Autonomous Mobile Robots Final Competition Report

May 15, 2019

Mabel Lawrence

Overview of Approach

To address the challenges presented in the final competition, our team decided to localize using dead reckoning with added beacon information and create a path to the waypoints with a rapidly exploring random tree (RRT). To begin localization, we had the robot execute a sequence of spinning and moving to open space until it could register a beacon and calculate its position using trigonometric functions cross referenced with depth information. After determining our starting position, we built our RRT to the mandatory waypoints. While building the RRT, we created the overall path by minimizing the number of edges from the most recently examined waypoint to one of the unexamined waypoints using MATLAB's distances function[1], and we repeated this process until all waypoints had been ordered. We initially planned to localize with an Extended Kalman Filter (EKF) using depth information but due to inaccuracies during testing, we decided to pursue a simpler method. Our initial algorithm also created an RRT to both mandatory and extra credit waypoints. We knew that extra credit waypoints would require knowledge of optional walls, so this algorithm's RRT tracked edges that went through optional walls[2] with the plan to update the map with the relevant wall present if the robot's bump sensor was triggered while traversing one of these edges. After updating the map, we planned to regenerate our RRT to the remaining waypoints with the new knowledge of the environment. In testing, we found that 3/5 times the RRT took over a minute to generate at the start of the run and would likewise take a long time to regenerate when accounting for updated optional walls. In our final implementation, we therefor choose to optimize our time rather than pursue the extra credit waypoints and optional wall identification. As such, at the start of the motion control program, we modified the given competition map to assume all of the optional walls were present and then created our RRT within the adjusted environment to find the mandatory waypoints.

Flowchart of Solution

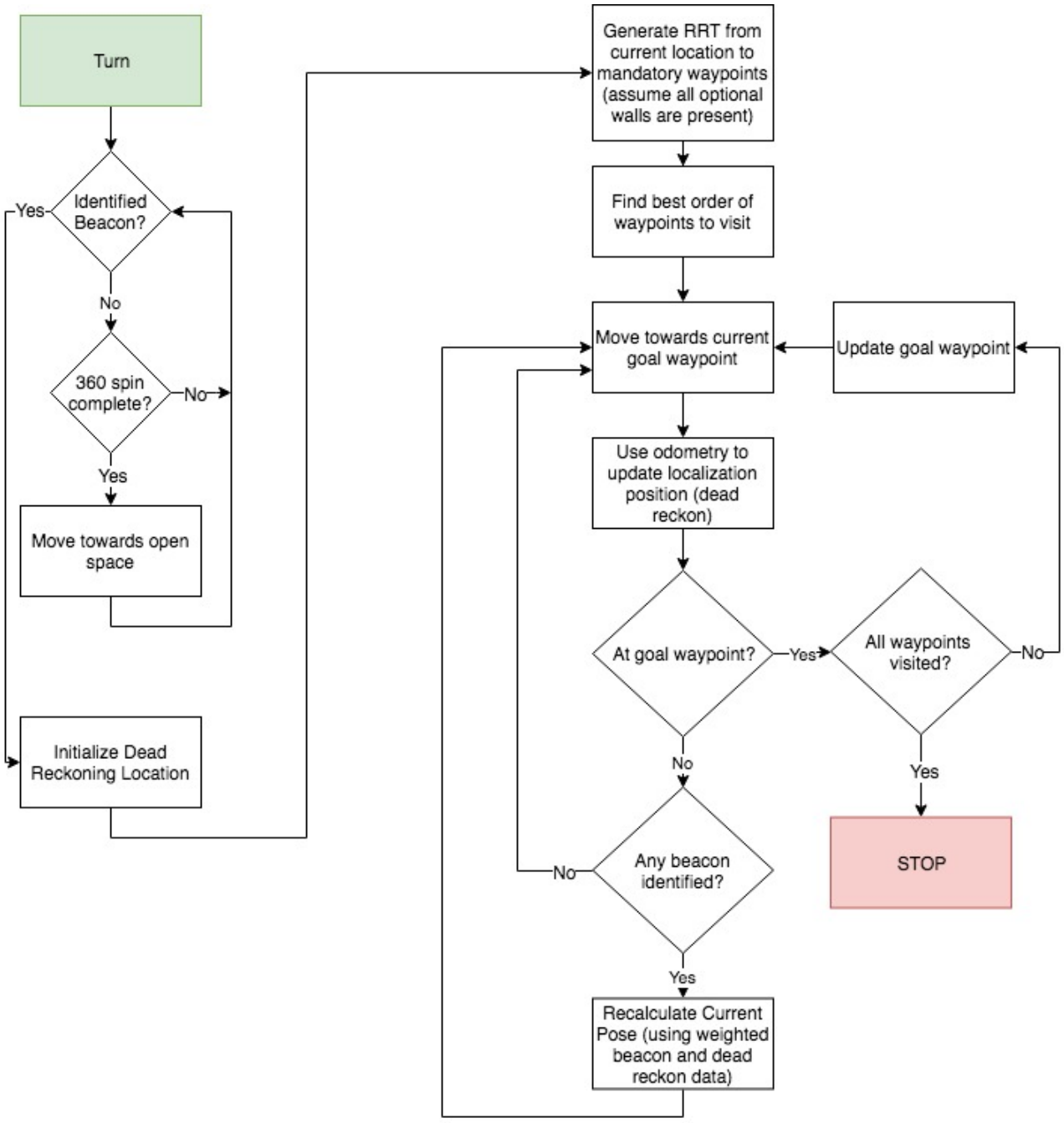

Figure 2: Flowchart of final implemented algorithm

Individual Contribution

I created an RRT path plan because our team found it a consistently effective algorithm to get from qStart to qGoal based on our homework and lab experiences. Furthermore, the one-way RRT requires building only one tree regardless waypoint quantity, so the complexity of implementing a BIRRT outweighed its speed advantages. Before building the RRT, I modified the map walls to have thickness to account for the robots turning radius when traversing edges. The function then continued to add nodes and edges to the tree until a path existed to all waypoints. Although it was not ultimately used, it was challenging to create path information that could be used to determine optional walls during motion control. I addressed this by checking if each edge was in free space assuming none of the optional walls were there and then again assuming only the optional walls were there; if an edge fell in the former category its coordinates would be stored as an edge, and if it fell in both categories its coordinates and which optional wall was intersected would also be stored as an optional edge. Once completing the tree, I found the number of edges between all of the waypoints using the distances function. I ordered the waypoints by connecting the waypoint most recently added to the path to the waypoint the fewest edges away until there was a complete path to each waypoint. However, this method did not always minimize path distance because tree edges were added regardless of length if they could reach a waypoint, but I created the path by minimizing the number of edges. Finally, I input the ordered waypoints Dijkstra's algorithm$_3$ to find all node coordinates along the path. I spent ~12 hours before and after the competition debugging. I found that the way I modified my RRT for multiple waypoints created complicated variable references and concatenating issues when using Dijkstra's algorithm. Due to our timeline and difficulties with motion planning, we did not catch and address the issues that came with integration until after the competition.

Competition Performance

During the final competition, our program failed to execute the tasks in the challenge because it timed out while creating the RRT. During post-competition analysis, I found that the problem causing the time out likely came from how we referenced the competition map file in our motion control function: before the competition, we did not remove the practice map file from our folder and so the actual map file was renamed to be properly opened in simulation. I believe this caused issues in the RRT generation logic because edges were not connecting to goal points properly and were going through walls[4]. Both during the competition and post-competition analysis calls directly to the RRT function, the tree went through the slanted walls on the left side of the map (as can be seen in figure 4), but the rest of the edges were within free space as expected. Since the RRT was not properly connecting to waypoints, the function ran until it timed out after attempting to add 2000 nodes and no path was found, giving the robot nowhere to move. After the competition, I created a new mat file with the competition map information and changed our motion control file to reference it properly which allowed the RRT function performed as expected when called outside of simulation[5]. Figure 5 shows the generated RRT with the new file name and the original algorithm that assumed we were going to check for optional walls and visit extra credit waypoints. Regardless of the file naming issue, the RRT regularly took 45 to 100 seconds to generate with trialed step sizes from .5 to 1.5, so with our given algorithm it was a good choice to only pursue the mandatory waypoints in hopes of completing the main task quickly and efficiently. If we were to refigure our plan, we could have used a greedier planning algorithm like a probabilistic roadmap that has the potential find a path sooner than an RRT. Alternatively, we could have generated RRTs to one waypoint at a time, visiting each as we went so as to tackle task planning completion in segments rather than waiting to generate a full path.

After fixing the RRT generation through renaming the file, I was able to run the simulator with our motion controller to the mandatory waypoints and the RRT generated consistently in 15 – 30 seconds after starting the simulation. While doing this, I found a significant issue in the creation of the complete path where I hardcoded in the index of the starting node input to Dijkstra's algorithm to be the first node in my node matrix – this logic came from how it was created for an RRT with only one goal point. In the context of building a path multiple waypoints, this meant that each time Dijkstra's was called both the start and end nodes needed to change. Before catching this, the list of nodes to visit along the path would repeatedly cycle back to node one after making it to each goal point which created both an inefficient path and confusion in localization because the robot "thought" it needed to return its initial position after reaching a waypoint but had no path to get there. While debugging, I also found issue with the way I was searching for the indexes of waypoint within my node matrix which occasionally caused the RRT to throw errors when looking for the path to a given point. After addressing these issues, the RRT generated as intended and the robot was able to begin motion in simulation.

To perform analysis on our algorithm when it was working as we had intended for the competition, I ran the simulation with the settings given in-class and found that our localization method had inaccuracies that routinely caused the robot to get stuck at an obstacle before reaching all of the waypoints. When starting the robot at [-4.5, -3.5] as was done in competition, the robot made it to the waypoints at [0, -3.5] and [-1.5, 3.5] but got lost when trying to turn around near [-3.5, -1.5] because there were no beacons in that area. Through further runs, I found

that this area, as well as the area around the [0, -3.5] waypoint caused incorrect estimates in localization because there were no beacons in sight. Despite the robot performing backup bump motions, if it got stuck in an alcove with no beacons, it would regularly misestimate its position and eventually get stuck in a corner. The robot performed better when starting at the [4.5, -1] waypoint, as there were more beacons visible along the majority of its path, but it can be seen that it still eventually got stuck in one of the aforementioned alcoves without beacon information[7]. From Figure 6, it can also be seen that the robot misidentified the waypoint at [-1.5, 3.5]. This happened because the robot clipped the side of the wall to the right of the mentioned waypoint and was nudged to the right side of this wall, but never registered that it was no longer on the edge to the waypoint because there was no beacon information, and the robot did not bump and move away from the wall but rather slid along it. It can be seen from where the true and dead reckoning positions diverge in this area that the robot incorrectly "thought" it had jumped to the other side of the wall and made it to the waypoint.

The localization problems could have been addressed had we used EKF with depth information as we had originally planned. When creating our motion planning and localization algorithm, we had been more hopeful that there would be more beacons present, and therefore relied too heavily on beacon information in our localization plan. To improve our implemented algorithm, we also could have programmed a sequence to find open space and scan for a beacon similar to the one used when initializing our localization to recalibrate our position if our bump sensor was repeatedly triggered a certain amount of times (an indicator that something is likely wrong). When starting the robot at [-1.5, 3.5], it can be seen that the algorithm to find open space and locate with a beacon was effective because the program accurately plotted the walls that can be seen from the robot's position each time it spins checking for a beacon and cannot find one[8,9]. I tested the robot starting at this point because it was the second start point chosen by the professor in the competition, and I wanted to see how our algorithm would perform if we were in actual competition. Although the RRT generated properly from this starting point – correctly locating a beacon from free space before creating the RRT[10] – the robot still had trouble fairly quickly due to the lack of beacons in the area around where it was started[11] as can be seen from Figure 10.

After resolving issues with the RRT and path generation, our robot was able to begin motion and traverse the tree to waypoints when presented with ideal starting conditions that had visible beacons spaced consistently along the path. When the robot was in started in or moved to positions not near beacons, the robot eventually got off track of its path every time. The issues with localization also compiled upon each other because there was neither a way for the robot to know when it was off the path or how to get back on it if there were no beacons nearby. If doing this project again, I would put more time into localizing with EKF with depth information to get more accurate information about the robot's position. This would also free up the ability to pursue extra credit waypoints and identifying optional walls because the robot would more reliably be able to find its way back onto a path if it got off track. Additionally, due to the time cost of generating the initial RRT and the potential cost of regenerating RRTs if pursuing the proposed algorithm to find extra credit waypoints and optional walls, I would create smaller individual RRTs to one waypoint at a time or use a probabilistic roadmap.

Appendix

1. Using MATLAB's built in distances function
   https://www.mathworks.com/help/matlab/ref/graph.distances.html

2.

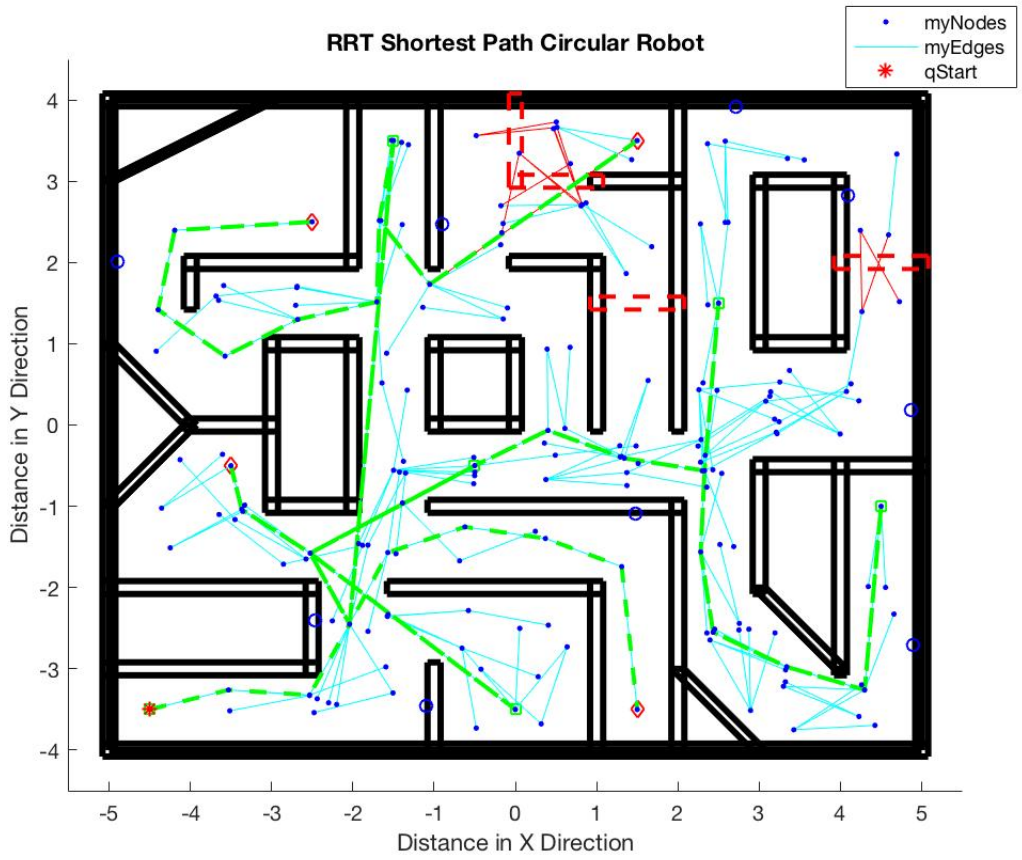

Figure 1: Example of generated RRT to all mandatory and EC waypoints. The shortest path is in green, and the red edges indicated edges through optional walls.

3. Dijkstra's algorithm from Joseph Kirk:
   Copyright (c) 2006, Joseph Kirk
   All rights reserved.
   https://www.mathworks.com/matlabcentral/fileexchange/12850-dijkstra-s-shortest-path-algorithm

4.

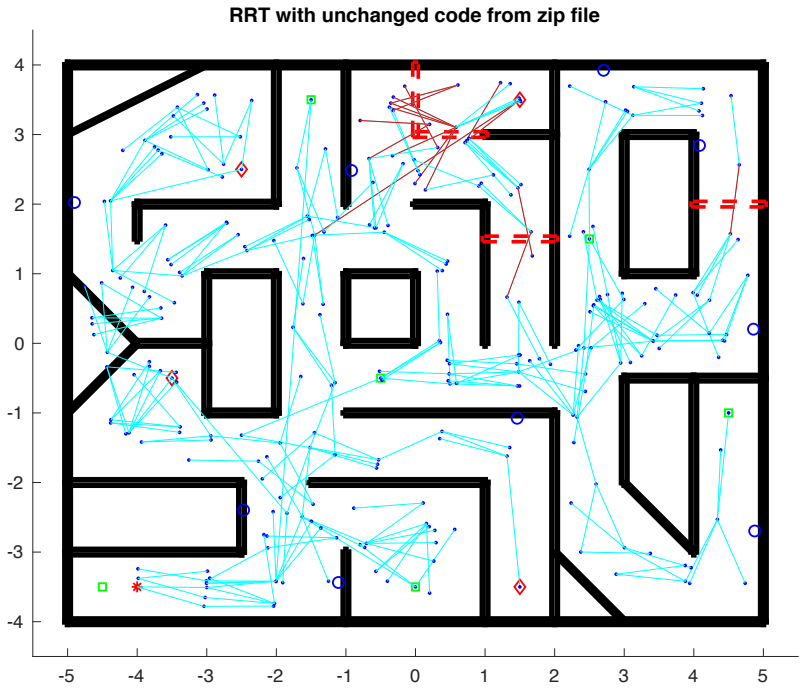

Figure 3: RRT generated from direct call to built RRT function downloaded from CMS competition zip file without edits.

5.

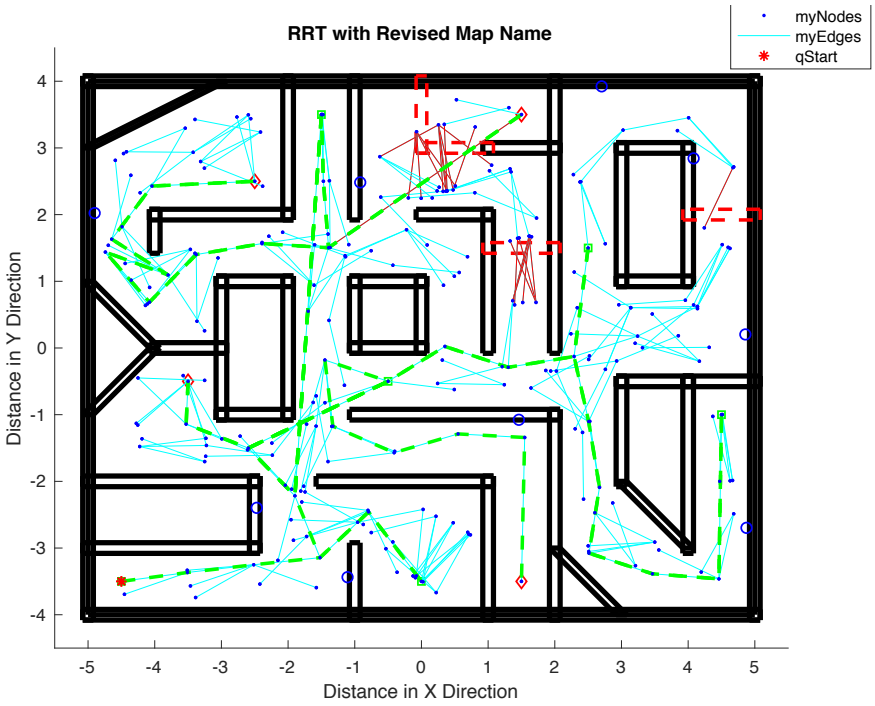

Figure 4: RRT generated from isolated call to buildRRT after competition with new map file name, and original algorithm accounting for optional walls and edges.

6.

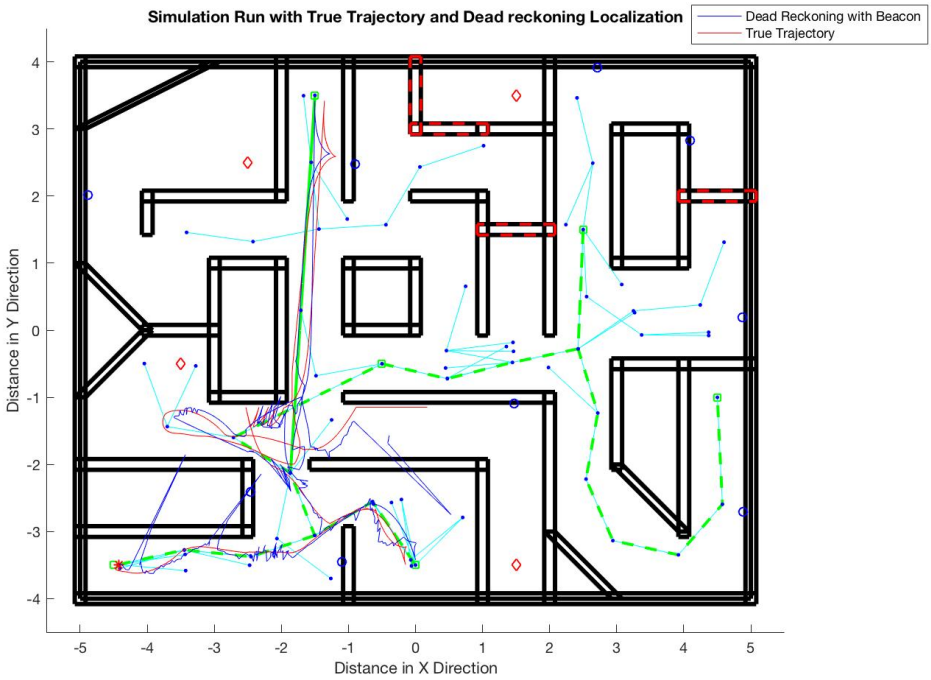

Figure 5: Simulation run starting at [-4.5 -3.5] with intended path in green, true trajectory in red, and our believed dead reckoning trajectory in blue.

7.

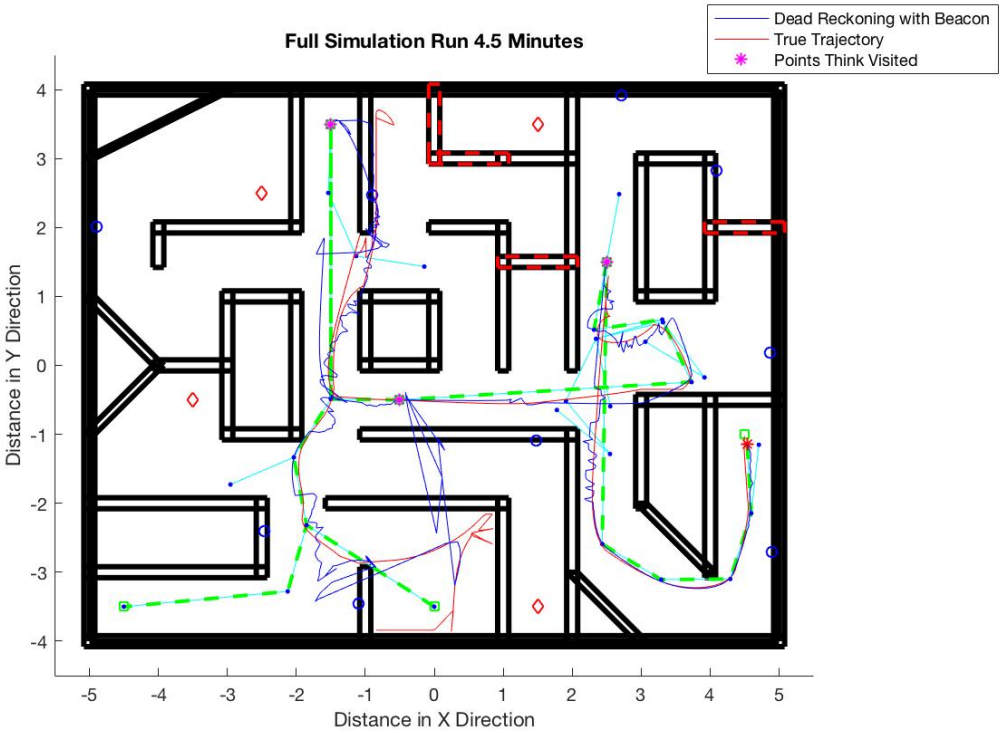

Figure 6: Run in competition settings starting at [4.5 -1] until robot got stuck after 4.5 minutes.

8.

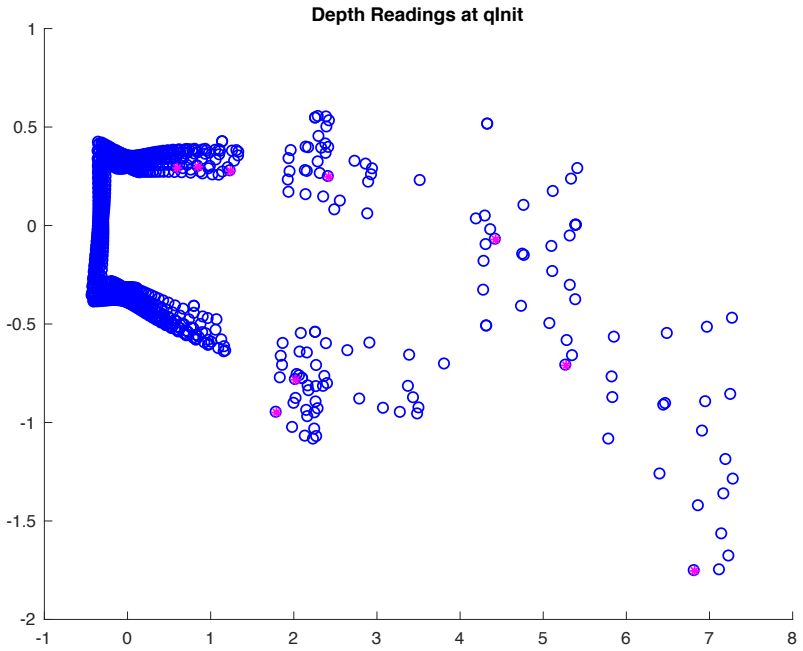

Figure 7: Plot of the depth readings registered by the robot at its initial position when no beacon is seen. The blue dots indicate obstacles, and the robot knows to move to the area with lower density of depth readings.

9.

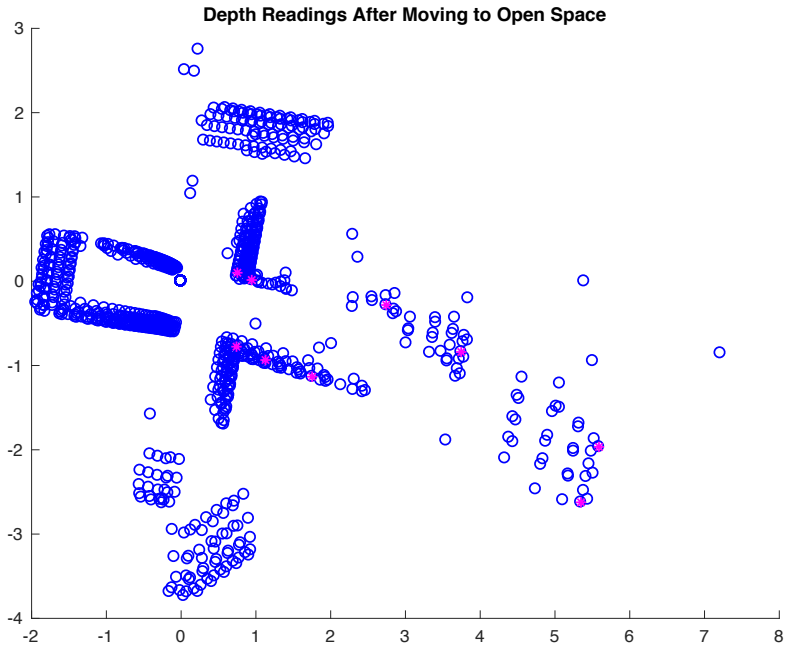

Figure 8: Plot of depth readings after first time robot moved to free space somewhere around [-1.5 1.5] in the map.

10.

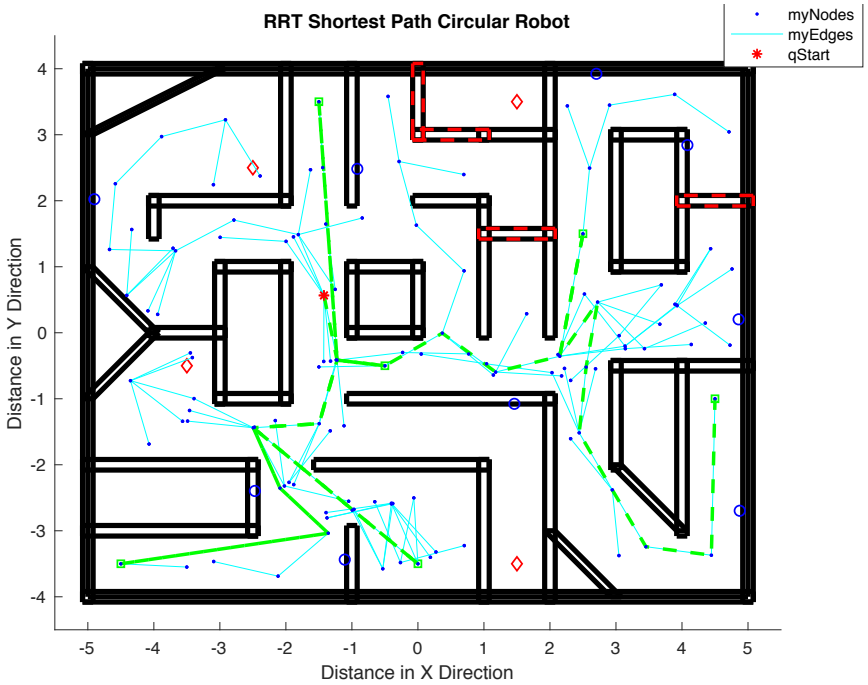

Figure 9: Properly generated RRT after robot initialized localization in site of a beacon in open space.

11.

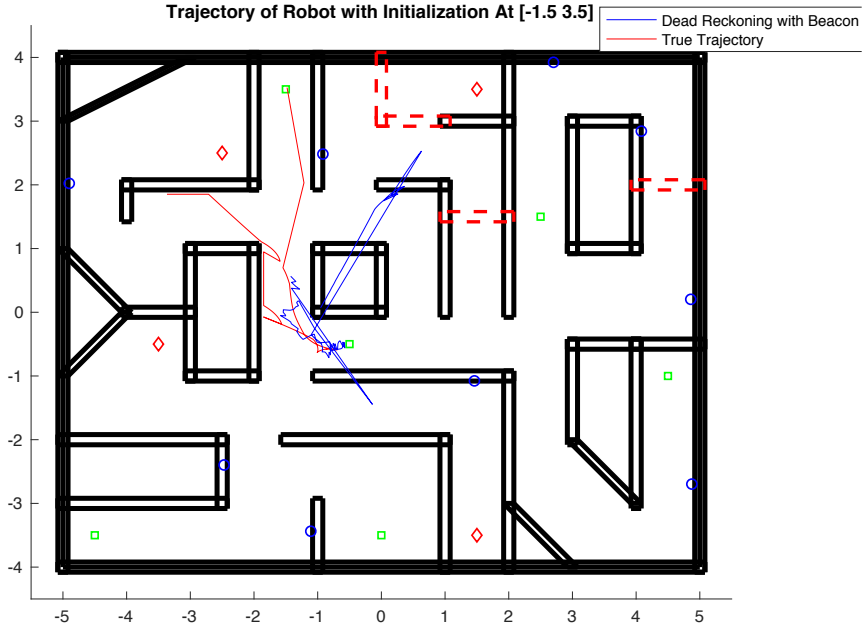

Figure 10: Plot of robot true and dead reckoning trajectory when initialized not near beacons.