Robot Planning and Its Application Project Report

Bussola Riccardo ID: 240080, Zilio Nicola ID: 240187

12 January 2024

0.1 Approach

Starting from the provided environment for the Shelfinos, we've designed and developed our project as a generic framework to explore, test, and execute different path-planning algorithms. Our main problem of interest is "Coordinated Evacuation" and what we will describe will refer to it. Furthermore, we have adopted what was implemented to also provide a solution to the problem of "Target Rescue".

The framework provides a GUI component (*env_map*) that enables the problem's visualization, and a component that manages the roadmap execution (*orchestrator*). These two nodes provide a common way to display and actuate the solution computed regardless of the path-planning algorithm adopted. Based on these components, we have implemented two roadmap generation solutions: the first one adopts the construction of a generalized Voronoi diagram to search for the best path using the Dijkstra algorithm, and the second one implements the RRT* algorithm with built-in Dubins' path planning.

The specific reasons behind these implementation choices, along with their pros and cons will be treated in the next chapters. We can reason on the fact that having both combinatorial and sampling-based planning is a good way to evaluate and find out the characteristics of these two realms.

0.1.1 The path-planning framework

We have decided to organize the system as a framework to maximize the reusability of our components and then fasten the development of new solutions. The integration of a new roadmap planner requires the only implementation of a node that, once subscribed to the environment's main topics (/obstacles, /gate_position, etc...), publishes the calculated paths on the dedicated topics. The orchestrator will check and correct the presence of path collisions and then send the final paths to the nav_2's dedicated node.

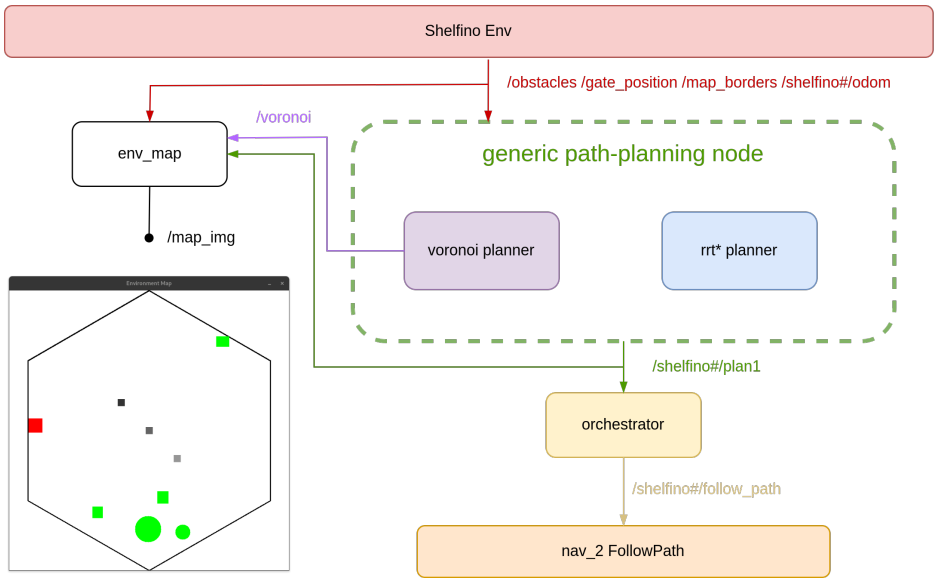

Figure 1: Path planning architecture built on top of the Shelfinos' envirmonment

Fig. 1 provides an overview of the path-planning framework, it is possible to observe how a new planner could be easily added without the need to introduce changes of any type to also inherit the collision avoidance feature.

Environmental map GUI (env_map)

This node provides a set of visualization tools that enable a more clear understanding of the environment state. Compared to RVIZ is easier to use and also enables more custom visualization such as the Voronoi diagram provided by one of the planning nodes. The GUI is made using OpenCV, it provides drawing functionalities for obstacles, gates, Shelfinos, and paths. A functionality that is developed but not used is the publishing of the environment state as an image through a dedicated topic. This functionality is intended for all the possible nodes that adopt a visual

state representation to compute the paths (our idea was to develop also an RL-based node for the path-planning, discarded after for lack of time)

Orchestrator Node (orchestrator)

This node is responsible for avoiding robot collisions with other robots. Once all the paths are received, for each robot it calculates the starting delay in case of a possible collision. This is made possible by two factors: robots move at the same constant velocity (0.2 [m/s]) and all the paths are discretized in steps of 0.1 [m]. The collision avoidance algorithm is heavily inspired by the X* algorithm: it starts iteratively from step 0 s_0 to look for collisions (robot distance $<$ safe distance e.g. 0.5[m]), if a collision at some step s is found, a delay time-step is added to the shortest path of the two involved, then it restarts from s_0 and it repeats all the steps until no conflicts are founded. The reason behind adding a delay to the shortest path comes from the fact that adding a delay to the longest path in a collision might generate an increment in the total time to evacuate the room. Then the delay time steps are converted into seconds and a corresponding timer is activated for each path. Once the timer is triggered it sends the path to the nav2 Node.

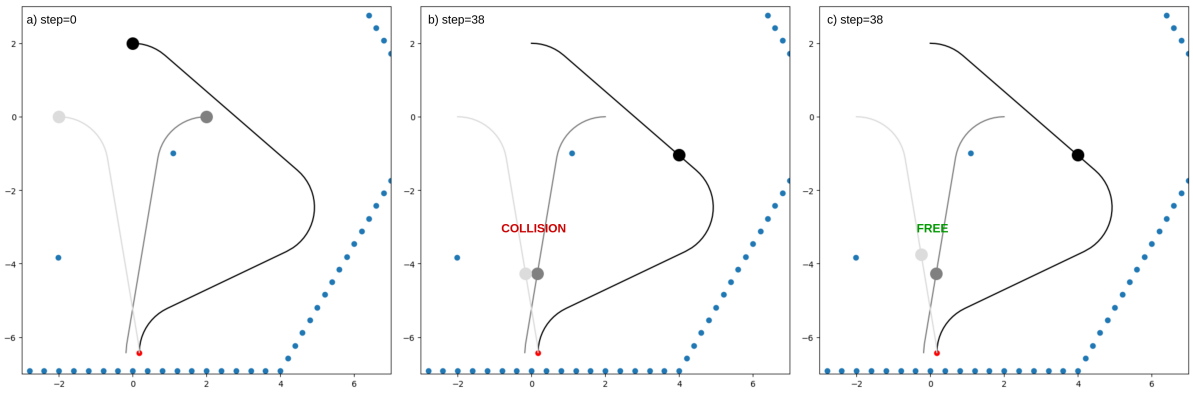

Figure 2: a) initial state, b) a collision is found at $s = 38$, c) the collision is avoided by adding a step delay to the Shelfino 2

Fig. 2 shows the effect of the orchestrator collision avoidance system. Since this node assumes the velocity to be constant and the path to be discretized with the same distance step, it works independently from the path-planning node that has sent the roadmap, as long as the node follows these two characteristics. Finding a collision between the paths is rare, so in the example of Fig. 2, the collision was created by assigning the same path to two Shelfinos and then swapping the coordinates x along the y-axis of one of them.

0.2 Problem 1 Coordinated Evacuation

Approach Followed

As described in the previous section, our main focus was the "Coordinated evacuation". In this chapter, we will expose all our development choices, our problems, and our results. Before discussing these, we must mention a preliminary work that we have done ahead of receiving the Shelfino's ROS2 environment. A grid-based approach with an A* algorithm was developed as a proof of concept in Python. After this small deviation, we will discuss the final solutions that involve a Dijkstra search algorithm inside the nodes generated by a generalized Voronoi diagram, and the RRT* algorithm applied in combination with Dubin's curve planning.

0.2.1 Grid-Based Approach

The main idea behind the Grid-Based approach was to use an occupancy grid in order to express the status of the environment in which the robot has to move. In order to achieve so we proceeded step-wise:

- The first step consists in the creation of the grid map. Since we are not always dealing with square maps, what we did was to inscribe the shape of the map into a square, in order to be able to divide the map in square

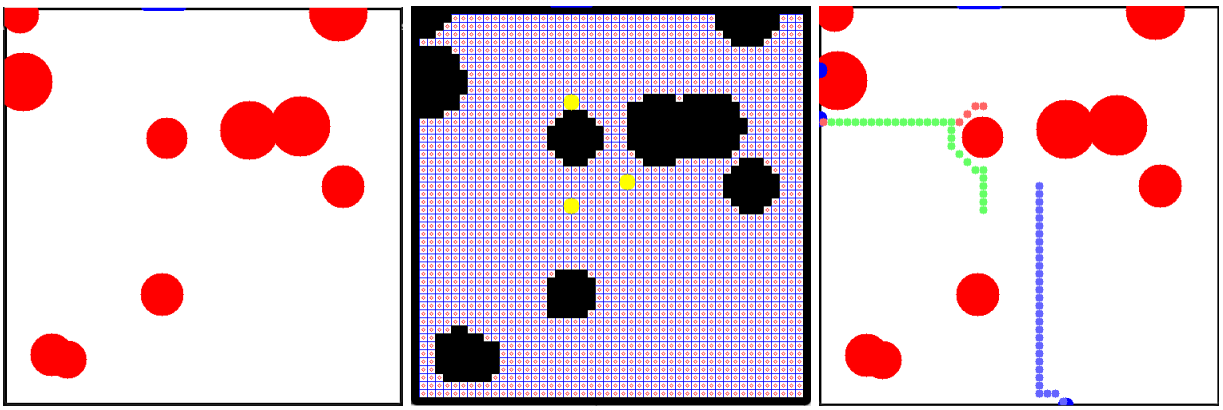

Figure 3: 1.1 Map with obstacles, 1.2 Map with grid,robots and obstacles, 1.3 Path found for each Shelfino

cells. Once the map was subdivided into a grid (in our experiments we tried sizes of 10x10 and 25x25), we decided to treat the grid as an occupancy grid, hence for each cell of the grid, we marked the cell as occupied if it was containing at least one part of an obstacle or the border of the map. In order to avoid collisions a priori, when we set the occupancy of a cell, we enlarged the borders of the map of a distance that was equal to the diagonal size of the Shelfino. For what regards the obstacles on the other side, we decided to transform all obstacles into circular ones, in this way we define an area of occupation of the obstacle. Also for those obstacles we augmented the radius of the circle by the diagonal of the Shelfino in order to avoid collisions a priori.

- Once the grid was created, taking as reference the center of the cell, a graph was build on top of it. While creating the graph we both tried a 4-directions and 8-direction graph build, meaning that for each cell in the grid, the graph will have an edge to an other cell if the difference in coordinates is (0,1),(0,-1),(1,0),(-1.0). In the second case in addition to the four possible movements, we added the possibility to move diagonally, so where the difference is (1,1),(-1,1),(1,-1),(-1,-1). In the second approach, when the movement is diagonal we assigned an higher weight to the edge, since we want to move diagonally only when doing this is more convenient with respect to combinations of horizontal and vertical movements.

- Once the graph was created, we placed our Shelfino on the closest node with respect to its position, and then we applied A-star [2] algorithm in order to find the best path that allows the Shelfino to reach the goal.

- After the path for each Shelfino was found, we implemented a priority mechanism in order to check if at anytime collisions may appear. We aligned all the paths and we checked if for each time-step, one or more Shelfinos are occupying the same node on the graph. If this happens then the Shelfino that has a shorter path was delayed on three time-steps in order to avoid conflicts. The choice of delaying the one with the shortest path comes from the fact that if we delay the longer one than the total time of the evacuation may increase, on the other hand if we delay the shorter one it will not have that much impact on the total time.

While testing this possible solution in Python, after we had a working prototype, we realised that this solution was creating us lots of problems, both in terms of the time required to generate the graph, the time required for the search and the complexity of the paths. In addition to this the Python library that we were exploiting didn't have a transposition in C++, hence this solution was abandoned. Hereafter we will show some screenshots of what was happening in that approach.

0.2.2 Voronoi-Based Approach

Motivation of The Choice

The generalized Voronoi diagram, also known as the maximum clearance roadmap provides a set of points along with their connection information (graph) that guarantees the maximum distance from obstacles in the map. Since it is a combinatorial-planning method, once the roadmap composed of vertices is created it can efficiently be reused to invoke multiple queries (so multiple robots can benefit from that). Since the roadmap obtained guarantees to be furthest from obstacles, not only by staying away within the minimal safety distance, the path is sub-optimal

concerning the length.
It is designed to maximize safety, and to then find the shortest path we have adopted the Dijkstra search algorithm. The reason behind choosing Dijkstra instead of more efficient search algorithms such as A* is its optimality characteristics, in this way, we know that concerning our roadmap, which is suboptimal in the lengths, we will result in an optimal path concerning the obtained graph.

The node relies on the Boost C++ library which provides an efficient implementation of the Voronoi diagram. Then, our search algorithm adopted the KDTree structure to quickly query the nearest neighbors. Despite the diagram providing a roadmap with maximum clearance, it is not guaranteed that a path satisfies the needed safety distance. To avoid collisions, our algorithm, during the post-processing of the diagram checks for possible intersections between nodes and obstacles. The obstacles are discretized along their perimeters as a set of points with a distance smaller than the robot radius (0.353 [m]). This discretization step is also needed to create a user-defined structure required by Boost to generate the diagram. Once the vertices of the Voronoi are founded, the search includes the start and the goal point and Dijkstra produces the optimal path.

Considering the planning of the Dubins curve as a built-in step of the path generation in this case is highly nontrivial. For this reason, the generated path are simple linear connection between two nodes graph. We had to dive deeper into the problem analyzing also a state-of-the-art approach [4] but we decided to not try to implement their solution since the complexity of the algorithm is high and might probably increase noticeably the computation time. Finally, once the path is computed, it is discretized in steps of 0.1 [m] to satisfy the orchestrator's requirement and obtain its collision avoidance capabilities.

Difficulties Encountered

The first difficulty to mention was the usage of the Boost library, the documentation is not very clear and the site lacks examples. The main issue was also the default requirement to construct Points and consequentially Segments struct with int coordinates. This is not good since we deal with real value coordinates. As a workaround, we adopted a class called CoordinateMapper that we have developed mainly for the env_map Node. This class provides methods to switch between real-world (Gazeboo) coordinates (double) and image coordinates (int). The second annoying problem was the spawning of obstacles inside the wall, this generated heavy artifacts in the Voronoi diagram since from the documentation of Boost, no segments must intersect. Despite that, our search algorithm still works. The final main problem was the KDTree library, after some trials with various implementations, we've adopted and modified to our needs the library provided by J. Frederico Carvalho https://github.com/crvs/KDTree. Unfortunately, this library is slow and lacks functionalities that we have added such as providing the distance between near nodes.

Results

This nodes provides a good solution to our path-planning problem. Unfortunately, though it is slow and intuitively highly dependent on the number of nodes. The average path planning accounts for a total of 6/9 s which in our opinion is not that great for a good path planner.

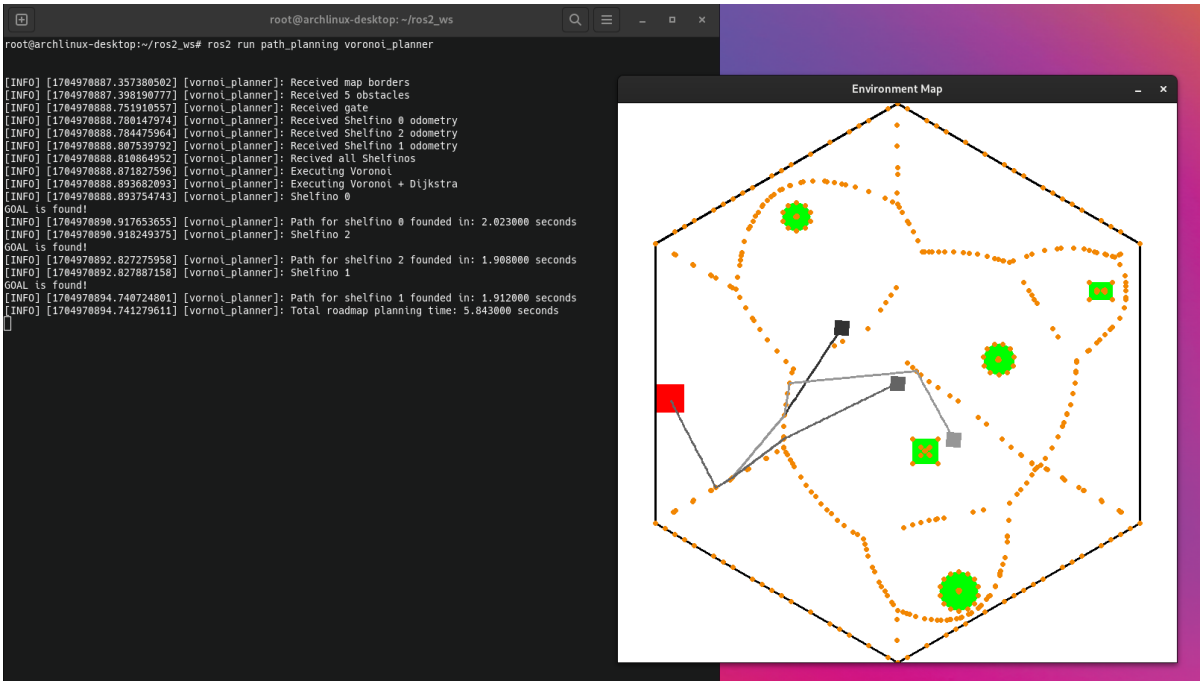

Figure 4: Voronoi path-planning and its execution time

0.2.3 RRT Star-Based Approach

Motivation of The Choice

After realizing that the Voronoi approach works but is slow and lacks Dubins planning we have decided to implement a new faster algorithm. That's why we have opted for a sampling-based planner, more precisely, RRT* which is demonstrated to be asymptotically sub-optimal. Instead of generating new nodes as a linear connection between two points, our algorithm adopts Dubins to plan between two configurations. This is great since allows us to plan directly in a viable motion way for our robots. To avoid collision between robots and obstacles, in this implementation, we have approximated the obstacles as a circle with a radius equal to half of the diagonal (for polygonal obstacles, cylindrical remains as the provided circle description). The map borders are discretized as a set of small circles with a distance sufficient to avoid the passage of the robot. Our algorithm during the search will check if a new node and the path calculated interests an obstacle providing in this way criteria to discard possible threatening paths.

Difficulties Encountered

This new method shows high improvements in terms of speed but introduced some hyperparameters to tune such as Dubin's curvature ratio, and not always find a viable solution although might exist. As a workaround, the algorithm, after a maximum number of iterations, starts again the search from 0 (for the single robot) since the initial node might have produced an inefficient configuration. This is the main drawback of using sampling-based planning. The second problem that we have solved is the speed of KDTree implementation, we've removed the library and we have adopted a couple of methods to search faster inside the tree structure generated using the RRT* algorithm.

Results

Compared to the Voronoi method RRT* can provide viable paths in a few tenths of a second once the way is not occluded, a couple of seconds instead if obstacles are in the way. z‘

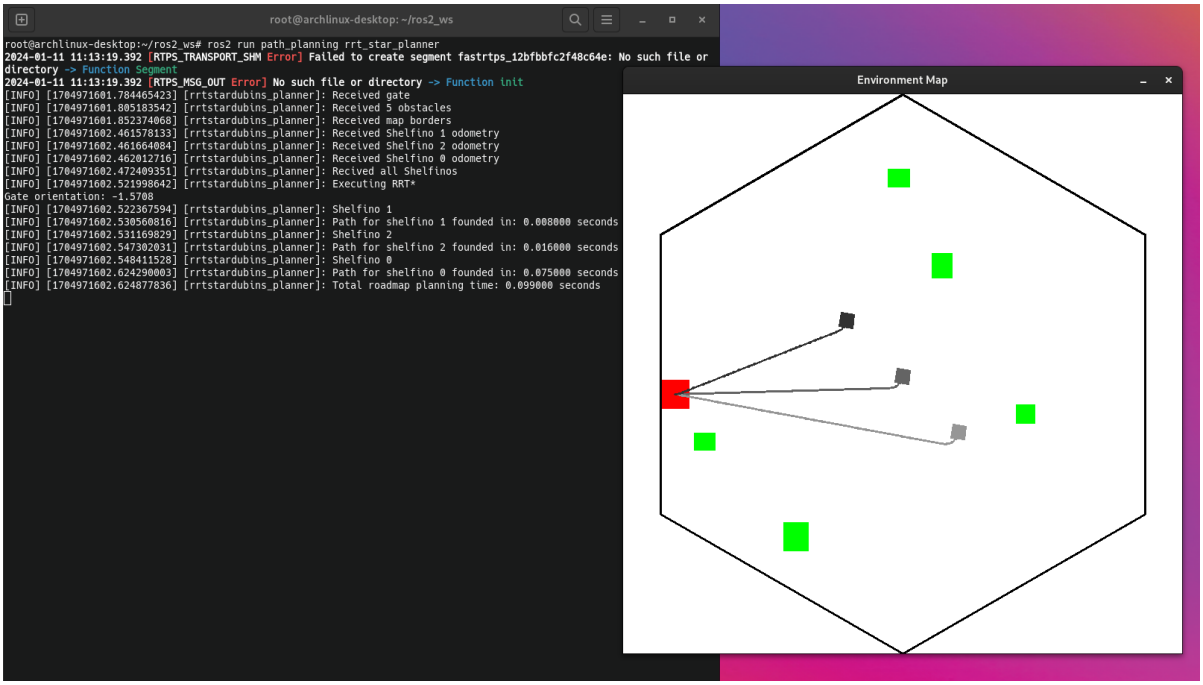

Figure 5: RRT* path-planning and its execution time

0.3 Problem 2 Target Rescue

0.3.1 Approach Followed

This problem has been tackled after we have solved the first one. So components such as the RRT star Dubins planner were already implemented. The first step we did in order to solve this problem was to analyze the requests of the problem and the task we wanted to accomplish. At first we created all the supports in order to better understand the environment. We built some subscribers that are able to read from the *victims* topic. Once we have read the data we proceeded in processing them. One of the places where those data come useful is the *env_map* node. The purpose of this node is to provide a visual 2D representation of the environment. In this representation the victims to be rescued are represented as blue circles having radius of 0.5. Once the representation of the problem had been build, we started thinking about the way we can solve this problem. Since with the RRT star algorithm we are able to find paths between two points inside of our environment we proceeded as follows:

- First we build our map, as in the problem before, by reading borders, obstacles, victims and shelfino positions.
- Once the map is built the first step is to try to find a path to the goal. We considered very important to calculate the path to the goal as the first thing, since if just reaching the goal from the starting position requires the majority of the time we have, than rescuing any victim will be nearly impossible. If the path to the goal is found then we already have our baseline in terms of time.
- We then try to find a path to reach the victims. There is no guarantee to find the path, since the RRT star as we are using it, is not meant to continue the search until a solution is found. This had been chosen in order to reduce the total computation time, since also the computation time for the road-map is a key point in the whole process. Once a feasible path has been found, we proceed to also compute the path from the goal to the victim. This is a crucial part of this phase, since we can say that we are able to reach a victim in the given amount of time, but we have no guarantee to reach the goal in time. Since not reaching the goal in time will result to null score than we compute the whole path start-victim-goal and check if the total duration of the path is lower than the total time. Throughout this whole process, we keep track of the path having the best value, this will allow us to achieve the maximum score given the victims in the scenario.
- After the path for the first victim of each possible path is computed then what follows is a greedy-like approach. For each one of the remaining victim what we did was to try to find a path that links the first victim of the

path to the one we are taking into consideration. If the path first_victim-second_victim-goal is feasible with respect to the time and the algorithm is able to find the path (considering collisions and reachability) then we update the path according to best value. This procedure will then be applied until all the victims that are in the environment are taken into consideration.

- When the final path is ready, we send it to the *orchestrator*, that will process it in order to create a path that can be executed on the Shelfino. For this particular case, the orchestrator that had been built for the first project has been simplified since we are dealing with just one Shelfino and the collisions between multiple agents should not be checked.

0.3.2 Motivation of The Choice

The main reason why we chose to use this type of approach in order to solve the problem starts after we have already solved the first one, hence we already tried to implement all the methods we described above and RRT-star [3] was the one that in our opinion would have worked better in this case. We started to develop out approach knowing that we were able to build Dubins paths between two locations. Given those assumptions we started at possible solutions. The one that was convincing us more was to develop a greedy-like approach. The problem that gave us the inspiration to do so was the knapsack problem [1]. After a brief discussion we decided to take inspiration from this sort of problems and to develop a greedy approach, that is not the best in terms of scalability, but for the simple cases we are considering (3 victims) it is not that demanding in terms of complexity of the algorithm, since the algorithm has to go over all the possible combinations of the victims.

0.3.3 Difficulties Encountered

We can summarize the main difficulties we have encountered as follows:

- **Spawn problems:** Since the spawn of the obstacles and of the victims are random, there are some cases in which our algorithm fails to achieve the goal. At first if the goal is surrounded by obstacles and the robot is not able to pass through them, then the problem has no solution. Secondly if at spawn there are obstacles in the surrounding of the robot (in particular in front) than it is very difficult for our algorithm to find a solution, since we suppose that the robot is not able to rotate on its place (we force it to use Dubins maneuvers for the motion). The third case is related to the position of the victims, if they are near a wall, or surrounded by obstacles the algorithm may not find a solution, resulting in empty plans.
- **Implementation problems:** Since the victims are expressed as pointers inside our code, the management of those pointers is not always trivial, in particular when erasing ad pushing back pointers containing the victims we encountered lots of problems.
- **Randomicity of the algorithm:** since that the algorithm is using are randomly generated, the algorithm is not always finding the path in a limited time of iterations, so sometimes the planner returns that it is not able to find the path despite with some more iterations it will find one.

0.3.4 Possible Future Works

One possible way to improve this work is to better develop the approach angle for the victims. In our work we decided to approach the victim always with a fixed angle of 45 degrees. The improvement may be to find for each victim that the robot wants to approach, the best angle in order to reduce the duration of the movement, the computation time and to ensure that a possible path is found.

0.3.5 Results

Hereafter we will present the results of this part. The plots 6 hereafter will represent the time required for 10 test-cases in which we run the algorithm.
As we can see from the plots 7 , except from the third experiment, in which the scenario was built in a way such that it was not possible to achieve the final task, the algorithm is able to find a solution for each of the scenarios. For what regards the duration of the paths, it is highly dependent from where the victims

are placed and how the path is built in order to reach them. In fact it is possible that in order to approach the victim with the desired angle, the path shall be elongated in order to perform the Dubins maneuver that allows the robot to turn and reach the center of the victim. For what concerns the time that is necessary to calculate the path, also this is strictly scenario-dependent. In fact since the algorithm for path finding evolves in a random way, there is no mathematical model in which we can have an expectancy on the duration. Also if the algorithm is not able to find the path at the first iteration, it may require more than one iteration to completely solve the goal, hence it will require more time.

The image below will present what the problem will output. What we can see from the image above are three views. The first, on the left, is the 2D visualization of the problem. The obstacles are represented in green, the victims in blue, the gate in red, and the Shelfino is the little gray square. The path that the Shelfino will follow is represented by the continuous dark grey line. The second view on the bottom right is the *nav2* trajectory planned for the robot, and the other is the Gazebo simulation, in which we can monitor the execution of the motion on the simulated robot.

Summing it all up, the results for this problem are in our opinion very satisfactory based on the approach we followed.

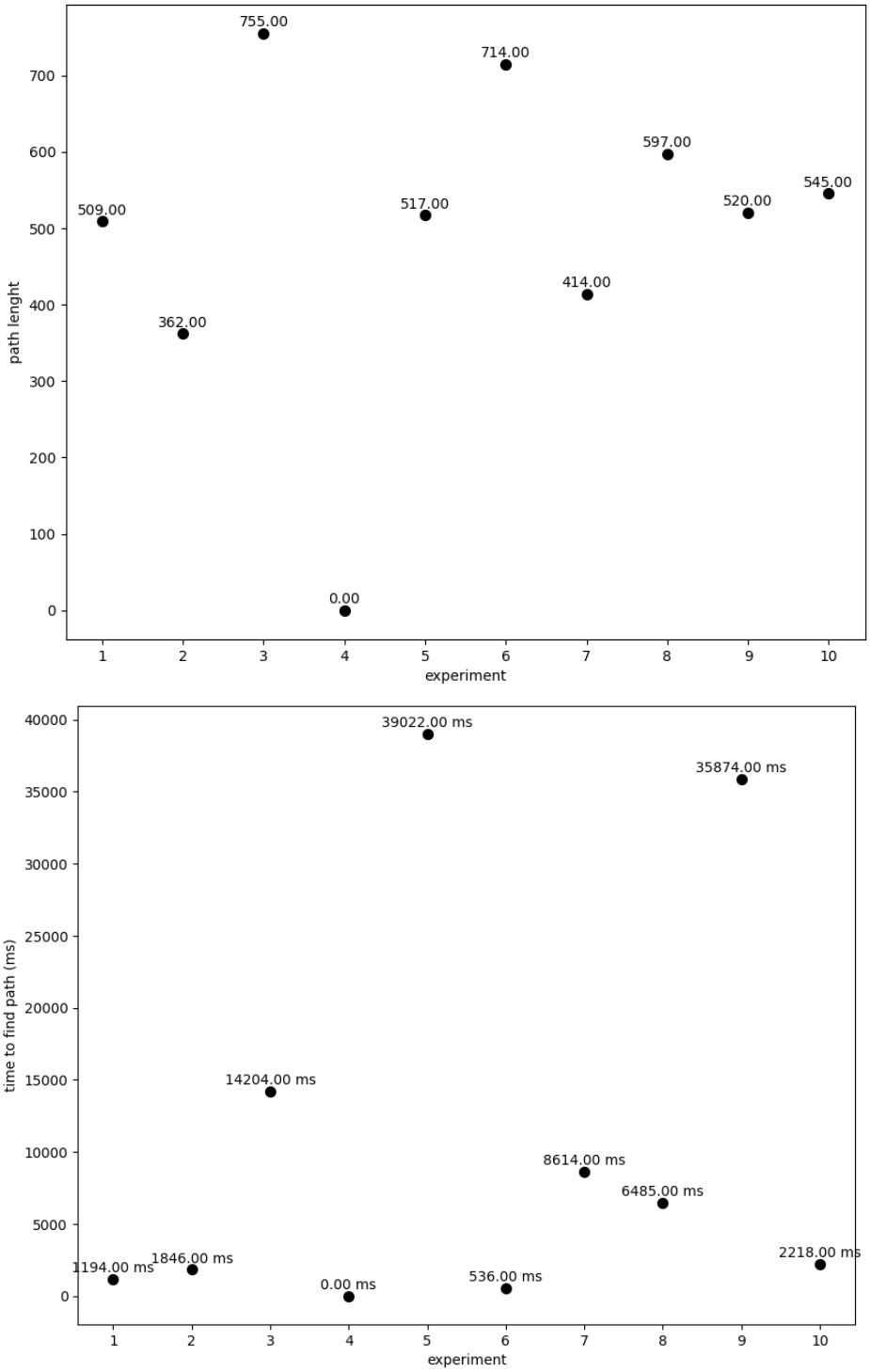

Figure 6: Path and Time Plots

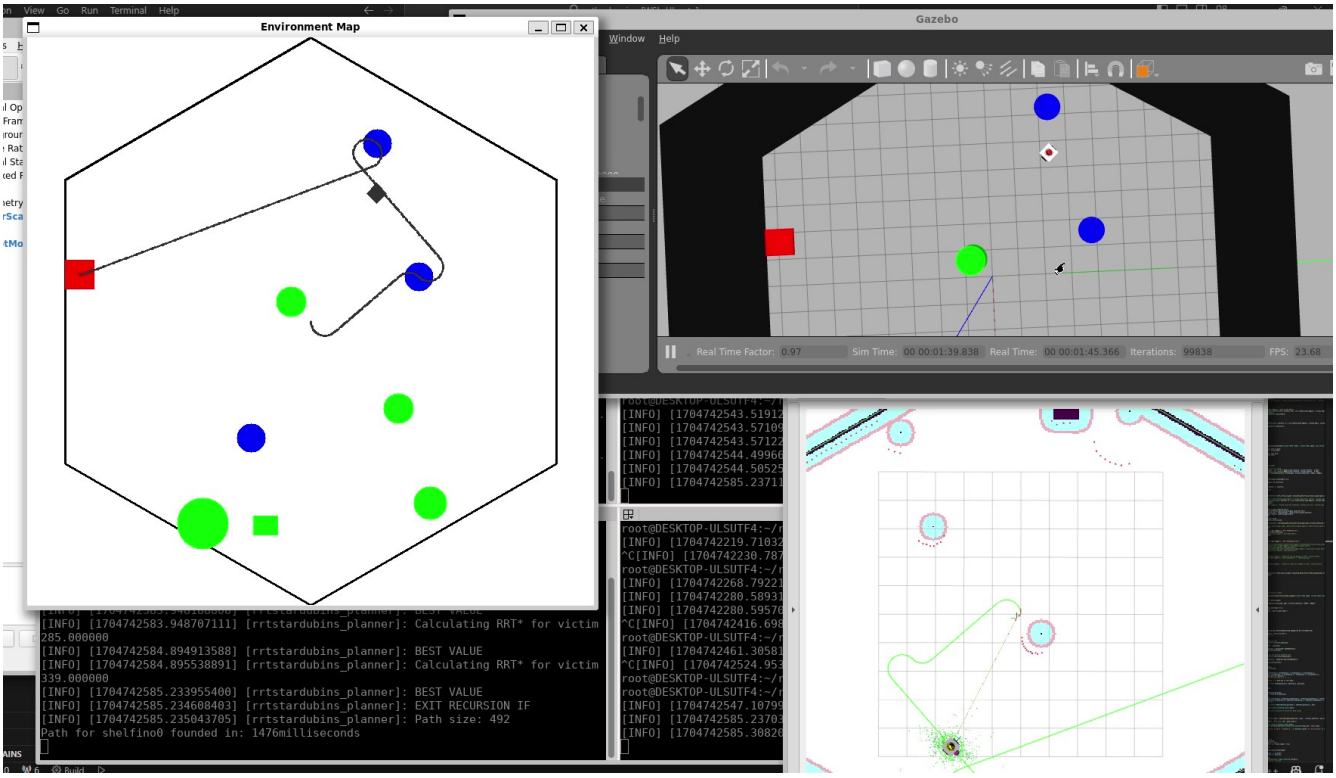

Figure 7: Output

Bibliography

[1] Montresor A. Algoritmi e strutture dati, 13-pd1.pdf, 2024.

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[3] Steven LaValle. Rapidly-exploring random trees: A new tool for path planning. *Research Report 9811*, 1998.

[4] Marco Schoener, Eric Coyle, and David Thompson. An anytime visibility–voronoi graph-search algorithm for generating robust and feasible unmanned surface vehicle paths. *Autonomous Robots*, 46(8):911–927, 2022.