Implementing Path Planning with Probabilistic Roadmaps (PRM)

Sritoma Nandan

Abstract— **Path planning is a crucial coding strategy that helps automated robots to make its way through obstacles. This task aims for the same. It asks us to implement PRM path planning technique to solve a maze . This same algorithm can also be fed into any aerial or ground robot os that it can make its way while looking for obstacles and by staying in the configurational or danger free zone assigned to it .**

I. INTRODUCTION

The probabilistic roadmap planner is a motion planning algorithm in robotics, which solves the problem of determining a path between a starting configuration of the robot and a goal configuration while avoiding collisions.

The basic idea behind PRM is to take random samples from the configuration space of the robot, test them for whether they are in the free space, and use a local planner to attempt to connect these configurations to other nearby configurations. The starting and goal configurations are added in, and a graph search algorithm is applied to the resulting graph to determine a path between the starting and goal configurations.

PyGame-Learning-Environment (PLE) is a learning environment, mimicking the Arcade Learning Environment interface, allowing a quick start to Reinforcement Learning with Python. It focuses on allowing the practitioner to focus on the model allowing fast iteration cycles.

II. PROBLEM STATEMENT

2D Image Implementation

Develop a Python program that implements the PRM algorithm for the given 2D image map maze.png representing the environment. We need to run the algorithm for both Start Easy and Start Hard

Requirements

1. The program should read the image map provided and identify obstacles.
2. Implement the PRM algorithm to find a path between the start point and the endpoint.
3. Visualize the implementation on the original image map.

Pygame environment Implementation

Utilize the autonavsim2D library to create a more dynamic environment.Adapt the PRM implementation to function within the Pygame environment provided by autonavsim2D

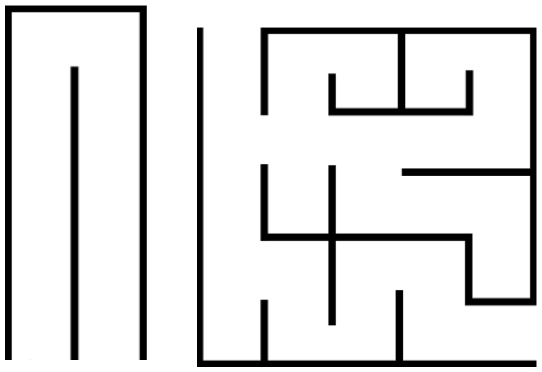
Requirements

1. Allow the user to define the robot's starting and goal configurations within the Pygame environment.
2. Utilize the PRM algorithm to find a collision-free path between the start and endpoints.
3. Visually represent the robot navigating the path within the Pygame environment.

III. INITIAL ATTEMPTS

I started out with clearing my concepts about probabilistic roadmaps and Dijkstra's algorithm. I started off with raw coding uniform sampling , valid point identification ,PRM calculation and finally the dijkstra's . I faced several problems after that .

Firstly ,I did not focus on the image maze.png provided to us for which the external blank space are was also being considered as the configuration space and hence hampering the program . So I made non technical attempts to extract the easy maze and hard maze for checking my code . the pictures are given below .



Next , I realized that there was problem in sampling that was recurring and had to debug various no. of times . The function for edge detection wss later included as the path parsed through the obstacle boundary.

The Dijkstra's algorithm was hard to implement but finally after coming across the heapq library .It was comparatively easy to implement.

IV.FINAL APPROACH

The use of externally manipulated image needed to be removed and hence after getting a desired path I processed the given image maze.png and added extra obstacles to close the maze in a copied image which was not displayed in the output.

The whole program has been divided into different functions,which are as follows :

Valid point or not : This functions checks if the pixel of the passed co-ordinate is in the configurational space of not so that the sampling can be done .

```
def is_valid_point(point, maze):
   x,y= point
   rows, cols = maze.shape
   return 0 <= y< rows and 0 <=x< cols and maze[y,x] == 255
```

Valid edge or not : This functions checks if the line joining two chosen points passes through any obstacle or maze wall which would make it invalid for consideration.

```
def is_valid_edge(p1, p2, maze):
   x1, y1 = p1
```

```
    x2, y2 = p2
    dx = x2 - x1
    dy = y2 - y1
    steps = max(abs(dx), abs(dy))
    if steps == 0:
        return False
    for i in range(1, steps + 1):
        x = int(x1 + float(i) * dx / steps)
        y = int(y1 + float(i) * dy / steps)
        if not is_valid_point((x, y), maze):
            return False
    return True
```

Sampling : This functions uses the randInt() function from the random library to sample out random points from the maze area that helps make possible nodes for the roadmap.

```
def sampling(maze):
    while true:
        x = int(np.random.randInt(0, maze.shape[1] - 1))
        y = int(np.random.randInt(0, maze.shape[0] - 1))
        point = (x, y)
        if is_valid_point(point, maze):
            return point
```

Probabilistic RoadMap : This functions initializes a list 'roadmap' with two points (the start and the end points), which is then added onto using the previously defined sampling function until the list reached the maximum no. of sampling allowed i.e num_nodes.

```
def prm(maze, start, goal, num_nodes):
    roadmap = [start, goal]
    while len(roadmap) < num_nodes:
        random_point = sampling(maze)
        if is_valid_point(random_point, maze):
            roadmap.append(random_point)
    return roadmap
```

Dijkstra's Algorithm : It implements bawsic djikstra's algorithm, for finding the shortest path in a graph, on the passed dictionary that contains distance from start point to that point and its co-ordinate. The algorithm maintains a priority queue (pq) to explore nodes in order of their distances from the start node. It iteratively selects the node with the shortest distance from the priority queue, explores its neighbors, updates the distances, and continues until the goal node is reached. It then backtracks from the goal node to the start node to reconstruct the shortest path. If no path exists, it prints a message and returns an empty list. Finally, it returns the shortest path as a list of nodes.

```
def dijkstra(graph,maze, start, goal):
    pq = [(0, start)]
    heapq.heapify(pq)
    visited = set()
    parent = {}
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    while pq:
        current_dist, current_node = heapq.heappop(pq)
        if current_node == goal:
            break
        if current_node in visited:
            continue
        visited.add(current_node)
        for neighbor, weight in graph[current_node]:
            if neighbor in visited:
                continue
            new_dist = current_dist + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                parent[neighbor] = current_node
                heapq.heappush(pq, (new_dist, neighbor))
    path = []
    while goal != start:
        path.append(goal)
        if goal not in parent:
            print("No path found from start to goal.")
            return []
        goal = parent[goal]
    path.append(start)
    return path[::-1]
```

Path plotting: The shortest path passed found by dijkstra's algorithm is now shown on the original image maze.png using this function . All the sampled points and the start and the goal points are also shown through this .

```
def plot_maze(img, start, goal, roadmap, path):
    for point in roadmap:
        img = cv2.circle(img, tuple(point), 1, (255, 255, 0), -1)
    for i in range(len(path)-1):
        cv2.line(img, tuple(path[i]), tuple(path[i+1]), (255, 0, 0), 2)
    img = cv2.circle(img, tuple(start), 3, (0, 255, 0), -1)
    img = cv2.circle(img, tuple(goal), 3, (0, 0, 255), -1)
    cv2.imshow('maze', img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

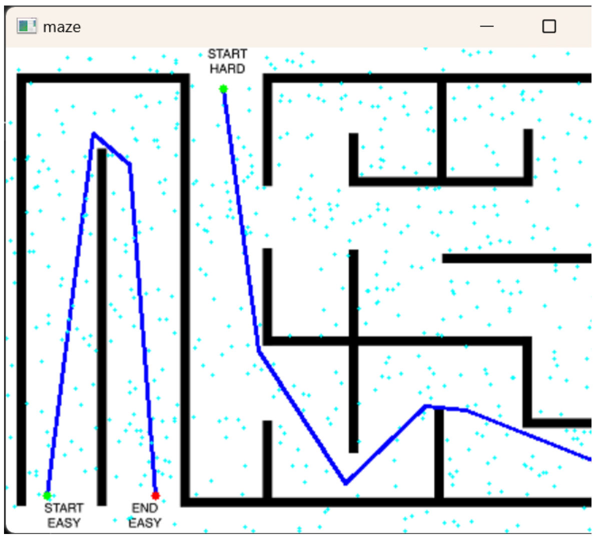
Path plotting: this function mainly aims at calling all the above defined functions in order and passing the correct order of parameters for the smooth working of the program.

```
def implement_prm(maze,start,goal,num_nodes):
    roadmap = prm(maze, start, goal, num_nodes)
    graph = {node: [] for node in roadmap}
    for i, node in enumerate(roadmap):
        for j in range(i + 1, len(roadmap)):
            neighbor = roadmap[j]
            if is_valid_edge(node, neighbor, maze):
                dist = np.linalg.norm(np.array(node) - np.array(neighbor))
                graph[node].append((neighbor, dist))
                graph[neighbor].append((node, dist))
    shortest_path = dijkstra(graph, maze, start, goal)
    plot_maze(image, start, goal, roadmap, shortest_path)
```

Main method: The main() method hereby copies the image read (maze.png) to make additional obstacles on it and then processed accordingly .All the start and end points are initialized and then through a loop both easy and hard maze are processed one after the other .

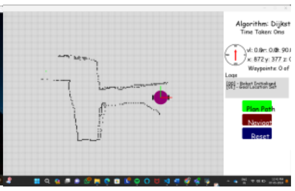
```
def main():
    img=image.copy()
    img=cv2.line(img,(11,340),(128,340),(0,0,0),10)
    img=cv2.line(img,(135,20),(190,20),(0,0,0),10)
    img=cv2.line(img,(445,280),(445,330),(0,0,0),10)
    maze_gray = cv2.cvtColor(img,
cv2.COLOR_BGR2GRAY).astype(np.uint8)

ret,maze=cv2.threshold(maze_gray,243,255,cv2.THRESH_BINARY)
    start_easy=(30,330)
    end_easy=(110,330)
    start_hard=(160,30)
    end_hard=(435,305)
    num_nodes = 300

    for i in range (0,2):
        if(i==0):
            start=start_easy
            goal=end_easy
            implement_prm(maze,start,goal,num_nodes)
```

```
else:
   start=start_hard
   goal=end_hard
   implement_prm(maze,start,goal,num_nodes)
```



Follwing this I went on to try using autonavsim2D. I installed pygame and autonavsim2D and ran the code to get an idea of what it was . followed by window size changing and trying to implement custom_planner in the given pygame environment .

AutoNav2D Simulator
Write code, Simulate Behaviour, Deploy To Innovation
What would you like to do?
Autonomous Mobile Robot Simulation - Differential Drive
Autonomous Mobile Robot Simulation - Skid-Steer Drive
4-Wheel Self-driving Vehicle



Results and Observation

Although my code successfully executed what it was supposed to execute , it showed time complexity error and was slower than expected . Probably other path planners like RRT or RRT* would have been more efficient here.

CONCLUSION

In conclusion, the implementation of the Probabilistic Roadmaps (PRM) algorithm for maze-solving showcases its effectiveness in navigating complex environments while avoiding obstacles. By efficiently sampling nodes, constructing roadmaps, and utilizing Dijkstra's algorithm, the PRM solver successfully finds collision-free paths from start to goal points. While variations in maze size and complexity may influence performance, fine-tuning parameters can optimize efficiency. Overall, the PRM maze solver demonstrates its versatility and reliability, offering a valuable tool for robotic path planning in diverse maze-like scenarios.

References

[1] OpenCV documentation
[2] GeekforGeeks
[3] Stack Overflow
[4] Kavraki, L. E., et al. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces.
[5] autonavsim2D