ME 5751 Module 4: Probabilistic Road Map

Cal Poly Pomona
Pomona, CA
Jonathan Jenkins
jjenkins@cpp.edu

Abstract*— In this project we developed a probabilistic roadmap or PRM algorithm that will generate an optimal travel path from a user suppled map, start point, and goal point. The methods used to test the controller involve a gui (graphic user interface), which simulates the robot, and a custom python algorithm. The mathematics involved were sourced from "Introduction to autonomous mobile robots", 2nd Ed. by Siegwart, Nourbakhsh, and Scaramuzza. We compared the effectiveness of the PRM algorithm to the A* algorithm. Our findings in this project demonstrate that our PRM algorithm can solve certain simple maps faster than the A* algorithm. However, it also becoming less useable with more complexed maps. Cluttering became a major inefficiently in our PRM algorithm and prevented the planer from correctly developing travel paths in more complex maps. Better optimization is needed to solve cluttering of nodes and allow the PRM algorithm to be used on more complexed maps.

Keywords—Motion Planning, Grid Map, A* Algorithm map, Probability Roadmap, PRM, Python

I. INTRODUCTION

The goal of any robotic motion planner is to construct a collision- free path between some initial position to some goal position. Every motion planner balances a variety of metrics including processing speed, completeness, optimality, and feasibility of solution. Previously we have tested an A* algorithm which had a good balance between all these merits. In this experiment we developed a different algorithm that could produce results far faster with fewer programing iterations using probability. This algorithm is called probabilistic roadmap or PRM algorithm.

The PRM algorithm does not try and find the fastest, shortest, or most efficient path, but rather it aims to use probability to find a valid path solution as quickly as possible with fewer iterations and nodes then other algorithms.

In this project, we will be implementing a custom PRM algorithm that can develop a robot travel paths from a user supplied environmental image, a starting position, and a goal position. After the development of our custom algorithm, we will conduct three experiments to determine the effectiveness of the code. This paper will go into depth on different elements of this project including control theory, python programing, and final evaluation.

II. BACKGROUND

Every day you drive, bike, or walk you are actively path planning. Generating a good travel path is critical to one's ability to navigate within its environment. A path can be created to be the optimal route depending on travel time, travel distance, safest path, or some other metric. As shown in Figure 1 an incorrect path planner can make the difference between reaching your desired goal and crashing.

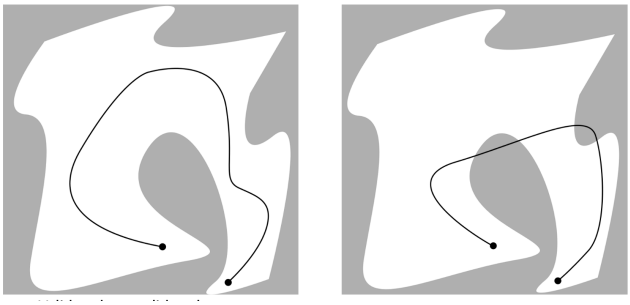


Figure 1 Good vs Bad Path Panning

A path planner begins with a map. Maps can be used to represent a variety of information. In this project we are using two different types of maps, a position map and a cost map. A position map in our case is a grid that shows occupied and unoccupied spaces. This map will be given by the user for each custom environment. An example of a position map is shown in figure 2.

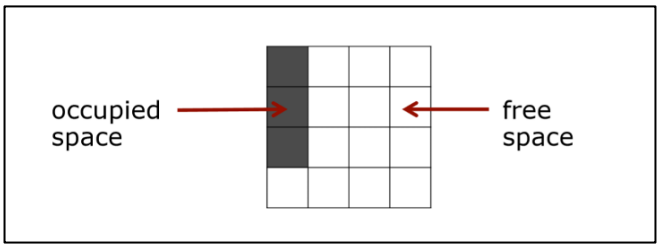


Figure 2 Position Map

The second type of map we will be using is called a potential cost map. The potential cost maps assign a risk value to all empty spaces. Spaces at a closer distance to an occupied space will be given a higher risk or cost and spaces that are farther away are assigned a lower risk. An example of a potential cost map is shown in figure 3.

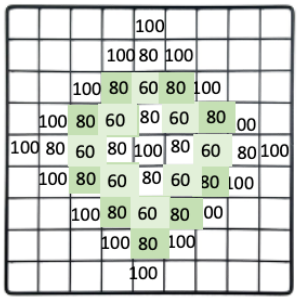


Figure 3 A Potential Cost Map

Our specific potential cost map consists of 3 important concepts. First an occupied space has the highest risk since it represents a physical object. Occupied spaces will have a risk value of 1000. Around all occupied spaces is a "no go zone" or safety zone that is treated like an occupied space. We have this space to compensate for the robot's chasse radius and prevent accidental collisions with the wall. The safety zone will have a cost between 500-800. Lastly a gradient of cost spreads from the end of the safety zone. This cost gradient becomes a cost of 0 once the distance is 1 full robot diameter away from the wall. To assign the cost to each grid a linear equation, as shown in 2.1, was used. A visual representation of our potential cost map is shown in figure 4.

$$y = mx + b \quad m = \frac{y_2 - y_1}{x_2 - x_1} \qquad (2.1)$$

Figure 4 Our Potential Cost map

With a Cost map in hand, we then use a user provided starting position and goal position as inputs to our path planner. The PRM algorithm uses these three inputs to calculate the path between the start position and goal position. An example of a path is shown in figure 5.

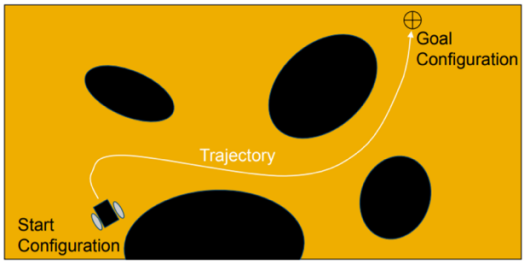


Figure 5 Path from Start to Goal

Fundamentally, a path planner decides what grids should the robot move to reach the desired location. Each grid that can be moved to is called a node. Connections between nodes are called edges. Nodes and edges together form the path between the start and goal position. It is up to the path planner to derive a usable collision-free path. A visual example of nodes edges and a final path is shown in figure 6.

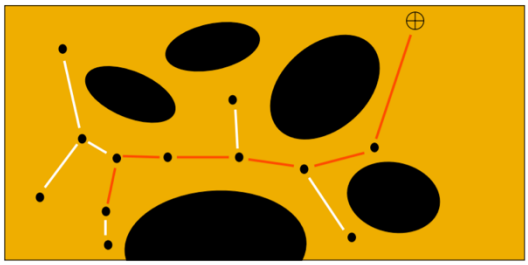


Figure 6 Nodes, Edges, & Final Path

Each node created is saved in a list array. As mentioned, all nodes are connected to another node by an edge. Theses connected node are referred to as a parent and child. This relationship for each grid is stored in a dictionary queue. When reconstruction of path is ready, we pull the list of each parent and each parent's parent beginning from the goal position. This list will eventually lead directly from the goal position to the start position forming the desired path the algorithm calculated. Figure 7 shows parents and child nodes.

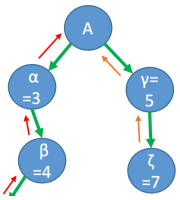


Figure 7 Parent & Child Nodes

With the foundation of what make up a node tree system we then implement the PRM algorithm. It is this algorithm that generates a new node and a new edge and guides the development of the pathway towards the goal position. The generalized PRM algorithm procedure is listed below in figure 8.

- 1. Add start configuration c_{start} to R(N, E)
- 2. Loop
  - 3. Randomly Select New Node c to expand
  - 4. Randomly Generate new Node c ' from c
  - 5. If edge e from c to c ' is collision-free
  - 6. Add (c', e) to R
  - 7. If c ' belongs to endgame region, return path
  - 8. Return if stopping criteria is met

Figure 8 PRM Procedure

PRM algorithm begins by randomly selecting a known node from the saved list of nodes. This node is referred to as *Node C*. Second a new child node is generated by randomly selecting a nearby unoccupied space. This new potential node is referred to as *Node C'*. Next an edge is formed between both nodes. Only if this edge is collision free will the child node C' be considered valid and saved to the list of nodes. If the edge is not collision free a different location is selected, and this process repeats until a valid node is found. Figure 9 shows the expansion of the first node.

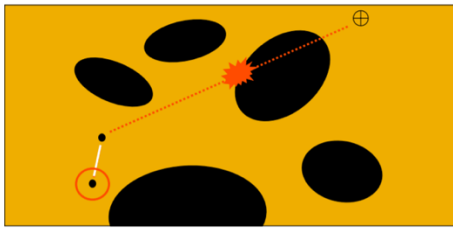


Figure 9 PRM Iteration 1

During each iteration of the PRM algorithm more nodes and edges are added to the node tree. As the node tree becomes denser it will spread out to occupy a larger portion of the map. This is shown in figure 10.

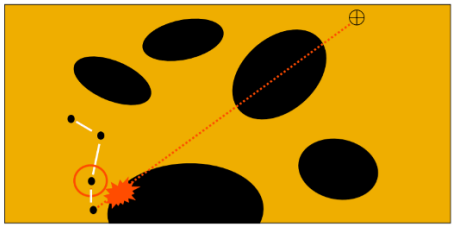


Figure 10 PRM Iteration 3

As the PRM algorithm developed new child nodes and edges it will inspect to see if an edge can be made between the child node and the goal position. If an edge can be made without any collision, then a valid path has been found. Once a valid path is found no additional nodes needs to be expanded and the PRM loop ends. Figure 11 shows the child edge finding a valid edge to the goal node.

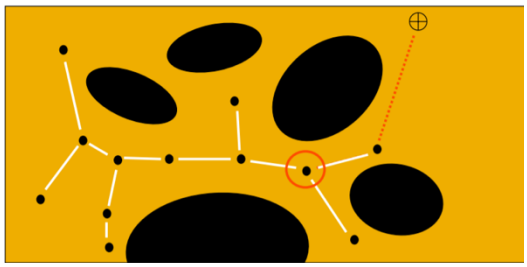


Figure 11 PRM Iteration 11

Now with the optimal path found between the goal and start point the algorithm needs to reconstruct the discovered path into a clear list of instructions. This process is shown in figure 12.

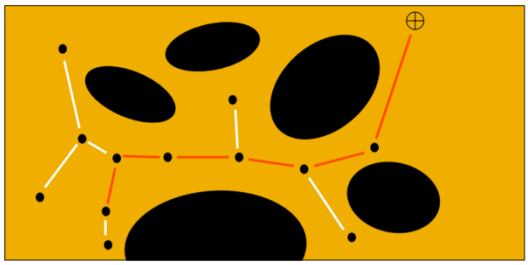


Figure 12 PRM Completed Path

III CONTROL DESIGN

The objective of our algorithm is to generate a path between our robot starting position and a goal position. Implementing a Probabilistic Road Map algorithm our code will take a given cost map, and randomly create nodes as it searches for a path to the goal position. To implement our custom algorithm onto our virtual robot we developed a custom python script to derive the PRM path into three key steps.

Before our PRM algorithm can begin it needs to be given the starting location and goal location. Each location is given in the world coordinate space but if transferred to the image grid space for the path planner function. The code used for declaring our goal and starting location can be seen in figure 13.

```
# Specify Starting Location (0,0)
self.set_start(world_x = 0, world_y = 0)
#Set Goal Location in World coordinate ...  Location will be updated to
self.set_goal(world_x = 126.0, world_y = 59.0, world_theta = .0)
#Run Path Planner Function
self.plan_path()
# Show Calculated Path
self._show_path()
```

Figure 13 Set Start/Goal Positions

Step 1 of our code is used to setup the necessary variables, arrays, queues, and grids that our algorithm will use. We begin by importing our cost map that we will use to check if a space is occupied or not. This cost map will show occupied and unoccupied spaces with assign cost values. Occupied spaces are assigned a value of 1000. While unoccupied spaces are assigned a value of 0-800. Unoccupied spaces with a cost of more than 500 are within a "no-go" zone we called safety zone. We will treat all spaces with a cost value of 500 or more as an occupied space.

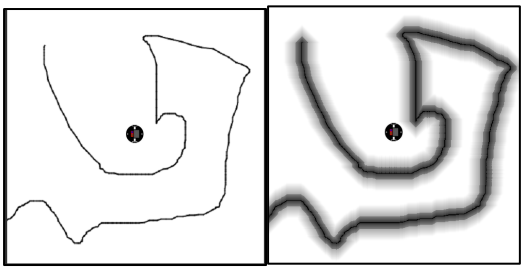


Figure 14 Cost Map

Step 1 also creates the node list and edge list. The queue called *node_list* is used to store the grids position of all created nodes. A node within this queue will be randomly selected when choosing the node to expand from. Lastly the parent of each node is saved in the array called *node_edge*. The code used for step one can be seen in figure 15.

```
#STEP 1:  Setup/Variables/Arrays
#Grids
grid= self.costmap.costmap
self.row_length = len(grid)
self.col_length = len(grid[0])
#Arrays
start =(self.start_state_map.map_i,self.start_state_map.map_j)
goal =(self.goal_state_map.map_i,self.goal_state_map.map_j)
#variables
prm_loop_counter=0
path_found=False
#Queues and Dictionary
node_list=[]                                                    # Lis
node_list.append(start)                                         # Add
node_edge = dict()                                              # Sto
node_edge[start] = None                                         # Ass
```

Figure 15 Step 1 Code

Step 2 contains the bulk of the PRM algorithm logic loop. Set 2 is divided into 5 sub steps correlating to the PRM outline in figure 8. The algorithm begins with step 2a where a new node c is randomly selected from the list of nodes. In step 2b the node c' is generated by randomly expanding from node c a certain distance. After the new node c' is chosen then the edge connecting both nodes is calculated by using the Bresnahan algorithm. In step 2C this edge is checked to ensure no points crosses an occupied space. If no points cross an occupied space, then the new node c' and edge is considered valid and is added to their respective queues.

Lastly, we check if the new node c' has a clear pathway to the goal position, if it does, we immediately add the goal position to the pathway. If not, we repeated the loop of randomly expanding out our nodes, creating new nodes and edges while searching for a pathway to the goal position. The code used in step 2 can be seen in figure 16.

```
# STEP 2A RANDOMLY SELECT NODE C  FROM LIST
random_node_c = random.choice(node_list)            # Randomly select a node c from the
(rx,ry) = random_node_c                             # get the node position  from the

# STEP 2B RANDOMLY GENERATE NEW NODE C' FROM C
random_node_c_prime_valid = False                   # Reset the logic bit to false
ri = random.randint(0,self.map_width-1)             # Random numver i variable
rj = random.randint(0,self.map_height-1)            # Random numver j variable
random_node_c_prime = (ri,rj)                       # save current grid  position to be
if grid[random_node_c] < 500    :                   # Make sure random node is in a unc
    random_node_c_prime_valid = True
if(self.check_vicinity(self.goal_node.map_i,self.goal_node.map_j,ri,rj,2.0)):
    print ("We hit goal!")
    random_node_c_prime_valid = False

# STEP 2C GENERATE EDGE E FROM NODE C' FROM C  - MAKE SURE IT IS COLLISION -FREE
if  random_node_c_prime_valid == True:                          # Perfem only if
    points = bresenham(rx,ry,ri,rj)                             # Generate pint li
    hit_obstacle = False                                        # Reset Hit obstac
    for p in points:                                            # check if straigh
        if(self.costmap.costmap[p[0]][p[1]]) >= 500:            # path is not clea
            hit_obstacle = True
            break
    if(hit_obstacle==False):                                    # Path is clear
        node_list.append(random_node_c_prime)                   # Add new node to
        node_edge[random_node_c_prime] = random_node_c          # store the parent

# STEP 2D GENERATE EDGE BETWEEN NODE C' TO GOAL - MAKE SURE IT IS COLLISION FREE
if  random_node_c_prime_valid == True and hit_obstacle==False:
        points = bresenham(ri,rj,self.goal_state_map.map_i,self.goal_state_map.map_j)
        hit_obstacle = False                                            # Reset Hit ob
        for p in points:                                                # check if str
            if(self.costmap.costmap[p[0]][p[1]]) >= 500:                # Path is not
                hit_obstacle = True
                break
        if(hit_obstacle==False):                                # path is clear .. pat
            path_found=True                                     # Variable to be used
            node_edge[goal] = random_node_c_prime               # store the parent nod
```

```
# STEP 2E OPTIMALSATION GENERATE EDGE E FROM NODE C' FROM C  - MAKE SURE IT IS COLLISION -FREE
if  random_node_c_prime_valid == True:                                          # Pe
    points = bresenham(self.start_state_map.map_i,self.start_state_map.map_j,ri,rj)        # Ge
    hit_obstacle = False                                    # check if straightline from sta
    for p in points:                                        # check if straightline from the
        if(self.costmap.costmap[p[0]][p[1]]) >= 500:        # depends on how you set the val
            hit_obstacle = True
            break
    if(hit_obstacle==False):                        # We didn't hit an obstacle
        node_edge[random_node_c_prime] = start      # determine if new node can directly go
```

Figure 16 Step 2 Code

Step 2 is nested inside a loop that will continue to loop until a pathway to the goal is found. However, it is possible for the algorithm to never find the goal position and in this senecio the algorithm would be stuck in an endless loop. To prevent this a loop counter is used to break the loop after 100,000 iterations. With this counter we placed an upper limit of how many iterations we will allow the algorithm to conduct. The code used for this is shown in figure17.

```
#STEP 2 PRM ALGORITHM LOOP
#Perform Path Plan search
while  path_found == False:
```

```
#STOPPING CRITERIA
prm_loop_counter = prm_loop_counter+1
if prm_loop_counter >= 100000:
    path_found=False
    break
```

Figure 17 While loop and PRM Loop Counter

The third step in the algorithm consist of reconstructing the path if a path was found during the PRM loop. Now with the final position found (goal) we can search who the parent node was for each child node all the way back to the starting position. As mentioned before the parent for each node is stored in the dictionary queue called *node_edge*. Then using the Bresnahan function a line connecting each node is used to reconstruct a path between each node. Finally, the program will export the calculated final path. This code is shown in figure 18.

```
#SET 3   Generate output path
# Output... Depends if path was found or not
if path_found==True:
    print ("Constructing Found path..... ")
    points = reconstruct_path(node_edge, start, goal)
    for p in points:
        self.path.add_pose(Pose(map_i=p[0],map_j=p[1],theta=0))
    self.path.save_path(file_name="Log\prm_path.csv")
if all_path_found==True:
    for p in points_all:
        self.path.add_pose(Pose(map_i=p[0],map_j=p[1],theta=0))
    self.path.save_path(file_name="Log\prm_path.csv")
if path_found==False:
    print("Iteration Limit Reached: NO PATH TO GOAL FOUND")
```

Figure 18 Path Reconstruction and Output

IV. METHOD

After completing the development of our PRM algorithm, we then moved to develop suitable challenges to test the code. The algorithm will be tested on multiple different maps, and on each map the algorithm will be evaluated by three metrics. First, we will evaluate the successful creation of the path planner without any collision with occupied spaces. A successful map generation is shown in figure 19.

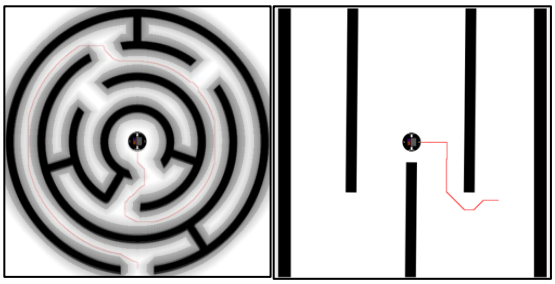


Figure 19 Exported Cost Map Overlay

Second, we will evaluate the number of loops the PRM algorithm performed to find the pathway to the goal position. The same PRM loop counter used for the stopping criteria will be used here as well. The PRM node counter is shown in figure 20.

```
#STOPPING CRITERIA
prm_loop_counter = prm_loop_counter+1
if prm_loop_counter >= 100000:
    path_found=False
    break
```

Figure 20 Node Counter Variable

Lastly, we will compare how the PRM algorithm to the A* algorithm developed in our previous report. An Example of the A* algorithm is shown in figure 21. Using this process, we will evaluate the effectiveness of the PRM algorithm as a path planner.

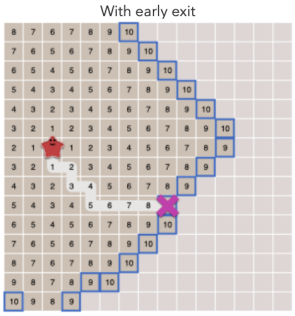
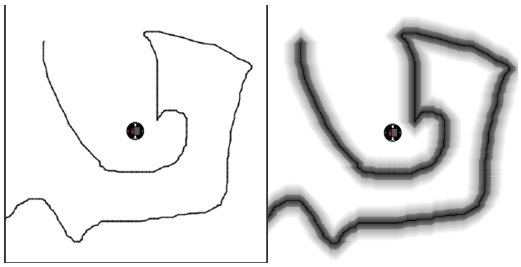


Figure 21 A with Early Exit*

V. EXPERIMENT RESULTS AND DISCUSSION

As mentioned, to determine the effectiveness of our PRM algorithm we tested the program with three custom maps. Each test will require the program to successfully develop the individual grid cost, find a path to the goal and finally export it to the path planner.

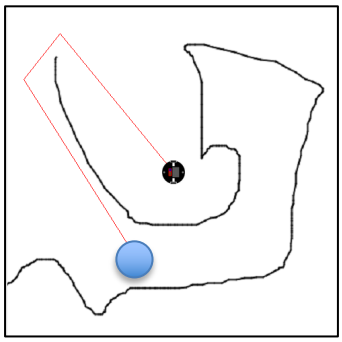
The first experiment uses the professor's provided test map. This map represents an irregular, unstructured obstacle. Figure 22 shows the given map, the cost map, and the robots' starting and goal position.



```
# Specify Starting Location (0,0)
self.set_start(world_x = 0, world_y = 0)
#Set Goal Location in World coordinate ...  Location will be updated to
self.set_goal(world_x = -152.0, world_y = -73.0, world_theta = .0)
```

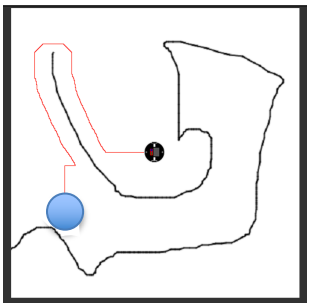
Figure 22 Experiment 1 Map

We begin by performing the PRM algorithm and recorded the results. As shown in figure 23 the PRM was able to find the path in 1,729 iterations and created 1,095 nodes during the process. During this same time the A* algorithm took 395,907 iterations to calculate a path to the goal. This represents a 99% reduction of iterations needed to calculate a path planner solution



```
Path To Goal Found
Constructing Found path.....
Number of Iterations
1729
Number of nodes
1095
```

Figure 23 Experiment 1 PRM Algorithm



```
PATH TO GOAL FOUND
the number of nodes searched with Early Exit
395907
```

Figure 24 Experiment 1 A Algorithm*

Next, we tested changing the goal position to different locations within the map. On every location chosen the A* algorithm was able to successfully develop a pathway to the goal position. Unfortunately, the PRM algorithm was not able to find a path when the goal position was placed in certain locations. An example of this failure is shown in figure 25. After 100,000 iterations the prm algorithm was not able to find a pathway to the goal position. This became a constant issue on majority of maps.

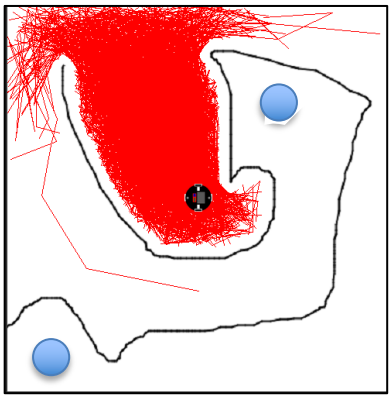


Figure 25 Experiment 1 PRM Failure

The second experiment uses a custom maze map. This map will test the algorithm ability to solve a more complex map with walls and other sharp corner. As shown in figure 26 the PRM was able to find a path to the goal in 65 iterations and created 29 nodes during the process. The A* algorithm took 173,733 iterations to calculate a path to the same goal. This represents a 99% reduction of iterations needed to calculate a path planner solution. Figure 27 shows the number of nodes searched by the A* algorithm.

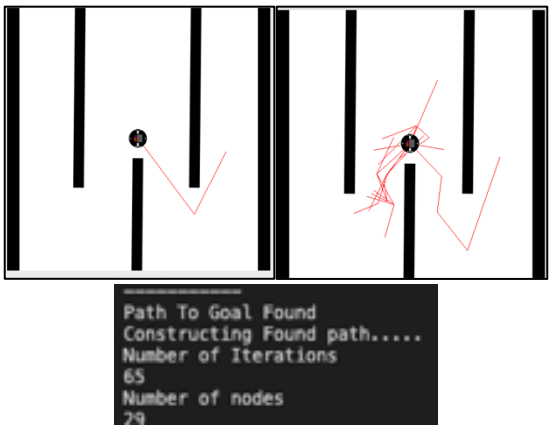


Figure 26 Experiment 2 PRM Algorithm

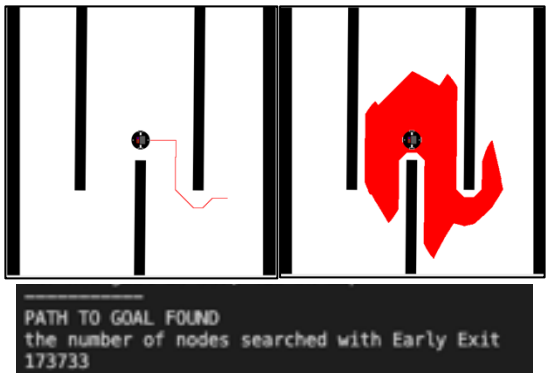


Figure 27 Experiment 2 A Algorithm*

The PRM algorithm was eventually able to solve every tested goal position attempted on this map. However, the amount of iteration time varies dramatically. Even keeping the goal position at the same position, the number of iterations varied from 65 iterations all the way to 1607 iterations. This is up to 2472% variation on number of iterations. Meanwhile, the A* algorithm consistently took the same number of iterations to solve the path to the same goal position.

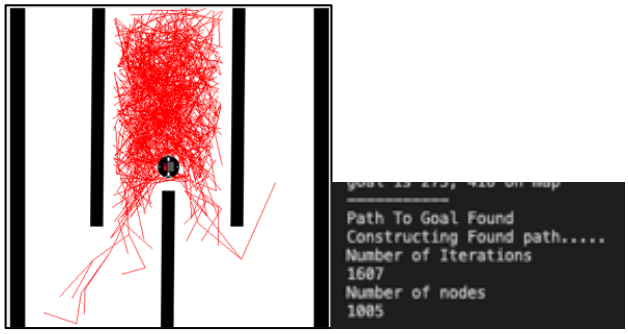


Figure 28 Experiment 2 PRM Algorithm Variation

The final experiment uses a custom maze map. This map represents complex obstacle with many false paths. We begin by evaluating the number of iterations the A* algorithm took to complete this maze. The A* algorithm completed he pathway in 234,636 iterations. Unfortunately, the PRM algorithm was unable to successfully develop the pathway for this map. This is likely due to the complexity of the map. The random selection of nodes ended up being concentrated in the center of the map and was never able to progress through the map. Figure 29 and figure 30 shows the outputs of A* and PRM algorithms respectfully.

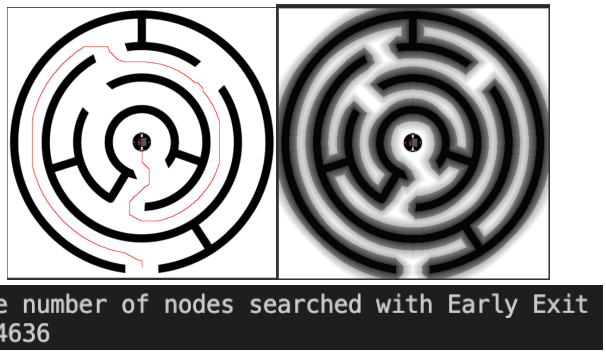


Figure 29 Experiment 3 A Algorithm*

Figure 30 Experiment 3 PRM Algorithm Failure

On every map a common flaw was discovered with the PRM algorithm, Clustering. Clustering is when the density of randomly selected nodes tends to stay relatively close to the starting location and does not evenly distribute across the map. A visualization of this can be seen on figure 31.

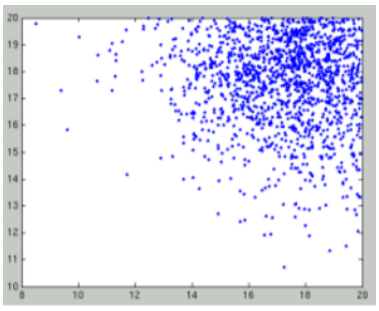


Figure 31 PRM Cluttering

Cluttering prevented the PRM algorithm for finding pathways to the goal on more complex maps or where the goal position requires a long non-strait path. Cluttering tends to occur when randomly selecting nodes without any bias. Implementing a more advance PRM such as Rapidly expanding Random Tree should solve or at least mitigate this error. Overall, the PRM program was successfully able to develop simple pathway but failed for more complexed path planning. When a path was found no visible errors appear in the final map.

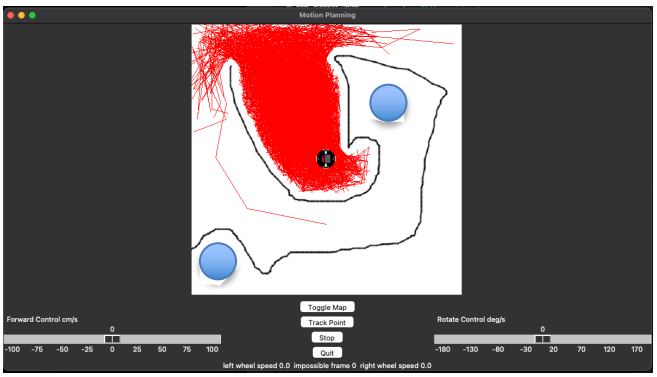


Figure 32 Cluttering on a Map

VI. CONCLUSION

In modern robotics good path planning is critical for effective navigation. A robot following a poor path can cause unnecessary wasted travel time or even cause the robot to collide with its environment. The goal of this lab is to develop an PRM algorithm capable of generating a viable path to take the robot from point A to point B. The PRM algorithm does not to find the fastest, shortest, or most efficient path, but rather it aims to use probability to find a valid path as quickly as possible with fewer iterations and nodes. We evaluated the algorithm effectiveness by testing it on three different maps, each being progressively more difficult.

On certain maps the algorithm was successful at creating viable paths. Depending on the map the PRM algorithm had up to a 99% reduction of iterations performed compared to A* algorithm. However, on other more complexed maps the PRM algorithm becoming slower than A* and at times completely fails at finding a valid solution. Cluttering became a major inefficiently in our PRM algorithm and prevented the planer from correctly developing travel paths in more complex maps. Better optimization is needed to solve cluttering of nodes and allow the PRM algorithm to be used on more complexed maps.

VII. ACKNOWLEDGEMENTS

None

VIII. REFERENCES

Jenkins, Jonathan and Erik Duque. "ME 5751 Module 1: Proportional Control for Point Tracking." 2022. *ME5751_Module1*. https://github.com/jonjenkins31/ME5751_Module1

Red Blob Games. *Introduction to the A* Algorithm*. 26 May 2014. 10 2022. https://www.redblobgames.com/pathfinding/a-star/introduction.html

Siegwart, Roland, Illah R Nourbakhsh and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. Second Edition. Massachusetts Institute of Technology, 2011.