Students:
Syma Afsha [symaafsha.eece@gmail.com]
Lisa Paul Magoti [paullisa051@gmail.com]

Lab4: RRT

Introduction

In this lab, we programmed the Rapidly exploring random tree (RRT) sampling-based path planning algorithm to solve a 2D path planning problem. The working of this algorithm is such that a random configuration point is selected from the obstacle free space of the environment and it is connected to the nearest configuration in the tree, provided there are no obstacles between them. If there is at least one obstacle, the nearest point is connected to a configuration point in the direction of the randomly generated point before any obstacle is encountered. At the beginning, the tree only has the configuration of the starting point. The tree is expected to grow following this algorithm until it reaches the goal point. We also implemennted the RRT* algorithm which is similar to the RRT, but with two additional optimizations, cost and rewire. The implementation of both algorithms into a code is explained below:

Implementation

In the implementation of this algorithm, we wrote different classes and functions which are explained in detail in this section:

class Point

We created this class to make the handling of 2D point operations easier. The constructor of the class defines the x and y variables of the point together with its parent node. Two other methods are written in this class which are **dist** method that returns the distance between the point and another point, and **__str__** method which returns the point itself. The implementation of this class is shown below:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.parent = None

    def dist(self, p):
        return math.sqrt((self.x - p.x)**2 + (self.y - p.y)**2)

    def __str__(self):
        return "({}, {})".format(self.x, self.y)
```

Figure 1: class Point implementation

create_random_point

This function takes as input a probability value p, maximum x and y values, and the goal point. It returns a random 2D point within the range of the maximum x and y values with the probability of that point not being the goal point being p. The implementation of this function is shown below:

```python
# Function to create a random point in the space
def create_random_point(probability, xmax, ymax, goal_point):
    if random.random() > probability:
        random_point = Point(random.randint(0, xmax), random.randint(0, ymax))
    else:
        random_point = goal_point
    return random_point
```

Figure 2: create_random_point implementation

find_nearest_node

This function takes as input the random point generated above, and the part of the tree that has already been generated and returns the point in the tree that is nearest to the randomly generated point. This nearness is calculated by considering the Euclidean distance. The implementation of this code is shown below:

```python
# Function to find the nearest node in the graph to a given point
def find_nearest_node(random_point, graph):
    distances = [vertex.dist(random_point) for vertex in graph]
    min_index = distances.index(min(distances))
    return graph[min_index]
```

Figure 3: find_nearest_node implementation

move_towards_point

Given a maximum distance, a start point and an end point, This function returns a point that is at most at a maximum distance from the start point in the direction of the end point. This function is used to stepwise move from the nearest point in the tree to the new random point. The implementation of this function is shown below:

```python
# Function to move towards a point by a fixed distance
def move_towards_point(step_size, start_point, end_point):
    if start_point.x == end_point.x and start_point.y == end_point.y:
        return start_point
    direction = Point(end_point.x - start_point.x, end_point.y - start_point.y)
    direction_length = math.sqrt(direction.x**2 + direction.y**2)
    normalized = Point(direction.x/direction_length, direction.y/direction_length)
    return Point(math.floor(start_point.x + step_size * normalized.x), math.floor(start_point.y + step_size * normalized.y))
```

Figure 4: move_towards_point implementation

determine_new_node

This function determines the position of the new node considering the maximum step size allowed. It returns the the random point if it is within the maximum distance. If it is not, it calls move_towards_point function to move towards the direction of the random point but only up to the step size allowed. The implementation of this code is shown below:

```python
# Function to determine a new node's position considering the step size
def determine_new_node(closest_node, random_point, step_size):
    distance = random_point.dist(closest_node)
    if distance < step_size:
        return random_point
    return move_towards_point(step_size, closest_node, random_point)
```

Figure 5: determine_new_node implementation

check_segment_free

This function checks whether the connection between two points is obstacle free or not. It returns True if it is and False if it is not. The implementation of this function is shown below:

```python
# Function to check if a line segment is free from obstacles
def check_segment_free(start_node, end_node, grid_map):
    num_points = math.floor(start_node.dist(end_node))
    for i in range(1, num_points + 2):
        point = move_towards_point(i, start_node, end_node)
        if grid_map[point.y, point.x] == 1:
            return False
    return True
```

Figure 6: check_segment_free implementation

construct_path

Given the tree and a node point, this function returns a path from the starting point to that node point. The implementation of this function is shown below:

```python
# Function to construct the path from start to goal
def construct_path(graph, current_node):
    path = [current_node]
    while current_node in graph:
        current_node = current_node.parent
        if current_node is not None:
            path.insert(0, current_node)
    return path
```

Figure 7: construct_path implementation

path_distance

Given a path, this function returns total cost of the path. The implementation of this function is shown below:

```python
def path_distance(path):
    total_distance = 0
    for vertex in range(len(path)-1):
        dist = math.sqrt((path[vertex].x - path[vertex+1].x)**2 + (path[vertex].y - path[vertex+1].y)**2)
        total_distance = total_distance + dist

    return total_distance
```

Figure 8: path_distance implementation

rrt

This function performs RRT algorithm by applying all the above functions and classes, either directly or indirectly through other functions. It initializes the tree with the starting point, then iteratively for K steps or until the goal point is added to the tree, creates a random point, then finds the nearest point from the graph to the random point. Afterwards, it finds the new point in the direction of the random point from the nearest point and checks if their connection is obstacle free and if it is, it joins them and plots their connection. This function returns the path and number of iterations it took to reach the goal. The implementation of this function is shown below:

```
#RRT Algorithm
def rrt(grid_map, K, delta_q, p, qstart, qgoal):
    G = [qstart]
    plt.figure(figsize=(10,10))
    for k in range(K):
        qrand = create_random_point(p, len(grid_map[0])-2, len(grid_map)-2, qgoal)
        qnear = find_nearest_node(qrand, G)
        qnew = determine_new_node(qnear, qrand, delta_q)

        #Plotting the random paths
        if check_segment_free(qnear, qnew, grid_map):
            plt.subplot(1,2,1)
            plt.scatter(qnew.x,qnew.y, c='black', marker="+")
            plt.plot([qnear.x,qnew.x],[qnear.y,qnew.y],'g--')
            G.append(qnew)
            qnew.parent = qnear
            if qnew.x == qgoal.x and qnew.y == qgoal.y:
                path = construct_path(G, qnew)
                return path, k
    return None
```

Figure 9: rrt implementation

rrt_smoothing

Given the path obtained in rrt, this function smoothes this path to obtain a more optimal path. The function aims to find a valid path from the initial starting point (qstart) to the goal point (qgoal). If the direct connection encounters an obstacle, the algorithm iteratively attempts to create a valid path by trying to connect from progressively closer positions in the path to the goal. Once a valid partial path is established, the algorithm resets the starting point to this partial path's endpoint and the goal point to the original starting point, repeating the process until a complete, obstacle-free path from the start to the goal is found. The implementation of this function is shown below:

```
#Smoothing the resulting path
def rrt_smoothing(grid_map, path):

    start_counter = 0
    goal_counter = len(path) - 1

    start = path[start_counter]
    goal = path[goal_counter]
    smooth_path = [goal]
    while goal.x != start.x or goal.y != start.y:
        if check_segment_free(start, goal, grid_map):
            smooth_path.insert(0, start)
            goal = path[start_counter]
            start_counter = 0
            start = path[start_counter]
        else:
            start_counter = start_counter + 1
            start = path[start_counter]
    return smooth_path

def path_distance(path):
    total_distance = 0
    for vertex in range(len(path)-1):
        dist = math.sqrt((path[vertex].x - path[vertex+1].x)**2 + (path[vertex].y - path[vertex+1].y)**2)
        total_distance = total_distance + dist

    return total_distance
```

Figure 10: rrt_smoothing implementation

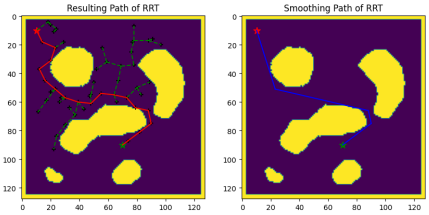
main

The main function serves as the entry point for the RRT-based path planning algorithm. It takes command-line arguments specifying the image file path, the number of iterations (K), the step size (delta_q), the probability of selecting the goal as a random point (p), and the coordinates of the start (qstart) and goal (qgoal) points. It initializes these parameters and the grid map using the provided image. Then, it invokes the rrt function to generate an initial path, followed by the rrt_smoothing function to refine the path. The distances of the original and smoothed paths are calculated, and the resulting paths are printed. Finally, the paths are visually displayed using Matplotlib. If a solution is not found, a corresponding message is printed. The code structure ensures that the main function is executed when the script is run as the main program. The implementation of this function is shown below:

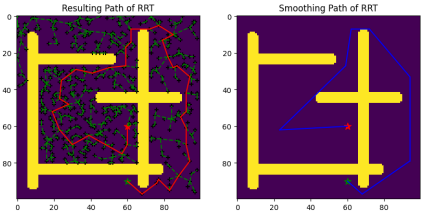
Figure 11: main function implementation

Result

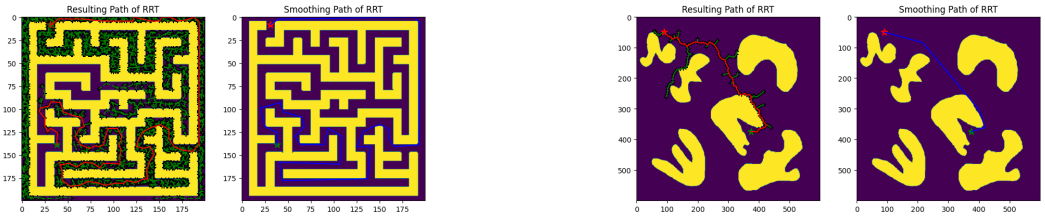
After developing the method, we tested our code on the specified maps for the lab sessions. The path planning outcomes of RRT are depicted in the images Figure 12a, 12b, 13a, and **??**.



(a) map0



(b) map1



(a) map2 (b) map3

Figure 13: RRT star graphs

RRT*

The implementation of this algorithm followed a similar approach as RRT but with few additional functions explained below:

Rewire

This function takes as inputs the new node, nodes near to it and the grid map and for all the nodes near to new node, it checks if it is possible to reach any of these nodes from it with less cost than with the previous path. Its implementation is shown below:

```
# Function to rewire the tree
def rewire(new_node, near_nodes, grid_map):
    for node in near_nodes:
        new_cost = new_node.cost + new_node.dist(node)
        if new_cost < node.cost and check_segment_free(new_node, node, grid_map):
            node.parent = new_node
            node.cost = new_cost
```

Figure 14: rewire implementation

find_new_near_nodes

This function returns nodes that a near the new node within a specific distance. Its implementation is shown below:

```
# Function to find the nearest nodes within a maximum distance
def find_near_nodes(new_node, nodes, max_dist):
    near_nodes = []
    for node in nodes:
        if node.dist(new_node) <= max_dist:
            near_nodes.append(node)
    return near_nodes
```

Figure 15: find_near_nodes implementation

calculate_cost

This function returns the cost to a node from the start node. Its implementation is shown below:

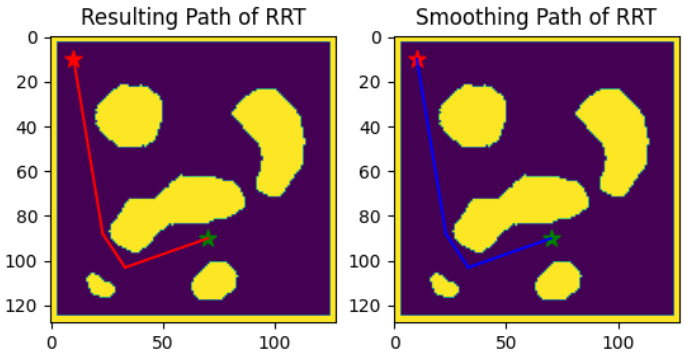


Figure 17: RRT* graph for map0

```
Path found in  78  iterations
Path Distance:  136.32101237345196
Resulting Path [(10, 10), (23, 88), (33, 103), (70, 90)]
Smooth Path Distance:  136.32101237345196
Smoothing Path [(10, 10), (23, 88), (33, 103), (70, 90)]
```

Figure 18: RRT* path distance for map0

```
# Function to calculate the cost of a node from the start node
def calculate_cost(node):
    cost = 0
    while node.parent is not None:
        cost += node.dist(node.parent)
        node = node.parent
    return cost
```

Figure 16: calculate_cost implementation

0.1 RRT* Result

After applying the RRT* algorithm, we have found the following result with the optimum path , shown in Figure 17 and 18.

Problems

We ran encountered only one problem when building the move-towards-point function. We explored numerous strategies as we attempted to approach the random node sequentially. However, a quick internet search led us to a method that included moving pixels by pixels.

Conclusion

The lab was successfully complicated and a deeper understanding on the sampling based path planning algorithm was gained.