Waste Taxi Robot Project

Dr. Daniel Montrallo Flickinger, RBE 550: Motion Planning

Sidhant Karamchandani, Tamir Lieber, Azzam Shaikh
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: tlieber@wpi.edu, skaramchandani@wpi.edu, ashaikh2@wpi.edu
GitHub Repository: github.com/azzamshaikh/RBE_550_Motion_Planning_Project

***Abstract*—Mobile robots are quickly becoming an integral part of society. They come in many different shapes and sizes, and are applied in many different industries. Companies, such as Amazon, have been pioneering the deployment of robots in their fulfillment centers to increase productivity and deliver to their customers faster. Warehouses and fulfillment centers introduce a variety of risks and challenges for robots who are tasked to move goods to and from locations.**

For this project, a mobile robot, a Turtlebot3, will be deployed in a warehouse to assist the facility in maintaining its 5S activities. The robot will focus on the 'shine' aspect of the 5S methodology by picking up waste and dropping it in a recycle bin. The robot will be tasked to achieve this with the presence of static and dynamic obstacles, mimicking a real-world scenario. In order to effectively evaluate the performance of the robot to achieve its objective, various global planning algorithms will be deployed onto the robot. These planning approaches will identify which methodology generates the safest and most efficient route.

A simulated world environment has been successfully created, providing a platform for testing and experimentation. This environment includes a realistic representation of a warehouse, as well as the placement of static and dynamic obstacles, such as boxes, pallet jacks, and dummy robots. The overall taxi service to move the robot to and from its various goal destinations was also successfully implemented. In addition, four custom global planners were developed to generate plans for the robot, an Dijkstra planner, an A* planner, a RRT planner, and PRM planner. Using the different planners, the taxi service could be ran and tested to find the fastest planner for this application. Based on the results of the testing, the best planner for this application is the A* planner as it provided the fastest solution for picking up waste from various locations and throwing it away in a recycle bin, while minimizing collisions. The heuristic nature of the algorithm provided a direct and fast path to the goals.

***Index Terms*—mobile robot, motion planning, obstacle, graph search algorithm, costmap, planner, collision**

I. Introduction

Mobile robots are quickly becoming an integral part of society. They come in many different shapes and sizes, and are applied in many different industries. These applications include, but are not limited to, self-driving commercial vehicles, autonomous taxis, autonomous delivery vehicles, warehouse robots, underwater exploration devices, aerial drones, and many more. Each of these types of mobile robots requires a focused use of motion planning algorithms in order to navigate complex environments and achieve its goals while minimizing or avoiding collisions.

II. Background

Motion planning itself is a complex field that involves multiple facets such as trajectory generation, graph searches, global and localization mapping, and more. This project focuses on these three mentioned techniques to solve the challenge of developing mobile taxis robots. Mobile taxi robots can be used for the transport of good and/or people from one location to another. There are many environments that such an application can be deployed. These environments could include obstacles such as pedestrians and cars for autonomous vehicles or buildings and vegetation for aerial drones. Due to the presence of obstacles, either static or dynamic, the task of moving from point A to point B safely and efficiently is not trivial. It's also important for mobile taxis to optimize for the time and cost it takes to move from point to point. Thus, various approaches can be considered to solve the search problem and find an optimal route for the robot while taking obstacles into consideration. This enables more the robot to complete more trips within a certain time frame while increasing productivity in the environment.

Diving further into this topic, companies such as an Amazon have been pioneering the introduction of robots in their fulfillment centers to increase productivity and deliver to their customers faster [1]. While Amazon has pushed the boundaries of what is possible with robots, a significant portion of the activities in the fulfillment centers require human operators. These workers move throughout the facility carrying boxes, driving carts, interacting with machinery and more. This presents mobile robots with a unique operating environment that requires knowledge regarding its own tasks and proper decision making with respect to its environment.

Warehouses and fulfillment centers introduce a variety of risks and challenges for robots who are tasked to move goods to and from locations. The environment is cluttered with various static obstacles like equipment, boxes, crates, as well as dynamic objects such as humans, moving equipment like pallet jacks, and even other moving robots. The development of autonomous robots that are able to generate effective plans in a warehouse environment can speed up the tasks moving goods for various processes.

For example, a mobile robot can be deployed to pick up waste inside a warehouse to help maintain a facilities 5S

activities. In the effort of maintaining the 'shine' aspect of the 5S methodology, the productivity of the manufacturing within the facility can increase. For the robot to support this objective, the robot would require knowledge on waste pick up locations as well as the location of the waste bin. Using these points, a path can be planned within the environment for the robot to follow. While the general layout of the warehouse can be provided to the robot for planning, the presence of dynamic obstacles would require additional maneuvers to avoid collision and potential HSE risk. Various global and local planning approaches can be deployed to find the most efficient route to achieve this task.

Based on this example, the purpose of this project involves the creation of a warehouse environment where the mobile robot will be deployed to taxi waste from various pickup locations and drop the waste at a recycle bin. Within the warehouse environment, static and dynamic obstacles will be present for the robot to navigate around. For the robot to achieve this task, various planning algorithms will be implemented and deployed onto the robot. These approaches will be evaluated and compared to find the best planner suitable for this application.

A. Objectives and Timeline

From the initial project proposal, the the goals for the project was described as the following:

1) Guide a vehicle through a dynamic environment safely and efficiently
2) Meet multiple stops safely and efficiently
3) Implement different motion planning algorithms
4) Stretch goal: establish global and local planners
5) Stretch goal: enhance simulation with ROS/Gazebo or add visuals to the Python simulation

Throughout the project and the work completed in preparation for the Progress Report Status Update, the scope of the project was narrowed down to a mobile taxi operating in a warehouse environment. With this redefined scope, the stretch goal of visualizing the simulation with ROS/Gazebo was realized with the current work completed. In addition, after studying the topic of global and local planners within ROS more extensively, it came to the realization that a global planner is effectively the path planning aspect of the program while the local planning is the controller portion of the robot and is used for moving the robot and avoiding obstacles. Thus, this stretch goal has been modified to establish a custom local planner only since a global planner is tacked in bullet three. With this information, the project goals and scope has been modified as shown below:

1) Guide a vehicle through a dynamic environment safely and efficiently
2) Meet multiple stops safely and efficiently
3) Implement three different path planning algorithms via global planners
4) Stretch goal: establish a custom local planner

The project schedule of work can be seen below:

- **Jan 22:** Project kickoff, meet the team
- **Jan 28:** Finalize proposal report and present proposal
- **Feb 26:** Finalize the environment (world, obstacles, mobile taxi)
- **Mar 25:** Implement 3 motion planners
- **Apr 29:** Complete performance analysis; submit project report and present project

III. Methods

The proposed methodology used to tackle this project will be considered in a 3-D, simulated environment. ROS1 Noetic and Gazebo will provide the foundation for visualizing the simulation environment. In the simulation, a taxi will be placed in a warehouse environment and static and dynamic obstacles will be added to the warehouse. Various search algorithms will be implemented for the robot to utilize to navigate the environment. The objective of implementing various search algorithms will provide insight regarding the relative success of each approach as the mobile taxi picks up various waste and drops it off at the recycle bin. The performance of the algorithms will be compared by obtaining the time required to complete the overall pickup process. Using the performance data, the best planner can be identified. To ensure appropriate collection of data, the various sites that the robot will pick up waste from will be queued randomly. This ensures that different paths can be generated from point to point. In addition, the random nature of the dynamic obstacles will play a real impact in the movement of the robot.

Each of these aspects of the methodology will be described below.

A. Simulation Environment

For the warehouse simulation environment, as mentioned, the simulation was setup using ROS1 Noetic and Gazebo. An environment developed by AWS was available via GitHub and was utilized as the base environment for this simulation [2]. This environment contains various shelves, boxes, bins, and other objects in the environment. Figure 1 shows a visual of the simulation environment. With the base environment defined, the world can be expanded by adding a robot and dynamic obstacles.

B. Robot

The robot selected to navigate this environment is a Turtlebot3 Waffle Pi. This differential drive robot comes equipped with a built-in navigation stack with a 360-degree LiDAR sensor and camera. With this hardware on the robot, the robot can communicate with the ROS 2D navigation stack [3]. The Turtlebot3 uses the ROS Navigation stacks amcl package for localization in its environment. This functionality will be utilized for this project. In addition, the Turtlebot3's turtlebot3_drive executable which controls the motion of the robot will be also be utilized for this project. Modifications may be made to the robot to allow backward movement or update the movement and rotation speeds.

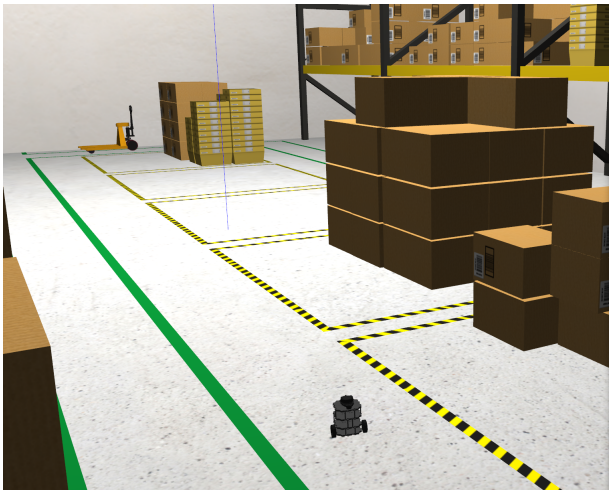

Fig. 1. Simulation environment provided by AWS.

C. Dynamic Obstacles

Since the environment already contained static obstacles, dynamic obstacles had to be added to the environment. To accomplish this, three moving bots were added, as shown in Figure 2. A Gazebo plugin package was created to move the objects in the environment. Two bots move back and forth along the green lines while the third bot moves along the center within the bounds. A pallet jack also moves to and from the bins in the fourth bound. All obstacles have collision boundaries defined to allow interaction with the robot and any LiDAR sensor.

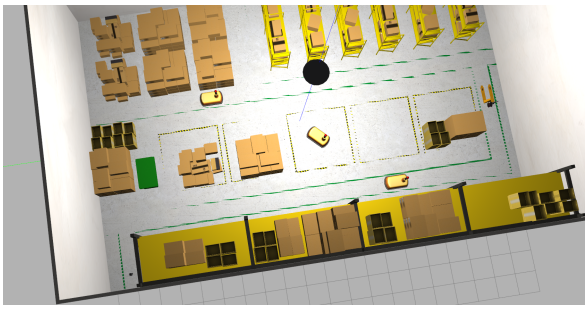

Fig. 2. Dynamic obstacles introduced to the environment.

D. ROS Navigation Overview

With the environment defined, the motion planning of the robot itself can be discussed. ROS1 includes a Navigation stack that can be used to manipulate robots in its environment. It includes different packages for localization and navigation. In order to implement the various search algorithms, the ROS1 Navigation move_base package will be utilized. The move_base package provides a action service that allows a global planner and a local planner to work together to reach a given goal state. A visual of the move_base package can be seen in Figure 3.

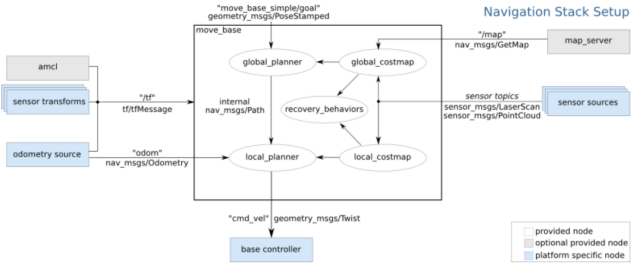

Fig. 3. move_base package high level structure [4].

For the taxi service, a goal, in this case the waste pickup and dropoff points, will be sent to the move_base node via the move_base/goal action topic. The move_base package will then call the global and local planners to generate a path to reach that goal.

In addition to the planners, the move_base package also consists of global and local costmaps. A costmap provides the planners with information regarding the environment in the form of an occupancy grid. The grid is generated from sensor data and a map server. The grid includes cell costs for evaluating a space in the grid. If a space is free, it has a cell cost of 0. If a space is occupied, the space will have a value from 0 to 254. The specific value reflects the possibility of collisions, based on a user defined obstacle inflation radius. The higher the value, the greater the danger of collision. These values will be used by the planners to identify how close a path can be to an obstacle [5].

Both local and global planners will take in a costmap for their respective packages. When a new goal is received, the global planner generates a path to the goal based on the costmap. The defined path is relayed to the local planner, which implements the motion control to follow the path.

Since the move_base package calls the global and local planners as action services, the planner functions are called at a user defined frequency. This allows both the global and local planners to receive real-time information about about the environment via sensors and allows the planners to adjust their plans as needed.

E. Implementation of Global Path Planning

For the implementation of the various search algorithms, the global planner portion of the move_base package will be modified. The default global planner implemented in the move_base package points to the navfn package, which utilizes a unique implementation of Dijkstra's algorithm for navigation [6].

In a StackExchange post [7], it was described that the path found by navfn is based on the path "potential". The potential is the relative cost of a path based on the distance from the goal and from the existing path itself. The potential is determined by the cost of traversing a cell and the distance away that the next cell is from the previous cell. It is mentioned that this approach is an implementation of a Wavefront expansion algorithm. The search approach is very similar to that of

Dijkstras algorithm, however, instead of creating a tree of neighbors based on cost traversal, it assumes initially that all cells are at high potential except the start which is at zero potential. It then assigns finite potential values to each cell during traversal and then uses gradient descent to trace a path back from end to start.

This default planner will act as a baseline planner to provide a reference to the success and/or failure of the other planners.

In order to override the default global planner, custom path planning packages will be developed based on different path planning algorithms. These packages will consist of a script to determine the path from a robots pose to goal pose. In order to run the navigation with these custom planners, the packages will be deployed as a plugin into the move_base package.

F. Custom Global Planners

For the custom planners, a total of four planners were implemented. Each planner will be discussed. In order to test and visualize the results of the planner, the RVIZ 2D Goal function is used to generate a path.

1) Dijkstra Planner: The first planner is a grid search based, Dijkstra planner. This custom planner would provide a reference against the default planners approach to the same algorithm.

Dijkstra's algorithm is "a systematic search algorithm that finds single-source shortest paths in a graph" [8].

The pseudocode for the algorithm is shown in Figure 4:

```
1  function Dijkstra(Graph, source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] ← INFINITY
5          prev[v] ← UNDEFINED
6          add v to Q
7      dist[source] ← 0
8
9      while Q is not empty:
10         u ← vertex in Q with min dist[u]
11         remove u from Q
12
13         for each neighbor v of u still in Q:
14             alt ← dist[u] + Graph.Edges(u, v)
15             if alt < dist[v]:
16                 dist[v] ← alt
17                 prev[v] ← u
18
19     return dist[], prev[]
```

Fig. 4. Dijkstra pseudocode [9].

Using this algorithm, the dijkstra package was created to implement this approach and was exported as a plugin for the simulation. Figure 5 shows the result of the planner visualized by RVIZ. The black line represents the path output from the planner to get to the goal destination. The grey cells visualize the expanded nodes searched by the algorithm.

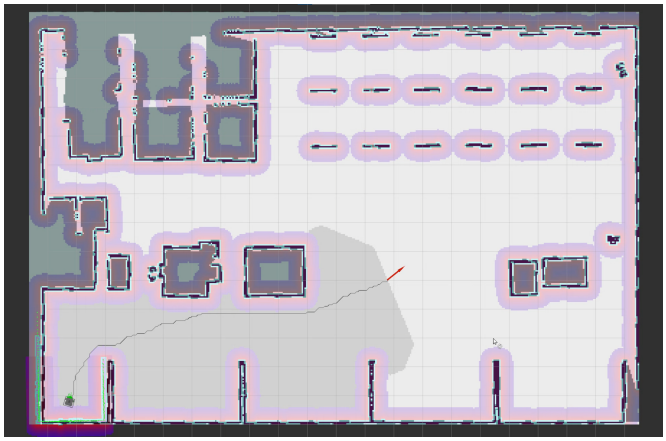

Fig. 5. Dijkstra planner visualization.

2) A Planner:* The second planner is another grid search planner based on the A* algorithm. This algorithm builds upon the Dijkstra planner by adding a heuristic cost that is computed by calculating the Euclidean distance from the current position to the goal position. This helps minimize the number of neighbors searched and provides a focused path toward the goal.

Using this algorithm, the astar package was created to implement this approach and was exported as a plugin for the simulation. Figure 6 shows the result of the planner visualized by RVIZ. The black line represents the path output from the planner to get to the goal destination. The grey cells visualize the expanded nodes searched by the algorithm. Compared to the Dijkstra planner, it can be seen that fewer nodes have been searched. This is expected as the heuristic cost associated with each node expansion helps guide the algorithm to find the goal in fewer steps.

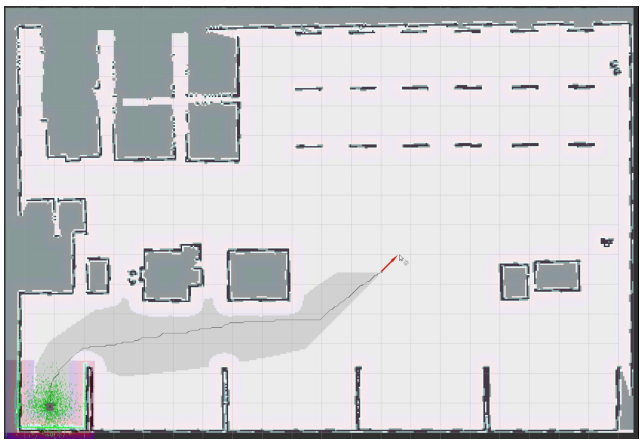

Fig. 6. A* planner visualization.

3) RRT Planner: The third planner is a sampling-based planner, based on the Rapidly-exploring Random trees (RRT) algorithm.

RRT is a method of to develop a path in an environment based on a search tree. The search tree is constructed by

incrementally expanding the tree based on randomly sampled points in the space. Based on the number of samples, the tree will continue to expand and grow. If obstacles are present, the expansion can be modified to be only in the valid free space. This assists in generating fast and probabilistically complete search paths in the environment [8]

The algorithm pseudocode can be seen below in Figure 7

```
Algorithm BuildRRT
    Input: Initial configuration q_init, number of vertices in RRT K, incremental distance Δq
    Output: RRT graph G

    G.init(q_init)
    for k = 1 to K do
        q_rand ← RAND_CONF()
        q_near ← NEAREST_VERTEX(q_rand, G)
        q_new ← NEW_CONF(q_near, q_rand, Δq)
        G.add_vertex(q_new)
        G.add_edge(q_near, q_new)
    return G
```

Fig. 7. RRT planner pseudocode [10].

The rrt package was created to implement this approach and was exported as a plugin for the simulation. Figure 8 shows the result of the planner visualized by RVIZ. The red lines represents the search tree generated to get to the goal destination.

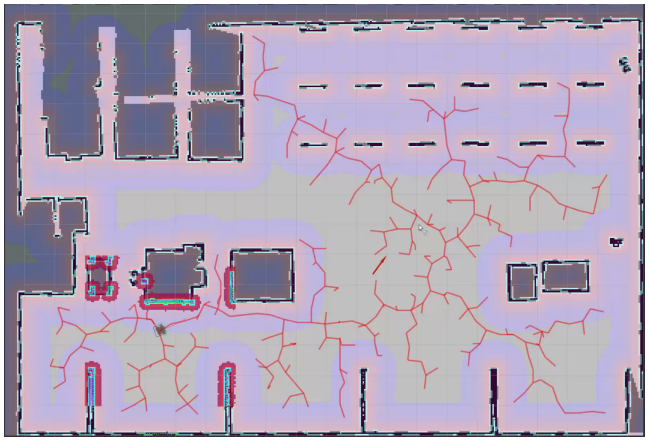

Fig. 8. RRT planner visualization.

4) PRM Planner: The last planner is another sampling-based planner, based on the Probabilistic Roadmap (PRM) algorithm.

PRM is a method to develop a path in an environment based on a roadmap created from randomly sampled nodes. First, random locations are chosen across the entire workspace (warehouse). Each location is cross-examined to make sure it isn't occupied by an obstacle. If it is, it is thrown out as invalid. Then, all the valid nodes are examined. Every single valid node is compared with all other valid nodes. If the compared node is within a maximum distance of the current node, and a straight line can be drawn between them without obstruction, that edge between nodes (vertices) is recorded in the roadmap. Eventually, a roadmap of roughly the entire workspace is constructed. Once this graph is constructed, technically any graph search algorithm can be applied to it. [8]

The algorithm pseudocode can be seen below in Figure 9

```
// Function to generate random samples within the bounds of the costmap
function generateRandomSamples(global_costmap, num_samples):
    for i from 1 to num_samples:
        x = random number between 0 and global_costmap.width
        y = random number between 0 and global_costmap.height

        // Check if the sample is in free space
        if global_costmap[x, y] equals 0:
            add (x, y) to samples

    return samples

// Function to construct roadmap using generated samples
function constructRoadmap(global_costmap, samples, num_neighbors):
    for i from 0 to length(samples):
        neighbors = empty_list []
        for j from 0 to length(samples):
            if i is not equal to j:
                dist = euclideanDistance(samples[i], samples[j])

                // Check if the distance is within the specified range and there is a clear path between the nodes
                if dist is less than or equal to max_distance and isClearPath(global_costmap, samples[i], samples[j]):
                    add j to neighbors

                // Break if enough neighbors are found
                if length(neighbors) equals num_neighbors:
                    break

        roadmap[i] = neighbors

    return roadmap
```

Fig. 9. PRM planner pseudocode.

The values found to work the best in this simulation were: 2000 samples, 20 neighbors, 10 foot maximum distance, a 10,000 iteration maximum for path finding, and a maximum cost of 25 (slightly less than semi-occupied) on cells lying on graph edges. With these values, valid paths were continuously being found without severely compromising CPU time. Figure 10 shows an example roadmap visualized in the project workspace.

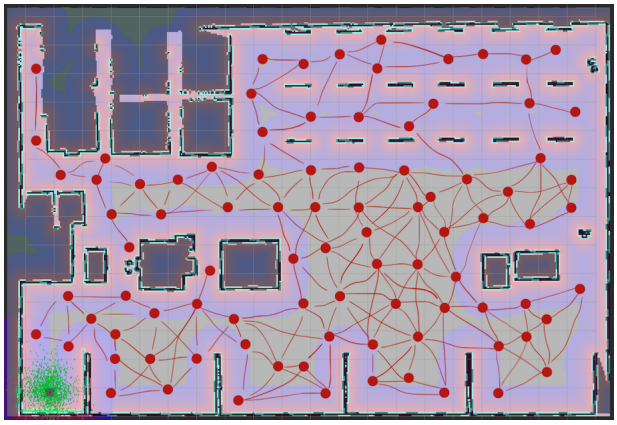

Fig. 10. PRM planner visualization.

G. Taxi Service

As mentioned in the Background, the purpose for the Turtlebot3 is to taxi waste from pickup location to the recycle bin. In the warehouse where this robot has been deployed, there are 5 known waste pickup spots. Figure 11 shows the locations of the 5 waste spots as well as the location of the recycle bin. During deployment, the robot will go to these pickup points and collect waste. The order at which the sites will get queued is completely random. This ensures that the dynamic obstacles have a real impact on the local planning of the robot.

A taxi service package has been developed to send the pickup location queue to the move_base package. The move_base package is responsible for providing data to the global and local planners about the goal and having the robot

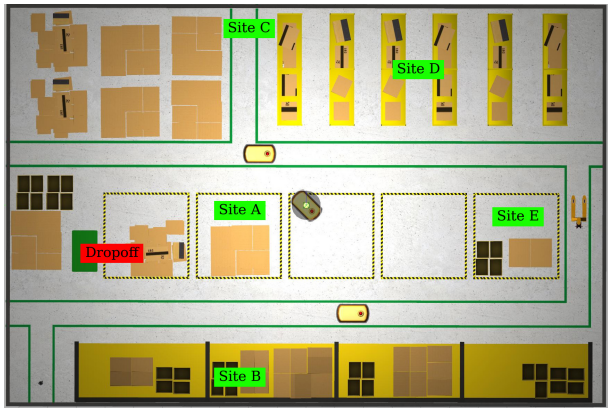

Fig. 11. Locations of waste pickup sites.

reach its goal. Once the robot is at a pickup site, it will then go to the next pickup site. From there it will repeat this cycle until all sites have been reached. Lastly, it will go to the recycling bin for finish the Taxi Service. Figure 12 shows an example path to Site E. The visualization is accomplished using RVIZ.

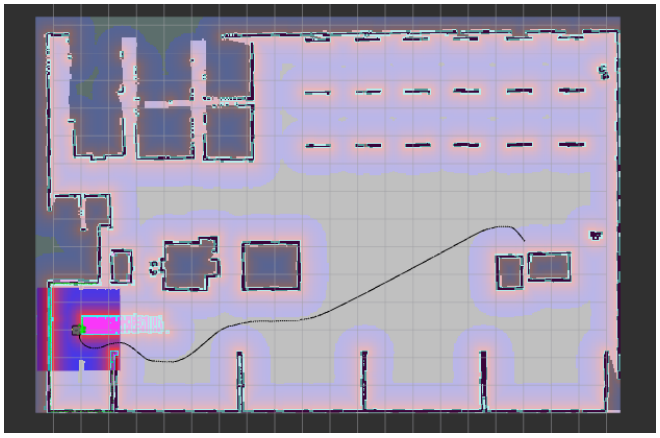

Fig. 12. RVIZ global planning visualization.

In addition, the taxi_service node will display information regarding the timings from pickup point to pickup point and the overall time required to collect all the waste and dump the items in the recycle bin. Figure 13 shows an example output from a taxi service run.

With the move_base package and its default global and local planners, the robot is successfully able to generate paths to reach its goal destinations.

IV. Challenges

Over the course of the project, various challenges presented themselves in different areas of the work.

Since the move_base package provided a usable interface for managing planners, it was elected to use the package versus creating something from scratch. While this reduced the amount of overhead for developing a base architecture, the implementation of the planners had to follow a specific paradigms. Since a lot of the processes and flow of data was

```
[ INFO] [1708191423.301949110]: Taxi service successfully initialized!
[ INFO] [1708191423.302674818]: Running Taxi Service!
The pickup site sequence is as follows:
Site A
Site D
Site C
Site E
Site B
Going to Site A
[ INFO] [1708191423.892083776, 8.401000000]: Sending the robot the pickup location!
[ INFO] [1708191423.892188228, 8.402000000]: Robot is on the way!
[ INFO] [1708191473.036703368, 57.502000000]: Hooray, the robot is at the pickup location!
It took 49.1447 seconds to reach this site.
[ INFO] [1708191473.036805035, 57.502000000]: Picking up object right now...
[ INFO] [1708191478.041374862, 62.503000000]: Pickup complete!
Going to Site D
[ INFO] [1708191478.041414895, 62.503000000]: Sending the robot the pickup location!
[ INFO] [1708191478.041477350, 62.503000000]: Robot is on the way!
[ INFO] [1708191523.079192810, 107.503000000]: Hooray, the robot is at the pickup location!
It took 45.0378 seconds to reach this site.
[ INFO] [1708191523.079277145, 107.504000000]: Picking up object right now...
[ INFO] [1708191528.086075176, 112.504000000]: Pickup complete!
Going to Site C
[ INFO] [1708191528.086125117, 112.504000000]: Sending the robot the pickup location!
[ INFO] [1708191528.086194014, 112.505000000]: Robot is on the way!
[ INFO] [1708191555.912999246, 140.305000000]: Hooray, the robot is at the pickup location!
It took 27.8269 seconds to reach this site.
[ INFO] [1708191555.913048216, 140.305000000]: Picking up object right now...
[ INFO] [1708191560.918791005, 145.305000000]: Pickup complete!
Going to Site E
[ INFO] [1708191560.918827813, 145.305000000]: Sending the robot the pickup location!
[ INFO] [1708191560.918890759, 145.305000000]: Robot is on the way!
[ INFO] [1708191625.368562267, 209.705000000]: Hooray, the robot is at the pickup location!
It took 64.4498 seconds to reach this site.
[ INFO] [1708191625.368616027, 209.705000000]: Picking up object right now...
[ INFO] [1708191630.371858546, 214.705000000]: Pickup complete!
Going to Site B
[ INFO] [1708191630.371900974, 214.705000000]: Sending the robot the pickup location!
[ INFO] [1708191630.371961516, 214.705000000]: Robot is on the way!
[ INFO] [1708191683.219785928, 267.505000000]: Hooray, the robot is at the pickup location!
It took 52.8479 seconds to reach this site.
[ INFO] [1708191683.219831402, 267.505000000]: Picking up object right now...
[ INFO] [1708191688.224319451, 272.506000000]: Pickup complete!
[ INFO] [1708191688.224359966, 272.506000000]: All waste has been picked up!
[ INFO] [1708191688.224385133, 272.506000000]: Sending the robot the dropoff location!
[ INFO] [1708191688.224450925, 272.506000000]: Robot is on the way!
[ INFO] [1708191739.865268455, 324.106000000]: Hooray, the robot is at the dropped off location!
It took 315.973 seconds to complete the Taxi service!
[ INFO] [1708191739.865312466, 324.106000000]: Dropping the object right now...
[ INFO] [1708191739.865336240, 324.106000000]: Dropoff Complete!
[ INFO] [1708191739.865354464, 324.106000000]: --- Pickup and dropoff complete! ---
[ INFO] [1708191744.967537335, 329.203000000]: Taxi service completed successfully!
```

Fig. 13. Default planner results.

completed under the hood, this lead to a lot of challenges with understanding the general process and workflow of what is occurring in the background and troubleshooting/debugging issues. Additionally, there is not a lot of material available describing a step by step process on the development of a global planner plugin. Thus, heavy analysis of the ROS Navigation navfn source code [6] and ROS Wiki [11] was required to implement the planner. Even with the examples that are present, there isn't sufficient information describing the usage and implementation of different variables, files, and parameters. This lead to a development process that relied on trial and error to see and understand what worked and what didn't.

For example, after implementing the first Dijkstra global planner, preliminary testing was executed. However, when running the RVIZ 2D Goal navigation or taxi service, several failures occurred.

In one case, certain site pickup points resulted in a failure to get a path, as show in Figure 14. Upon further inspection, the goal point for certain pickup points was an invalid location because it was within the inflation radius of an object and thus, was unreachable. This was not an issue for the default planner however. Fine tuning of the inflation radius was required to enable access to all the sites.

In another instance, when going to a reachable goal, due to the inflation radius of the dynamic object, the fastest path was blocked. Figure 15 visualizes this scenario. In some cases, when posed with this challenge, the robot stopped for a few seconds before updating a path to go in between both objects inflation radius; this was the ideal response. However, in other

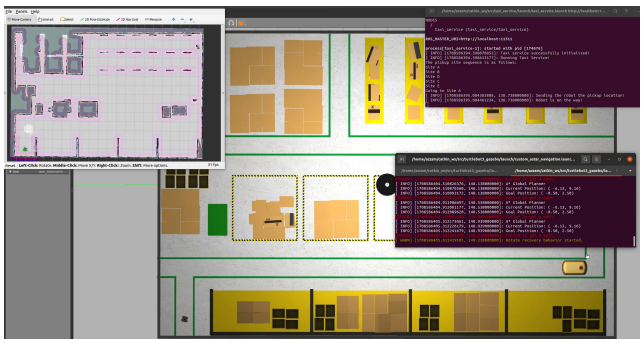

Fig. 14. Dijkstra failing to obtain path.

cases, even after the dynamic objected moved out of the way, the costmap had old visual artifacts of the object which prevented the robot from moving entirely. The costmap is supposed to use sensor data to mark and/or clear any obstacle information as needed. Furthermore, it was unclear which of the planners, either global or local, was causing this issue.

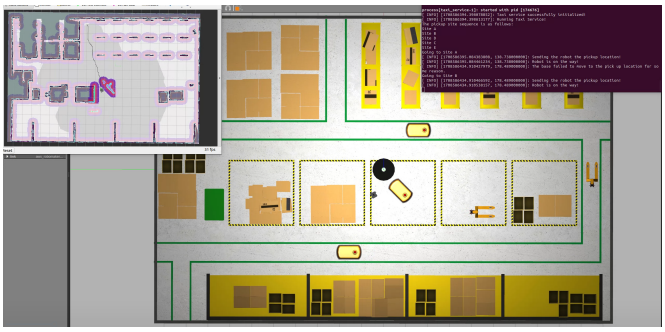

Fig. 15. Dijkstra challenged with two obstacles blocking the path.

In another instance, when receiving a reachable goal, the robot failed to move. Figure 16 visualizes this scenario. The planner had received the new goal from the taxi service but the robot did not generate any commands to move and follow the path. It initiated its recovery sequence after not moving and cleared its costmaps but the robot still failed to move. This instance occurred multiple times during testing. Tt was unclear which of the planners, either global or local, was causing this issue.

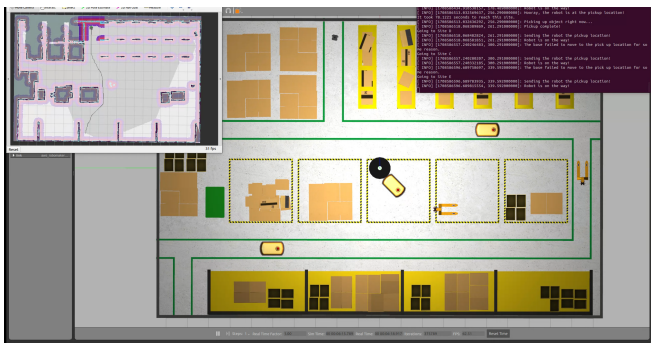

Fig. 16. Robot failing to move to a valid path provided by Dijkstra.

Another situation that was exhibited was the robots failure to move to a goal destination when the planner creates a path that is close to objects. As mentioned in Proposed methods section, the default planner uses a gradient descent approach to finding a solution. With that approach, the planner is able to find paths between obstacles that this graph search planner is not able to do.

Overall, from the initial implementation of the global planner, while it worked to a degree, it was not very effective. The initial implementation relied on passing the all the costmap information - from the static map and sensor data - to the global planner and using the default local planner. Thus, significant optimization was needed to get the planners to work at a similar level as the default planner. After significant troubleshooting with the global planner parameters, it was unclear if the global planner was causing these issues or if the local planner was causing these issues. Based on this, it was selected to utilize the dynamic window approach (DWA) controller provided with the Turtlebot3 navigation package; this change would remove the question of which planner was causing the issues. In addition, the global planner was modified to only utilize the map server data. This offloaded the sensor data to be read by the local planner and then passed to the global planner. After implementing this change, a lot of the issues previously seen had resolved themselves. With these challenges resolved, the testing could be completed.

It should be noted that, with this change, the objective for implementing a custom local planner as a stretch goal was removed. In the status update, this stretch goal was defined since the robot was struggling to move in certain situations and it was assumed the default local planner was the culprit of these challenges. Since this need was resolved with the usage of the Turtlebot3 DWA local planner, the stretch goal was no longer needed.

V. Results

With the planners, taxi service, and dynamic obstacles implemented, the tested could be completed to determine which approach is best suited for this application.

The testing will be accomplished as follows. Each planner will execute the taxi service for a total of 5 runs. In each run, the timings and number of collisions will be observed and the times will be averaged. From these averaged values, the best planner can be identified.

A. Default Planner

The results from the testing of the default planner can be seen in Table I.

Overall, with the default planner, the Turtlebot3 was successfully able to get to each goal destination that it received, with an average time of 197 seconds for a full run. The Turtlebot3 only collided with 1 obstacle during the fifth run.

During the various runs, it was noted that the planner avoided obstacles by a large margin. Thus, this resulted in the robot taking longer paths to reach a certain goal. This is most likely due to the default global planner that is being used.

TABLE I
TESTING RESULTS FOR THE DEFAULT PLANNER.

	Time Between Sites [s]				
Run #	Site 1	Site 2	Site 3	Site 4	Site 5
1	26.04	28.65	20.04	28.44	26.54
2	43.75	29.84	35.54	22.73	21.73
3	43.46	30.04	35.06	30.15	24.63
4	37.57	19.34	15.23	33.85	15.03
5	27.17	24.12	16.45	29.59	31.07

Run #	Total Time [s]	Collisions
1	207.81	0
2	212.88	0
3	204.80	0
4	179.64	0
5	182.16	1
Average	197.46	0.2
Max	179.6	
Min	212.88	

With these baseline results, the other planners can be tested relative to this.

B. Dijkstra Planner

The results from the testing of the Dijkstra planner can be seen in Table II.

TABLE II
TESTING RESULTS FOR THE DIJKSTRA PLANNER.

	Time Between Sites [s]				
Run #	Site 1	Site 2	Site 3	Site 4	Site 5
1	35.35	35.57	20.24	24.74	13.23
2	23.26	28.05	12.82	23.14	15.54
3	42.07	26.85	20.33	30.04	20.13
4	43.69	25.95	15.52	20.64	42.26
5	40.48	20.54	39.00	28.38	26.25

Run #	Total Time [s]	Collisions
1	186.05	0
2	157.10	0
3	197.41	0
4	199.75	1
5	195.99	0
Average	187.26	0.2
Max	157.10	
Min	199.75	

Overall, with the Dijkstra planner, the Turtlebot3 was successfully able to get to each goal destination that it received, with an average time of 187 seconds for a full run. The Turtlebot3 only collided with 1 obstacle during the fourth run.

During the various runs, compared to the default planner, it was noted that the planner took shorter paths, which shaved down approximately 10 seconds. These shorter paths resulted in the robot going much closer to obstacles. In a real-world scenario, this risk would need to be taken into consideration as it could lead to higher chances of collisions and damage.

C. A* Planner

The results from the testing of the A* planner can be seen in Table III.

TABLE III
TESTING RESULTS FOR THE A* PLANNER.

	Time Between Sites [s]				
Run #	Site 1	Site 2	Site 3	Site 4	Site 5
1	40.67	15.73	26.75	34.67	19.73
2	41.25	25.25	23.04	20.44	31.95
3	23.34	27.53	13.41	28.25	21.13
4	25.35	32.24	22.73	15.43	29.84
5	31.39	20.87	58.60	12.62	20.87

Run #	Total Time [s]	Collisions
1	178.32	0
2	191.51	0
3	176.06	0
4	175.68	0
5	198.40	0
Average	183.99	0
Max	175.68	
Min	198.40	

Overall, with the A* planner, the Turtlebot3 was successfully able to get to each goal destination that it received, with an average time of 183 seconds for a full run. The Turtlebot3 had no collisions with any obstacles, within these 5 runs.

During the various runs, compared to the Dijkstra planner, even shorter paths were take by the robot to reach its goal destination. This shaved down an additional 4 seconds off the average run time. Due to the even shorter paths taken, the clearance between obstacles was minimal, which poses even higher risks for potential collisions.

D. RRT Planner

The results from the testing of the RRT planner can be seen in Table IV.

The implementation of the RRT planner had interesting results. Due to the overall architecture of the move_base package, the planning algorithm is constantly called and ran at a user defined frequency. When called, the plan generated from the previous frame gets cleared and a new path is made. This provides decent results with the combinatorial planners since they are generally guided by a heuristic. However, for a random sampler, for every iteration, a randomly made tree structure is generated every cycle. It was noted that after a random amounts of time, the planner would freeze on the most recent path and fail to create a new tree structure. When this occurs, based on the process of the move_base package, it indicates that the rrt package was successfully able to complete the `makePlan` function but the move_base package failed to call that function on the next cycle. The `makePlan` function is called via a service as part of the rrt package. Thus, it is unclear if the move_base package is failing to call the service or if the rrt package service is failing to call the `makePlan` function. There is no warning or error messages were showing up saying any processes have died. Furthermore, when sending

a new goal, either via the Taxi Service or via RVIZ, the new goal was not updated by the `makePlan` function, indicating that the function was not being called. Thus, the move_base and rrt package had to be shut down and restarted to resolve the issue.

When running the RRT planner with the taxi service, depending on the goal received, if the goal is far from the robot, several issues occurred. Since the tree is completely random, the overall path is changing constantly, leading to the local planner having to follow different trajectories constantly. This caused issues when the robot was attempting to follow a certain path in a narrow area and then receiving a new path that had a different trajectory. This would result in the robot getting stuck and failing to move. Thus, it was attempted to manually send goals to the robot that were closest to it. This at least allowed the robot to move and reach a goal without getting stuck. When doing this, in two runs, all the goals were able to be reached, providing some metric regarding the effectiveness of the planner. Since these goals were send with the shortest distance in mind, it is not an accurate comparison. However, even with the fastest order of goals, the average time, for 2 runs, is around the same time as the A* planner for completely random paths. This is indicating that the RRT planner paths were not extremely efficient for reaching its goals. This is confirmed by the different paths taken by the robot, which were seen during testing. Since the trees were constantly making new paths, the local planner made the robot move slowly and was constantly adjusting its path. Furthermore, the random nature of the RRT path was not the fastest path.

An ideal approach would be one that generates an RRT path once and allows the local planner to traverse the path completely. This would reduce the need to recompute the tree constantly.

TABLE IV
TESTING RESULTS FOR THE RRT PLANNER.

	Time Between Sites [s]				
Run #	Site 1	Site 2	Site 3	Site 4	Site 5
1	27.33	28.74	Crashes		
2	38.67	37.05	Crashes		
3	28.40	26.50	16.87	Crashes	
4	45.61	45.69	22.76	Crashes	
5	38.90	Crashes			
6	30.68	27.80	19.64	25.76	20.82
7	26.64	24.67	18.40	24.81	28.04

Run #	Total Time [s]	Collisions
1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	196	0
7	168	0
Average	182	0
Max	196	
Min	168	

E. PRM Planner

The results from the testing of the PRM planner can be seen in Table V-E.

TABLE V
TESTING RESULTS FOR THE PRM PLANNER.

	Time Between Sites [s]				
Run #	Site 1	Site 2	Site 3	Site 4	Site 5
1	210	203	Crashes		
2	145	163	Crashes		
3	78	50	66	83	Crashes
4	77	84	107	Crashes	
5	81	64	92	112	78
6	89	68	101	108	91

Run #	Total Time [s]	Collisions
1	-	-
2	-	-
3	-	-
4	-	-
5	427	0
6	457	0
Average	442	0
Max	457	
Min	427	

The PRM planner could be considered a feasible planner, although it would require some improvements before deploying in production.

There were a few runs in which all waste sites were visited, but this was a bit overshadowed by the runs in which the planner got stuck after not being able to find a valid path. There are a few explanations for these challenges. Firstly, there is a high computational cost to sampling grid nodes around the workspace and producing a roadmap from them. The controller, local planner, and global planner require constant, low-latency communication in order to operate smoothly, and repeated generation of samples and roadmap was disrupting that. After many experiments, a workable balance was found between the number of samples and neighbors, and an acceptable performance. Secondly, the architecture of the ROS nodes and their intercommunication made it impossible to sample just once, without doing a major refactor of the project. Lastly, high traffic areas of the warehouse naturally have less sampled points. Because of all the obstacles (both static and dynamic) in the high traffic areas, most sampled nodes here were seen as invalid, consequently reducing the number of paths available to the robot taxi. And due to this, whenever multiple consecutive high-traffic waste sites were provided as destinations, the planner had a hard time navigating a path between them. The longer run times of the PRM planner highlight the regeneration of the roadmap as well as the struggle to find a path in high-traffic areas.

Outside the struggles, there were bright moments. No collisions were recorded between the robot taxi and obstacles (both static and dynamic). Also, the paths taken were smooth, as opposed to jittery and zigzagged, as can be seen in the accompanying demo videos.

F. Combined Results

Table VI shows the average run time of all the planners for the taxi service.

TABLE VI
COMBINED TESTING RESULTS.

Planner	Average Time [s]	Average Collisions
Default	197.46	0.2
Dijkstra	187.26	0.2
A*	183.99	0
RRT	-	-
PRM	442	0

Based on the results, it can be seen that the A* planner had the fastest time for the taxi service to reach all waste pickup points and drop them off at the recycle bin. This is an expected result as the A* planner is known for providing effective and efficient paths toward the goal with its heuristic implementation. While the Dijkstra planner is not far behind, the A* planners prioritization of the search with the goal in mind helps shave a few seconds off. Since the RRT planner failed to complete a single taxi run, no data was included. The manual goal sending approach described in Section V.D details some of the capabilities of the RRT planner. While it is capable of guiding the robot to the goals, it is not an effective at finding the fastest path, which is expected with a random searching planner. The PRM planner had significantly longer average run times. This was potentially due to limited computational resources on the machines that measured the results. In a real scenario, it would still not be expected for the PRM to outperform the A* planner due to the random nature of the PRM sampling process, unless the warehouse workspace was significantly larger (10x or more), which would force discrete planners to store and parse an unreasonably large amount of data.

Overall, in an application where there is full knowledge of the environment, a grid based planning approach provided effective results for the ability to get to the goals in a timely and quick manner. While sampling based approaches provide probabilistically complete solutions, it may not provide the fastest solutions. Had the environment been unknown to a degree, an RRT planner may have been a better planning solution as compared to an grid based planner. The overall selection of planner for each application needs to be taken into consideration such that the objective of the robot can be statisfied with the environmental constraints.

VI. CONCLUSION

The purpose of this project involved the creation of a warehouse environment where the mobile robot will be deployed to taxi waste from various pickup locations and drop the waste at a recycle bin. Within the warehouse environment, static and dynamic obstacles will be present for the robot to navigate around. For the robot to achieve this task, four planning algorithms were developed and implemented deployed as global planner plugins for the robot to use. The goal is to find the best planner suitable for this application.

The objectives for this project included the following:

1) Guide a vehicle through a dynamic environment safely and efficiently
2) Meet multiple stops safely and efficiently
3) Implement three different path planning algorithms via global planners
4) Stretch goal: establish a custom local planner

After establishing the environment, obstacles, and global planners, the robot could be deployed to the warehouse to taxi waste. The results from the testing is tabulated in VI.

Based on the results of the testing, the best planner for this application is the A* planner as it provided the fastest solution for picking up waste from various locations and throwing it away in a recycle bin, while minimizing collisions. The heuristic nature of the algorithm provided a direct and fast path to the goals.

The objectives for (1) guiding a vehicle through a dynamic environment safely and efficiently, (2) reaching multiple stops safely and efficiently, and (3) implementation of three different global planners, were all successfully met and exceeded. The stretch goal was removed, as described in the Challenges section.

This project provided an excellent, yet challenging learning experience with the ROS platform, in specific, its navigation stack and connection to the move_base package. Since this existing architecture was utilized to implement global planners, significant time was spent learning how to implement a planner. While it provided an effective solution for moving a robot within its environment, the overall architecture can be slightly restrictive. Since the plugin is constantly called and regenerates a path, for a random, sampling based planner, like the RRT and PRM planners, it defeated the purpose of the path generation. In an ideal scenario, instead of regenerating the path, a singular path should have been found from the start to the goal and then allowed the local planner to navigate the environment and avoid obstacles as needed. However, since the global planner took costmap data regarding its environment in real-time, every planner call would provide an updated path and solution. While this may work to a degree, it is not ideal in a computational sense. Another key lesson learned from this project was the usage of partially occupied grid cells. Algorithms were significantly more restrictive and under-performing when searching for completely free grid cells. Being more lenient and allowing the consideration of partially occupied cells allowed far more paths to be found and navigated by the taxi.

Regarding future work, the primary goal would be stabilization of the random sampling-based planners. It would be interesting to develop a custom global planner architecture from scratch that generates a single path once, and allows the local planner to manage the motion of the robot and obstacle collisions. Alternatively, the current architecture could continue to be used but perhaps tree and roadmap files could be saved on disk or memory so that they are generated and

written a minimal number of times, but read an indefinite number of times. Another future goal would be to implement other TurtleBot models, each of which would have different kinematics and dynamics associated with it.

APPENDIX A
INDIVIDUAL CONTRIBUTIONS

This appendix will describe the individual contributions by each team member.

A. Sidhant Karamchandani

Initiated code reviews of the different packages, plugins, services, and documentation. Launched different modules and documented/reported any bugs. Ran multiple simulations for the default global planner and observed/recorded results.

Helped develop and test the PRM planner. Extensive debugging to ensure accurate roadmap generation and path navigation. Experimenting and tweaking parameters (as highlighted in Methodology) for optimal performance. Comparing sampling-based algorithm performance vs discrete planning algorithm performance to see where future improvements can be made.

Contributions to major sections of the presentation and paper.

B. Tamir Lieber

Dynamic obstacles were added to the environment in Gazebo using the motion plugin. These obstacles help create a realistic environment to test so the results can be applied to real engineering problems. The Taxi Service, dynamic obstacles and default motion planner were all tested to ensure the simulation could be run on any PC environment. Contributed warehousing and distribution knowledge to ensure the proposal had realistic goals and could be applied outside of this class. Contributed to the report and presentation. Developed functional modules of the PRM planner using ROS and Gazebo to make sure the roadmap generation is accurate and the search algorithm finds the goal in an efficient way. Tested PRM planner by logging data and viewing feedback from ROS and Gazebo. Logged data from taxi service package for results and conclusions.

C. Azzam Shaikh

Developed overall project architecture by implementing the following:

- Developed warehouse simulation environment and imported TurtleBot3 into the world.
- Developed gz_motion_plugin package to provide objects in the world with dynamic motion. Applied plugin to the three moving warehouse bots.
- Developed taxi_service package to give goal messages to the robot to navigate to. Determined robot pickup and dropoff locations and setup taxi_service to randomly move to all the sites to pickup waste and drop them in the recycle bin.

Developed several custom planners by implementing the following:

- Developed the dijkstra, astar, and rrt global planner package plugins to override the default planner.
- Tested the default, dijkstra, astar, and rrt global planner packages to collect taxi service run times.

Drafted significant portions of report and presentations.

REFERENCES

[1] S. Dresser, “Amazon announces 2 new ways it’s using robots to assist employees and deliver for customers,” About Amazon, https://www.aboutamazon.com/news/operations/amazon-introduces-new-robotics-solutions (accessed Feb. 18, 2024).

[2] AWS-Robotics, “AWS-robomaker-small-warehouse-world: This gazebo world is well suited for organizations who are building and testing robot applications for warehouse and logistics use cases.,” GitHub, https://github.com/aws-robotics/aws-robomaker-small-warehouse-world (accessed Feb. 18, 2024).

[3] ROBOTIS, “TurtleBot3 Features,” TurtleBot3 e-Manual, https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications (accessed Feb. 18, 2024).

[4] “move_base,” ROS Wiki, https://wiki.ros.org/move_base (accessed Feb. 23, 2024).

[5] ”costmap_2d,” ROS Wiki, https://wiki.ros.org/costmap_2d (accessed Apr. 20, 2024).

[6] J. Lee, “navfn,” ROS Wiki, https://wiki.ros.org/navfn (accessed Feb. 18, 2024).

[7] Haruharu, DimitriProsserDimitriProsser, 2ROS0, indraneel, and PLui, “Ros answers se migration: Navfn algorism,” Robotics Stack Exchange, https://robotics.stackexchange.com/questions/34631/ros-answers-se-migration-navfn-algorism (accessed Feb. 22, 2024).

[8] S. M. LaValle, Planning Algorithms. New York (NY): Cambridge University Press, 2014.

[9] Wikipedia contributors. ”Dijkstra’s algorithm.” Wikipedia, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm (accesssed 20 Apr. 2024).

[10] Wikipedia contributors. ” Rapidly exploring random tree.” Wikipedia, https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree (accesssed 20 Apr. 2024).

[11] A. Koubaa, “Writing A Global Path Planner As Plugin in ROS,” ROS Wiki, https://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS (accessed Feb. 22, 2024).