Students:
Atharva Patwe [patweatharva@gmail.com]
Precious I. Philip-Ifabiyi [philipifabiyiprecious@gmail.com]

Lab4: RRT and RRT* Path Planner

1 Introduction

The aim of this report is to investigate the problem of path planning with the use of sampling-based algorithms. The two algorithms implemented in this report are the Rapid-Exploring Random Trees (RRT) and the RRT Star (RRT*). These algorithms work by randomly building trees and are well-suited for high-dimensional configuration spaces and complex robotic systems with many degrees of freedom. The report is organized in the following way: The algorithms and logic developed are explained in the Methodology section (Section 2). The subsequent section on results (Section 3) demonstrates all the results obtained while implementing RRT and RRT* algorithms in different environments. Finally, the optimality and the completeness of the algorithms are discussed in the Discussion and Conclusions section (Section 4).

2 Methodology

In this section of the report, the programming approach, tools, and methods employed to implement the RRT and RRT* algorithms on a grid map will be discussed.

Important! To properly run this code, you will have to import heapq library using import command. If the package is not present in the machine, the code can be modified to run using heapq_max package. This can be installed by running the following command in the terminal.

```
pip install heapq_max
```

Listing 1: Install Heap

Verify Installation! You can verify that the library is installed correctly by creating a Python script and importing it. For example:

```
import heapq
import heapq_max
```

Listing 2: Verify Installation

2.1 Rapidly-Exploring Random Trees (RRT)

In order to implement the RRT algorithm, two classes were defined, ***tree_node*** and ***RRT***. The ***tree_node*** class is used to define the properties of a tree node such as its position in the grid map

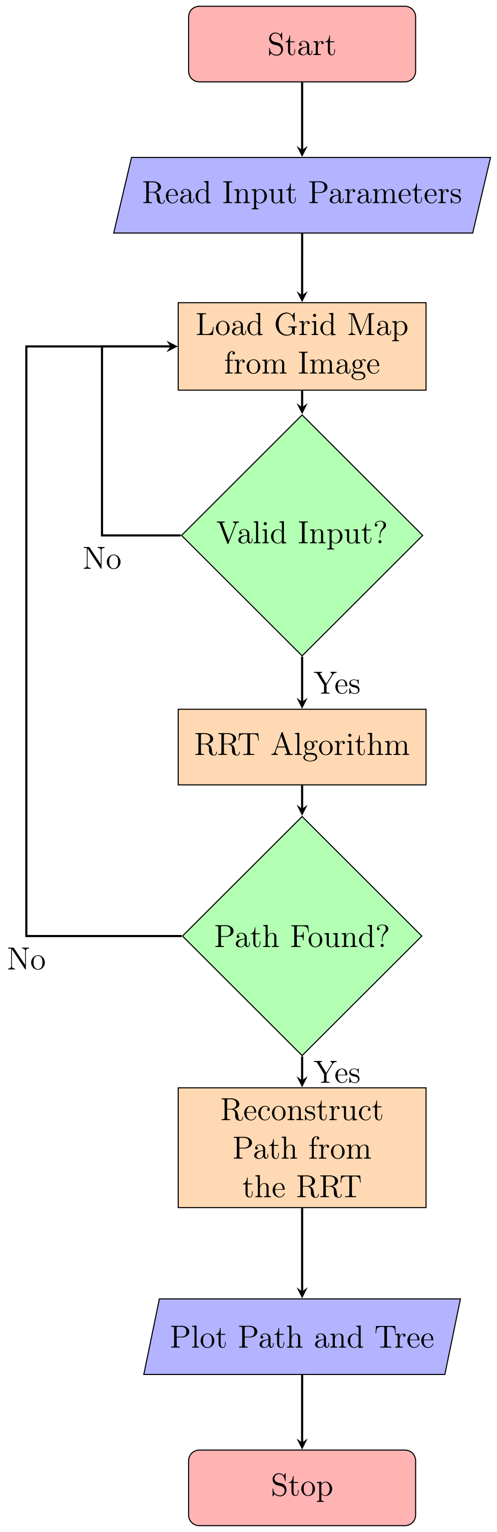


Figure 1: RRT Global Path Planner

and parent node. The ***RRT*** class contains methods for the implementation of the RRT algorithm, path generation, and path plotting. The RRT algorithm iteratively chooses random positions in space with $(1 - p)$ probability and the goal position with $\boldsymbol{p}$ probability. The position chosen is called ***q_ rand***. The nearest tree node to **q_rand** is chosen, and the algorithm tries to extend to ***q_ rand*** from the nearest node by a distance Δq. If there is no obstacle within its extension line, the extended position is called ***q_ new***, and it is added to the edges and nodes of the tree. The following subsections discuss the implementation of each method in the ***RRT*** class.

Algorithm 1: Class tree_node

```
Input  : parent, pos: tuple, cost = None
Output: None
// Constructor
1 def __init__(self, parent, pos: tuple, cost=None) → None:
2     self.pos ← pos;
3     self.parent ← parent;
4     self.cost ← cost;
```

2.1.1 Initialization

An object of the ***RRT*** class is instantiated by passing the path of the grid map, the start, and goal points in the grid map. Properties of this class are lists to store tree_nodes, edges, and the path from the start to the goal position.

2.1.2 Gridmap Method

This method is used to create a *numpy* array representation of the grid map. In this array, 0 represents free space and 1 represents obstacles in the grid map.

2.1.3 GenerateRRT Method

This method implements the RRT algorithm for path planning. The parameters of this method are the following:

- `iteration (int)`: This specifies the maximum number of iterations allowed to find the path from start to goal.
- `prob (float)`: This parameter specifies the probability of the random points being the goal position.
- `del_q (float)`: This specifies the maximum length of the edges in the tree.

The RRT algorithm is shown in Algorithm 2. As mentioned earlier, the algorithm chooses a random node in the space using method ***q_ random*** and extends a step ***del_ q*** in that direction using the method ***extend_ tree***. Method ***is_ segment_ free*** returns whether the segment is free in the space. If found free, the new node and edge are added to the tree. The tree develops iteratively, and the iteration stops when the goal is found or the number of iterations is equal to `iteration`.

Algorithm 2: generate_RRT(iteration, prob, del_q)

```
Data: self, iteration, prob, del_q, goal_thresh
Result: None
1  Initialize self.goal_reached to False;
2  Add start node to the Tree Structure;
3  for iter ← 1 to iteration do
4      Generate a random node q_rand using self.q_random method;
5      Find the nearest neighbor q_near using self.q_nearest(q_rand) method;
6      Extend the tree using self.extend_tree() to q_new;
7      if self.is_segment_free(q_near.pos, q_new.pos) then
8          self.add_node(q_new);
9          self.add_edge(q_near, q_new);
10         if q_new.pos == self.goal then
11             self.goal_reached ← True;
12             print("Path found in", iter, "iterations");
13             break;
14     else
15         continue;
```

2.1.4 QRandom Method

This method is used to randomly generate a node position in the map. The function first randomly generates a value between 0 and 1. This range is chosen to get a sense of probability. It is made sure that this value is chosen from the uniform distribution to avoid bias. This value is then compared with the probability of choosing the goal, ***p*** as a random node. If the value generated is less than ***p***, the goal is returned; otherwise, a random node is generated in the space considering the limits and bounds of the grid map.

2.1.5 QNearest Method

In this method, the nearest tree node to ***q_ rand*** is obtained. This is done by comparing the distances of all the nodes in the tree to ***q_ rand*** and choosing the tree node with the minimum distance.

2.1.6 IsSegmentFree Method

This method checks if the segment between two node positions is free. This was achieved by using the recursive subdivision strategy. The function recursively calls itself to calculate the midpoint(***q_ mid)***) of two input node positions and check if it belongs to any obstacle by looking at the grid. Since the map is discrete, the integer value of the midpoint is chosen for checking in the map. Initially, if the midpoint is an obstacle, it returns False, since the segment is not free in space. However, if the midpoint is not an obstacle, the nodes are updated to ***q1***, ***q_ mid*** and ***q_ mid***, ***q2*** for the next recursive call. This recursion is executed till the distance between the segments becomes 2 meaning the nodes are adjacent to each other. At this stage, the individual nodes are checked from the grid for the obstacles. This condition is the base condition that stops the recursive call.

Algorithm 3: is_segment_free(q1, q2)

```
Data: q1: tuple, q2: tuple
Result: bool: True if segment is obstacle-free, False otherwise
1  Function is_segment_free(q1, q2):
2      distance ← Distance Between q1 and q2;
3      mid_point←midpoint of q1 and q2;
4      if q1 or q2 are out of map limit then
5          return False;
6      if distance > 2 then
7          if mid_point is obstacle then
8              return False;
9      else
10         if q1 and q2 are free then
11             return True;
12         else
13             return False;
14     return is_segment_free(q1, mid_point) and is_segment_free(mid_point, q2);
```

2.1.7 ExtendTree method

This method is called once a random point (***q_ rand***) is returned by the ***q_ random*** method, so as to extend the tree in that direction. If the distance between ***q_ near*** and ***q_ rand*** is less than ***del_ q***, the position of the new node becomes ***q_ rand***. Otherwise, the method calculates the angle of the segment which is to be extended from ***q_ near*** using ***arctan2*** function. This angle is then used to find out the x and y positions of the new node. The step is the given as ***del_ q*** as shown in equation 2.

$$\theta = \arctan 2(q_{\text{rand}}[1] - q_{\text{near}}[1], q_{\text{rand}}[0] - q_{\text{near}}[0]) \quad (1)$$
$$\text{row} = (q_{\text{near}}[0] + \text{del\_q} \times \cos(\theta)) \quad (2)$$
$$\text{col} = (q_{\text{near}}[1] + \text{del\_q} \times \sin(\theta)) \quad (3)$$

2.1.8 AddNode and AddEdge

The ***RRT*** class contains tree nodes and edges stored as a list. The methods ***add_ node*** and ***add_ edge*** append to these lists when called with a node or an edge.

2.1.9 GeneratePath

In this method, the path from the start to the goal point is generated. If the goal node is found in the tree_nodes list, it means that there is a path from the start point to the goal location. A variable called ***current_ q*** is initialized, and the goal node is assigned to it. The parent node of ***current_ q*** is iteratively obtained. At each iteration, the parent node obtained is assigned to ***current_ q***. This is done until the start node is reached. All the positions of the current node during the iteration are stored in a list, which will be the path from start to goal.

2.1.10 DrawPath

In this method, the vertices of the graph, edges of the graph, and path generated from the start point to the goal point by the RRT algorithm are plotted. The smooth path is also plotted here. This is achieved by accessing the elements of the tree_nodes, edges, and smooth_path lists.

2.1.11 SmoothenPath

After generating the path from the start to the goal point, a smoothing algorithm was applied to obtain a shorter and less noisy path. This smoothing algorithm is implemented in the **smoothen_ path** method. The algorithm works as follows:
Let path[n] be the last element in the path list and path[0] be the first element in the path list. If the segment between the last element in the path list, path[n], which is the goal point and the first element, path[0], is free, an edge between the last and the first element is created. If it is not free, the algorithm proceeds to check if the segment between the last element in the path list, path[n], and the second element, path[1], is free. This process continues till it finds a valid edge or element. Once a valid element has been found, path[i], path[n] and path[i] are added to the smooth_path list. Then, the algorithm starts checking for free segments between path[i] and the first element of the path list, path[0]. This continues till it reaches the first element of the path.
The implementation is shown in Algorithm 4.

2.2 RRT*

RRT* is an extended version of the RRT algorithm with two modifications for optimality. The first modification is to extend the tree from a point of least resistance to the start. The resistance in this context is the cost to travel from any node to the start node. This implies the ***q_ new*** is connected to a node having the least cost. To make this modification, method ***q_ nearest_ n_ neighbors*** was developed to get the nearest neighbours from the new q. As it can be observed in the line number 9 of algorithm 5, every neighbour is checked for the connection. The next modification is done to rewire the tree as soon as a new node is added to the tree. All the nodes near to ***q_ new*** are checked if they can be reached through ***q_ new*** with less cost. If it is possible, the parent of that node is changed to ***q_ new*** and subsequent changes are made to the edges of the tree.

In this implementation, while most of the methods are similar, there are various modifications. Every node of the class ***tree_ node*** has a cost attribute, which is the cost to travel from that node to start. The method ***q_ nearest_ n_ neighbors*** returns a sorted list of neighbours according to their cost.

Algorithm 4: smoothen_path()

```
Data: None
Result: None
1  Function smoothen_path():
2      self.smooth_path ← empty list;
3      if the length of the path list is not zero then
4          Append the last position, which is the goal position to the smooth_path list;
5          current_pos ← goal position;
6          current_index ← Get the index of the current_pos in the path list;
7          while the start position is not in the smooth_path list do
8              new_list ← Get elements of the path list from 0 to current_index;
9              for i=0 to current_index-1 do
10                 if the segment from current_pos to the ith element of new_list is free then
11                     x ← ith element of new_list;
12                     Append x to smooth_path list;
13                     current_pos ← x;
14                     current_index ← Get the index of x in path list;
15                     break;
```

2.2.1 q_nearest_n_neighbors

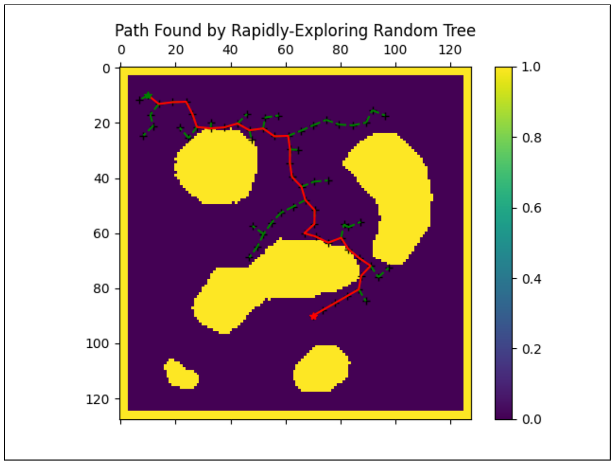
This function calculates the nearest n neighbours of a node. The function iteratively updates a priority queue of cost. Due to the implementation of a priority queue, the code runs significantly faster as compared to for loops. Neighbours with the least cost are then connected with the new node.

The rewiring is done using another ***for loop*** in the main ***generate_ RRT_ star*** method. The rewiring step considers this neighbour list and checks if that neighbour can be reached with less cost from the new node. If the neighbour can be reached with a lower cost, the parent of that node is updated to the new node and edges are added to the tree edge list.

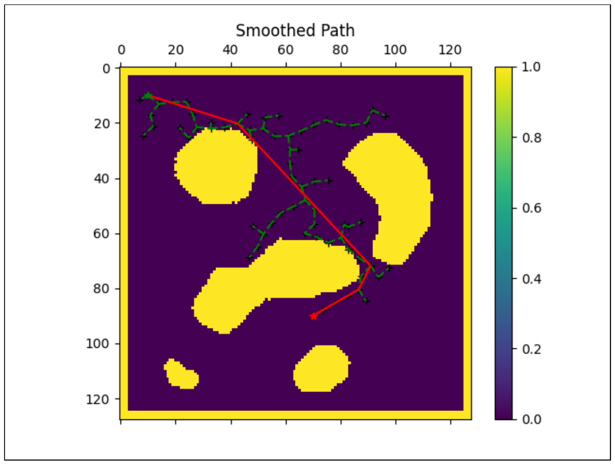
3 Results

3.1 RRT Results

The RRT algorithm was tested in all the given map environments. Figures 2 through 9 show the working of RRT in the environment. All the figures presented on the left side (Figure (a)) show the raw path obtained from the RRT. The right side figures demonstrate the working of the smoothing function. As can be seen, they are much more optimal and at the same time do not go through any obstacles.

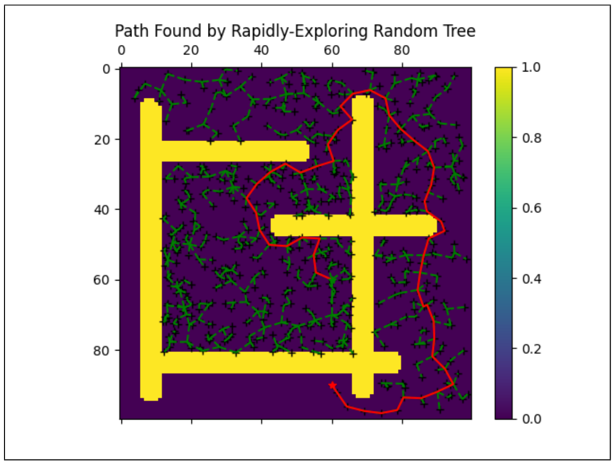


(a) Tree structure

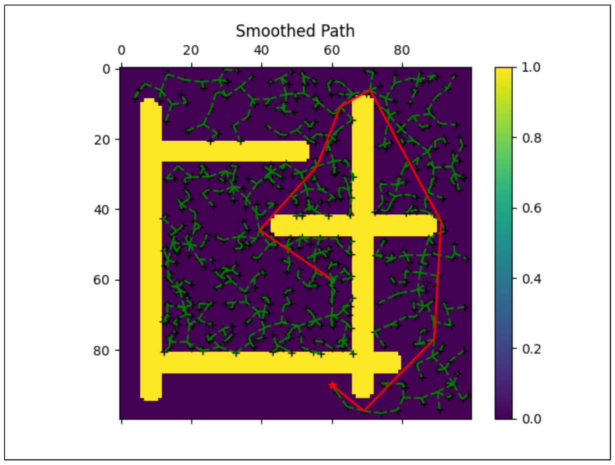


(b) Smoothed RRT Path

Figure 2: RRT Implementation on Map 0

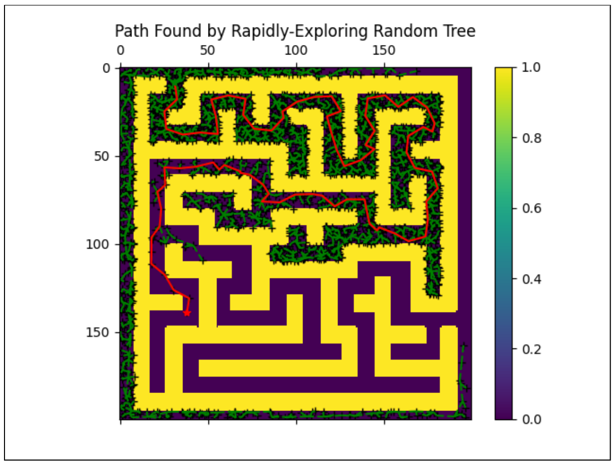


(a) Tree structure

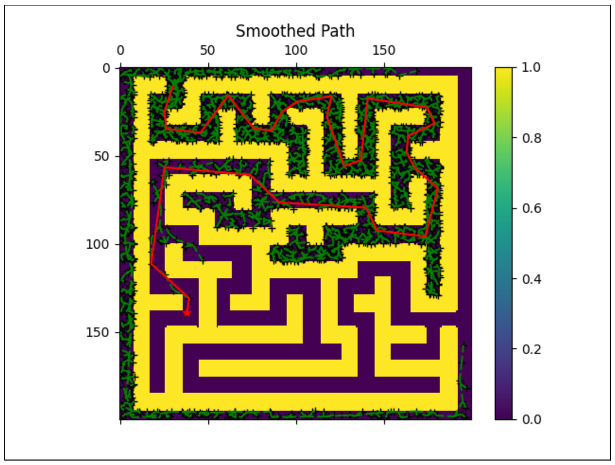


(b) Smoothed RRT Path

Figure 3: RRT Implementation on Map 1

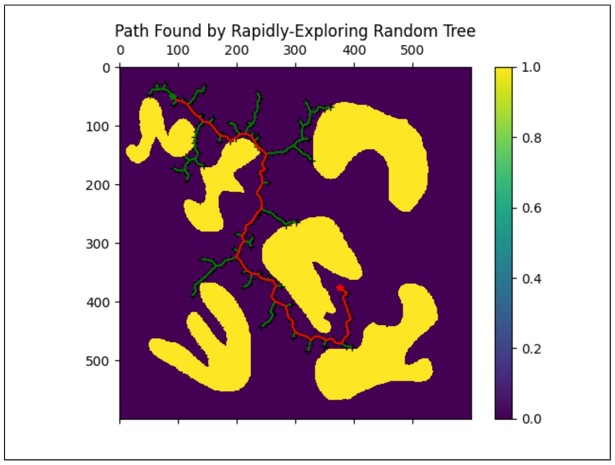


(a) Tree structure

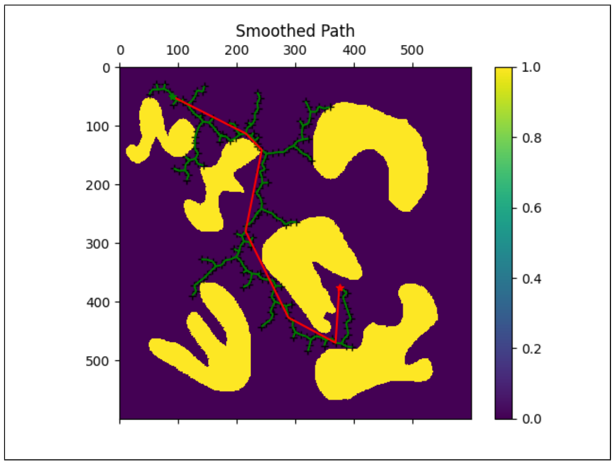


(b) Smoothed RRT Path

Figure 4: RRT Implementation on Map 2

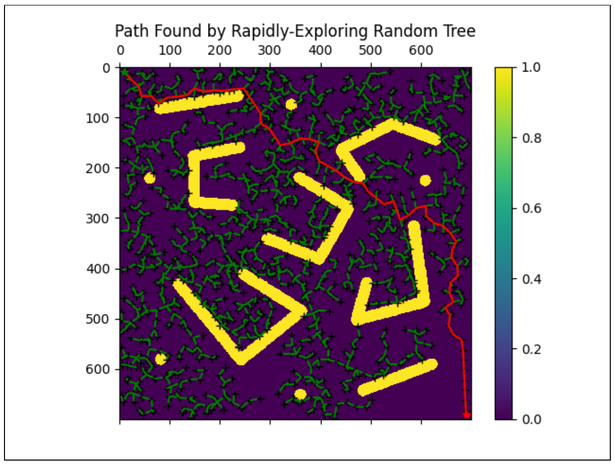


(a) Tree structure

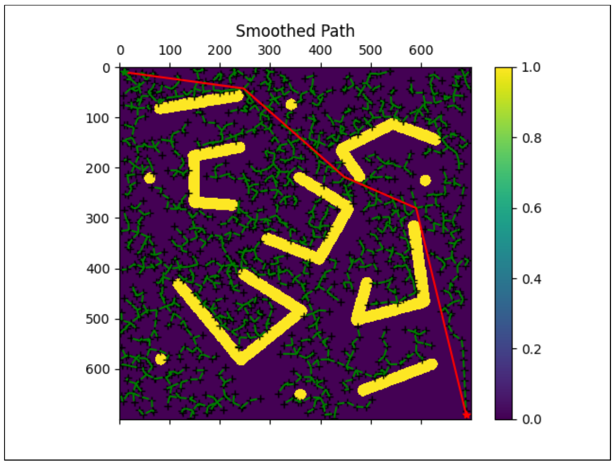


(b) Smoothed RRT Path

Figure 5: RRT Implementation on Map 3

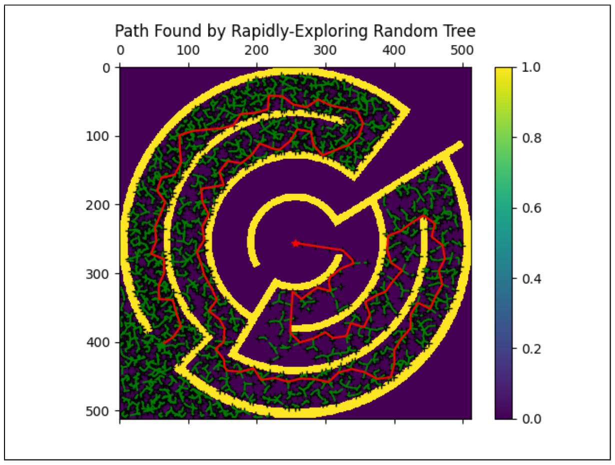


(a) Tree structure

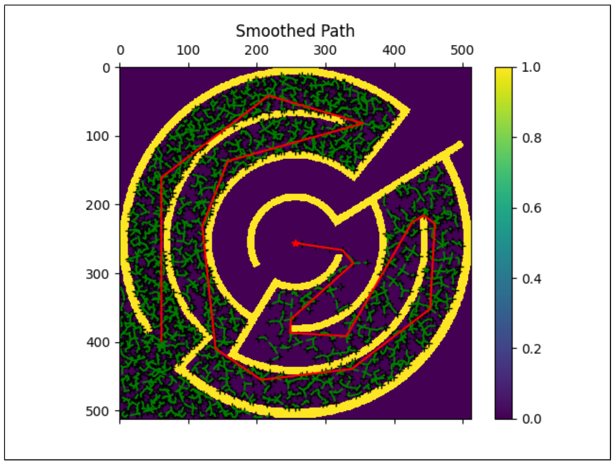


(b) Smoothed RRT Path

Figure 6: RRT Implementation on Map 4

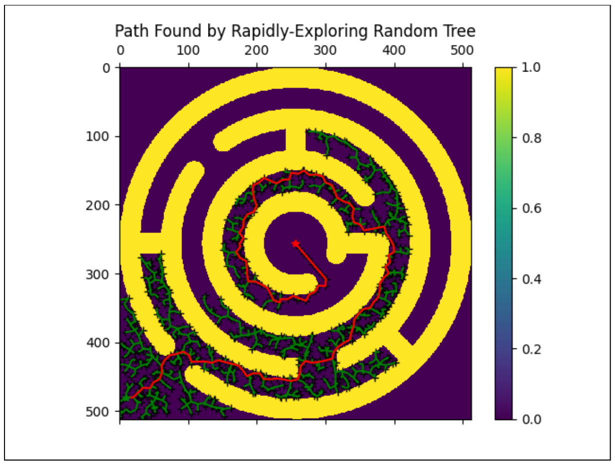


(a) Tree structure

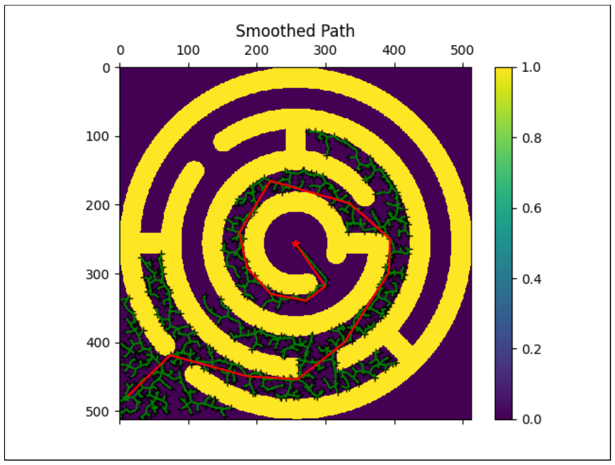


(b) Smoothed RRT Path

Figure 7: RRT Implementation on Map 5

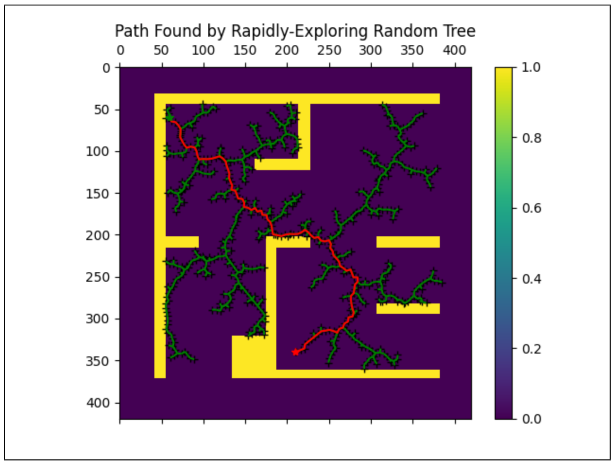


(a) Tree structure

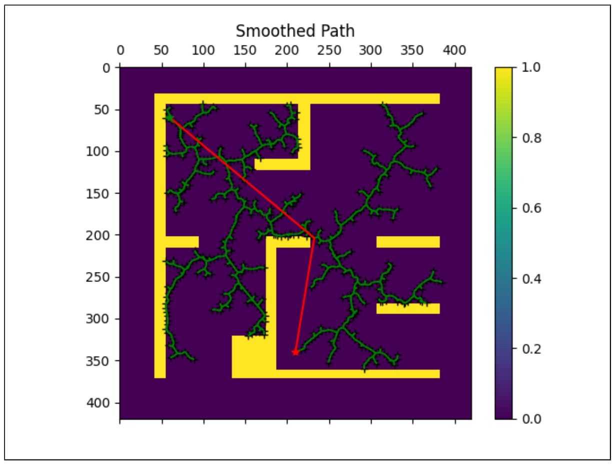


(b) Smoothed RRT Path

Figure 8: RRT Implementation on Map 6



(a) Tree structure

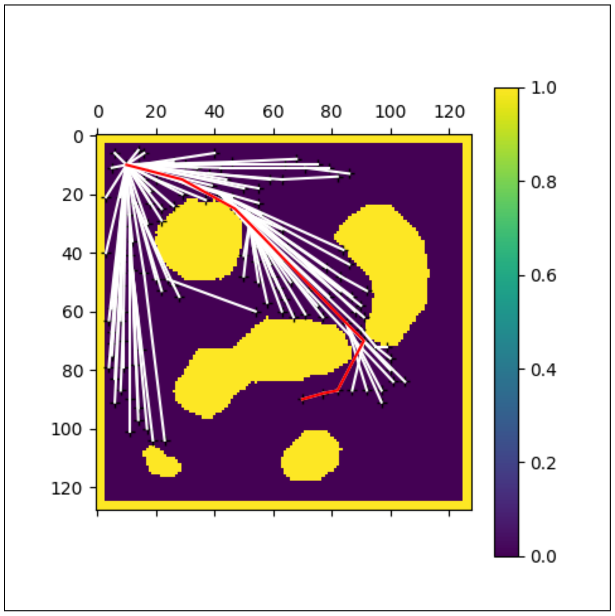


(b) Smoothed RRT Path

Figure 9: RRT Implementation on Map 7

3.2 RRT* Results

The maps given were tested with the RRT* algorithm with parameters iteration, step size, probability of sampling goal, and maximum distance. As soon as RRT* reaches the goal for the first time, the cost is calculated, and then iterations were continued. After every iteration, the tree structure becomes more optimal, reducing the cost of traveling. After the iterations were exhausted, the cost was compared, and it was found that the cost decreased with the iterations. All the results are shown from figure 10 to 14.


(a) After 198 Iterations

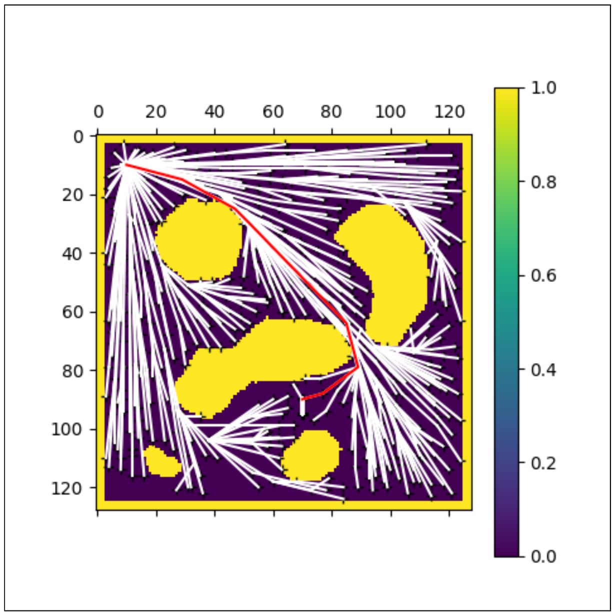
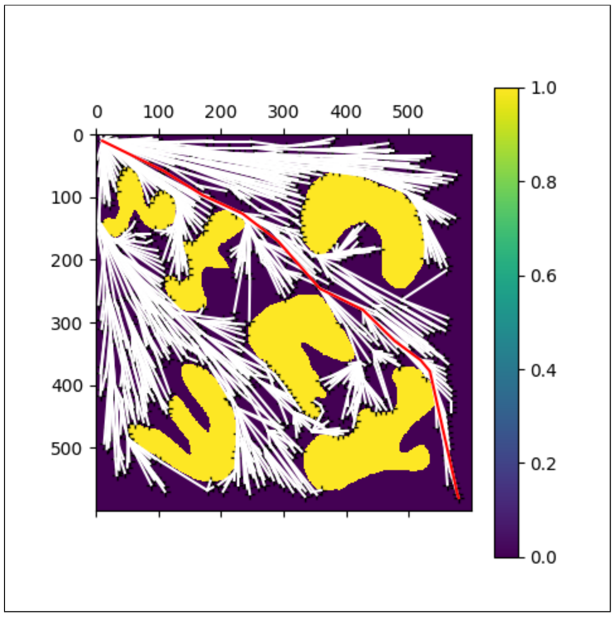

(b) After 1000 Iterations

Figure 10: RRT* Implementation on Map 0


(a) After 5863 Iterations

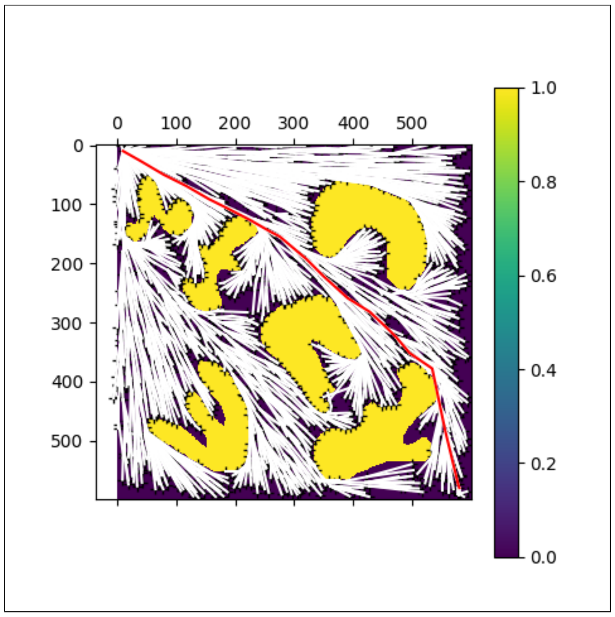
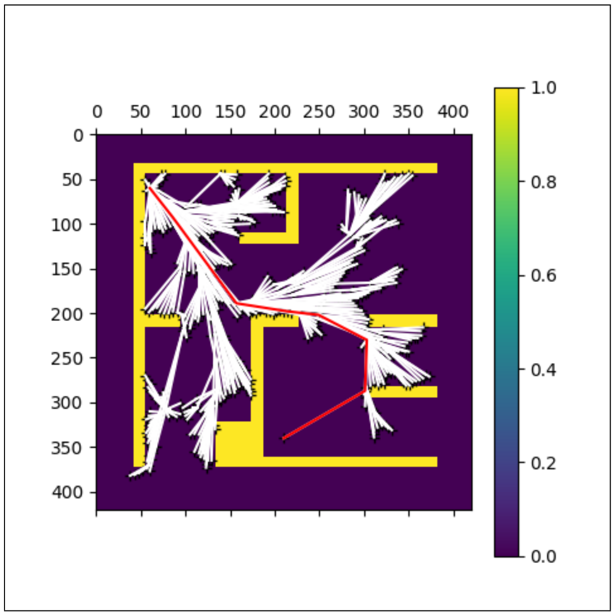

(b) After 10000 Iterations

Figure 11: RRT* Implementation on Map 3


(a) After 3578 Iterations

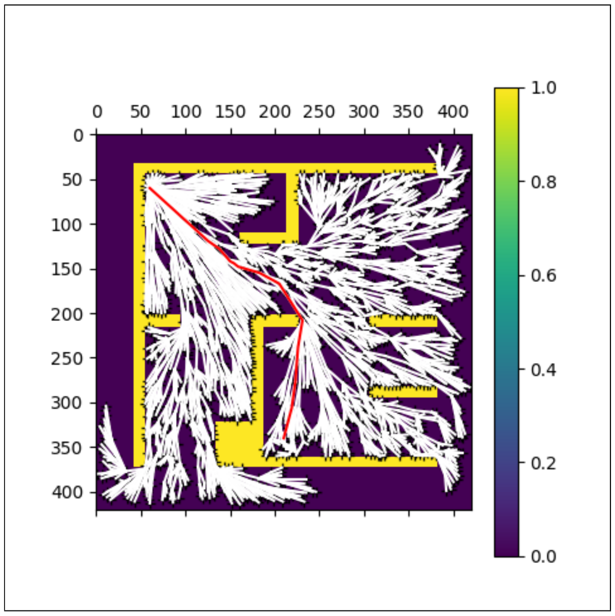
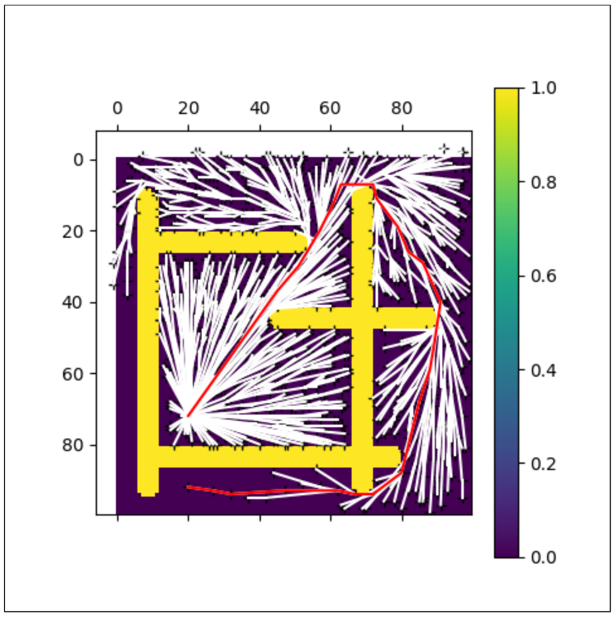

(b) After 5000 Iterations

Figure 12: RRT* Implementation on Map 7


(a) After 4598 Iterations

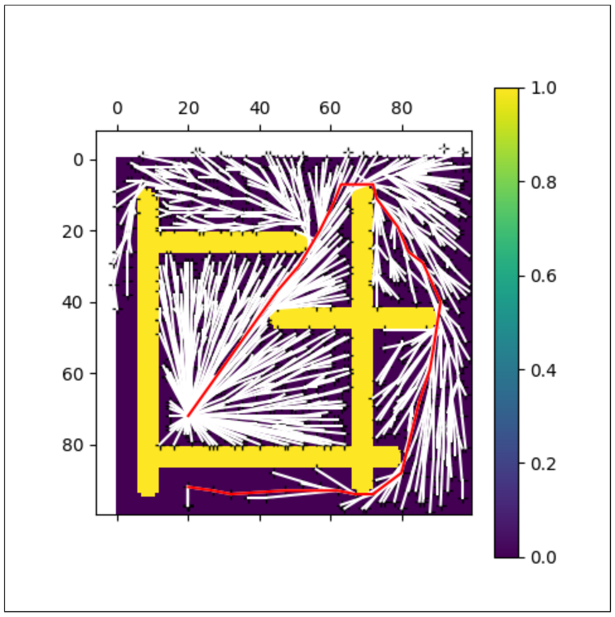
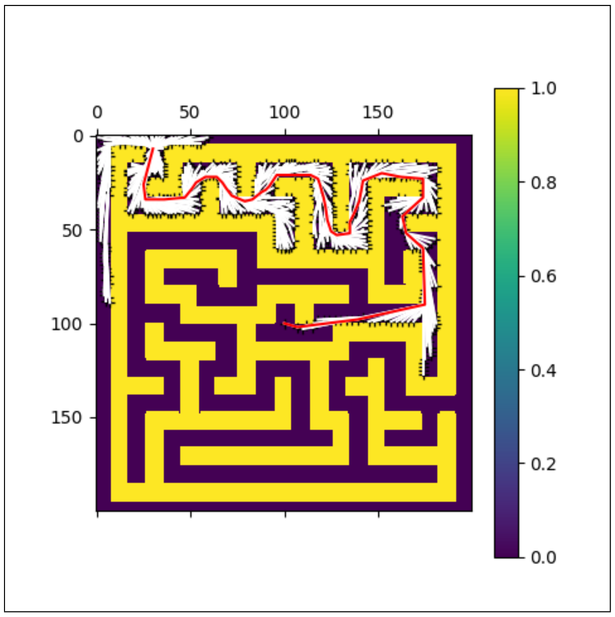
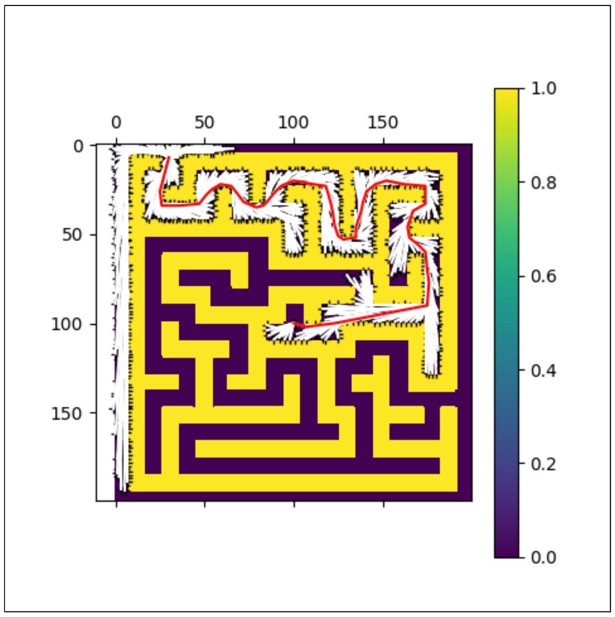

(b) After 5000 Iterations

Figure 13: RRT* Implementation on Map 1



(a) After 14598 Iterations



(b) After 20000 Iterations

Figure 14: RRT* Implementation on Map 2

4 Discussion & Conclusions

The results show that both RRT and RRT* demonstrated commendable performance in terms of finding the paths to the goal position from the start position. RRT exhibited faster computation times, especially in less cluttered environments, while RRT* showcased superior solution quality in complex scenarios.
Both algorithms demonstrated completeness, guaranteeing the finding of a solution if one existed. But RRT* showed a stronger inclination to reach optimality, especially in crowded circumstances where RRT sometimes offered less-than-ideal routes.
The algorithms exhibited differing levels of sensitivity to obstacles in the environment. RRT performed excellently in open and simple environments, while RRT* was more effective in cluttered spaces.
The paths generated by the RRT were not smooth due to its random nature. Post-processing techniques such as path smoothing greatly helped in producing smooth paths. On the other hand, RRT* generally produced smooth paths.
In difficult situations, RRT*'s iterative refining procedure (rewire and cost optimisation) led to a longer, but more effective convergence, while RRT displayed faster convergence because of its simpler exploration method.
In conclusion, the RRT and RRT* algorithms for path planning were implemented in this report. These algorithms were tested on different maps and scenarios. This offered insights into their strengths and weaknesses. The decision between RRT and RRT* is based on the particular needs of the robotic system as well as the peculiarities of the deployment environment.

Algorithm 5: generate_RRT_star(iteration, prob, del_q, n=100)

```
Data: iteration: int, prob: float, del_q: float, goal_thresh: int, n: int
Result: None
Input  : self: RRT_Star object
Output: None
1  Add start to tree;
2  for iter ← 1 to iteration do
3      q_rand ← self.q_random(prob);
4      q_near ← self.q_near(q_rand);
5      q_new ← self.extend_tree(q_near, q_rand, del_q);
6      if is_segment_free(q_near.pos, q_new.pos) then
7          q_min ← minim cost neighbor;
8          q_min ← q_near;
9          q neighbor list ← self.q_nearest_n_neighbors(q_new, n);
10         for q in q neighbor list do
11             if self.is_segment_free(q.pos, q_new.pos) then
12                 qcost ← q.cost + self.dist(q.pos, q_new.pos);
13                 if qcost < q_new.cost then
14                     q_min ← q;
15                     q_new.parent ← q_min;
16                     q_new.cost ← q_min.cost + self.dist(q_min.pos, q_new.pos);
17         Add q_min to nodes;
18         Add edge (q_min, q_new) to the edge list;
19         for q in q neighbor list do
20             if q ≠ q_min and self.is_segment_free(q_new.pos, q.pos) and q.cost >
                q_new.cost + self.dist(q_new.pos, q.pos) then
21                 Remove edge ((q.parent, q)) from edge list;
22                 q.parent ← q_new;
23                 q.cost ← q_new.cost + self.dist(q_new.pos, q.pos);
24                 self.add_edge(q_new, q);
25         if goal reached and already was not reached then
26             self.goal_reached ← True;
27             self.draw_path();
28             self.path ← [];
29             self.distance ← 0;
30     else
31         continue;
```