SLAM Challenge Report

(ROB 330)

Alexandru Otlacan (aotlacan) and Akash Sarada (asarada) Tian Yi Yang (tianyiy)

Overview

This report documents our implementation and results for the Exploration Challenge. All tasks are detailed in the following sections, including obstacle distance grids, A* planning, frontier identification, exploration behavior, escape logic, and a final reflection on our overall performance.

Task 1: Obstacle Distance Grid

The obstacle distance grid provides the robot with a map of the area it has explored, marking walls and obstacles along its path. This allows it to not only create a map around the area it is traversing, but also use said map to find the shortest distances between points without running into any obstacles. It allows for a more efficient search algorithm as it only needs to check if a cell is blocked along the shortest path, and navigate around it, instead of needing to check each orientation and location of each obstacle.

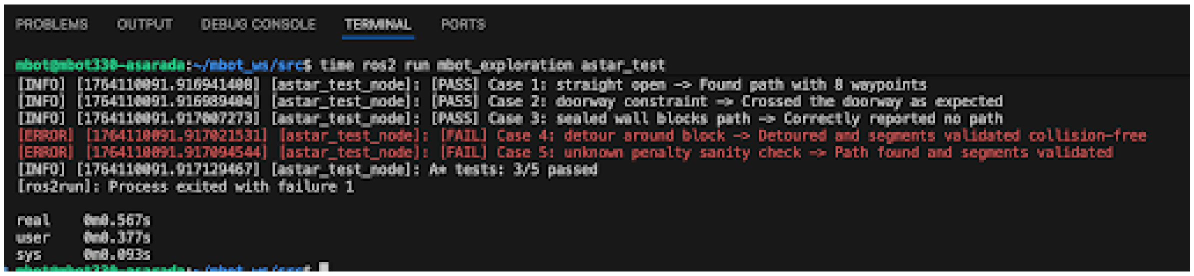
One assumption is that the area is mostly static, as the robot stores the locations of every obstacle as it maps, and only maps an area a few times. Additionally, another key assumption is that the robot can be modeled as a point, handling the physical size of the robot by growing the obstacles. This allows the robot to find the most efficient way to traverse through the grid while reducing the complexity.

The advantage of using an 8-way expansion is that it allows the robot to traverse in diagonal directions, which are usually the most efficient paths between any two points. This allows the robot to move more cautiously, without sacrificing quality or speed. Diagonal paths also allow for the use of Euclidean distance, which is computationally light while being extremely effective for traversal.

Task 2: A* Planning

The metric used for the heuristic cost was the Euclidean Distance, which was chosen due to its efficiency and performance. The Euclidean distance calculation allows the robot to take the most efficient path toward the goal, as the distance between any two points is a diagonal distance. Therefore, using the Euclidean heuristic ensures that the robot takes the shortest path possible. Furthermore, since we choose to use the 8-way expansion, we not only had the data, but were prepared to traverse diagonally, allowing us to utilize the full benefits of the Euclidian distance heuristic.

There are alternate heuristic cost functions we could have chosen, like the Manhattan Distance or the Diagonal Distance. The Manhattan Distance is used for a 4-way expansion, as it only checks the cardinal neighbors and calculates the distance between the cell and the goal. This allows it to efficiently find the next step that would move the robot closer to the goal. The diagonal distance is used for an 8-way expansion, moving in the direction of the largest difference between the x and y coordinates of the cell and the goal. This allows it to move in a diagonal direction without needing to check all 8 cells.



```
mbot@mbot330-asarada:~/mbot_ws/src$ time ros2 run mbot_exploration astar_test
[INFO] [1764110091.916941408] [astar_test_node]: [PASS] Case 1: straight open -> Found path with 8 waypoints
[INFO] [1764110091.916989404] [astar_test_node]: [PASS] Case 2: doorway constraint -> Crossed the doorway as expected
[INFO] [1764110091.917007273] [astar_test_node]: [PASS] Case 3: sealed wall blocks path -> Correctly reported no path
[ERROR] [1764110091.917021531] [astar_test_node]: [FAIL] Case 4: detour around block -> Detoured and segments validated collision-free
[ERROR] [1764110091.917094544] [astar_test_node]: [FAIL] Case 5: unknown penalty sanity check -> Path found and segments validated
[INFO] [1764110091.917129467] [astar_test_node]: A* tests: 3/5 passed
[ros2run]: Process exited with failure 1

real    0m0.567s
user    0m0.377s
sys     0m0.093s
```

Figure 1: A* test script output, with runtime.

Table 1: A* Test Runtime and Results

Test Case	Passed	Runtime (real / user / sys)
1	Y	0.567s / 0.377s / 0.093s
2	Y	0.567s / 0.377s / 0.093s
3	Y	0.567s / 0.377s / 0.093s
4	N	0.567s / 0.377s / 0.093s
5	N	0.567s / 0.377s / 0.093s

We had the last two test cases fail but it worked in real life. Even though these two test cases failed in the test script, there is a lot more noise and real sensor data that the robot is detecting in real time and then being able to plan based on that. So when running A star tests it's possible that the robot runs in unsafe conditions theoretically but in reality it never does pass through that.

If a rotational movement were to be added, we would add a factor to the action cost (g(x)), which would be formulated through the multiplication of the amount required to turn as a proportion and a tunable coefficient. This would allow for the robot to understand the energy it would require to turn and assess if it is the most efficient route to take.

We did not add any additional costs to optimize the planning performance, as the performance given throughout the lab guides was very efficient and performant.

Task 3: Identifying Frontiers

As per the lab guide, "A frontier is defined as the unknown space adjacent to known space on a map." Therefore, we determined that a frontier would exist if there exists a cell where the cell itself is free, but there is a neighboring cell which is unknown, and no neighboring cell which is not an obstacle. This would mean that this cell is the last cell before an unknown cell, making it a frontier cell. Once these cells are found, they are grown to combine with other frontier cells, creating a larger, connected frontier throughout the entire frontier space. This ensures that when the robot navigates to the frontier, it navigates to the middle, ensuring that the most area is mapped.

One limitation of the method we chose would be the lack of resistance to sensor noise. Since this implementation is heavily dependent on the sensor values and mapped cells, it is therefore vulnerable to sensor noise. Thus, this method may generate extremely small frontiers that the robot navigates unnecessarily. One alternative method would be utilizing a gain-based frontier detector, which would view the current map and estimate the additional cells mapped in the unknown space. It would then navigate to the most promising estimate, ensuring that the robot moves in the most efficient way possible.

Green and blue colored squares are the frontiers and green is the one that is being explored right now, as shown in Figure 2 on the next page.

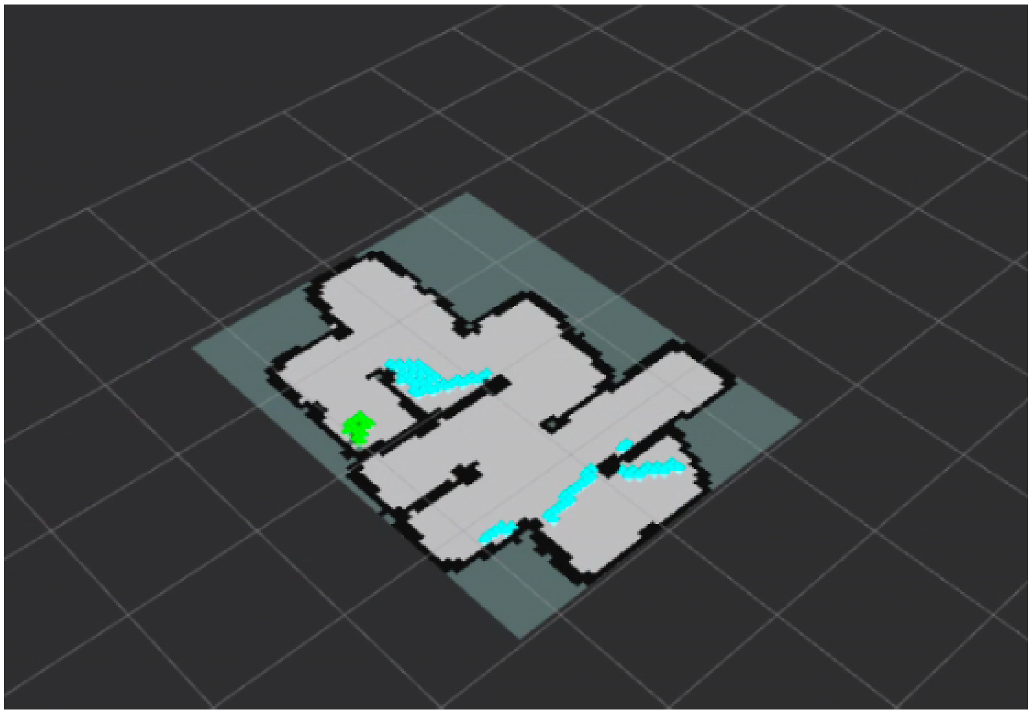


Figure 2: Frontier visualization.

Task 4: Exploration

Driving to the center of the centroids is not a robust solution, as the centroid of the unknown space may not be reachable, or it may be inside a tightly constricted area, creating a difficult scenario for the robot to navigate in. Since the robot navigates to the centroid, it does not consider the closest part of the frontier to the robot, which may not only be easier to navigate to, but also the closest point where the robot can start mapping. Furthermore, driving to the centroid of each frontier means that it treats every frontier equally, not exploring the frontier that has the most promise. Overall, this means that the robot uses more energy to map the area, and may even attempt to navigate to a more dangerous area. As a result, as we saw in many trials during the challenge, the robot did not always take the most effective path to explore and took a lot more time to create the whole map.

We did not change this approach, as the given approach of navigating to the centroid of the frontiers was good enough, as the walls were not tight enough, and most of the unknown space was reachable. There were a few moments where the robot was not able to explore the whole maze, but it was not large enough to warrant changing the basic, yet reliable approach.

We did not use a fixed-radius or node-count cutoff. The search was constrained by the grid size and safety-based obstacle threshold of 0.05m from occupied cells. Cells closer than this threshold were flagged and treated as blocked. Additionally, a safety weighting factor of 0.6 biases the path away from obstacles. There is no explicit node expansion limit or radial cutoff as the search runs until the goal is reached or no valid cells remain.

Task 5: Escape

After the removal of the edge, we found the new frontier to drive through by resetting the map, allowing the robot to start exploring the maze again, navigating toward the farthest frontier. This meant that the robot found the frontier to escape through by finding the largest and farthest frontier and navigating toward that first.

We did not modify the state machine logic, keeping the same state machine found in Lab 12. We kept the same logic as it was extremely simple and clear, moving from the initialization to the exploration stage, then exploring and mapping the maze before returning home. Then, it finished with the escaping stage, as it finds the farthest frontier and escapes through it.

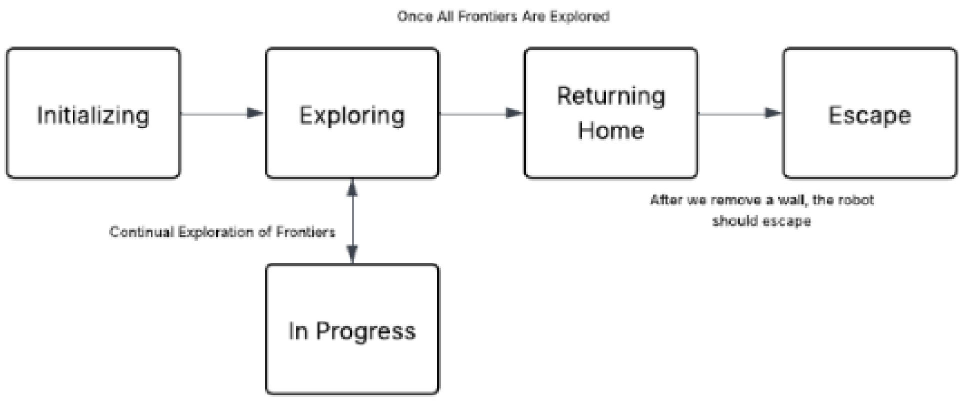

Once All Frontiers Are Explored
Initializing
Exploring
Returning Home
Escape
After we remove a wall, the robot should escape
Continual Exploration of Frontiers
In Progress

Figure 3: State machine diagram, including escape.

Reflection

One thing we would like to improve on is tuning and testing some of the different variables during our practice runs. On our best run, we were not able to escape due to a simple error in the safety distance calculations. After we attempted to update this, we slightly clipped a wall while navigating around one corner. From this, we know that some of our safety variables could be tuned slightly more, helping us achieve a better score. Furthermore, we would attempt to better assess the removal of the wall. This is because our current way of seeing the wall being removed was by resetting the map, which was allowed for the challenge, but not a standard way to address the issue. Therefore, we would want to change the way we assessed the wall by slowly phasing out the older map values for newer ones, detecting that the wall was removed, and creating a frontier to explore outside of it.