Introduction

It is important to note that I did all my work in Euclidean space, adding extra difficulty. Though this project should have been executed in a configuration space, I feel that I have gained a deeper understanding of the Euclidean space because of it. I, also, have a better grasp on the difficulty of working in a Euclidean space versus a configuration space.

Method I

For method one, I implemented an RRT (rapidly exploring random tree) algorithm. In my main script I created ten random obstacles, created a random robot, and defined a q_inital and q_goal which are the initial and goal configurations. With those four variables, I called my RRT function. The RRT algorithm creates a random configuration and sees if it is in collision with obstacles. If the configuration is not in collision with obstacles, we find the closest node already in the tree and add the configuration to a tree. Once the goal configuration is reached, a path is found from q_goal to q_init.

Because MATLAB does not have a built-in tree data structure, I modelled my tree using a 4xn array. The first three elements in my array would be the x position, the y position, and the theta associated with the node and the final element is the index of the parent node.

I initialized my tree with the initial configuration and made the parent index 1. Then in a while-loop I ran the RRT algorithm. The stopping condition for the while loop was if a random configuration was found that was close enough to the goal configuration. If this condition was met, we would add the goal configuration to the tree with the closest configuration as the index.

A problem I faced was that even though a random configuration was not in collision where it was placed, in order for the random /configuration to be reached, the robot would come into collision with an obstacle. In order to remedy this, I created a random configuration and then plotted it in the space. Then I found the closest configuration already in the tree and plotted that using MATLAB. I then created line segments between the current configuration and the random configuration, and if those segments were not in collision with any obstacles, the random configuration was added to the tree.

Once it got close enough to the goal configuration, I added the goal to the tree and searched for the path back to the initial configuration using the parent indexes in the tree. After establishing that path, I reversed it, so it was from q_init to q_goal instead, and returned the path.

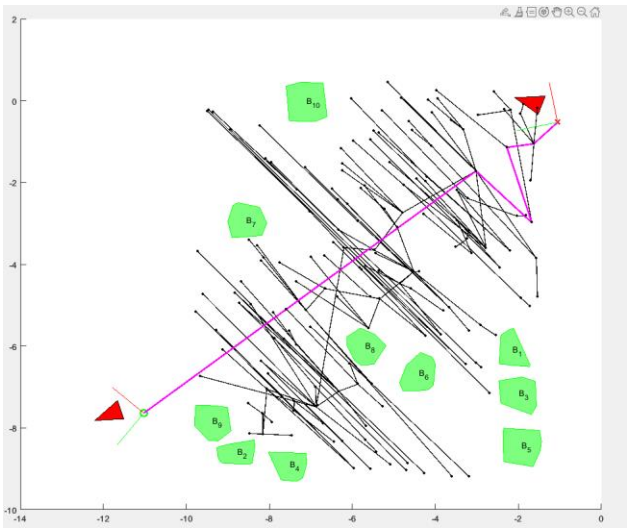

Figure 1: Example of what an RRT looks like in Euclidean Space.

Method II

The second method I implemented was the vertical cell graph in Euclidean space. Once I successfully completed that, I used depth first search (DFS) to compute the shortest path.

I used the vertical cell graph code that was submitted for homework, which leaves me with a configuration that has ten obstacles: the start and initial configurations, the upper and lower segments, and the midpoints. At the midpoints, I find a configuration that is not in collision with any obstacles. I picked a random theta value at each mid-point, and then if it was in collision, I rotated it until I found one that was valid. Some mid points don't have any configurations plotted because all possible configurations would be in collision with an obstacle.

After checking all the midpoints, I go through all the configurations and see what can be connected to what without being in collision. I used a similar method that I had in method one where I plotted all segments connecting two configurations to see if they would collide with an obstacle while travelling to one another. I am then left with a tree with the same format I had method one.

I plug the tree into the DFS algorithm, and it gives me an array with path indices. I find the valid configurations using those indices and print the path. This was the easiest of the three methods to create because most of this code had already been submitted as homework. The only thing I changed was that it is in Euclidean space, and I use DFS algorithm to find a path from q_init to q_goal.

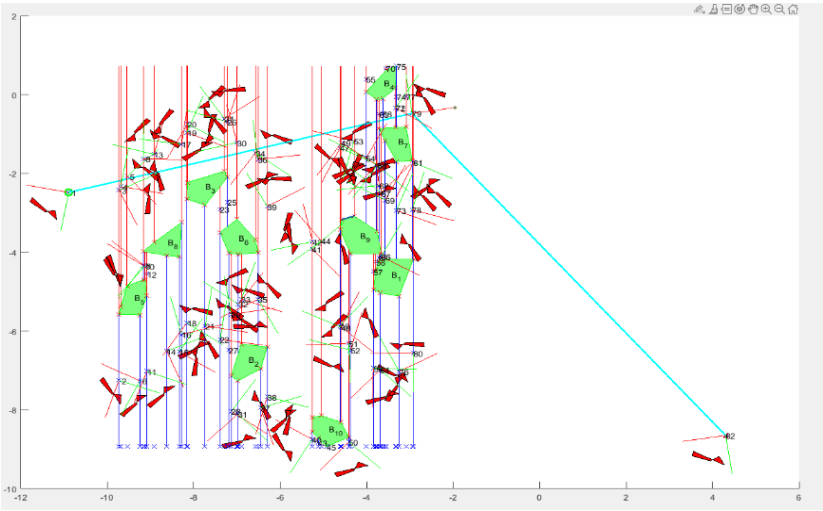

Figure 2: Example of what vertical cell decomposition looks like using DFS algorithm.

Method III

For method three I implemented PRM breadth first search (BFS) to find a path from q_goal to q_init. This was the method I chose to see how kinematic driving constraints and turning affects it. I created the workspace by plotting 20 obstacles and making the bounds significantly larger to allow for turning. I then generated a random radius to act as the car's turning radius and created a circle to be plotted as robots. PRM is very similar to RRT in how we sample things. PRM, however, first finds the valid configurations, creates a tree or adjacency matrix, and then finds a path instead of generating one valid configuration and connecting it to the tree.

For this method I decided to try something different and use an adjacency matrix. Once I had my adjacency matrix, I use BFS to find the shortest path from q_goal to q_init. The most challenging part about writing this method was creating the adjacency matrix. I was unsure of how to check if collisions would occur when one configuration travelled to another. What I ended up doing was taking four points on the circumference of the plotted circle and the center point on each circle and connecting them. If those segments were in collision with an obstacle when I used the intersectSegmentPolygon() method, then they were not added to the matrix.

One drawback of PRM is how long it takes to run. I ran mine taking only 60 random samples and it can take up to 5 minutes to find the shortest path. However, if you use more samples, you are more likely to find a path. I only returned the x and y values of the vertices in my path array because you should be able to be in that space in any configuration and get to the goal configuration from the initial configuration.

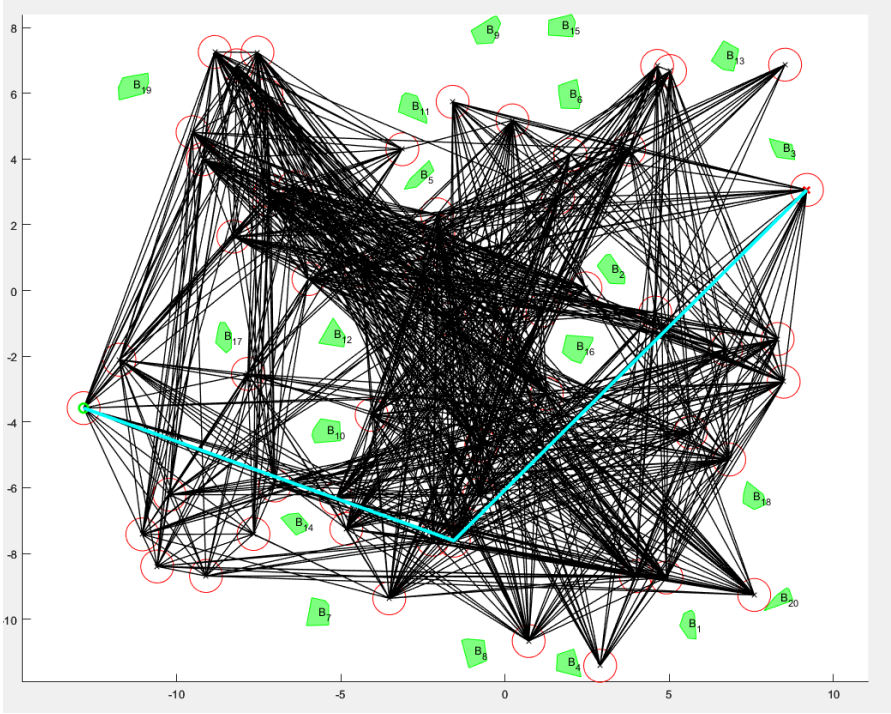

Figure 3: Example of what PRM looks like with BFS.

References

[1] “Robotic Motion Planning: RRT’s - CMU school of computer science,” Carnegie Mellon University, https://www.cs.cmu.edu/~motionplanning/lecture/lec20.pdf (accessed Aug. 17, 2023).

[2] “Probabilistic Path - Department of Computer Science, Columbia University,” Columbia University, http://www1.cs.columbia.edu/~allen/F19/NOTES/probabilistic_path_planning.pdf (accessed Aug. 17, 2023).

[3] “Breadth first search or BFS for a graph,” GeeksforGeeks, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ (accessed Aug. 17, 2023).

[4] “Depth first search or DFS for a graph,” GeeksforGeeks, https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ (accessed Aug. 17, 2023).

[5] G.D. Hager, "Algorithms for Sensor-Based Robotics: Sampling-Based Motion Planning," Baltimore, 2022, pp. 1-39.