Aerial Robotics Kharagpur Tasks 2 Implementing Path Planning with Probabilistic Roadmaps (PRM

Upendra Singh

Abstract— **This research paper explores the implementation of the Probabilistic Roadmaps (PRM) algorithm for path planning in a two-dimensional (2D) environment. The paper outlines the development of a Python program to implement the PRM algorithm and its integration into the Pygame environment using the autonavsim2D library. The program aims to find collision-free paths between start and goal configurations within a maze-like environment represented by an image map. Additionally, the paper discusses the broader applications of path planning algorithms in robotics, including autonomous navigation, industrial automation, and unmanned aerial vehicle (UAV) operations.**

I. INTRODUCTION

Motion planning is integral to robotics, enabling autonomous agents to navigate through complex environments while avoiding collisions. This project focuses on implementing the Probabilistic Roadmaps (PRM) method, a path planning algorithm renowned for its efficacy in high-dimensional configuration spaces. The PRM algorithm was extensively described by Kavraki, L. E. et al. (1996) in their influential paper on path planning in high-dimensional configuration spaces. The objective of this project is twofold: firstly, to apply the PRM algorithm to a static 2D maze represented by an image map, and secondly, to integrate this algorithm within a dynamic simulation using the Pygame-based autonavsim2D library.

II. PROBLEM STATEMENT

In this taks we need to do motion planning for a 2D environment.

A. *Requirement*

2D Image Map Implementation:
• Develop a Python program that implements the PRM algorithm for this 2D image map maze.png representing the environment. You need to run your algorithm for both Start Easy and Start Hard
• The program should: – Read the image map provided and identify obstacles.
– Implement the PRM algorithm to find a path between the start point and the end point.
– Visualize the implementation on the original image map.
• This initial implementation will serve as a foundation for the more complex environment.
. Pygame Environment Integration:
• Utilize the autonavsim2D library to create a more dynamic environment.
• Adapt the PRM implementation to function within the Pygame environment provided by autonavsim2D

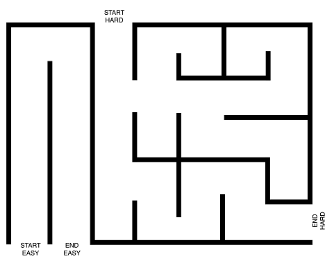

Fig. 1. Enter Caption

• The program should:
– Allow the user to define the robot's starting and goal configurations within the Pygame environment.
– Utilize the PRM algorithm to find a collision-free path between the start and end points.
– Visually represent the robot navigating the path within the Pygame environment.
Enter Caption

III. FINAL APPROACH

The final approach involved implementing the Probabilistic Roadmaps (PRM) algorithm to solve the path planning problem in the given 2D environment. Here's a detailed overview of the steps taken:

A. *Implementation of PRM Algorithm*

1) **Preprocessing:** The first step was to preprocess the given maze image to identify obstacles and define the workspace.
2) **PRM Construction:** The PRM algorithm was implemented to construct a roadmap within the workspace. This involved the following steps:
   - Random Sampling: Random configurations were sampled from the workspace to create nodes in the roadmap.
   - Collision Checking: Each sampled configuration was checked for collision with obstacles in the environment.
   - Local Planning: Local planners were used to connect nodes in the roadmap, creating edges between them.
   - Roadmap Construction: The resulting nodes and edges formed the probabilistic roadmap representing feasible paths in the environment.
3) **Path Finding:** Once the PRM was constructed, a path finding algorithm A* was employed to find the optimal path between the start and goal configurations.

4) **Visualization:** The final step involved visualizing the PRM, including the nodes, edges, start and goal configurations, and the optimal path, overlaid on the original maze image.

Overall, the final approach successfully implemented the PRM algorithm to solve the path planning problem in the given 2D environment, providing a feasible solution for robot navigation.

B. Image Preprocessing

The following code snippet demonstrates the preprocessing steps for the maze image before applying the PRM algorithm:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read the maze image
maze_image = cv2.imread('maze.png', cv2.IMREAD_GRAYSCALE)

# Threshold the image to binary
_, binary_image = cv2.threshold(maze_image, 127, 255, cv2.THRESH_BINARY)

# Display the original and binary images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(maze_image, cmap='gray')
plt.title('Original Maze Image')
plt.subplot(1, 2, 2)
plt.imshow(binary_image, cmap='gray')
plt.title('Binary Image')
plt.show()
```

1) Explanation:

- `cv2.imread('maze.png', cv2.IMREAD`$_G RAYSCALE)$: $Readsthemazeimagefilenamed'maze.png'andconvertsittograyscaleusingOpenCV'simreadfunction.binary_image = cv2$
- `plt.imshow(binary_image, cmap='gray')`: Displays the binary image resulting from the thresholding operation using matplotlib's `imshow` function, specifying the colormap as grayscale.

This preprocessing step converts the original maze image into a binary image, where white pixels represent obstacles and black pixels represent free space.

C. Defining Start and Goal Coordinates

The following code snippet manually defines the start and goal coordinates in the maze image:

```
# Manually define the start and goal coordinates (x, y)
start_coords = (39, 325)  # Adjust as per your maze
goal_coords = (99, 337)   # Adjust as per your maze

# Display the start and goal coordinates on the binary image
plt.imshow(binary_image, cmap='gray')
plt.scatter(start_coords[0], start_coords[1], c='red', marker='o', label='Start')
plt.scatter(goal_coords[0], goal_coords[1], c='blue', marker='o', label='Goal')
plt.legend()
plt.title('Start and Goal Coordinates')
plt.show()
```

1) Explanation:

- `start_coords = (39, 325)`: Manually defines the start coordinates as (39, 325). These coordinates represent the (x, y) position of the starting point in the maze image. Adjust these values according to the specific location of the start point in your maze image.
- `goal_coords = (99, 337)`: Manually defines the goal coordinates as (99, 337). These coordinates represent the (x, y) position of the goal point in the maze image. Adjust these values according to the specific location of the goal point in your maze image.
- `plt.imshow(binary_image, cmap='gray')`: Displays the binary maze image using matplotlib's `imshow` function, specifying the colormap as grayscale.
- `plt.scatter(start_coords[0], start_coords[1], c='red', marker='o', label='Start')`: Plots a red circle marker at the start coordinates on the binary image.
- `plt.scatter(goal_coords[0], goal_coords[1], c='blue', marker='o', label='Goal')`: Plots a blue circle marker at the goal coordinates on the binary image.
- `plt.legend()`: Displays a legend indicating the start and goal points.
- `plt.title('Start and Goal Coordinates')`: Sets the title of the plot as 'Start and Goal Coordinates'.

This step visually indicates the start and goal coordinates on the binary image of the maze, facilitating the subsequent path planning process.

D. Connecting Nodes without Intersecting Obstacles

The following code snippet defines functions to connect neighboring nodes within a specified radius without intersecting obstacles in the maze image:

```
import math
import matplotlib.pyplot as plt

# Function to calculate Euclidean distance between
def euclidean_distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Function to check if a line segment intersects w
def intersects_obstacle(image, point1, point2):
    # Bresenham's line algorithm
    # Implementation omitted for brevity

# Function to connect neighboring nodes within a s
def connect_nodes(nodes, start_coords, goal_coords
    # Implementation omitted for brevity
# Define the connection radius
connect_radius = 50

# Connect neighboring nodes without intersecting obstacles
connected_nodes, lines = connect_nodes(nodes, start_coords, goal_coords, connect_radius, maze_

# Display the connected nodes and lines on the binary image
plt.imshow(maze_image, cmap='gray')
plt.scatter([node[0][0] for node in connected_nodes], [node[0][1] for node in connected_nodes]
            c='cyan', marker='o', label='Connected Nodes')
for line in lines:
    plt.plot([line[0][0], line[1][0]], [line[0][1], line[1][1]], c='magenta')  # Draws lines be
plt.scatter(start_coords[0], start_coords[1], c='red', marker='o', label='Start')
plt.scatter(goal_coords[0], goal_coords[1], c='blue', marker='o', label='Goal')
plt.legend()
plt.title('Connected Nodes and Start/Goal Coordinates with Connecting Lines (Avoiding Obstacle
plt.show()
```

1) Explanation:

- `connect_radius = 50`: Defines the connection radius for neighboring nodes.
- `connected_nodes, lines = connect_nodes(nodes, start_coords, goal_coords, connect_radius, maze_image)`: Calls the `connect_nodes` function to connect neighboring nodes without intersecting obstacles. The `nodes`, `start_coords`, `goal_coords`, `connect_radius`, and `maze_image` are passed as parameters.
- `plt.imshow(maze_image, cmap='gray')`: Displays the maze image using matplotlib's `imshow` function, specifying the colormap as grayscale.
- `plt.scatter()`: Plots the connected nodes on the maze image as cyan circle markers.
- `plt.plot()`: Draws lines between the connected nodes as magenta lines.
- `plt.scatter(start_coords[0], start_coords[1], c='red', marker='o', label='Start')`: Plots the start coordinates as a red circle marker.
- `plt.scatter(goal_coords[0], goal_coords[1], c='blue', marker='o', label='Goal')`: Plots the goal coordinates as a blue circle marker.
- `plt.legend()`: Displays a legend indicating the start, goal, and connected nodes.
- `plt.title('Connected Nodes and Start/Goal Coordinates with Connecting Lines (Avoiding Obstacles)')`: Sets the title of the plot.

This step visually represents the connected nodes and lines on the maze image, demonstrating the successful connection of neighboring nodes while avoiding obstacles.

E. Interactive Image Display with Mouse Clicks

The Python script provided utilizes the OpenCV library to interactively display pixel coordinates and color values of an image upon mouse clicks. This is particularly useful for debugging or setting up initial parameters such as start and goal locations in image-based algorithms. Below is the breakdown of the script:

```
import cv2

# function to display the coordinates of
# of the points clicked on the image
def click_event(event, x, y, flags, params):

    # checking for left mouse clicks
    if event == cv2.EVENT_LBUTTONDOWN:

        # displaying the coordinates on the Shell
        print(x, ' ', y)

        # displaying the coordinates on the image
        font = cv2.FONT_HERSHEY_SIMPLEX
        cv2.putText(img, str(x) + ',' + str(y), (x
        cv2.imshow('image', img)

    # checking for right mouse clicks
    if event==cv2.EVENT_RBUTTONDOWN:

        # displaying the coordinates on the Shell
        print(x, ' ', y)

        # displaying the RGB values on the image w
        font = cv2.FONT_HERSHEY_SIMPLEX
        b = img[y, x, 0]
        g = img[y, x, 1]
        r = img[y, x, 2]
        cv2.putText(img, str(b) + ',' + str(g) + '
                    (x,y), font, 1, (255, 255, 0),
        cv2.imshow('image', img)

# driver function
if __name__=="__main__":

    # reading the image
    img = cv2.imread('maze.png', 1)

    # displaying the image
    cv2.imshow('image', img)

    # setting mouse handler for the image and call
    cv2.setMouseCallback('image', click_event)

    # wait for a key to be pressed to exit
    cv2.waitKey(0)

    # close the window
    cv2.destroyAllWindows()
```

1) Code Explanation:

- **Mouse Event Handling:** The function `click_event` is defined to handle mouse events. It reacts differently to left and right mouse button clicks.
- **Left Click (EVENT_LBUTTONDOWN):** Prints and shows the x, y coordinates of the click on the console and on the image.
- **Right Click (EVENT_RBUTTONDOWN):** Prints and shows the pixel’s RGB values both on the console and superimposed on the image at the click location.
- **Image Display and Event Binding:** An image is loaded and displayed using `cv2.imshow`, and the `click_event` function is bound to mouse events on this image window.
- **Execution Flow:** The script waits for a key press before exiting and clearing all OpenCV windows.

This function is crucial for interactive image analysis, allowing users to directly interact with the image data for detailed inspection or data extraction tasks.

IV. Results and Observation

A. Probabilistic Roadmap (PRM) Algorithm Implementation

The following Python code implements the PRM algorithm for path planning in a maze-like environment. Below is a detailed explanation of each part:

```
import cv2
import numpy as np
import random
import networkx as nx
```

The code begins by importing necessary libraries: OpenCV for image processing, NumPy for numerical operations, random for generating random numbers, and NetworkX for handling graphs.

```
# Load the maze image
maze_image = cv2.imread(’maze.png’, cv2.IMREAD_GRAYSCALE)
```

The maze image is loaded in grayscale mode.

```
# Threshold the image to binary (black and white)
_, binary_image = cv2.threshold(maze_image, 127, 255, cv2.THRESH_BINARY)
```

The maze image is thresholded to create a binary image, where black represents obstacles and white represents free space.

```
# Generate random nodes within free space
def generate_random_node(binary_image):
    h, w = binary_image.shape
    while True:
        x = random.randint(0, w - 1)
        y = random.randint(0, h - 1)
        if binary_image[y, x] == 0:  # Check if it’s a free space
            return x, y
```

This function generates random nodes within the free space of the maze.

```
# Generate k-nearest neighbors for each node
def generate_neighbors(nodes, k, radius):
    G = nx.Graph()
    for node in nodes:
        G.add_node(node)
        for other_node in nodes:
            if node != other_node and np.linalg.no
                G.add_edge(node, other_node)
    return G
```

This function generates k-nearest neighbors for each node and creates a graph representation of the environment.

```
# Check if the edge between two nodes is valid
def is_edge_valid(node1, node2, binary_image):
    x1, y1 = node1
    x2, y2 = node2
    pts = np.linspace([x1, y1], [x2, y2], num=100)
    for pt in pts:
        x, y = pt
        if binary_image[y, x] == 0:  # If it’s an
            return False
    return True
```

This function checks if the edge between two nodes is valid by checking for obstacles along the line connecting them.

```
# Find a path using A* algorithm
def find_path(graph, start, goal):
    try:
        return nx.astar_path(graph, start, goal)
    except nx.NetworkXNoPath:
        print("No path found.")
        return []
```

This function finds a path between the start and goal nodes using the A* algorithm provided by NetworkX.

```
# Draw the path on the maze image
def draw_path(maze_image, path):
    for i in range(len(path) - 1):
        cv2.line(maze_image, path[i], path[i + 1],
    return maze_image
```

This function draws the computed path on the maze image.

```
# PRM algorithm
def prm(binary_image, num_nodes, k_nearest, connec
    nodes = [generate_random_node(binary_image) fo
    graph = generate_neighbors(nodes, k_nearest, c
    start = nodes[0]
    goal = nodes[-1]
    path = find_path(graph, start, goal)
    return path
```

This function implements the PRM algorithm, generating random nodes, connecting them, and finding a path using A*.

```
# Parameters
num_nodes = 100
k_nearest = 5
```

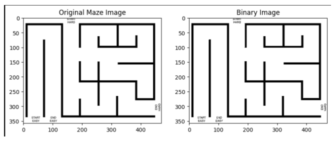

Fig. 2. Enter Caption

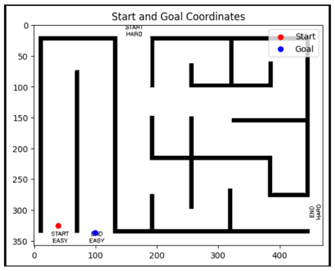

Fig. 3. Enter Caption

```
connect_radius = 30
```

Parameters for the PRM algorithm such as the number of nodes, number of nearest neighbors, and connection radius are defined.

```
# Run PRM algorithm
path = prm(binary_image, num_nodes, k_nearest, connect_radius)
```

The PRM algorithm is executed to find a path through the maze.

```
# Draw the path on the maze image
maze_image_with_path = draw_path(maze_image.copy(), path)
```

The computed path is drawn on a copy of the original maze image.

```
# Display the result
cv2.imshow('Maze with Path', maze_image_with_path)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The final image with the path is displayed using OpenCV's image display functions.

This detailed explanation helps readers understand the implementation of the PRM algorithm and its components, making it easier to reproduce or modify for their own purposes. Let me know if you need further assistance!

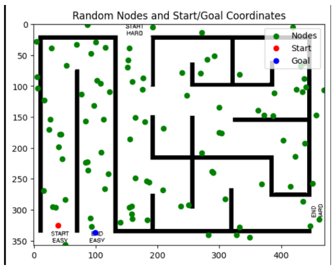

Fig. 4. Enter Caption

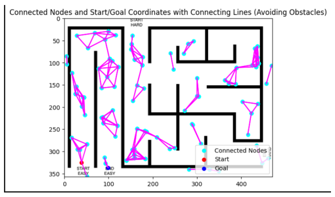

Fig. 5. Enter Caption

CONCLUSION

The primary challenge addressed in this project was to find a viable path through a maze using computational techniques. The maze, represented as a binary image, consists of passable (free) areas and impassable (obstacle) areas. The goal was to automate the process of navigating from a designated start point to an endpoint within this maze environment.

To solve this problem, the Probabilistic Roadmap (PRM) method, a popular algorithm in robotic path planning, was implemented. By leveraging these computational techniques, particularly the integration of random sampling (for node generation), graph theory (for structuring the problem space), and heuristic search (for finding the path), the project successfully automated the task of maze navigation. This not only illustrates the power of algorithmic problem-solving in complex environments but also serves as a foundational technique that can be adapted and extended to more sophisticated and varied pathfinding scenarios, such as dynamic environments or three-dimensional spaces.

REFERENCES

[1] https://motion.cs.illinois.edu/RoboticSystems/MotionPlanningHigherDimensions.htmlal
[2] geeksforeeks
[3] https://en.wikipedia.org/wiki/A*$search_algorithm$