previous chapter) for data collection and Simultaneous Localization and Mapping (SLAM) [9]. The smaller NxP MR-B3RB operates with greater freedom within these mediums, aiding in this comprehensive analysis. A practical scenario is illustrated where a primary robot maps an environment post-earthquake, followed by the deployment of smaller robots to navigate the disturbed area. Before delving into these algorithmic strategies, it is crucial to first discuss the methods of data acquisition, processing, and utilization integral to this optimization process.

A. Data acquisition

Data acquisition via sensors is fundamental to the process of robot path-planning, providing the essential information required for a robot to interpret and navigate its environment. The choice of sensors and their integration directly influence the accuracy and reliability of the path-planning system. Here we explore the types of sensors utilized for this paper, the integration of their data, and the challenges they face in the context of robot navigation.

1) The primary sensor used in this project is the LiDAR (Light Detection and Ranging). This kind of sensor works by emitting laser light pulses towards the environment and measuring the time it takes for the light to bounce back after hitting an object. The platform (Wheeltec S300) has two of this kind of sensors to map the environment, while the car has one.
2) In addition to the primary sensors, the omni-directional robot is equipped with six ultrasonic sensors to detect objects in close proximity. Similar to LiDAR, these sensors determine the distance to nearby objects by measuring the time it takes for sound waves to reflect back to the sensor.

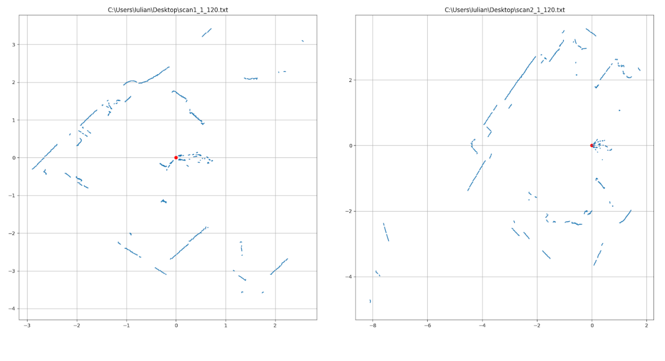

Fig. 1. Raw sensor data.

In Figure 1, we observe the raw data from the LiDAR sensors mounted on the platform. In the subsequent section, our attention will shift towards refining and filtering this information.

B. Data filtering

In this section, we'll explore various methods of data filtering, aiming to provide essential information for the algorithms discussed in the following chapter.

The first step involved in data filtering is removing information that falls outside the standard deviation of the LiDAR intensity output. This helps identify and mitigate outliers. For example, a really strong reading might be the LiDAR picking up the robot itself, and a very weak one could mean an object is too far off to confidently identify. The second step involved determining how to eliminate points that do not belong to any specific object or can be disregarded. This process is crucial for simplifying the data analysis by focusing only on the most relevant elements. For this purpose, the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [10] algorithm was implemented. DBSCAN is a clustering technique that identifies clusters in data based on the density of points. It operates by categorizing points into core points, border points, and noise, based on two parameters: epsilon (ϵ), which defines the radius around each point to search for neighboring points, and MinPts, the minimum number of points required within that radius to qualify a point as a core point. Points that do not meet these criteria are considered noise and can be ignored or removed in the analysis, effectively focusing on more significant, densely clustered data points. After multiple attempts, the optimal results were achieved when the algorithm was configured to ignore objects consisting of fewer than five points (MinPts = 5) within a radius of five centimeters (ϵ=0.05). A personal modification I made to this algorithm for this project was the addition of a minimum distance threshold for it to activate. This adjustment was made to account for smaller objects in close proximity to the robot.

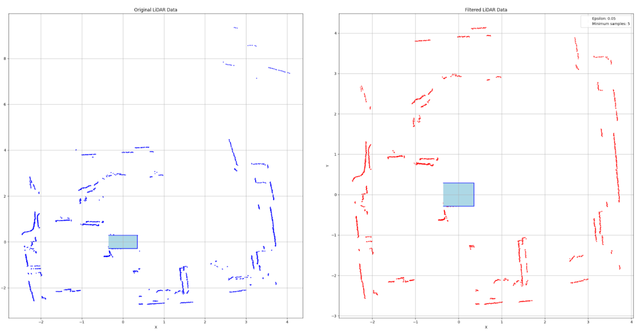

Fig. 2. Comparation before and after the filter

In Figure 2, we can observe how detection points that are far away and singulars are disregarded. This strategy enhances the focus on immediate and pertinent obstacles near the robot and aids in generating a more precise map during the stages before the focus of this paper.

The final step involves the quantization of space into a grid that categorizes potential states as robot, obstacle, extended obstacle, and goal. Initially, the center of the map is aligned with the robot's starting position, with the orientation also set to zero. This spatial representation simplifies the navigation process by giving accentuate to the areas accessible to the robot and identifying obstacles and objectives within the environment. The extended objects have the goal to simplify the representation of the robot to just a single point on the grid. This makes navigation and path planning a bit simpler by reducing the complexity associated with the robot's size. Each reduction of the robot is done in the center of two dimensional representation, using the top view. In this implementation, objects are expanded by a sufficient number of cells to allow the robot to maneuver freely within the usable space. Specifically, for the omni-directional robot, cells are expanded by half the diagonal length, whereas for the Ackerman model, the expansion is equal to the full diagonal length. The reasons are about the ability to do any maneuver without colliding with the obstacles.

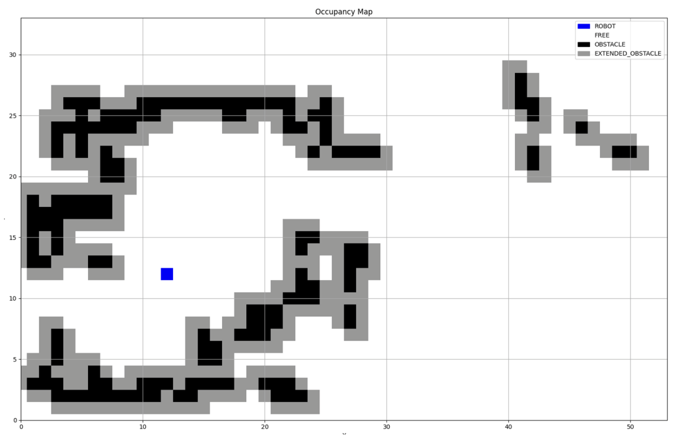

Fig. 3. Robot grid envirenment based on real data

In Figure 3, is one of the sceneries obtained from real data sensor acquisition, with the specified filters aplied and the robot reduction to a single point.

From here, we can shift our focus to the core subject: path planning algorithms and their application in this specific context.

III. Path planning algorithms

In this chapter, we will present the pseudocode for each evaluated algorithm and discuss their specific adaptations for validation on the actual mobile platform. We have selected a representative sample of algorithms from each major category mentioned to ensure a comprehensive evaluation. Each algorithm's pseudocode will be thoroughly explained, followed by a discussion on how it was modified or optimized to function effectively on the mobile platform.

A. *A* Algorithm (A star)*

In this the input is represented by the grid that represents the robot's environment, where each cell has a state from the next collection: FREE, ROBOT, OBSTACLE, EXTENDED_OBSTACLE and GOAL. The algorithm uses this grid to navigate from the starting point to the goal. Computing at each step the next move that minimize the cost function. Additionally, in this specific application, changes in direction are penalized. An extra cost is incurred whenever the robot's

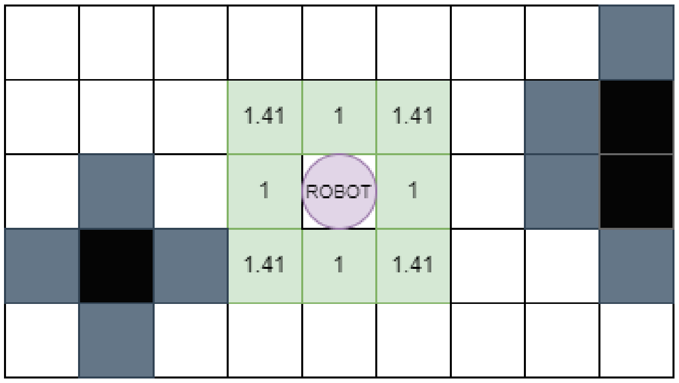

Fig. 4. Cost map of the robot movments.

next movement deviates from its previous direction to encourage smoother, more direct paths.

The robot, being omnidirectional, can perform the possible moves shown in Figure 4, each associated with its designated cost.

Algorithm 1 A* Algorithm

```
1: Let GRID be the robot environment
2: Let MOVES be the set of possible moves
3: Let OPEN be a empty priority queue
4: Initialize OPEN with the start node and initial cost
5: Initialize CLOSED with an empty set
6: **while** OPEN is not empty **do**
7:     $n \leftarrow$ node in OPEN with the lowest $f(n)$
8:     Remove $n$ from OPEN
9:     **if** $GRID(n)$ is GOAL **then**
10:        **return** Reconstructed path from start to goal
11:    Add $n$ to CLOSED
12:    **for** each possible move $dm = (dx, dy)$ in $MOVES$ **do**
13:        $new\_node \leftarrow n + dm$
14:        **if** $GRID(new\_node)$ is not FREE or GOAL **then**
15:            Try next possible move
16:        $g(new\_node) \leftarrow g(n) + \text{cost}(n, new\_node)$
17:        **if** $new\_node$ doens't improves path or in CLOSED **then**
18:            Try next possible move
19:        $h(new\_node) \leftarrow \text{HeuristicCost}(new\_node, \text{goal})$
20:        $total\_cost \leftarrow g(new\_node) + h(new\_node)$
21:        $f(new\_node) \leftarrow total\_cost$
22:        **if** $dm \neq$ last_move **then**
23:            Add a penality to the f(new_node) value
24:        update last_move
25:        parent$(new\_node) \leftarrow n$
26:        Add $new\_node$ to OPEN
27: **return** noSolution
```

B. The Rapidly-exploring Random Tree (RRT) algorithm

The algorithm initiates by placing the starting node at the robot's initial position. During each iteration, it randomly selects a point within the space. If this point coincides with an obstacle, the selection is repeated until a valid point is found. Once a suitable point is identified, the algorithm determines the closest existing node in the tree to this point [11]. If the distance to this nearest node exceeds a predetermined maximum length, the connecting line is shortened accordingly. For this implementation, the maximum allowable distance is tailored to match the minimum width of the obstacles, eliminating the need to verify for intermediate nodes between the new node and its nearest neighbor. This procedure continues until a valid node directly corresponds with the goal location. Ultimately, the algorithm reconstructs and returns the path from the starting node to the goal.

Algorithm 2 Rapidly-exploring Random Tree (RRT)

```
    Let GRID represent the robot's operating environment
2:  Let GOAL be the target location
    Initialize TREE with the start node
4:  Initialize MAX_DIST with the minimum width of the obstacles
    **while** not reached maximum iterations **do**
6:      $random\_point \leftarrow \text{SampleRandomPoint(GRID)}$
        **if** GRID(new_node) is not FREE or GOAL **then**
8:          Try next random point
        $nearest\_node \leftarrow \text{Nearest}(\text{TREE}, random\_point)$
10:     $nearest\_dist \leftarrow \text{Dist}(\text{nearest\_node}, random\_point)$
        **if** nearest_dist( is greater than MAX_DIST **then**
12:         Shorten random_point
        $new\_node \leftarrow \text{Extend}(\text{nearest\_node}, random\_point)$
14:     Add $new\_node$ to TREE
        **if** $GRID(new\_node)$ is GOAL **then**
16:         **return** Reconstructed path from start to goal
        Increment iteration counter
18: **return** noSolution
```

C. RRT-Connect

The first derivation from the Rapidly-exploring Random Tree (RRT) we will speak about is the RRT-Connect algorithm [12] [13]. It enhances the standard approach by employing a two-tree strategy to efficiently connect the start and goal nodes in the space. The process begins by initializing two trees: one at the robot's starting position and another at the goal. The algorithm places the starting node of the first tree at the robot's initial position and the starting node of the second tree at the goal. During each iteration, it selects a random point within the space. The algorithm then extends the nearest node from each tree towards this point. If the path to the point is blocked by an obstacle, the extension is halted, and the process repeats with a new random point.

For each tree, if the newly extended node is within a specified reach distance from the nearest node of the opposite tree, an attempt is made to directly connect them. If successful, the path between the start and goal is established through the connecting nodes of the two trees. The iteration continues until a connection is made, or a maximum number of iterations is reached. Finally, the algorithm reconstructs the path by connecting the nodes from the start to the goal through the connected trees.

To accelerate the convergence rate of the algorithm, the distance parameter is not limited to the minimum width of obstacles but is instead optimized for faster path connections. To ensure that paths between two grid points do not intersect with obstacles, the Bresenham line algorithm is utilized. The Bresenham line algorithm is an efficient method for drawing straight lines on a pixel grid. It uses integer arithmetic to determine which points should form the line, incrementing coordinates and adjusting for errors to ensure the line remains close to the ideal slope. This technique minimizes the number of grid points checked for obstructions, thereby validating the feasibility of adding new nodes to the tree. An extra procedural step is incorporated during the path reconstruction phase: long-distance node connections are refined by inserting additional points generated by the Bresenham line algorithm to ensure the path is both smooth and viable. From this point on, this method will be used for long distant nodes.

Next, we provide the pseudocode for the RRT-Connect algorithm, with the specified add-on:

Algorithm 3 RRT-Connect

```
    Let GRID represent the robot's operating environment
    Let START and GOAL be the initial and target positions
 3: Initialize TR1 with the start node at START
    Initialize TR2 with the start node at GOAL
    Define MAX_DIST as the maximum branch length
 6: Define REACH_DIST as the maximum reach distance for connection attempts
    while not reached maximum iterations do
        $random\_pt \leftarrow$ SampleRandomPoint(GRID)
 9:     for each $tree$ in [TR1, TR2] do
            $nearest \leftarrow$ Nearest($tree, random\_pt$)
            $new\_node \leftarrow$ Extend($nearest, random\_pt$)
12:         if Not IsValidNode($new\_node$, GRID) then
                Continue
            Add $new\_node$ to $tree$
15:         $other\_tree$ is the opposite tree to $tree$
            $aux\_nearest \leftarrow$ Nearest($other\_tree, new\_node$)
            $other\_nearest \leftarrow aux\_nearest$
18:         if notValidDist($new\_node, other\_nearest$) then
                Continue
            Use Bresenham line algorithm to check path between $new\_node$ and $other\_nearest$
21:         if path is clear then
                Connect $new\_node$ and $other\_nearest$
                return Reconstructed path
24: return noSolution
```

D. RRT (RRT Star)*

This version of the algorithm, known as RRT*, enhances the standard RRT by incorporating a step aimed at minimizing the path length. It does so by iteratively refining the path to each node, considering alternative, shorter routes that may emerge as the tree expands. This iterative refinement helps to reduce the path cost progressively, potentially culminating in an optimal path solution [14].

The algorithm initiates by setting the starting node at the robot's initial location. It follows the original RRT steps until it identifies a new valid point that, along with its neighboring nodes within a specified radius. In this specific implementation the radius is defined either as the length of the robot or a maximum of fifty nodes to prevent performance degradation in later stages due to a high density of nodes. The algorithm then evaluates which of these points would result in the most optimal path and adjusts the tree accordingly to incorporate this more efficient route. This process is repeated until either a satisfactory solution is achieved or the maximum number of iterations is reached.

Algorithm 4 Rapidly-exploring Random Tree Star (RRT*)

```
    Let GRID represent the robot's operating environment
    Let START and GOAL be the initial and target positions
    Initialize TREE with the start node at START
 4: Define RADIUS as the maximum of the robot's length
    while not reached GOAL and within iteration limits do
        $random\_pt \leftarrow$ SampleRandomPoint(GRID)
        $nearest \leftarrow$ FindNearestNode(TREE, $random\_pt$)
 8:     $new\_node \leftarrow$ Extend(nearest, $random\_pt$, RADIUS)
        if IsNotValidNode($new\_node$, GRID) then
            Continue
        $ns \leftarrow$ FindNeighborhood(TREE, $new\_node$)
12:     $best\_nd \leftarrow$ ChooseOptimal($new\_node, ns$)
        Connect $best\_nd$ to $new\_node$ in TREE
        if Distance($new\_node$, GOAL) is within a threshold then
            return Reconstructed path from START to GOAL
16:     Increment iteration counter
    return noSolution
```

*E. RRT-Divided**

The RRT-Divided* algorithm represents a combination of the RRT approach and a spacial division one, conceptualized to optimize the search process through systematic spatial division. Originating from a custom idea and implementation, this algorithm strategically divides the operational grid into four distinct quadrants from the robot's initial position. Within each quadrant, it randomly generates a point, subsequently evaluating the associated costs of reaching these points from the current node. The point that offers the most cost-effective route is selected and connected back to the originating node, establishing it as the new node for further division.

This process of recursive spatial division enables focused exploration of the grid, effectively minimizing the path length and reducing computational overhead by concentrating on the most promising areas first. As the algorithm advances, it continues to iteratively connect newly identified points to the nearest existing nodes in the tree, systematically building a coherent path towards the goal through successive refinements.

To counter corners that could results in the tree getting stuck for some time we add a bouncing effect that makes the robot to search for bigger distances if two direction with opositte effect take time one after the other.

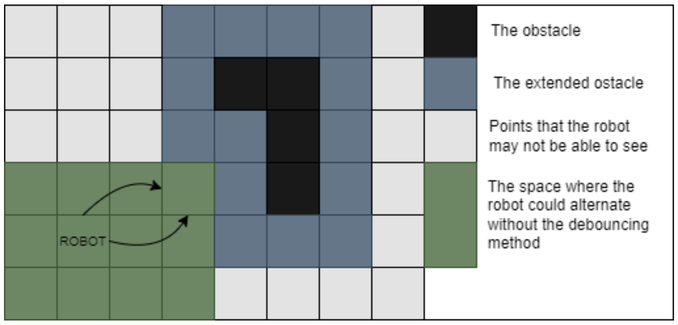
The obstacle
The extended ostacle
Points that the robot may not be able to see
The space where the robot could alternate without the debouncing method
ROBOT

Fig. 5. The bouncing problem.

The new aspect of RRT-Divided* lies in its ability to dynamically adjust exploration zones based on the developing path's requirements, converging in on optimal routes while avoiding less promising regions. This targeted exploration ensures that the algorithm remains efficient, even in complex operational environments. By continually repeating this process until the goal is reached or a set number of iterations are exhausted, RRT-Divided* efficiently constructs a path that is both direct and cost-effective.

F. FastRRT

The Fast RRT (FRRT) algorithm enhances the standard Rapidly-exploring Random Tree (RRT) method, specifically designed to speed up pathfinding in robotic navigation [15]. FRRT uses a biased sampling strategy that focuses search efforts directly from the start point to the goal, while maintaining enough randomness for thorough environmental exploration.

To converge more quickly toward the goal, this specific implementation of the FRRT dynamically adjusts the sampling probability, maintaining it within set thresholds to avoid the pitfalls of standard RRT, which either over-focuses on broad exploration or limits itself to areas near the goal.

Similar to fitness functions in evolutionary algorithms, the probability of generating a point closer to the goal increases if a series of successful points toward the goal has been generated. This adjustment in strategy promotes goal-oriented behavior throughout the search space. In each iteration, the FRRT algorithm selectively samples points nearer to the goal and extends the tree from the closest node to optimize the route. This process is repeated until the destination is reached or the maximum number of iterations is exhausted, thus enhancing both the speed and effectiveness of the robotic navigation path.

Algorithm 5 RRT-Divided* Algorithm

```
1: Let GRID represent the robot's operating environment
2: Let START and GOAL be the initial and target positions
3: Initialize TREE with the start node at START
4: $current\_node \leftarrow$ START
5: while within iteration limits do
6:     $quadrants \leftarrow$ divideGrid($GRID, current\_node$)
7:     $best\_point \leftarrow$ null
8:     $lowest\_cost \leftarrow$ infinity
9:     for each quadrant in quadrants do
10:        $point \leftarrow$ SampleRandomPoint($quadrant$)
11:        $cost \leftarrow$ ComputeCost($point, current\_node$)
12:        if $cost < lowest\_cost$ then
13:            $lowest\_cost \leftarrow cost$
14:            $best\_point \leftarrow point$
15:    if $best\_point = null$ then
16:        Continue
17:    Connect $best\_point$ to $current\_node$ in TREE
18:    $current\_node \leftarrow best\_point$
19:    Update random funciton for debouncing
20:    if $current\_node$ = GOAL then
21:        return path from START to GOAL through TREE
22: return noSolution
```

Algorithm 6 Fast Rapidly-exploring Random Tree (FRRT)

```
1: Let GRID represent the robot's operating environment
2: Let START and GOAL be the initial and target positions
3: Initialize TREE with the start node at START
4: Define MAX_ITERATIONS as the limit for algorithm execution
5: Define MAX_THRESHOLD and MIN_THRESHOLD for dynamic probability adjustment
6: $current\_node \leftarrow$ START
7: while iteration ¡ MAX_ITERATIONS do
8:     $prob \leftarrow$ UpdateProbability()
9:     $random\_point \leftarrow$ SamplePoint(GRID, prob)
10:    if IsNotValidPoint($random\_point$, GRID) then
11:        Continue
12:    $nearest\_node \leftarrow$ Nearest(TREE, $random\_point$)
13:    $new\_node \leftarrow$ Extend(nearest_node, $random\_point$)
14:    if IsNotValidConnection($new\_node$, GRID) then
15:        Continue
16:    Add $new\_node$ to TREE
17:    $current\_node \leftarrow new\_node$
18:    if $current\_node$ = GOAL then
19:        return path from START to GOAL through TREE
20:    $iteration \leftarrow iteration + 1$
21: return noSolution
```

G. Genetic algorithm

The Genetic Algorithm (GA) is a powerful optimization technique inspired by natural selection, ideal for complex problems like robot path planning. It starts with a population of potential solutions, each encoded as a sequence of actions, in our case: left, right, up, down, or diagonal steps. Each solution's effectiveness is evaluated using a fitness function considering path length, feasibility, obstacle avoidance, and goal achievement [16].

GA employs genetic operators like selection, crossover, and mutation to improve the population. Selection methods pick the fittest individuals, ensuring beneficial traits are propagated. Crossover combines two parents to create new offspring through crossover of data, introducing genetic variety. Mutation adds diversity by randomly altering parts of the genetic code, broadening the search for potential solutions. GA refines the population iteratively through repeated cycles of these genetic operations. Each cycle involves evaluating the current population, selecting the fittest individuals, applying crossover and mutation to produce a new generation, and replacing the older population.

Fitness is continuously assessed, with penalties for hitting obstacles or straying outside grid boundaries, reducing the path's fitness score. Through successive generations, GA refines these solutions, gradually extending the path length while capping it to avoid impractical solutions. This iterative process continues until a path with satisfactory fitness is achieved or a predetermined number of generations is reached.

Algorithm 7 Genetic Algorithm for Robot Path Planning

```
1: Let GRID represent the robot's operating environment.
2: Let START and GOAL be the initial and target positions.
3: Initialize population with random paths from START to GOAL.
4: Define MAX_GENERATIONS as the maximum number of iterations.
5: while generation count ¡ MAX_GENERATIONS and no solution found do
6:     Select the top-performing individuals based on fitness for reproduction.
7:     Perform crossover among selected individuals to produce offspring.
8:     Mutate the offspring at a predetermined mutation rate to introduce variability.
9:     Evaluate the fitness of each new individual in the offspring.
10:    Select the best individuals from the current and new generation for the next generation.
11:    if any individual reaches GOAL then
12:        return the path
13:    Increment generation count.
14: return noSolution
```

IV. Results

To evaluate the effectiveness of pathfinding algorithms implemented, it is necessary first to establish a set of metrics that quantitatively measure the quality of the generated paths. These metrics not only provide insights into the efficiency and practicality of the paths but also highlight aspects critical to navigation such as safety and adaptability under dynamic conditions.

1) **Path Length:** This metric assesses the directness and efficiency of the path, favoring shorter paths that potentially reduce travel time and conserve energy.
2) **Number of Turns:** It quantifies the navigational complexity of the path, with fewer turns indicative of easier maneuverability and reduced energy consumption during directional changes.
3) **Smoothness:** Analyzed through the angles between consecutive segments, where smaller angles suggest a smoother path facilitating higher travel speeds and enhanced control.
4) **Execution Time:** Critical in dynamic environments, the time taken to compute the path is crucial, especially where quick response and adaptation are necessary.
5) **Energy Cost:** Estimated based on the path length, changes in elevation, and the number of turns, providing an insight into the practical energy demands of the path.

To evaluate the algorithms used, we tested them in a variety of scenarios designed to simulate real-world challenges. These scenarios range from basic grid environments to complex, real-life data-driven maps, ensuring a comprehensive evaluation of each algorithm's performance.

We began with a simple 5x5 grid (from this point one point in the grid is 0.25m x 0.25m, like it would be from the real life data), a basic environment to assess the fundamental success rate of the algorithms in finding a path from the start to the goal. This straightforward grid allowed us to focus on the core functionality of each algorithm.

Following this, we tested the algorithms in a more complex 20x20 artificial grid simulating a warehouse environment. This grid included randomly placed obstacles representing shelves, crates, and other objects, presenting additional challenges for the algorithms. The increased size and random obstacle placement tested the adaptability and efficiency of each algorithm in a more intricate setting, closer to real-world applications.

Finally, we evaluated the algorithms on a map derived from real-life data acquisition. This map reflected the actual layout and obstacles found in a specific environment, such as a manufacturing floor or a logistics center. Testing on this map allowed us to assess the algorithms' performance under authentic conditions, ensuring they could handle the complexities and nuances of a real-world setting.

By systematically evaluating the algorithms across these varied scenarios, we gained a comprehensive understanding of their capabilities, limitations, and overall effectiveness in different path planning contexts.

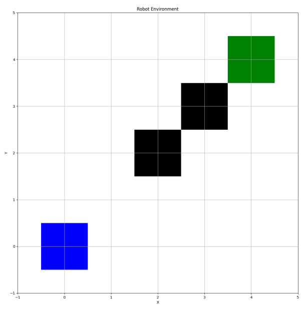

Fig. 6. Map utilized for the initial test.

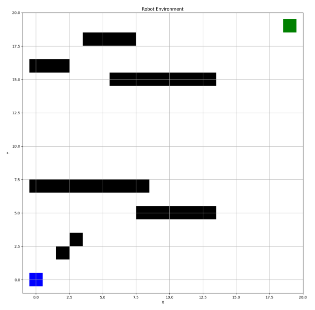

Fig. 8. An example of the warehouse scenary

A. The simplest map

As seen in Fig. 6, the first scenario features a 5x5 map with two obstacles to check for collisions. This map is used to test which solutions are unreliable from the start. The data represents the mean over 250 independent runs, with a time constraint of 60 seconds per run and a limit of 10,000 iterations.

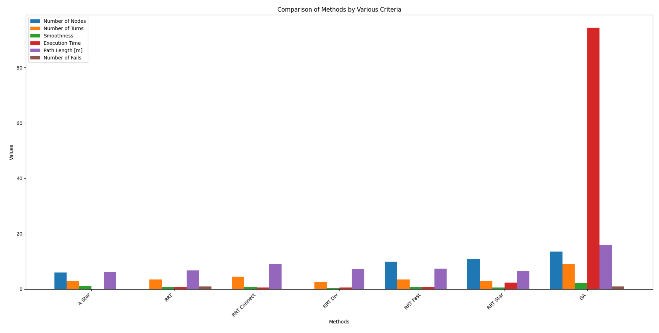

Fig. 7. Results for the first map.

As can be observed in Fig. 7, although all algorithms performed acceptably and found usable paths, the Genetic Algorithm is significantly slower and has more failed points in comparison. This aspect can be crucial in real-time or fast-reaction operations. The slow response is a consequence of the fact that the algorithm has no knowledge transfer from one iteration to another. In the future, this approach could be upgraded with the addition of a neural network.

B. Randomm warehouse map

In this scenario, the map has a fixed size of 20x20, featuring five randomly generated obstacles of varying lengths in a horizontal orientation. An example is shown in Fig. 8. The robot's starting point is at (0, 0), and the goal is located at the opposite corner of the map.

This scenario was tested over 100 runs, with the same constraints on the number of iterations and time limit per run.

As seen in Fig. 9, the fail rate is high, indicating that the time and iteration constraints are too harsh for the algorithms.

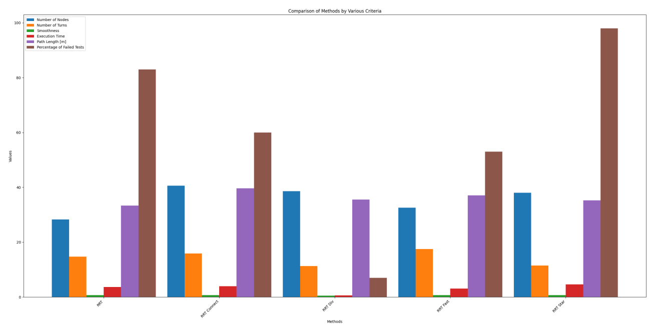

Fig. 9. Results on the bigger map.

Increasing the max_iteration to 50,000 yields the following results:

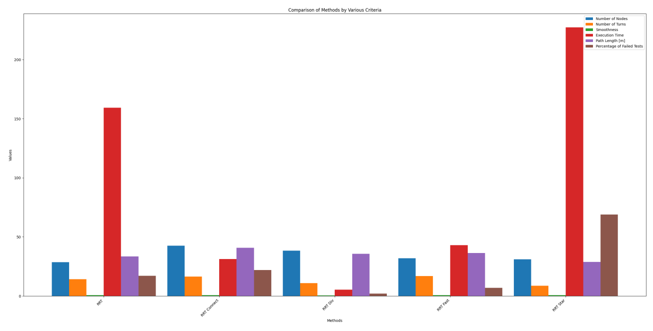

Fig. 10. Results with reduced constraints on the larger map.

C. Map from lidar data

The initial figure in this subsection depicts a map with dimensions of 31x54, as detailed in the first chapter. This map was created based on real-life data collected from lidars, which were introduced in the earlier chapters. The second figure illustrates the outcomes of 1000 runs, which required an increase in the maximum number of iterations, similar to the shift from a simple 5x5 map scenario to a more complex warehouse environment. These results strongly correlate with the findings discussed in the previous subsection. In the

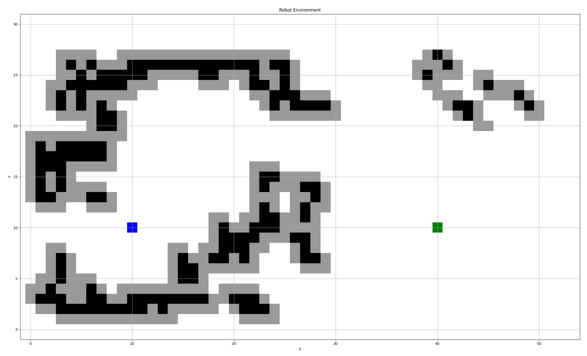

Fig. 11. Map from real lidar information.

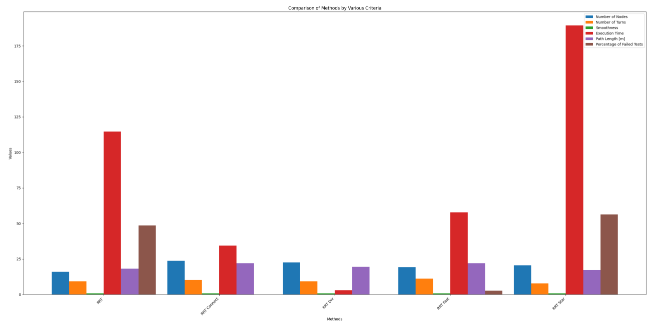

Fig. 12. Results for the real life data.

final chapter, the implications of these results are thoroughly analyzed, emphasizing their significance within the overall context of the study.

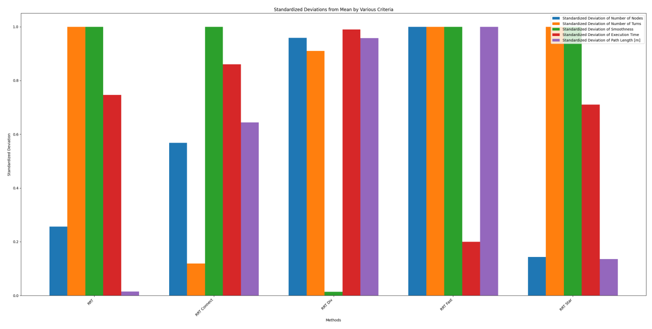

Fig. 13. Normalized deviation of the means of the results for the real life data.

Before drawing conclusions from the results, it is essential to consider the standard deviation of the data from the mean. The standard deviation serves as a critical statistical metric that quantifies the variability or dispersion of a dataset relative to its mean. In the context of experimental results, the standard deviation is particularly valuable because it provides insight into the reproducibility of the results. A smaller standard deviation indicates that the data points tend to be closer to the mean, suggesting higher consistency and reliability of the experimental outcomes across multiple trials. While a larger standard deviation suggests greater variability, indicating less predictability and lower reproducibility of the results. Thus, understanding this measure of spread is fundamental in assessing the robustness and reliability of the algorithms being tested, thereby ensuring that any conclusions drawn are well-supported by the data.

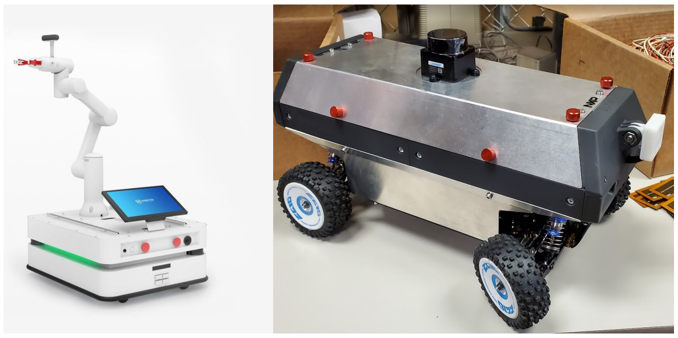

Fig. 14. Platform and car.

Both of the robots, which are the focus of this study, can be seen in the fig. 14. This visual representation provides context as we transition to the concluding remarks.

V. Conclusion

From the results from the previous chapter we can draw the following conclusions: Starting with the RRT algorithm, it generates a moderate number of nodes and turns. Additionally, the execution time is fairly high, making it less suitable for real-time applications. The fail rate for RRT is significantly high, suggesting that the time and iteration constraints were too restrictive for this algorithm.

On the other hand, RRT Connect generates a greater number of nodes compared to the standard RRT, indicating a more comprehensive exploration and slightly longer paths. This algorithm not only matches but potentially exceeds the performance of the standard RRT variant, while significantly reducing execution time. Most notably, it achieves this with zero failures.

RRT Div is an upgrade from both of the previous algorithms, having the lowest execution time from all methods with zero failures. The other aspects, such as and number of turns, are similar to the previous methods. However, its speed makes it a strong candidate for real-time applications.

RRT Fast shows very good performance across multiple metrics. It is a balanced method with a good execution time (a little worst than RRT Connect), a low failure rate (2-3%), and competitive results in other criteria, making it comparable with the other algorithms.

Lastly, RRT Star generates a high number of nodes, indicating extensive exploration and optimization. It has fewer turns, suggesting more optimized and direct paths, and the smoothness of the paths is high, showing highly optimized and direct routes. However, the execution time is quite high due to the optimization process, making it less ideal for quick decisions. Despite its efficient paths, the fail rate is very high, indicating that even the current constraints are too restrictive for RRT Star to perform effectively. This high fail rate, combined with the long execution time, highlights the limitations of RRT Star under the given conditions.

However, RRT Star and other algorithms with extensive exploration are strong candidates for future integration with rewire algorithms. When there are dynamic obstacles are encountered, these algorithms can more easily find alternative paths, enhancing their practicality and robustness in dynamic environments.

When considering the standard deviations:

1) "RRT" has the least variation in the standardized deviation of path length, making it the most consistent in maintaining a similar path length across different scenarios.
2) "RRT Connect" shows the lowest variation in the standardized deviation of the number of turns, smoothness, and execution time, demonstrating very high consistency in these aspects.
3) "RRT Div" has the most constant smoothness while maintaining mid-level performance in every other aspect.
4) "RRT Fast" is very consistent in its execution time.
5) "RRT Star" stands out as one of the most consistent algorithms overall, excelling in path length, number of nodes, and maintaining a relatively constant execution time.

In conclusion, the evaluation of various Rapidly-exploring Random Tree (RRT) algorithms provides distinct insights into their suitability for different applications within robotic path planning and navigation. RRT Div emerges as the optimal choice for real-time applications due to its outstanding speed and reliability, marked by the lowest execution times and zero failures. For general usage across a range of conditions and requirements, RRT Connect proves to be the most adaptable, balancing efficient exploration with quick execution and flawless operational performance. Meanwhile, RRT Star is best suited for scenarios involving dynamic obstacles. Its capacity for extensive exploration and optimization, although accompanied by higher fail rates and longer execution times, makes it invaluable for environments where conditions can change unpredictably, necessitating rapid re-routing and adaptation.

References

[1] M. Jain, "Lane line detection," *International Journal for Research in Applied Science and Engineering Technology*, vol. 12, pp. 4932–4938, 04 2024.

[2] Y. Guo and Z. Liu, "Uav path planning based on deep reinforcement learning," *International Journal of Advanced Network, Monitoring and Controls*, vol. 8, pp. 81–88, 03 2024.

[3] M. Laith, A. Humaidi, and E. Flaieh, "A comparison study and real-time implementation of path planning of two arm planar manipulator based on graph search algorithms in obstacle environment," *ICIC Express Letters*, vol. 17, pp. 61–72, 01 2023.

[4] Q. Zou, M. Le, and Y. Shen, "Rrt path planning algorithm with enhanced sampling," *Journal of Physics: Conference Series*, vol. 2580, p. 012017, 09 2023.

[5] H.-C. Chen, L. Shih An, T.-H. Chang, h.-M. Feng, and Y.-C. Chen, "Hybrid centralized training and decentralized execution reinforcement learning in multi-agent path-finding simulations," *Applied Sciences*, vol. 14, p. 3960, 05 2024.

[6] Y. Guo and Z. Liu, "Uav path planning based on deep reinforcement learning," *International Journal of Advanced Network, Monitoring and Controls*, vol. 8, pp. 81–88, 03 2024.

[7] J.-S. Shaw and S.-Y. Lee, "Using genetic algorithm for drawing path planning in a robotic arm pencil sketching system," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 02 2024.

[8] M. Sun, *Design of Path Planning Algorithm for Intelligent Robot Based on Chaos Genetic Algorithm*, pp. 127–135. 10 2023.

[9] Z. Lin, Q. Zhang, Z. Tian, P. Yu, and J. Lan, "Dpl-slam: Enhancing dynamic point-line slam through dense semantic methods," *IEEE Sensors Journal*, vol. PP, pp. 1–1, 05 2024.

[10] M. Begum, M. H. Shuvo, M. G. Mostofa, A. Kamrul, A. K. Islam Riad, M. Arabin, I. Talukder, S. Mst, Akter, and H. Shahriar, "M-dbscan: Modified dbscan clustering algorithm for detecting and controlling outliers," 04 2024.

[11] J. Yao, "Rrt algorithm learning and optimization," *Applied and Computational Engineering*, vol. 53, pp. 296–302, 03 2024.

[12] Z. Tu, W. Zhuang, Y. Leng, and C. Fu, *Accelerated Informed RRT*: Fast and Asymptotically Path Planning Method Combined with RRT*-Connect and APF*, pp. 279–292. 10 2023.

[13] Y. Li and S. Ma, "Navigation of apple tree pruning robot based on improved rrt-connect algorithm," *Agriculture*, vol. 13, p. 1495, 07 2023.

[14] H. Pu, X. Wan, T. Song, P. Schonfeld, and L. Peng, "A 3d-rrt-star algorithm for optimizing constrained mountain railway alignments," *Engineering Applications of Artificial Intelligence*, vol. 130, p. 107770, 04 2024.

[15] Z. Wu, Z. Meng, W. Zhao, and Z. Wu, "Fast-rrt: A rrt-based optimal path finding method," *Applied Sciences*, vol. 11, p. 11777, 12 2021.

[16] R. Mankudiyil, R. Dornberger, and T. Hanne, "Improved genetic algorithm in a static environment for the robotic path planning problem," pp. 217–230, 02 2024.