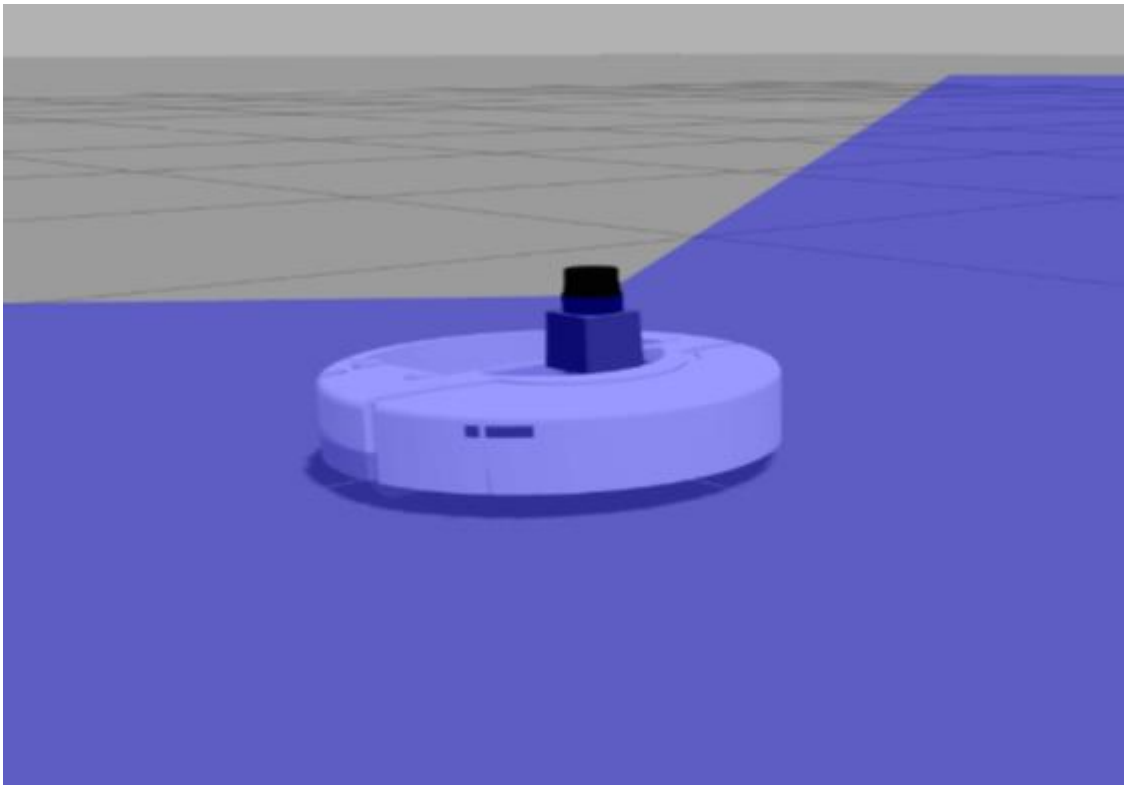
NASIR Amun, JOHANNES Raoul, CONSTANT Quentin, PEYRARD LEJEUNE Luc
SM3-F

21/04/2025

In424: Cooperative Frontier-Based Exploration with Multi-Robot Systemsa

Deploying Autonomous Agents for Efficient Mapping in Unknown Environments using ROS2 and Gazebo



NASIR Amun, JOHANNES Raoul, CONSTANT Quentin, PEYRARD LEJEUNE Luc
SM3-F

21/04/2025

Table des matières

I. Introduction

This project focuses on designing and deploying a multi-robot system capable of autonomously exploring an unknown environment in a coordinated and efficient manner. The main objective is to map a 40x40 grid environment as quickly as possible by leveraging collaborative behaviors between three autonomous mobile robots.

Each robot is equipped with:

- **Two motorized wheels**, allowing differential drive motion.
- A **LIDAR sensor** with the following specifications:
  - **Field of view**: $[-\pi;\pi]$ radians
  - **Angular resolution**: 10.25° per scan increment
  - **Sensing range**: 0.1 to 4 meters
- A **cylindrical body** with a **radius of 25 cm**

The robots are deployed within a simulated environment using the **ROS2** (Robot Operating System 2) **framework** and the **Gazebo simulator**, which provides realistic physics, sensor emulation, and communication middleware. ROS2 also enables inter-robot communication and modularity, making it ideal for developing scalable multi-agent systems.

To achieve rapid and complete exploration, we implement a frontier-based exploration strategy, where robots seek out the borders between known and unknown regions. Through real-time coordination, frontier grouping, and intelligent task allocation, the robots collaborate to minimize overlap and idle time, ensuring an optimized mapping process.

This project demonstrates the integration of sensing, control, and coordination in a simulated autonomous system and serves as a foundation for real-world multi-robot exploration scenarios.

II. Cartography

The cartography module plays a fundamental role in enabling autonomous robots to explore and understand an unknown environment. It is responsible for building a shared and updated occupancy grid map by transforming raw LIDAR sensor data into usable spatial information. The output map is then used for path planning and frontier-based exploration. In this section we will talk about how we manage to know the map near our robots.

A. Coordinate system alignment

One of the first technical challenges addressed in this part of the project was the alignment between coordinate systems. While Python matrices are typically computed with the origin located at the top-left corner, the occupancy grid in ROS2 considers the origin to be at the bottom-left. This mismatch required adjustments in how data points were interpreted and plotted onto the map to ensure spatial accuracy. So, our coordinates become:

$$(x, y)_{map} = (-y - 1, x)_{matrix}$$

B. Transformation from sensor data to map grid

To put our LIDAR sensor data on our map we need to perform 4 steps.

1. **Reading sensors data**
   Each robot collects readings from its LIDAR sensor, which provides distance and angle information for each detected point. These readings are used to construct a representation of the surrounding environment.
2. **Conversion to Local Coordinates (Robot Frame)**
   The sensor data is initially expressed in polar coordinates. It is converted to Cartesian coordinates in the robot's local frame using:
   $x_{local} = d.\cos(\theta), y_{local} = d.sin(\theta)$
3. **Transformation to Global Coordinates**
   The local coordinates are transformed to global coordinates using the robot's position and orientation (x_r, y_r, θ_r):

$$x_0 = x_r + (x_{local} * \cos(\theta_r) - y_{local} * \sin(\theta_r))$$
$$y_0 = y_r + (x_{local} * \sin(\theta_r) + y_{local} * \cos(\theta_r))$$

4. **Conversion to Grid Indices**
   Once in global coordinates, the points are converted into indices of the occupancy grid using the map resolution and origin:

$$i = \left\lfloor \frac{x_0 - x_{map}}{resolution} \right\rfloor, j = \left\lfloor \frac{y_0 - y_{map}}{resolution} \right\rfloor$$

C. Map Update Using Bresenham's Line Algorithm

To update the occupancy grid accurately, it is necessary to mark not only the endpoints of LIDAR rays (which typically represent obstacles) but also all the **free cells** between the robot and the detected obstacle. For this purpose, we implemented **Bresenham's line algorithm**.

Bresenham's algorithm is a rasterization method used to identify which discrete grid cells form a straight line between two points. It is computationally efficient and well-suited for real-time LIDAR data processing. In our case, the algorithm is applied as follows:

- Draw a line from the robot's current grid position to the grid cell corresponding to the detected obstacle.
- Mark all intermediate cells along this line as **free**.
- Mark the final cell (i.e., the LIDAR endpoint) as **occupied** if it lies within the sensor's range (i.e., $\leq$ 4 meters).
- If the LIDAR detects an object farther than 4 meters (or if no object is detected), the final cell is left as **free space**, as no obstacle is confirmed.

This approach significantly improves the accuracy of the constructed map, helping to avoid incorrect labeling of free or unknown space, which in turn enhances the efficiency and safety of the exploration process.

III. Frontiers

In the context of autonomous exploration, a frontier represents the boundary between known and unknown space. It is a critical concept in robotics exploration, as it directly defines the next potential areas to be discovered by the robots. By identifying and targeting these frontier regions, we ensure that the robots are always moving toward the edge of the known map, gradually expanding it until the entire environment has been explored.

We chose to implement a frontier-based strategy from the beginning because it naturally aligns with the goal of full map coverage. By computing the cost to reach each frontier and assigning them efficiently to robots, we maximize exploration efficiency while minimizing overlap or redundancy. The strategy also enables the system to dynamically react to the environment by constantly updating the map and redefining frontiers in real time.

A. Frontier detection

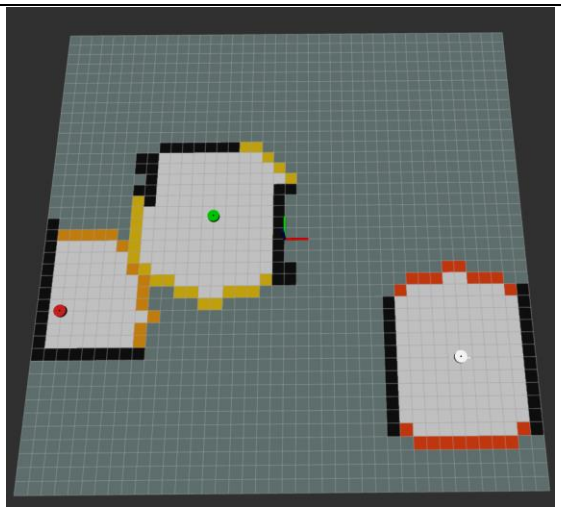
First, we needed to define what frontiers are; to do so we implemented a function that starts from a known free space—typically a cell marked as free in the occupancy grid (i, j). From this point, we examine its adjacent cells in a cross pattern (i.e., up, down, left, right). If one of the adjacent cells is still unexplored (unknown space), the current cell (i, j) is marked as a frontier cell.

This process is repeated over all known free spaces on the map. Every identified frontier cell is appended to a list, frontier_list, and temporarily marked with a distinct color or value based on the robot that detected it. This helped in tracking which robot contributed to which part of the map and was useful for debugging or visualization purposes.

B. Frontier grouping

To ensure meaningful and non-fragmented exploration, frontier cells are clustered into **groups** based on spatial adjacency.

As you can see below, Robot 3 has 1 frontier, Robots 2 & 1 both have 2.



The grouping algorithm iteratively selects a frontier cell, then performs a **breadth-first search (BFS)** to gather all neighboring frontier cells (including diagonals) into a single group. This process continues until all frontier cells have been assigned to a group.
Each group then represents a continuous unexplored area and forms the basis for further cost evaluation and task assignment.

C. Centroids computation

For each group of frontiers, we calculate the **centroid**—the average position of all cells in that group. These centroids serve as target candidates for exploration.
By reducing entire regions to a single representative point, we simplify decision-making for the robot strategy module, making it easier to compute and compare potential paths.

At this stage, each frontier is represented by centroids. The next step was to implement a decision system that evaluates and compares the **cost** for each robot to reach each centroid—using metrics like distance and the size of the frontier group.

IV. Path planning

Path planning algorithms are necessary in order to plan a route towards the centroids computed before. There exist multiple path planning algorithms such as A*, Djikstra or RRT. We will choose A* in our study because it finds the shortest path to a known point to reach. Djikstra algorithm explores all possible paths which is not efficient in our case. RRT is also not an option because it is more suitable for high-dimensional spaces and is harder to implement.

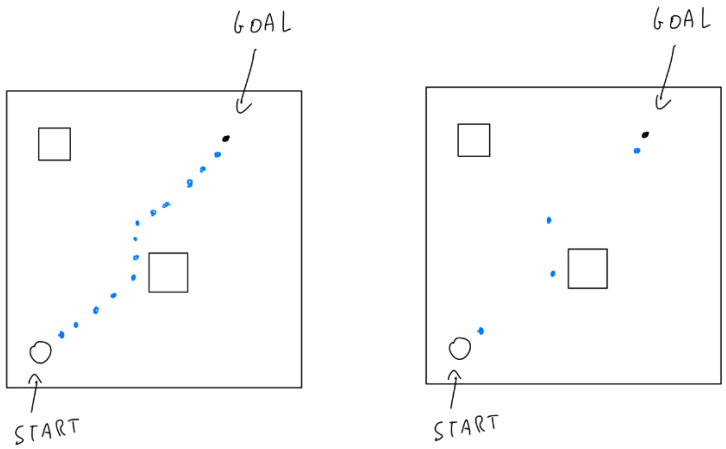
In pseudo-code A* works as follows:

- Define a start node and goal node
- Create an open list and put the start node in it
- For each node, we will compute:
    - g score: cost from start node
    - f score: g + h with h a heuristic function
- While the open list is not empty, we take the node with the lowest f score
- If that node is the goal, we are done
- Otherwise, for each neighbor of the current node, we compute the g score
- If the neigbor is not in open list, we add it otherwise we update its g and f
- Repeat until goal is found

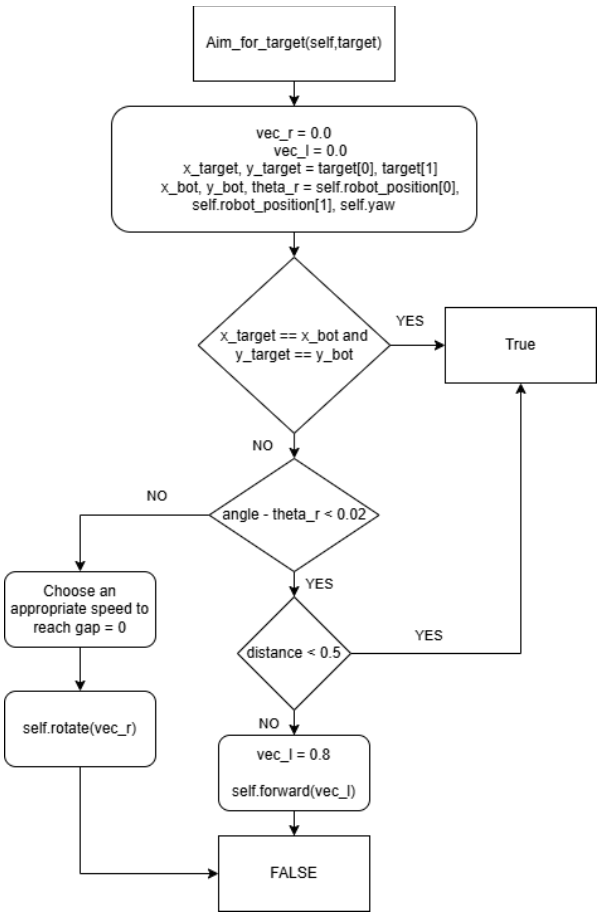
In our case, we tried to implement an 8-direction A* algorithm with the heuristic being the euclidian norm:

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

Finally, we removed intermediate points in the path of A* for efficiency like below:



V. Movement



The aim_for_target function is designed to guide the robot toward a specific target within a discrete grid, where the target is defined by its coordinates ranging from 0 to the grid's width and 0 to its height. The function operates by adjusting two vectors: vec_l and vec_r, which represent the linear and rotational movement commands respectively, corresponding to vec_linear and vec_rotation.

The function begins by calculating the necessary rotation for the robot to align with the direction of the target. While the robot's orientation is not yet aligned with the target direction, the function continuously calls the rotate function with a vec_rotation set to the robot's maximum rotational speed. As the robot gets closer to the desired orientation, this rotational vector is gradually reduced to allow for more precise adjustments. Once a safe alignment threshold is reached, the function calls rotate with vec_rotation = 0.0, effectively stopping any further rotation.

Following successful alignment, the function then calls the forward function with a vec_linear set to 1.0, initiating forward movement toward the target. This continues until the robot reaches the destination, at which point the forward function is called with vec_linear = 0.0, bringing the robot to a stop. This approach allows smooth and controlled navigation toward discrete grid targets using simple vector commands.

An area of improvement for the aim_for_target function would be to allow backward movement when it is more efficient than turning around. Currently, the robot always rotates to face the target before moving forward. However, in certain situations, especially when the target is behind the robot, it could be faster and more efficient to move in reverse rather than performing a full rotation. By enabling the function to send a negative vector forward, the robot could drive backward directly toward the target. Implementing this logic would require checking the angular difference between the robot's current orientation and the direction of the target and choosing the shortest or most optimal path—forward or backward—accordingly. This enhancement would make the navigation more flexible and efficient, particularly in tight or dynamic environments.

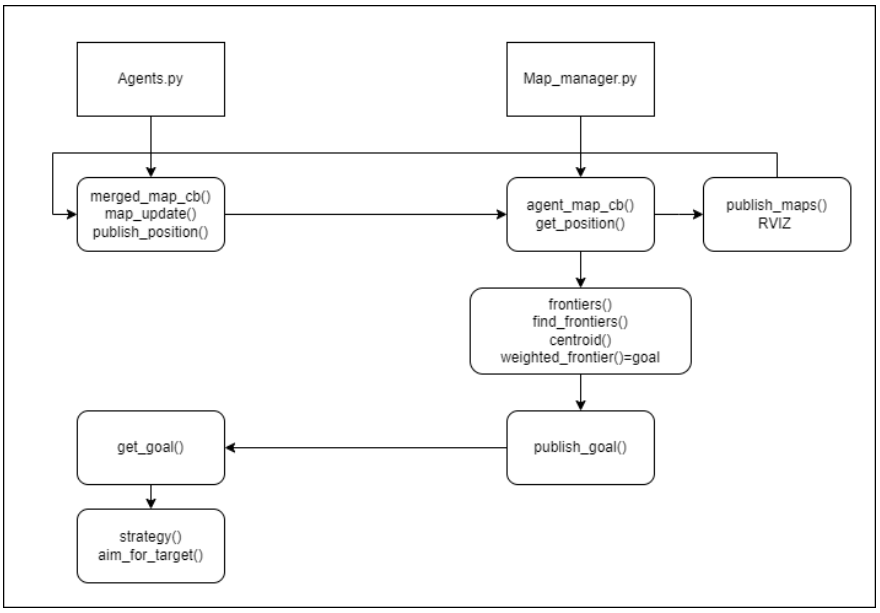
To conclude, the aim_for_target function is not well-suited for scenarios where a separate path-finding algorithm provides a full list of grid cells that the robot must traverse to reach a goal. In such cases, this function tends to stop too frequently to realign the robot toward each intermediate target, which can lead to inefficient and jerky movement. A more appropriate approach would be to implement a continuous vector-based navigation function, driven by external stopping conditions and smoother directional updates. Nonetheless, aim_for_target has proven sufficient for initial robot testing, offering reliable and straightforward control in simpler contexts. It remains a valuable foundation and a clear point of improvement for future development.

VI. Strategy

We tried to implement 2 strategies:

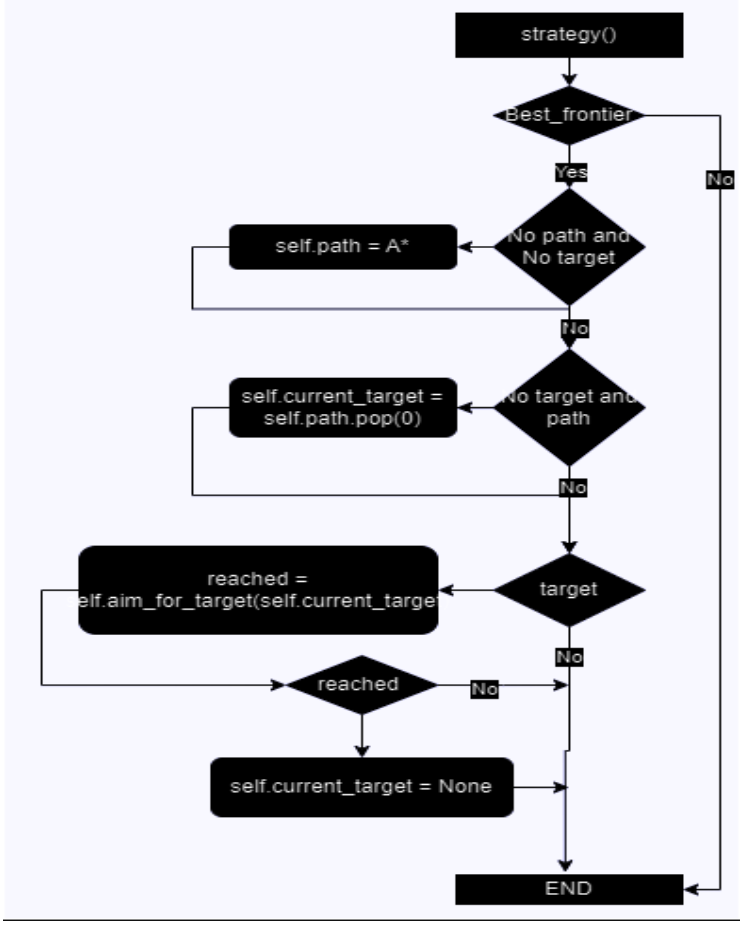
- FRONTIER-BASED, take a discrete map, compute frontiers' centroid and go to the lowest cost one.
- HEAT MAP, take a continuous map from 0 to 100 (0 for free space and 100 for obstacle). The robot goes to an unkown position

The structure of the code before diving deeper:



We opted for a centralized decision-making system. Agents.py has a publish_position() function that publishs the position of the robots which is received by the subscriber get_position() in Map_manager.py. Then, those coordinates are treated by frontiers() which detetects the frontiers and returns a list of all the concerned cells. Find_frontiers() clusters those frontiers into a list of lists while centroid() finds the centroid for each cluster. Finally, weighted_frontier() is the most critical function where all the logic is applied. This means that we compute the size, the distance from the robot (of the frontiers and centroids) here with the function returning the coordinate goal for the robot. Those final coordinates will be sent by the publisher publish_goal() to the agent which will execute the strategy function.

The strategy function is as decribed in the following flowchart :



The strategy function starts with best_frontier or best_centroid sent by the centralized system. Then, if we don't have a path and a target, we compute A* to get a path. If we have a path but not a target, we take a target from the path. Finally, if we have a target, we wait until it is reached and reset the variable.

One crucial assumption that we made is that A* could find a path through unknown spaces. This assumption is made because A* is computed on the local map of the agent but not on the global map which is located on the map_manager.py side. Therefore, A* could fail on the local map even through a path existed. This assumption is further backed by the fact that our map is small (so we have reduced unknown spaces) and according to the cost functions we used, the robot will always go to the closest frontier or closest and biggest frontier. Therefore, if the robot attempts to reach a more distant frontier, it is very likely that it has already explored its immediate surroundings and that its local map has merged with another robot's map, allowing it to find a valid path through the now-expanded known space.

A. Frontier-based strategy

We implemented several versions with different coordination levels and cost functions to see which has gotten us the best results.

1. Version 1

The rawest version that we had was a non-coordinated version with each robot going to the nearest frontier in euclidian norm. This caused problems with distance through walls and already explored spaces. All of this is detailed in the table below :

Strategy	Goal/Cost function	Time	Problem found
Nearest-based frontier allocation	Closest frontier in Euclidian norm	5 to 10 min	Robot explores known spaces

2. Version 2 and 3

We improved the previous algorithm by adding coordination to the robots. To achieve this purpose, first we created a cost matrix with each row representing a robot, each column a centroid and each value is the cost (normalized distance and/or size) just like below:

	Centroid 1	Centroid 2	Centroid 3	Centroid 4
Robot 1	0	0.33	0.66	1
Robot 2	1	0	0.33	0.66
Robot 3	0.66	1	0	0.33

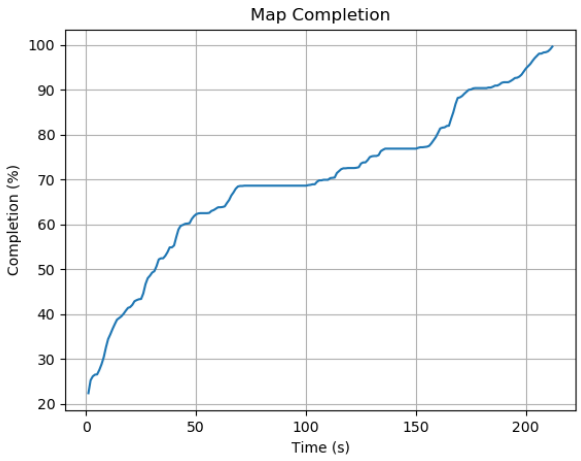
In this case, robot 1 will take centroid 1, robot 2 centroid 2 and robot 3 centroid 3. In order to optimize this cost matrix to get the minimal total exploration cost and avoid same centroid allocation, we applied the Hungarian algorithm through the scipy library.

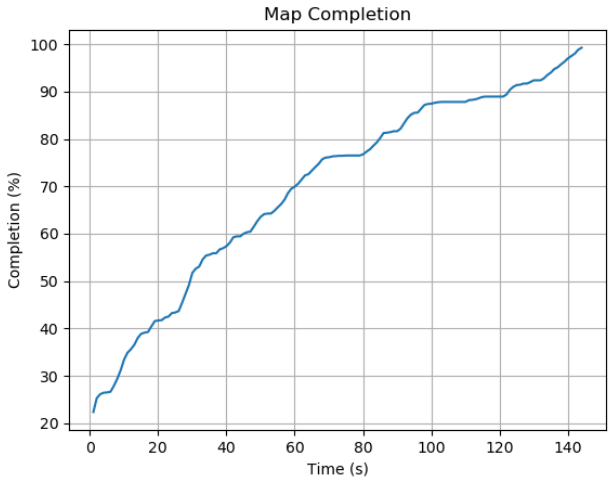
The results found are described in the following table and the graph on the left side represents version 2 while the other version 3:

Version	Strategy	Goal/ Cost function	Time	Problem found
Version 2		Cost function = distance (euclidian norm)	3 to 4 min	Robot explores known spaces

21/04/2025

Version 3	Rank-based allocation	Cost function = distance (euclidian norm) – frontier size	2 to 3 min	Robot explores known spaces



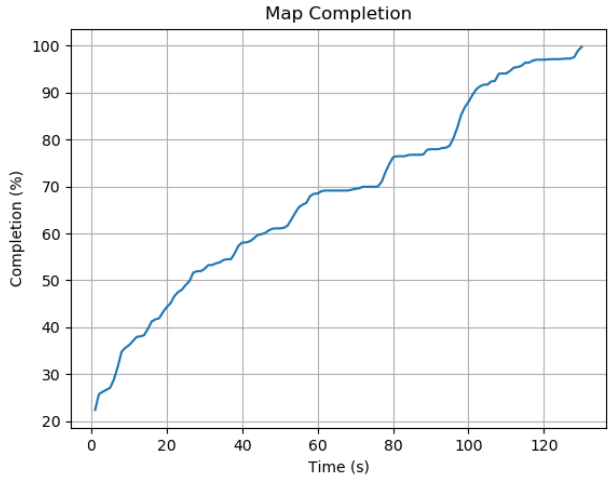


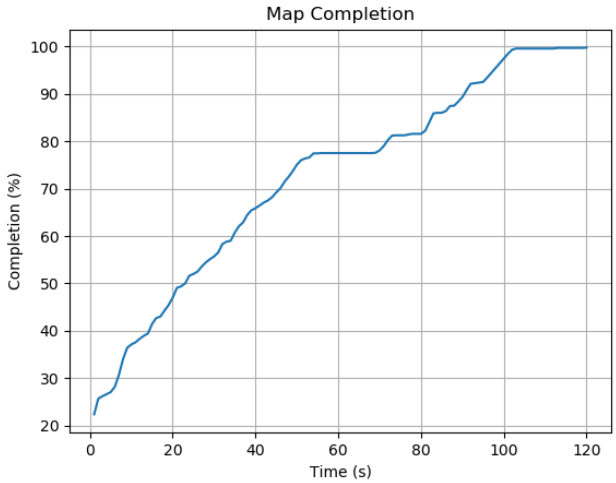
3. Version 4

In this final version, we changed the way the distance was computed for the cost function. Indeed, we took the A* path distance instead of the euclidian norm, which gave us more accurate results.

Strategy	Goal	Time	Problem found
Rank-based allocation	Closest frontier in A* distance	2min30 or less	Robot could get stuck

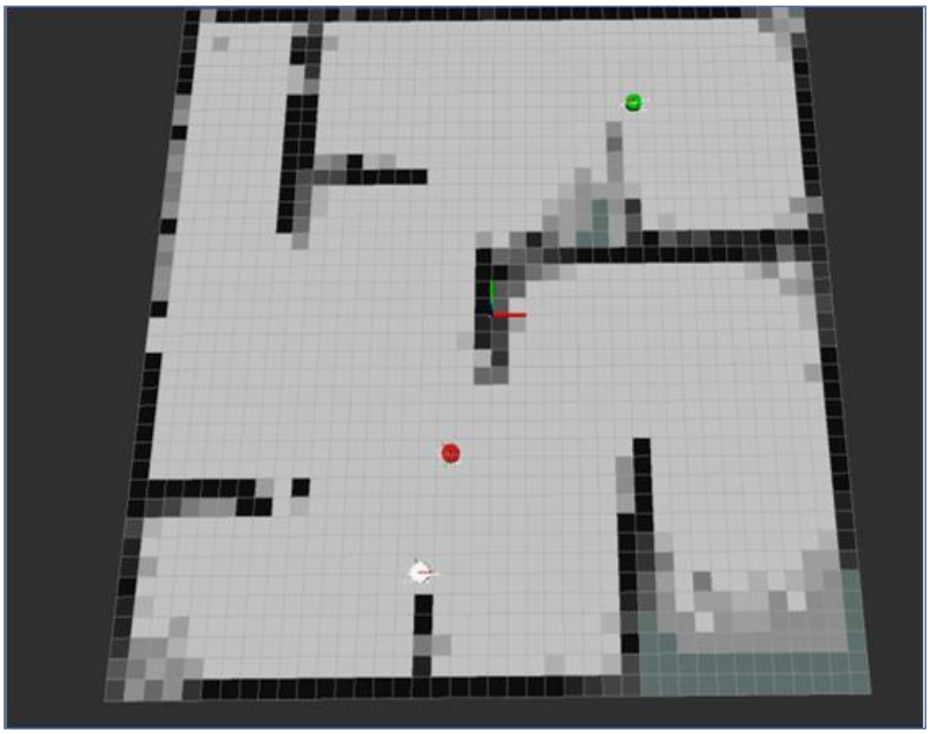
We obtained the following graphs for two cost functions, the first one with the cost = A* distance and the second one with cost = A* distance – frontier size.





We can add that the graphs showed randomness and could change a lot depending on the simulation so further testing is needed.

B. Heat-map Strategy



The heat map used by the robot represents its understanding of the environment as a grid where each cell holds a numerical value that evolves based on sensor readings. Initially, all unexplored cells have a value of -1. As the robot explores and its laser sensors pass through a cell without stopping, this cell is assigned an initial value of 50. Each time a laser traverses such a cell, its value decreases by a fixed amount (typically -5), simulating increased confidence that the space is free. This decrement continues until the value reaches 0, at which point the cell is locked and considered definitively free. Conversely, if a laser ray stops on a cell, indicating a potential obstacle, the value of that cell increases by a detection increment (usually +5). This increment continues with each detection until the value reaches 100, at which point the cell is also locked but now classified as an obstacle. This continuous adjustment of cell values allows the robot to refine its map over time, distinguishing between free space and obstacles through the accumulation of sensor data.

One of the main advantages of this heat map approach is that it allows the robot to avoid navigating through areas where the cell values are uncertain, thereby reducing

the risk of getting stuck. By assessing the average certainty values of the surrounding cells, a safety threshold can be defined to guide the robot through zones with reliable information. This ensures smoother navigation and decision-making. Moreover, once there are no more unexplored cells (i.e., cells with a value of -1), the robot can be directed toward areas where the confidence in the map is still low. This strategy enhances the precision of the overall map by refining partially explored regions. Another important benefit of this method is its ability to distinguish between permanent obstacles, like walls, and temporary or moving obstacles, such as other robots. Since moving objects typically do not remain in place long enough for their cells to reach the maximum obstacle value of 100, they are not immediately classified as true obstacles. As soon as they move, the cells they previously occupied can be updated again by passing laser rays, reducing their value accordingly. This dynamic behavior allows the map to remain flexible and reactive to changes in the environment. To further improve the distinction between static and dynamic obstacles, the increment or decrement values applied during detection can be adjusted. Slower updates mean that cells require more consistent evidence over time to be considered definitively free or occupied, thereby increasing the robustness of the system against false positives caused by transient objects.

This heat map approach offers several advantages: it increases navigation safety by avoiding uncertain areas, improves map precision through targeted exploration, and distinguishes effectively between static and dynamic obstacles. However, this method is more complex to implement and can be computationally demanding, as uncertain cells must be continuously updated based on sensor input. This constant adjustment requires careful tuning and sufficient processing power, especially in large or dynamic environments.

In the current implementation, most of these advanced features have not yet been developed due to time constraints. However, the core visualization of the heat map is fully functional and provides a solid foundation for future projects. It allows for clear observation of how the environment is progressively mapped and interpreted by robots. Work on this system is still ongoing, with efforts currently focused on improving several aspects, including the distinction between static and dynamic obstacles, and refining the confidence-based navigation logic.

VII. Conclusion

This project achieved fast and efficient exploration of a 40x40 unknown environment using three autonomous robots in ROS2 and Gazebo. By iterating on frontier allocation strategies, V4 with A* path cost proved the most effective, consistently completing the task in under 2 minutes 30 seconds with minimal overlap and idle time.

The heat map approach added potential for refining exploration, especially in partially known or dynamic areas. It introduced a confidence-based view of the environment that could be valuable for future improvements.

To go further, several enhancements could be explored:

- Better handling of dynamic obstacles using temporal filtering
- Path smoothing to improve motion efficiency
- Decentralized decision-making for scalability
- Integration of heat map data into path planning
- Real-world testing to validate the system beyond simulation

Overall, the project demonstrates an effective framework for collaborative autonomous exploration, with decently strong results and several promising directions for future work.