Autonomous Systems

Lab 4: Sample-Based Motion Planner algorithms

Delivered by:

Mazen Elgabalawy u1999109
Solomon Chibuzo Nwafor u1999124

Supervisor:

Sebastian Realpe Rua

Date of Submission:

08/12/2024

Table of Contents

1. Introduction:

Over the years, sample-based motion planners (SBMP) algorithms have been developed and shown to be effective in solving challenging motion planning problems that involve high dimensional space and more degrees of freedom (DoF) [1]. These algorithms include Probabilistic Roadmap planners, Rapidly-exploring Random Trees (RRT), and Rapidly-exploring Random Trees star (RRT*).. The SBMP also operates directly in continuous spaces which allows for more precise and flexible path planning in real time over traditional search-based planners like A* and Dijkstra's algorithm which requires discretized representation of the state space and a feasibility graph [2]. There are basic principles of SBMP which includes; configuration space (c-space), random sampling, and incremental connections. The c-space is a representation of all possible pose (position and orientation) of the robot. In this space, there is free space (set of configurations that avoids the obstacles), target space (where we want the robot to move to), and obstacle space (where the robot cannot move to) [3]. Each point is a unique configuration of the robot while obstacles are regions which the robot must avoid. The SBMP randomly samples points within the c-space and constructs paths in the space while avoiding obstacles [1]. By connecting sampled points that are not in collision, SBMP incrementally builds a roadmap or tree that represents potential paths through the free space which enables efficient path finding and exploration in complex environments [4]. The SBMPs are used in various fields beyond the traditional robotics in solving complex problems. For instance, in video games, SBMP can generate realistic movements for digital multiple characters or agents, especially in ensuring efficient navigation in complex environments [5]. Also, SBMP can assist in modeling the conformational changes of biomolecules, aiding in understanding protein folding and interactions in computational biology [6].

–**RRT** creates trees that are initialized from the start configuration and iteratively sampling random points in which these trees are extended towards the goal while ensuring that the paths does not collide with the obstacles. This approach efficiently searches high-dimensional spaces and is particularly useful for single-query problems [7]. However, it does not necessarily find the shortest path. On the other hand, **RRT*** is an extension of RRT by rewiring procedure that ensures the path cost converges to the optimum as the number of samples increases, achieving asymptotic optimality (the number of samples increases) [7]. The RRT* is designed to find the shortest path than the RRT but it is computationally more intensive due to its optimization process.

In this lab, we implemented two SBMP algorithms—RRT and RRT*—to find an optimal path in reaching a set goal configuration from the start configuration, and make some assumptions in order to reduce computational cost by sampling free path configurations [1]. This lab report is arranged as follows: In section 1, we introduced SBMP algorithms, their principles, and applications. Section 2 is the methodology where we explained in detail how the RRT and RRT* were implemented. And in Section 3, we discussed the results obtained for RRT and RRT*. In Section 4, we conclude the report.

2. Methodology:

In this section we explain the implementation of both the RRT and RRT* algorithms. In our approach we used **Object-Oriented-Programming (OOP)**, where we made **RRT** and **RRTStar** classes which encapsulate all the functions needed to implement both RRT and RRT*. Due to the similarity between both algorithms, both classes share most of the functions. In section 2.1, we explain those similar functions as well as give the pseudo code for them. Sections 2.2 and 2.3, explain the main algorithm for RRT and RRT* respectively.

2.1. Helper Functions:

2.1.1. sample_random_point:

This function samples a random point from the set of valid configurations in the gridmap, with a probability to sample the goal occasionally. The function takes the valid configurations and the probability of sampling the goal as input, and returns a randomly sampled point. The Pseudocode is given below.

```
FUNCTION sample_random_point (valid_rows, valid_cols, p):
        # Randomly choose an index from valid_cols
        rand_idx ← RANDOMLY select an index from valid_cols

        # Get the row and column corresponding to the random index
        x ← valid_rows[rand_idx]
        y ← valid_cols[rand_idx]

        # Determine the sampled point based on probability
        IF a RANDOM number between 0 and 1 < p:
        RETURN goal
        ELSE:
        RETURN a Point object at coordinates (x, y)
```

2.1.2. nearest_vertex:

Given a tree and a point in the gridmap, this function finds the node in the tree closest to the point. This is later used in the RRT and RRT* algorithms to connect a new node to the tree. The function takes as input the randomly sampled point and the tree and returns the closest point to the random node. The pseudocode is given below.

```
FUNCTION nearest_vertex (qrand, tree):
# Initialize the minimum distance to infinity
min_distance ← infinity

# Loop over all vertices in the tree
FOR each point in tree.keys:
        # Calculate the distance between qrand and the current point
```

```
        distance ← qrand.dist(point)

        # Check if the current point is the closest so far and not the same as qrand
        IF distance < min_distance AND distance != 0:
        # Update the minimum distance
        min_distance ← distance
        # Update the nearest vertex
        qnearest ← point

# Return the nearest vertex
RETURN qnearest
```

2.1.3. get_qnew:

This function steers towards the randomly generated point from its nearest neighbor belonging to the tree. The inputs to the function are the randomly sampled point, its nearest neighbor and the maximum distance to move in the direction of **qrand**. The function returns a new point **qnew.**

```
FUNCTION get_qnew (qrand, qnear, dq):
# Calculate the direction vector from qnear to qrand
direction ← qnear.vector(qrand)

# Calculate the distance between qnear and qrand
distance ← direction.norm()

# Calculate the unit vector in the direction of qrand from qnear
unit_vector ← direction.unit()

# Check if qrand and qnear are the same to avoid division by zero
IF distance == 0:
        RETURN qnear

# Calculate the step vector scaled by the smaller of dq or the actual distance
step ← unit_vector * min(dq, distance)

# Generate the new vertex by adding the step vector to qnear
qnew ← qnear.__add__(step)

# Return the new vertex
RETURN qnew
```

2.1.4. is_segement_free:

This function is used to check if the line connecting two points pass through an obstacle or not. This function is later used to prevent connecting two nodes of the tree if there is no obstacle-free line to connect them. The function takes as input two nodes and return a boolean value of **True** if the path connecting the two nodes is free, and returns **False** otherwise. The pseudocode is given below.

```
FUNCTION is_segment_free (p1, p2):
# Convert points p1 and p2 to NumPy arrays for calculations
p1 ← p1.numpy()
p2 ← p2.numpy()

# Divide the line segment between p1 and p2 into 50 equally spaced points
ps ← INTEGER array of 50 points linearly spaced between p1 and p2

# Check each point on the segment for obstacles
FOR each point (x, y) in ps:
        # Check if the point lies in an obstacle
        IF gridmap[x, y] == 1:
        RETURN False  # The segment is not free

# If no obstacles are found, the segment is free
RETURN True
```

2.1.5. reconstruct_path:

After reaching the goal, this function is used to find the full path from the starting node to the goal point. The function takes as input the tree and the end point, and returns a list containing the nodes making the path and the total cost of the path.

```
FUNCTION reconstruct_path (tree, q):
# Initialize the path reconstruction
current ← q  # Set the goal node as the current node
path ← [current]  # Start the path with the goal node
path_cost ← 0  # Initialize the total path cost to zero

# Traverse the tree from the goal node to the start node
WHILE current EXISTS in tree.keys:
        current ← tree[current]  # Move to the parent node
        APPEND current to path  # Add the current node to the path

# Reverse the path to order it from start to goal
REVERSE path

# Calculate the total path cost
FOR i FROM 2 TO length(path[1:]):
        path_cost ← path_cost + path[i - 1].dist(path[i])  # Add the distance between consecutive
points

# Return the reconstructed path (excluding goal and start nodes) and its total cost
RETURN path[1:-1], path_cost
```

2.1.6: Plotting functions:

In addition to the functions used in the classes, we implemented the plotting functions to ease the visualization of the resulting trees and maps. We have the **plot** function which is used to plot one map with one path, and the **plot2**, mainly used in RRT to plot both the path resulting from RRT and the smoothed path. The Pseudocode for both functions is given below.

```
FUNCTION plot (gridmap, start, goal, tree, path, show_vertices):
        # Initialize a visualization figure
        INITIALIZE a figure with size 10x10

        # Display the grid map
        VISUALIZE gridmap using matshow

        # Optional: Plot vertices if show_vertices is True
        IF show_vertices IS True:
        FOR each vertex v and its index i in tree.keys:
        PLOT vertex v at coordinates (v.y, v.x) as a green plus sign
        DISPLAY index i as text next to the vertex

        # Plot edges in the tree
        FOR each parent and child pair in tree:
        IF child IS NOT None:
        PLOT a white line connecting (parent.y, parent.x) and (child.y, child.x)

        # Plot the given path
        FOR each consecutive pair of points in path:
        PLOT a red line connecting the two points

        # Mark the start position
        PLOT a red star at (start[1], start[0]), labeled as 'Start'

        # Mark the goal position
        PLOT a blue star at (goal[1], goal[0]), labeled as 'Goal'

        # Add a legend
        DISPLAY a legend
```

```
FUNCTION plot2 (gridmap, start, goal, tree, original_path, smooth_path, show_vertices):
        # Initialize a visualization figure
        INITIALIZE a figure with size 10x10

        # Display the grid map
        VISUALIZE gridmap using matshow

        # Optional: Plot vertices if show_vertices is True
```

```
IF show_vertices IS True:
FOR each vertex v and its index i in tree.keys:
PLOT vertex v at coordinates (v.y, v.x) as a green plus sign
DISPLAY index i as text next to the vertex

# Plot edges in the tree
FOR each parent and child pair in tree:
IF child IS NOT None:
PLOT a white line connecting (parent.y, parent.x) and (child.y, child.x)

# Plot the original path
FOR each consecutive pair of points in original_path:
PLOT a red line connecting the two points
ADD label "Original-Path" only for the first segment

# Plot the smoothed path
FOR each consecutive pair of points in smooth_path:
PLOT a green line connecting the two points
ADD label "Smooth-Path" only for the first segment

# Mark the start position
PLOT a red star at (start[1], start[0]), labeled as 'Start'

# Mark the goal position
PLOT a blue star at (goal[1], goal[0]), labeled as 'Goal'

# Add a legend
DISPLAY a legend
```

2.2. RRT Implementation:

For the RRT algorithm, we created the **RRT** class, which takes as input the following:

Max_iter: the maximum number of iterations to run the algorithm

dq: The maximum distance to extend a node

p: The probability of sampling the goal point

start: The start point

goal: The goal point

In addition to the helper functions mentioned above, we implemented the **run** and the **smooth** functions, both of which are explained below.

2.2.1. run:

This function executes the RRT algorithm given the class attributes. The function takes no inputs and is run directly on class objects. If a path is found in the limit of **max_iter**, the function returns the resulting tree, the path found and its cost, as well as the number of iterations required to reach the goal. If no path is found, the function returns the tree, and an empty path and infinite cost (since the goal could not be reached using the resulting tree). We provide the pseudocode for the run function below.

```
FUNCTION run ():
# Initialize the tree with the starting node as the root (no parent)
tree ← EMPTY dictionary
tree[start] ← None

# Perform the RRT algorithm for a maximum number of iterations
FOR i FROM 0 TO max_iter - 1:
        # Sample a random point from the grid
        qrand ← sample_random_point(valid_rows, valid_cols, p)

        # Find the nearest vertex in the tree to qrand
        qnear ← nearest_vertex(qrand, tree)

        # Steer toward qrand from qnear to generate a new vertex qnew
        qnew ← get_qnew(qrand, qnear, dq)

        # Check if the segment between qnear and qnew is free of obstacles
        IF is_segment_free(qnear, qnew):
        # Add qnew to the tree with qnear as its parent
        tree[qnew] ← qnear

        # Check if qnew coincides with the goal
        IF qnew.dist(goal) == 0:
        # Add the goal to the tree with qnew as its parent
        tree[goal] ← qnew

        # Reconstruct the path from the start to the goal
        path, path_cost ← reconstruct_path(tree, goal)

        # Print success message and return the result
        PRINT "Path found in " + str(i) + " iterations"
        RETURN tree, path, path_cost, i

# If no path is found after maximum iterations
PRINT "No Path Found"
RETURN tree, [], infinity, i
```

2.2.2. smooth:

The smooth function is run on the resulting path after running the RRT algorithm to provide a smoother and more direct path towards the goal, as the path resulting from RRT might have unnecessary movements that increase the cost of the path. The function uses a greedy approach where we check directly from start to goal if there is a viable path, if not we check between the next point on the path and the goal and so on, until an obstacle-free path is found. We then repeat again until we have a smoother, more efficient path to the goal. The function takes as input the path resulting from RRT and returns a smoother path and the cost of the smooth path. The pseudocode is provided below.

```
FUNCTION smooth (path):
# Check if the input path is empty
IF path == []:
        # Return an empty path and infinite cost
        RETURN [], infinity

# Initialize variables
next_node ← path[-1]  # Set the goal as the initial next_node
i ← 0  # Index for iterating through the path
smooth_path ← [path[-1]]  # Start the smoothed path with the goal
smooth_path_cost ← 0  # Initialize the total cost of the smoothed path

# Perform smoothing until the start node is reached
WHILE smooth_path[-1] != path[0]:
        # Check if there is a direct, obstacle-free path from path[i] to next_node
        IF is_segment_free(path[i], next_node):
        # Add path[i] to the smoothed path
        smooth_path.APPEND(path[i])
        # Update the smoothed path cost with the distance to next_node
        smooth_path_cost ← smooth_path_cost + path[i].dist(next_node)
        # Update next_node to path[i]
        next_node ← path[i]
        # Reset the index to 0
        i ← 0
        ELSE:
        # Move to the next waypoint in the original path
        i ← i + 1

# Reverse the smoothed path to ensure it is ordered from start to goal
smooth_path.REVERSE()

# Return the smoothed path and its total cost
RETURN smooth_path, smooth_path_cost
```

2.3. RRT* Implementation:

For the RRT* Algorithm, we created the **RRTStar** class, that takes as input the following attributes:

Max_iter: the maximum number of iterations to run the algorithm

dq: The maximum distance to extend a node

p: The probability of sampling the goal point

Max_search_distance: The maximum distance between **qnew** and a node in the tree where cost optimization and rewiring can take place.

start: The start point

goal: The goal point

The main difference between RRT and RRT*, is the cost optimization and rewiring phases, where after finding a new node **qnew**, we first find the best node in the maximum search radius that minimizes the cost of **qnew**. Then after cost optimization for **qnew**, we rewire the neighbors of **qnew** in the search radius to minimize their cost further if possible. These two extra steps ensure that RRT* is asymptotically optimal and if run long enough is guaranteed to find the optimal path.

For running the actual algorithm, we implemented the **run** function similar to the one in **RRT** class but with the added steps of **Cost Optimization** and **Node Rewiring**. The pseudocode for the function is given below.

```
FUNCTION run ():
# Initialize the tree with the start node as the root (no parent)
tree ← EMPTY dictionary
tree[start] ← None

# Create a copy of the initial tree for storing the first path found
first_tree ← tree

# Initialize costs of nodes, starting with the cost of the start node as 0
costs ← EMPTY dictionary
costs[start] ← 0

# Flag to mark if the goal has been reached for the first time
reached_goal ← False

# Perform RRT* iterations
FOR i FROM 0 TO max_iter - 1:
        # Sample a random point from the grid
        qrand ← sample_random_point(valid_rows, valid_cols, p)

        # Find the nearest vertex in the tree to qrand
        qnear ← nearest_vertex(qrand, tree)
```

```
        # Steer toward qrand from qnear to generate a new vertex qnew
        qnew ← get_qnew(qrand, qnear, dq)

        # Check if the segment between qnear and qnew is obstacle-free
        IF is_segment_free(qnear, qnew):
        # Update the cost of reaching qnew
        costs[qnew] ← costs[qnear] + qnew.dist(qnear)

        # Initialize a list of neighbors for qnew and set qnear as the initial parent
        qnew_neighbors ← EMPTY list
        qmin ← qnear

        # Cost optimization
        FOR each vertex j in tree.keys:
        IF qnew.dist(j) < max_search_distance AND is_segment_free(qnew, j):
        APPEND j to qnew_neighbors
        new_cost ← costs[j] + qnew.dist(j)

        # Update parent and cost of qnew if a lower cost path is found
        IF new_cost < costs[qnew]:
                qmin ← j
                costs[qnew] ← new_cost

        # Add qnew to the tree with qmin as its parent
        IF qnew NOT IN tree:
        tree[qnew] ← qmin

        # Check if qnew coincides with the goal and if this is the first time the goal is reached
        IF qnew.dist(goal) == 0 AND NOT reached_goal:
        reached_goal ← True
        tree[goal] ← qnew
        first_tree ← DEEP COPY of tree
        first_path, first_path_cost ← reconstruct_path(tree, goal)
        costs[goal] ← first_path_cost
        first_iter ← i
        PRINT "First path found after " + str(i) + " iterations"

        # Rewiring step
        FOR each neighbor in qnew_neighbors:
        IF neighbor != qmin:
        new_neighbor_cost ← costs[qnew] + qnew.dist(neighbor)

        # Update parent and cost of neighbor if a lower cost path is found
        IF new_neighbor_cost < costs[neighbor]:
                tree[neighbor] ← qnew
                costs[neighbor] ← new_neighbor_cost

# After all iterations, check if the goal is in the tree
IF goal IN tree.keys:
        final_path, final_path_cost ← reconstruct_path(tree, goal)
```

```
        RETURN first_tree, first_path[:-1], first_path_cost, first_iter, tree, final_path, final_path_cost

# If no path was found
PRINT "No Path Found"
RETURN EMPTY dictionary, EMPTY list, infinity, 0, tree, EMPTY list, infinity
```

2.4. Remarks for Execution

- For the RRT script rrt.py, use the following interface:

$ path-to-dirctory/rrt.py path-to-grid-map-image max_iters dq p qstart_x qstart_y qgoal_x qgoal_y

- For the RRT* script rrt_star.py, use the following interface:

$ path-to-dirctory/rrt.py path-to-grid-map-image max_iters dq p max_search_radius qstart_x qstart_y qgoal_x qgoal_y

- For RRT*, don't use a very large number of iterations; **it will result in the script freezing**
- For **Maze-like** maps (e.g., Map2 or Map5), don't use a large search radius with RRT*, as it will result in unnecessary checks between neighbors that cannot be reached, exponentially increasing execution time.

3. Results:

This section provides the results for both RRT and RRT* on four different grid maps. During testing, we set the minimum distance to choose the goal as the final vertex to zero. We also tested on more grid maps, which are presented in the appendix section.

3.1. RRT:

3.1.1. Map0:

max_iter = 1000, **dq** = 10, **p** = 0.2 , **start**=(10,10), **goal**= (90,70),

Path to follow: (10.0, 10.0), (19.61, 12.78), (28.67, 17.0), (38.63, 16.12), (45.53, 23.36), (52.43, 30.6), (60.62, 36.33), (60.49, 46.33), (60.36, 56.33), (56.71, 65.65), (56.96, 75.64), (63.93, 82.81), (68.0, 87.0), (76.07, 92.91), (85.61, 89.9), (87.76, 80.14), (89.92, 70.37), (90.0, 70.0)

Smooth Path: (10.0, 10.0), (38.63, 16.12), (52.43, 30.6), (56.96, 75.64), (76.07, 92.91), (90.0, 70.0)

Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124

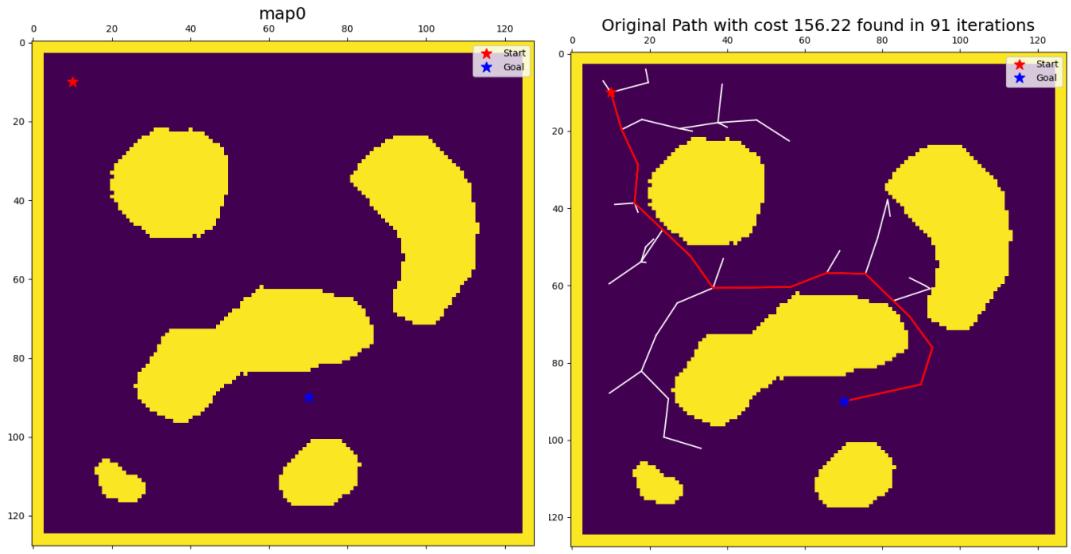


Figure 1: RRT of map0 path

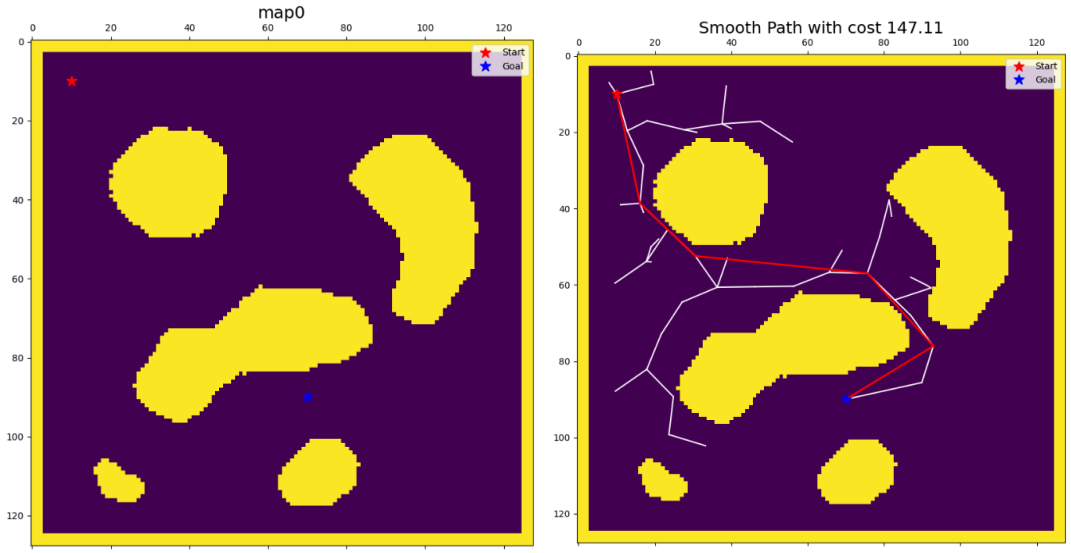


Figure 2: Smooth Path of map0

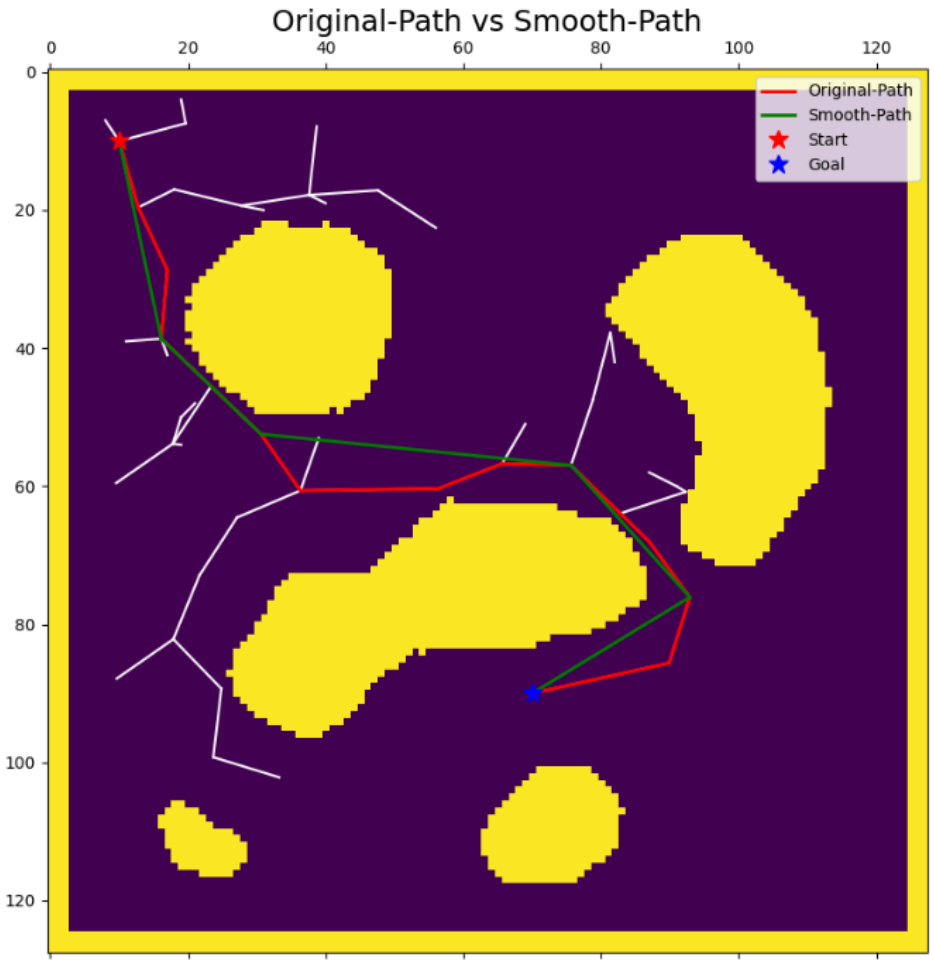


Figure 3: Combined RRT and Smooth-Path of map0

3.1.2. Map1:

max_iter = 1000 **dq** = 10 **p** = 0.2 **start** = (60, 60) **goal**= (90, 60)

Path to follow: (60.0, 60.0), (61.06, 50.06), (57.88, 40.57), (48.44, 37.3), (42.31, 29.39), (32.41, 30.86), (27.51, 39.57), (27.74, 49.57), (26.43, 59.49), (17.95, 64.78), (19.0, 60.0), (13.76, 51.48), (11.0, 53.0), (6.0, 51.0), (1.0, 55.0), (4.0, 60.0), (4.0, 70.0), (5.19, 79.93), (15.01, 81.84), (23.79, 86.62), (27.88, 95.74), (35.0, 90.0), (44.62, 92.75), (54.36, 90.52), (55.0, 92.0), (63.94, 96.47), (73.94, 96.85), (83.93, 96.45), (93.92, 96.13), (94.0, 96.0), (97.59, 86.67), (95.0, 86.0), (92.13, 76.42), (95.99, 67.2), (90.0, 60.0).

Smooth Path: (60.0, 60.0), (48.44, 37.3), (26.43, 59.49), (13.76, 51.48), (4.0, 70.0), (27.88, 95.74), (83.93, 96.45), (95.99, 67.2), (90.0, 60.0)

Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124

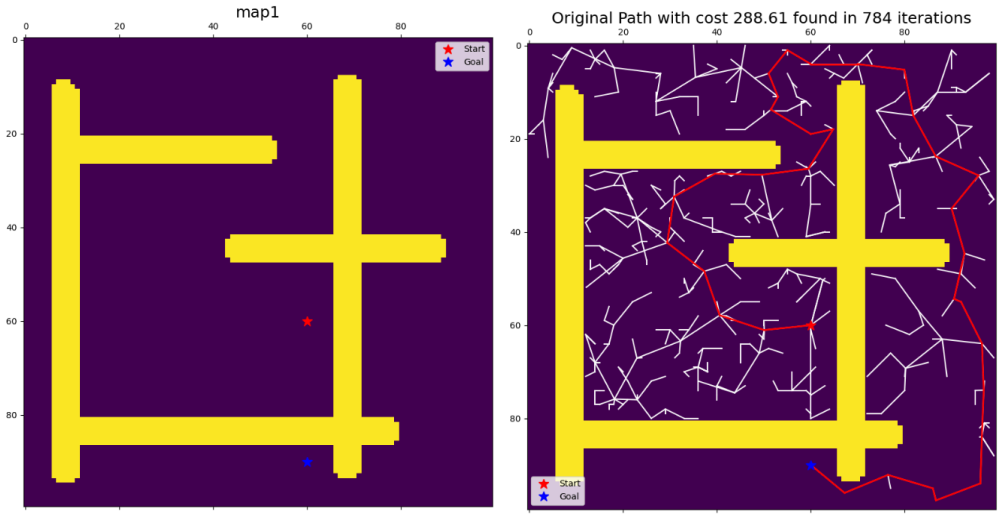


Figure 4: RRT of map1 path

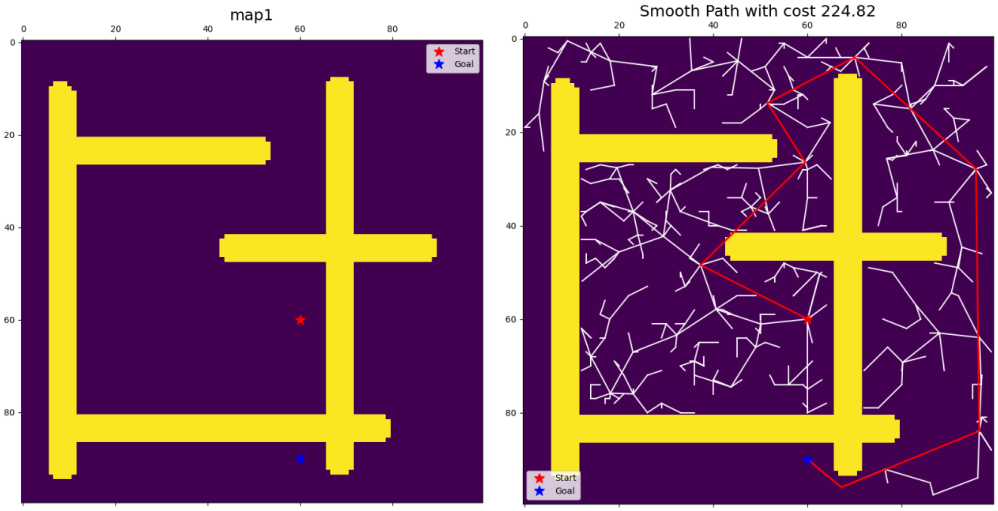


Figure 5: Smooth Path of map1

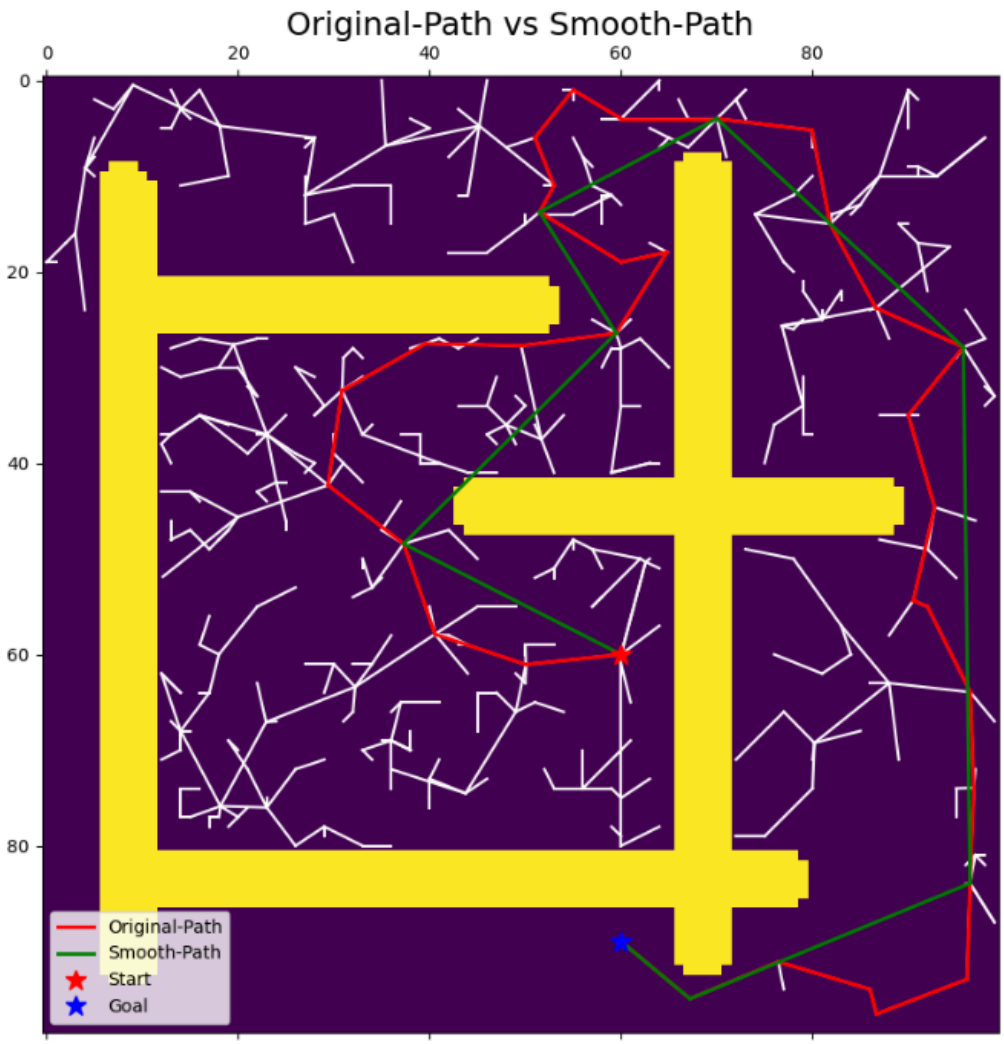


Figure 6: Combined RRT and Smooth-Path of map1

3.1.3. Map2:

max_iter = 15000 **dq** = 10 **p** = 0.2 **start** = (8, 31) **goal**= (139, 38)

Path to follow: (8.0, 31.0), (17.98, 30.37), (20.0, 26.0), (29.98, 25.32), (39.91, 26.48), (40.79, 36.44), (39.27, 46.32), (29.71, 49.25), (21.25, 54.58), (22.17, 64.54), (27.06, 73.26), (36.81, 75.48), (34.72, 85.25), (26.0, 88.0), (19.0, 90.0), (19.0, 92.0), (17.0, 100.0), (19.04, 109.79), (23.21, 118.88), (30.8, 125.39), (36.0, 124.0), (38.0, 121.0), (47.28, 124.71), (53.0, 124.0), (55.79, 133.6), (54.07, 143.45), (48.0, 144.0), (39.68, 138.45), (38.0, 136.0), (34.0, 140.0), (28.0, 136.0), (21.3, 143.42), (20.81, 153.41), (21.39, 163.39), (17.24, 172.49), (21.34, 181.62), (23.0, 180.0), (32.16, 175.99), (37.0, 176.0), (39.43, 166.3), (45.0, 161.0), (54.28, 164.71), (55.0, 173.0), (64.86, 174.68), (74.43, 177.56), (84.07, 174.89), (94.03, 175.8), (91.0, 173.0), (96.24, 164.48), (92.71, 155.13), (96.92, 146.06), (94.37, 136.39), (85.0, 136.0), (80.0, 135.0), (76.0, 129.0), (77.0, 128.0), (77.0, 121.0), (71.0, 114.0), (71.0, 109.0), (78.0, 106.0), (72.0, 98.0), (72.45, 88.01), (76.0, 81.0), (70.0, 76.0), (65.0, 75.0), (58.0, 78.0), (54.58, 68.6), (53.0, 64.0), (55.0, 58.0), (58.0, 49.0), (53.0, 50.0), (53.0, 49.0), (54.64, 39.14), (59.0, 34.0), (56.0, 34.0), (54.0, 33.0), (52.0, 31.0), (52.0, 27.0), (57.0, 26.0), (54.0, 20.0), (58.0, 20.0), (67.0, 24.0), (65.0, 19.0), (68.0, 19.0), (77.28, 22.71), (87.21, 23.91), (88.0, 23.0), (96.0, 24.0), (99.0, 18.0), (105.0, 21.0),

(114.9, 22.41), (123.3, 27.84), (125.15, 37.67), (135.15, 37.91), (139.0, 38.0)

Smooth Path: (8.0, 31.0), (20.0, 26.0), (29.98, 25.32), (39.91, 26.48), (39.27, 46.32), (21.25, 54.58), (22.17, 64.54), (36.81, 75.48), (34.72, 85.25), (19.0, 90.0), (23.21, 118.88), (53.0, 124.0), (55.79, 133.6), (21.3, 143.42), (21.34, 181.62), (32.16, 175.99), (45.0, 161.0), (54.28, 164.71), (55.0, 173.0), (64.86, 174.68), (94.03, 175.8), (96.92, 146.06), (80.0, 135.0), (76.0, 81.0), (58.0, 78.0), (57.0, 26.0), (114.9, 22.41), (123.3, 27.84), (125.15, 37.67), (139.0, 38.0)

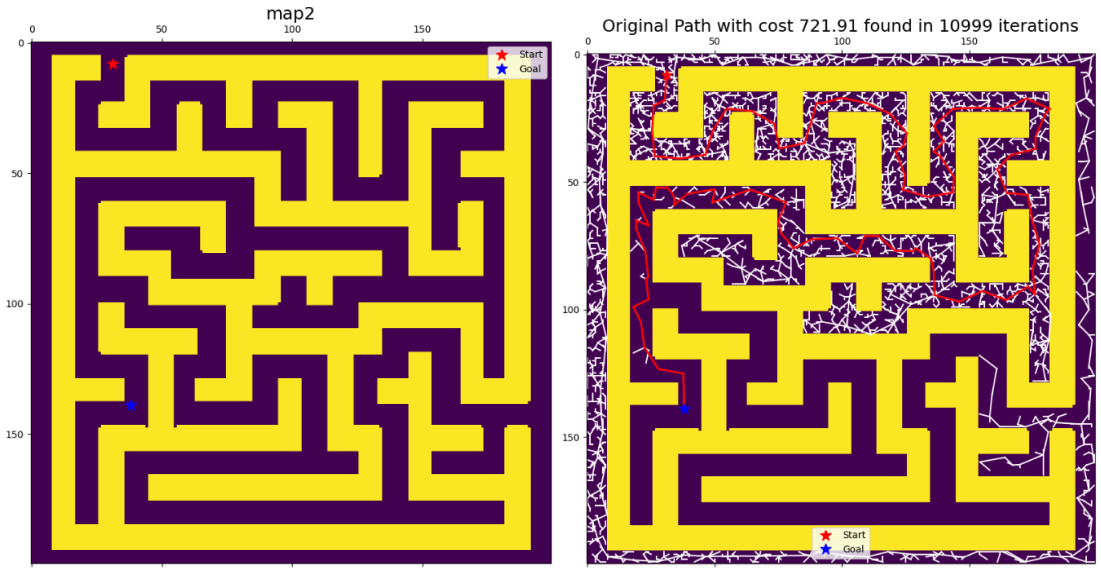


Figure 7: RRT of map2 path

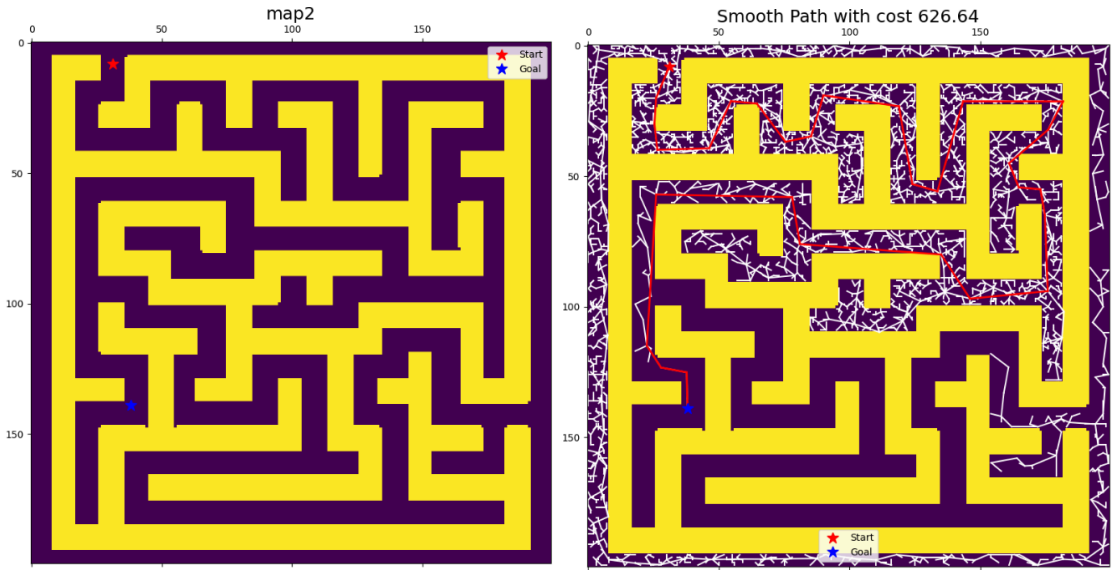


Figure 8: Smooth Path of map2

Figure 9: Combined RRT and Smooth-Path of map2

3.1.4. Map3:

max_iter = 1000 **dq** = 20 **p** = 0.2 **start** = (50, 90) **goal**= (375, 375)

Path to follow: (50.0, 90.0), (49.34, 109.99), (50.81, 129.93), (46.91, 149.55), (63.39, 160.88), (59.91, 180.57), (74.81, 193.91), (72.72, 213.8), (90.37, 223.21), (97.67, 241.83), (115.85, 250.18), (133.86, 258.86), (137.61, 278.5), (156.14, 286.04), (176.14, 285.93), (194.04, 294.85), (212.32, 302.95), (227.92, 315.47), (245.86, 324.32), (249.15, 344.04), (251.52, 363.9), (258.72, 382.56), (277.76, 376.45), (291.48, 391.01), (307.99, 402.29), (307.41, 422.28), (327.13, 425.65), (340.93, 440.12), (360.93, 439.72), (373.95, 424.54), (374.38, 404.55), (374.8, 384.55), (375.0, 375.0).

Smooth Path: (50.0, 90.0), (97.67, 241.83), (360.93, 439.72), (375.0, 375.0).

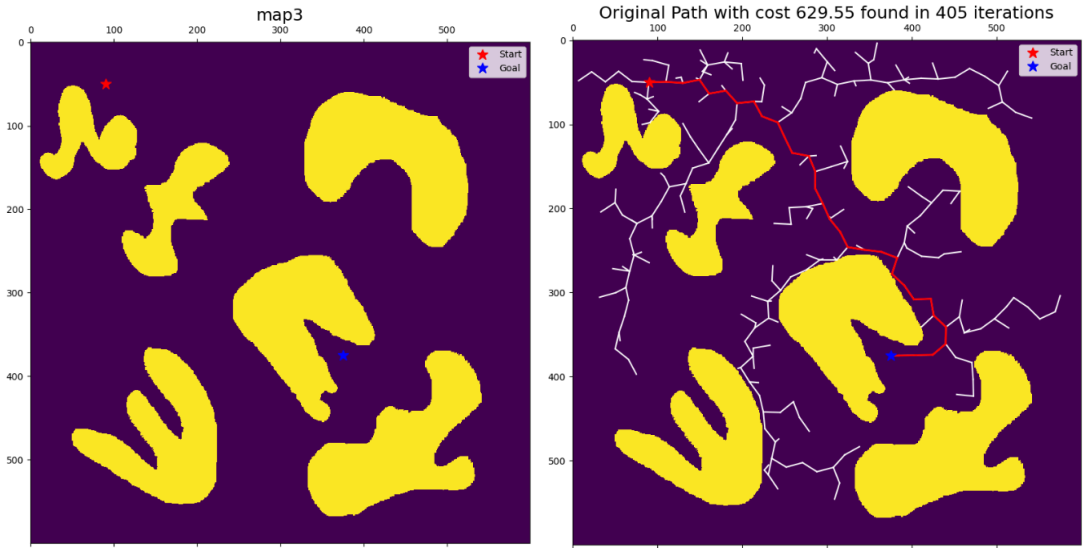


Figure 10: RRT of map3 path

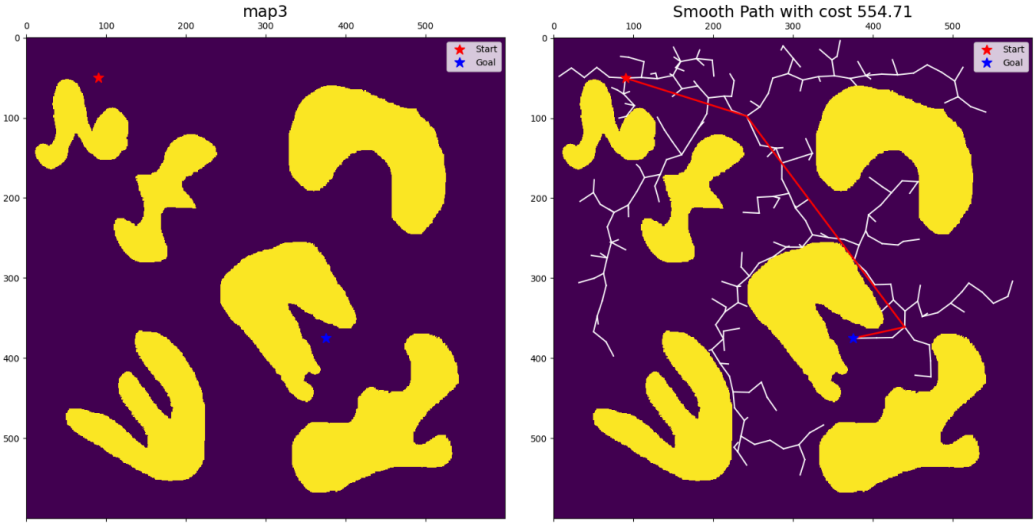


Figure 11: Smooth Path of map3

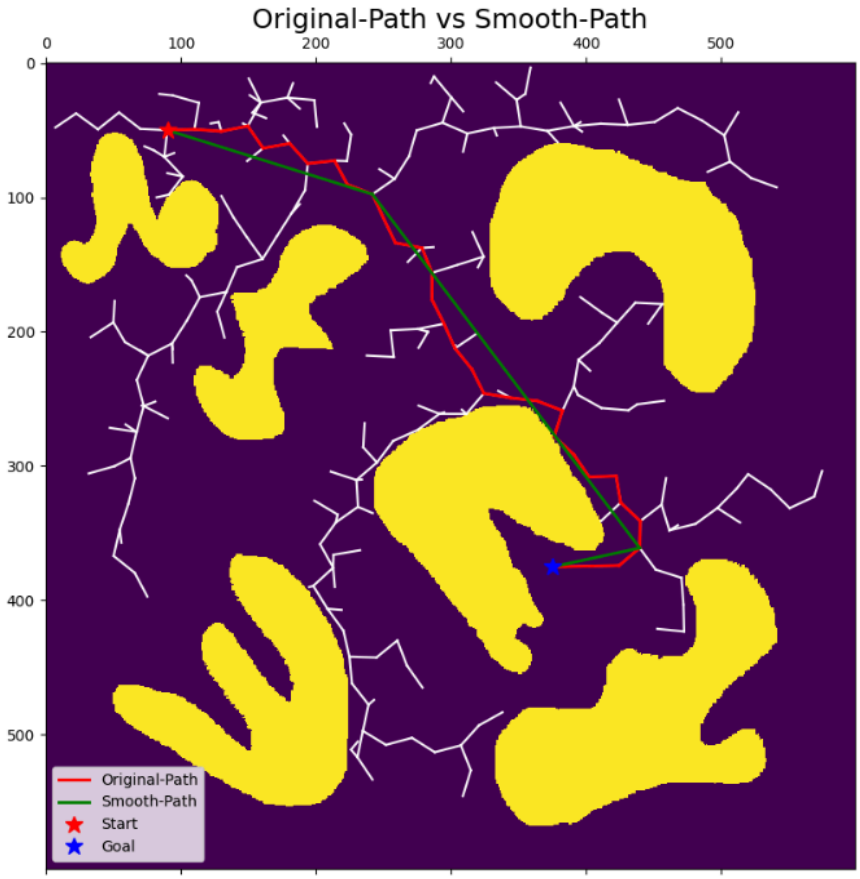


Figure 12: Combined RRT and Smooth-Path of map3

Table 1: Performance of the RRT and Smooth-Path on the four grid maps

Map	**Path found**	**Total Path Cost**	**Smooth Path Cost**
Map0	91 iterations	156.2176987655537	147.11287660976743
Map1	784 iterations	288.6083849275888	224.81575383462115
Map2	10999 iterations	721.9057416525919	626.6373078206044
Map3	405 iterations	629.5540592352531	554.7114459185377

From the results shown above, it can be seen that applying smoothing to the obtained path can significantly reduce the cost of the path.

3.2. RRT*:

3.2.1. Map0:

max_iter = 1000, **dq** = 5, **p** = 0.2, **max_search_radius** = 30, **start**=(10,10), **goal**= (90,70)

Path to follow: (10.0, 10.0), (13.0, 19.0), (22.41, 45.66), (41.0, 68.0), (51.37, 77.68), (76.14, 90.64), (90.0, 70.0)

Final Path to follow: (10.0, 10.0), (30.1, 13.69), (53.0, 18.46), (75.0, 23.0), (91.0, 26.0), (97.0, 34.0), (98.0, 42.0), (90.0, 70.0), (90.0, 70.0)

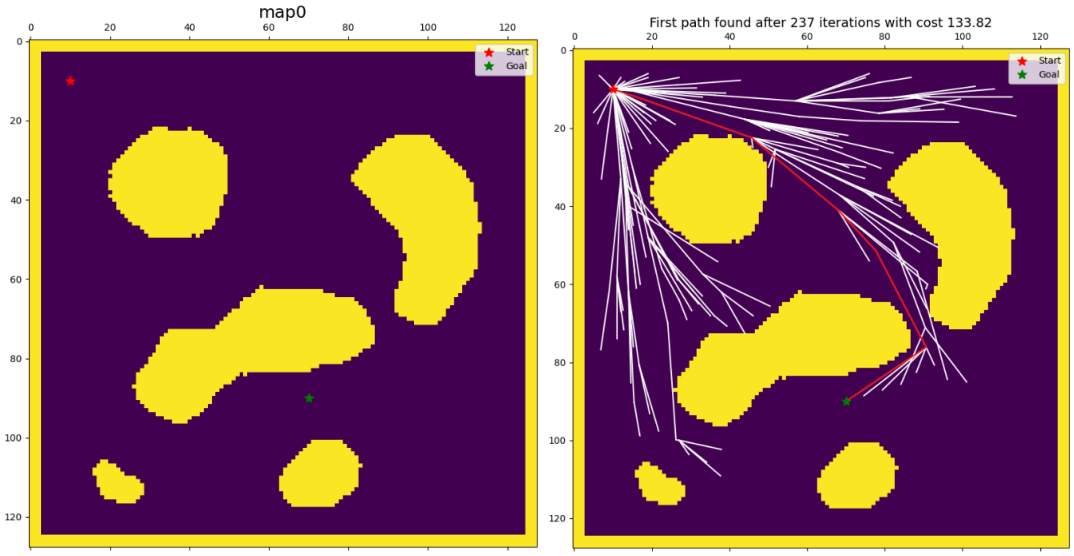


Figure 13: First path obtained from RRT* for map0

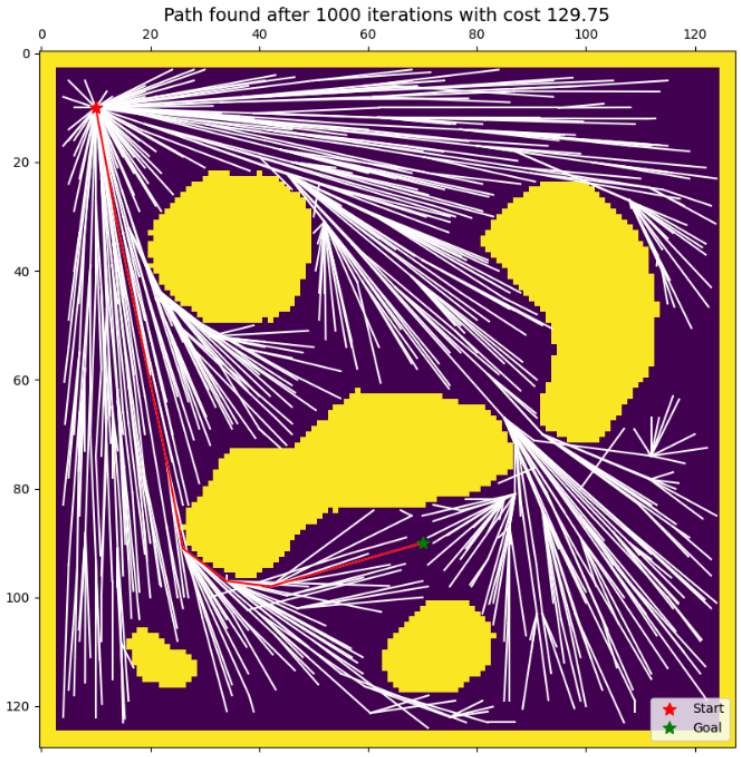


Figure 14: RRT* of map0 Final Path

3.2.2. Map1:

max_iter = 2000, **dq** = 10, **p** = 0.2, **max_search_radius** = 20, **start**=(60,60), **goal**= (90,60)

Path to follow: (60.0, 60.0), (49.0, 44.0), (47.0, 41.0), (40.0, 43.0), (25.0, 56.0), (15.0, 61.0), (6.0, 68.0), (14.0, 84.0), (29.0, 89.0), (43.0, 91.0), (59.0, 89.0), (68.0, 88.0), (85.0, 82.0), (94.0, 74.0), (97.0, 65.0), (90.0, 60.0)

Final Path to Follow: (60.0, 60.0), (49.0, 44.0), (44.0, 41.0), (32.0, 50.0), (28.0, 53.0), (13.0, 63.0), (7.0, 67.0), (7.0, 71.0), (17.0, 76.0), (29.0, 85.0), (39.0, 90.0), (47.0, 90.0), (63.0, 85.0), (76.0, 81.0), (86.0, 81.0), (96.0, 69.0), (90.0, 60.0)

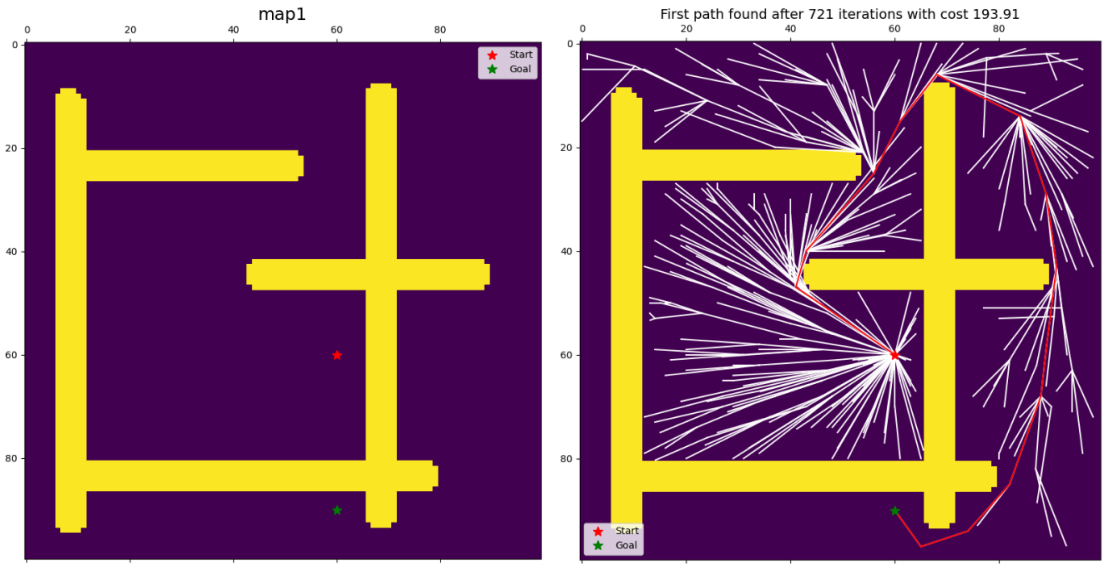


Figure 15: First path obtained from RRT* for map1

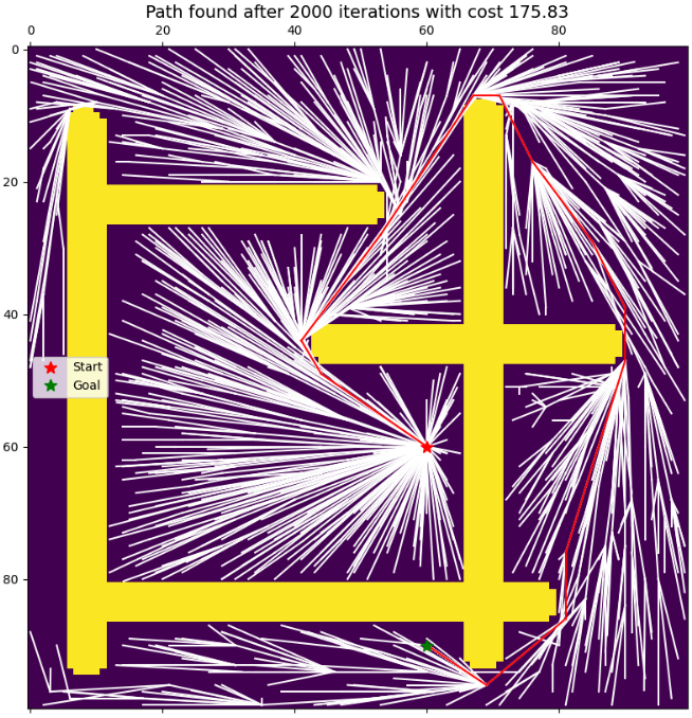


Figure 16: RRT* of map1 Final Path

3.2.3. Map2:

max_iter = 20000, **dq** = 10, **p** = 0.2, **max_search_radius** = 20, **start**=(8,31), **goal**= (139,38)

Path to follow: (8.0, 31.0), (22.0, 26.0), (33.0, 25.0), (33.0, 40.0), (33.0, 47.0), (24.0, 55.0), (22.0, 59.0), (23.0, 65.0), (33.0, 75.0), (33.0, 81.0), (33.0, 86.0), (23.0, 93.0), (21.0, 100.0),

(22.0, 116.0), (25.0, 117.0), (28.0, 118.0), (47.0, 124.0), (52.0, 126.0), (52.0, 135.0), (34.0, 141.0), (22.0, 145.0), (20.0, 160.0), (23.0, 174.0), (34.0, 174.0), (42.0, 164.0), (52.0, 165.0), (61.0, 174.0), (80.0, 174.0), (90.0, 174.0), (90.0, 171.0), (90.0, 166.0), (91.0, 147.0), (88.0, 141.0), (81.0, 137.0), (79.0, 133.0), (79.0, 129.0), (78.0, 117.0), (77.0, 106.0), (72.0, 87.0), (64.0, 79.0), (61.0, 76.0), (58.0, 62.0), (58.0, 43.0), (59.0, 27.0), (71.0, 22.0), (78.0, 20.0), (93.0, 19.0), (109.0, 20.0), (120.46, 26.63), (123.0, 35.0), (132.83, 36.84), (139.0, 38.0)

Fianl Path to follow: (8.0, 31.0), (22.0, 26.0), (33.0, 25.0), (33.0, 40.0), (33.0, 47.0), (24.0, 55.0), (22.0, 59.0), (23.0, 65.0), (33.0, 75.0), (33.0, 81.0), (33.0, 86.0), (23.0, 93.0), (22.0, 103.0), (22.0, 116.0), (34.0, 121.0), (52.0, 126.0), (52.0, 127.0), (52.0, 135.0), (47.0, 138.0), (30.0, 142.0), (24.0, 144.0), (22.0, 146.0), (22.0, 160.0), (23.0, 174.0), (31.0, 174.0), (34.0, 174.0), (41.0, 166.0), (43.0, 164.0), (52.0, 165.0), (61.0, 174.0), (80.0, 175.0), (90.0, 174.0), (90.0, 158.0), (91.0, 146.0), (85.0, 140.0), (79.0, 134.0), (79.0, 123.0), (76.0, 109.0), (72.0, 91.0), (71.0, 85.0), (63.0, 76.0), (61.0, 75.0), (59.0, 56.0), (60.0, 47.0), (58.0, 32.0), (62.0, 25.0), (70.0, 25.0), (87.0, 24.0), (104.0, 24.0), (121.0, 26.0), (125.0, 35.0), (134.0, 38.0), (139.0, 38.0)

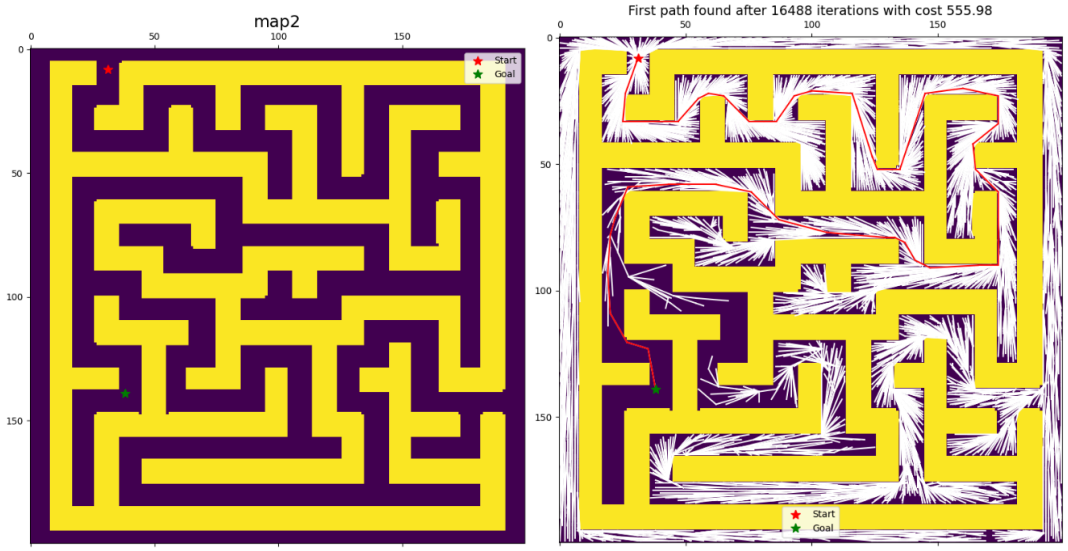


Figure17: First path obtained from RRT* for map2

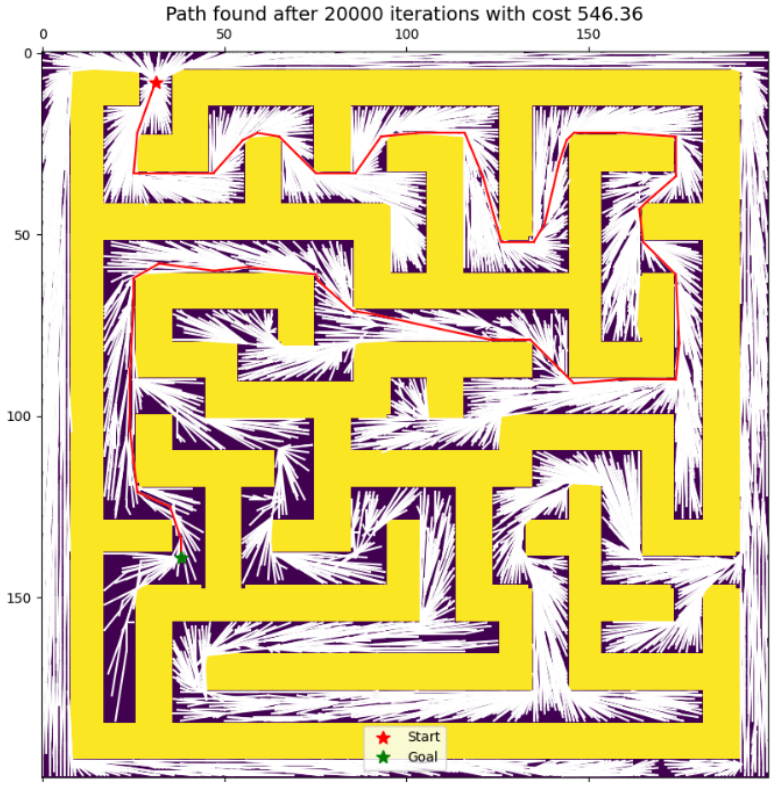


Figure 18: RRT* of map2 Final Path

3.2.4. Map3:

max_iter = 5000, **dq** = 5, **p** = 0.2, **max_search_radius** = 20, **start**=(50,90), **goal**= (375,375)

Path to follow: (50.0, 90.0), (53.76, 93.3), (65.91, 106.06), (76.36, 119.3), (90.88, 133.02), (101.8, 148.51), (110.27, 162.21), (112.18, 166.83), (114.93, 185.18), (117.83, 201.19), (122.5, 218.79), (129.23, 232.51), (132.49, 248.14), (134.94, 254.95), (146.16, 265.86), (160.49, 276.86), (172.4, 284.82), (179.16, 291.35), (188.76, 301.39), (198.5, 310.69), (215.65, 319.63), (234.63, 324.88), (244.1, 328.09), (252.15, 345.87), (256.41, 361.71), (260.36, 372.53), (275.05, 385.3), (288.47, 389.93), (295.6, 393.57), (308.06, 399.38), (321.89, 407.9), (328.43, 412.67), (343.27, 424.0), (359.67, 416.28), (364.89, 402.22), (366.63, 397.53), (373.88, 379.64), (375.0, 375.0)

Final Path to Follow: (50.0, 90.0), (53.76, 93.3), (65.91, 106.06), (76.36, 119.3), (90.88, 133.02), (100.0, 146.0), (105.0, 155.0), (112.18, 166.83), (115.0, 182.0), (117.83, 201.19), (122.5, 218.79), (129.23, 232.51), (141.24, 243.56), (151.0, 257.0), (162.0, 271.0), (172.4, 284.82), (187.0, 295.0), (202.65, 301.31), (220.84, 307.11), (232.7, 315.6), (240.0, 321.0), (244.1, 328.09), (252.15, 345.87), (253.34, 354.7), (261.82, 367.75), (274.34, 380.35), (284.37, 387.07), (295.0, 393.0), (308.06, 399.38), (326.67, 406.42), (343.91, 415.93), (354.67, 416.03), (363.15, 406.91), (368.0, 395.0), (373.88, 379.64), (375.0, 375.0)

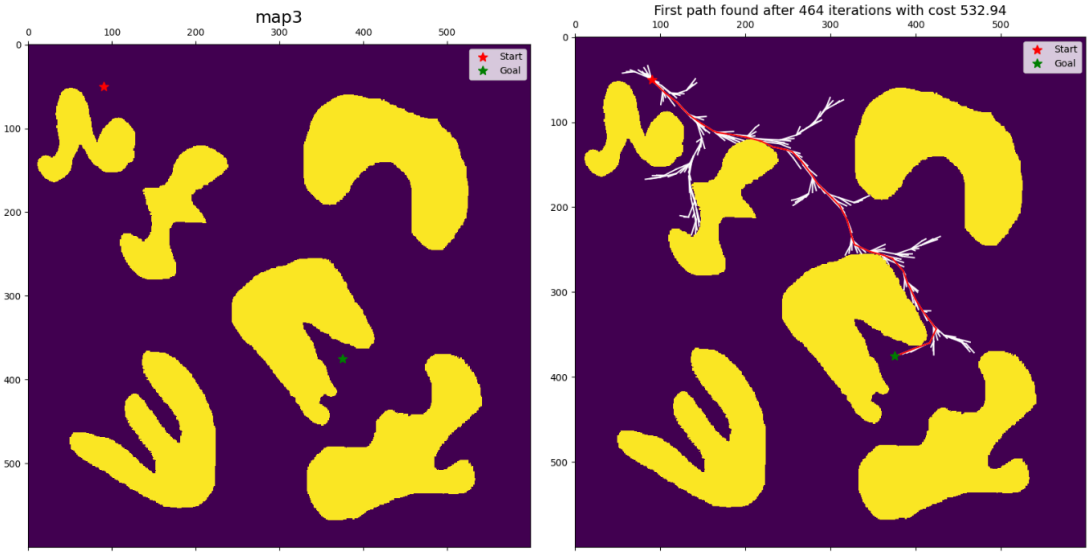


Figure19: First path obtained from RRT* for map4

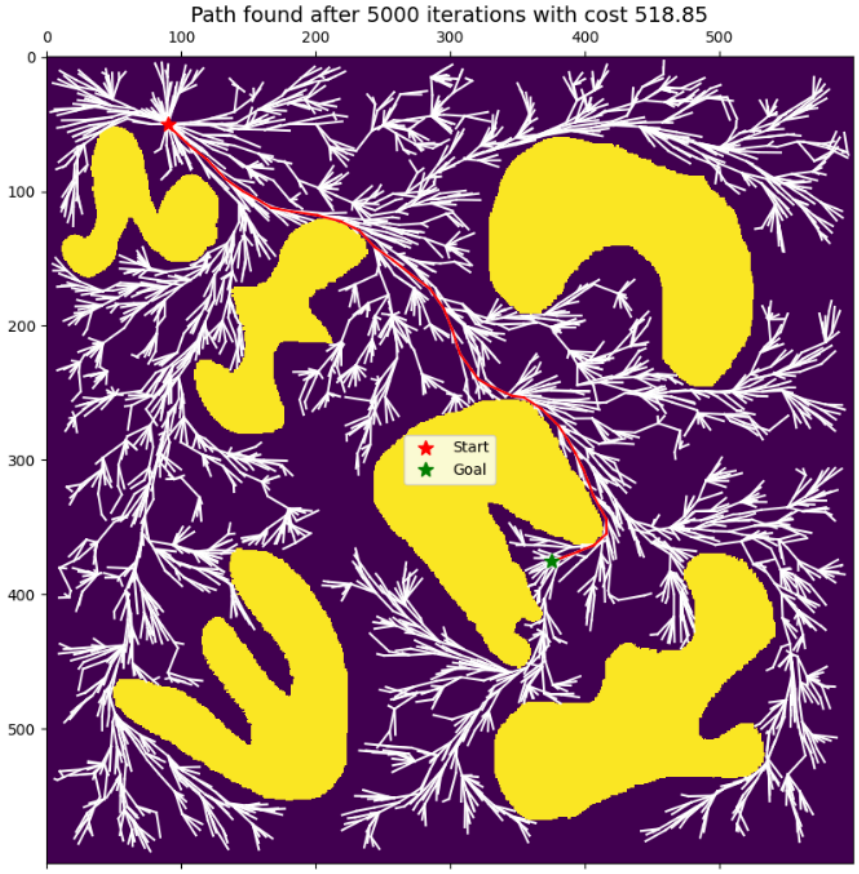


Figure 20: RRT* of map4 Final Path

Table 2: Performance of the RRT* on the four grid maps

Map	**First Path found**	**First Path Cost**	**Final Path Cost**
Map0	237 iterations	133.82235502367473	129.752449312443
Map1	721 iterations	193.9141539664273	175.83200291182766
Map2	16488 iterations	555.9759584803489	546.3571114340813
Map3	464 iterations	532.9377551828592	518.8481066121777

It is seen from Table 2 and the above figures, that running RRT* for more iterations results in a more optimal and efficient path.

4. Challenges:

This section discusses the challenges that we have encountered during our implementation of the algorithms for this lab.

4.1. Code Optimization:

Both RRT and RRT* contain parts where we loop over all nodes that have been added to the tree. During our first implementation, we had multiples of those loops in our code, resulting in very slow execution times especially as the maximum number of iterations was increased.

To Solve this, we had to rewrite our code in a more optimized way that reduces the number of loops to reduce the execution time.

4.2. Self-Referencing Nodes:

During our testing, we noticed that the program might get stuck in an infinite loop sometimes. When debugging, we found out that some nodes of the tree were referencing themselves as their own parents (self-referencing). This resulted in the program being stuck while reconstructing the path as it can no longer traverse the tree properly.

Our solution was to check if a sampled node already exists in the tree and ignore the node if that is the case. This ensures that a node can never self-reference, as each node can be created only once.

5. Conclusion:

In this lab, we implemented the RRT and RRT* algorithms which are Sample-Based-Motion-Planners (SBMP). It was seen that both algorithms performed well in different grid map environments, finding a path from start to goal configurations in the C-Space. We implemented RRT as well as applied smoothing to the resulting path to obtain a more cost efficient and optimal path.

The testing of RRT* showed that the more iterations that are run, the better and more optimal the resulting path is, however, this came at the cost of execution time. However, in comparison to other motion-planning algorithms seen in previous labs (e.g Potential Functions or A*), we have seen that SBMP offer real-time performance as they don't require a visibility graph beforehand, as well as the operation in continuous space without the need to discretize the C-Space.

References

[1] Lindemann, S.R., LaValle, S.M. (2005). Current Issues in Sampling-Based Motion Planning. In: Dario, P., Chatila, R. (eds) Robotics Research. The Eleventh International Symposium. Springer Tracts in Advanced Robotics, vol 15. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11008941_5

[2] The realm of Sampling-Based Planning (2023)

https://www.weeklyrobotics.com/articles/2023_04_21_sampling_based_planners/?form=MG0AV3

[3] Motion Planning https://en.wikipedia.org/wiki/Motion_planning [Accessed: 6/12/2024]

[4] Sampling-Based Motion Planning - Pieter Abbeel

https://people.eecs.berkeley.edu/~pabbeel/cs287-fa12/slides/SamplingBasedMotionPlanning.pdf

[5] Motion Planning for Robotics: A Review for Sampling-based Planners (2024) https://arxiv.org/html/2410.19414v1?form=MG0AV3

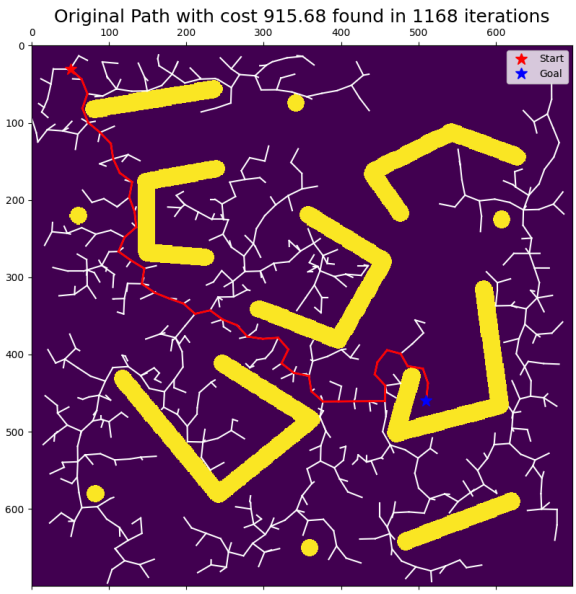
[6] Ekenna, C., Thomas, S. & Amato, N.M. Adaptive local learning in sampling based motion planning for protein folding. BMC Syst Biol 10 (Suppl 2), 49 (2016). https://doi.org/10.1186/s12918-016-0297-9

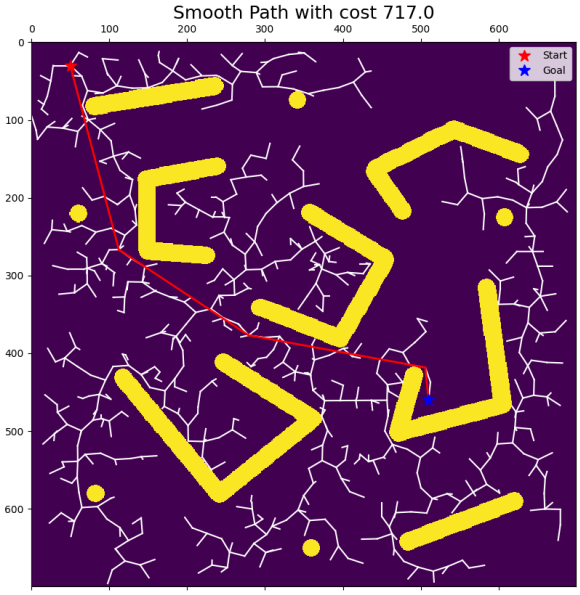
[7] https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree

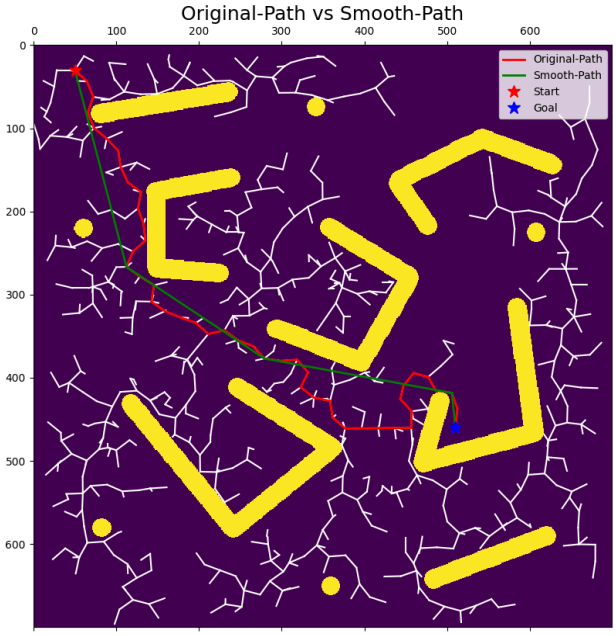
Appendix

1 RRT

1.1 Map4: max_iter = 2000 dq = 20 p = 0.2 start = (30,50) goal= (460,510)

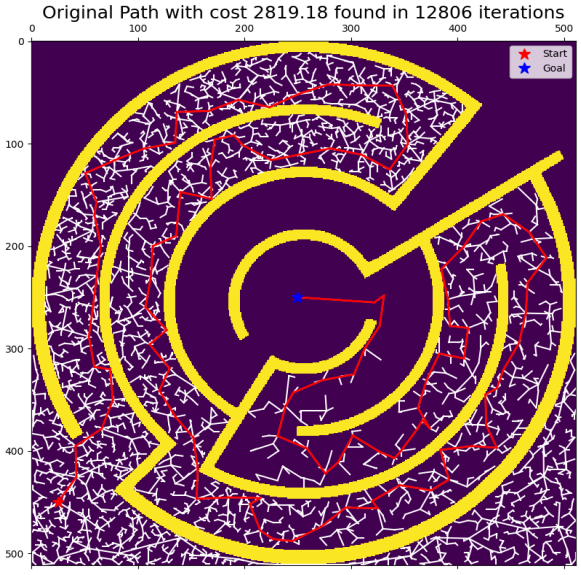


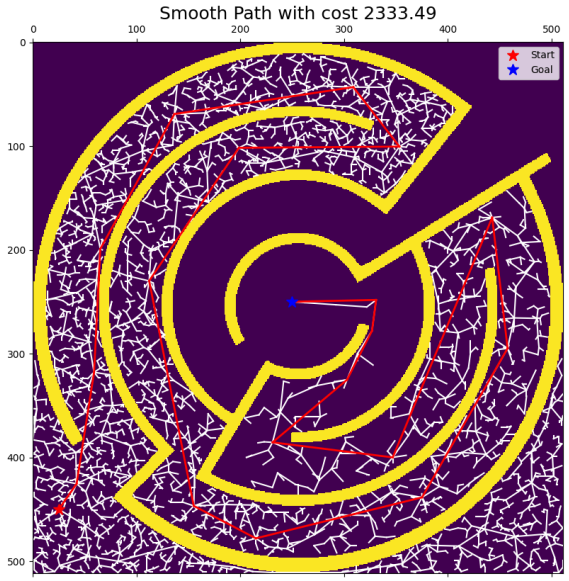


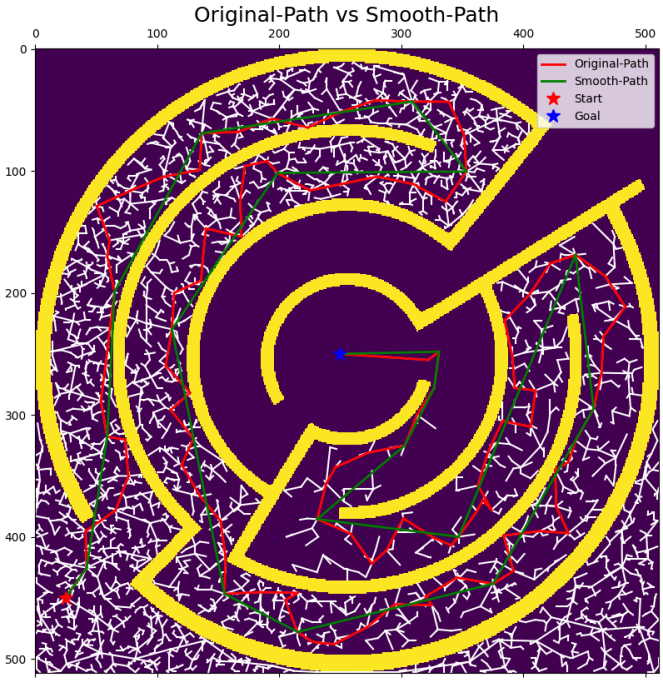


Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124

1.2. Map5: max_iter = 15000 dq = 30 p = 0.3 start = (450,25) goal= (250,250)

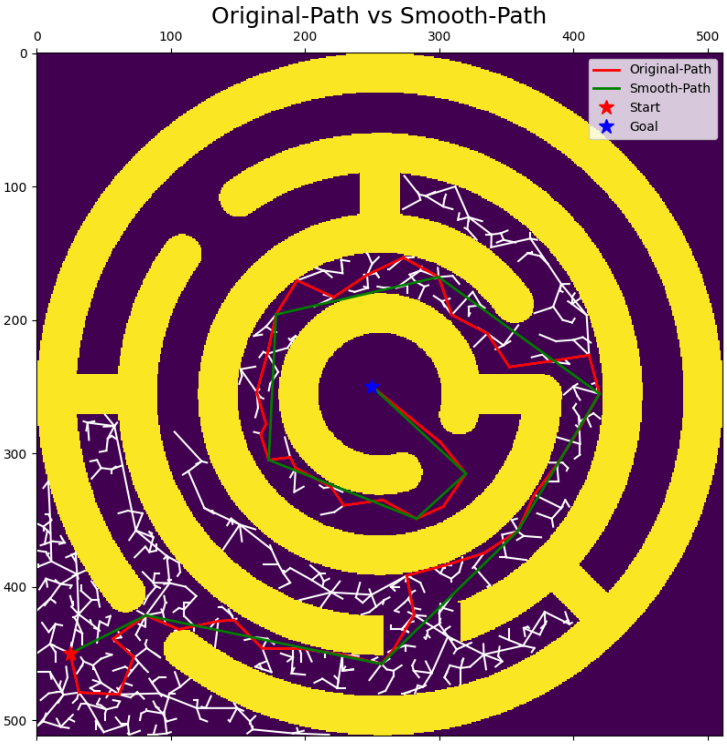






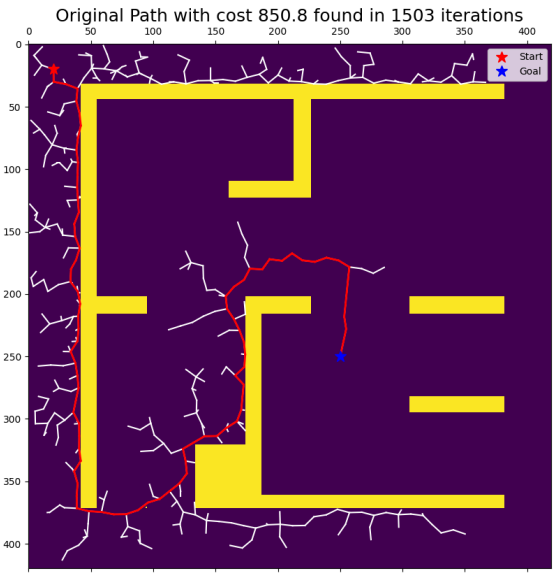
1.3. Map6: max_iter = 5000 dq = 30 p = 0.2 start = (450,25) goal= (250,250)

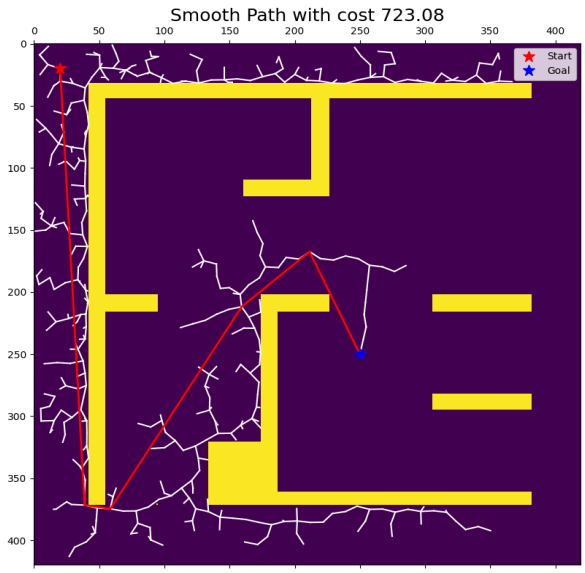
Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124

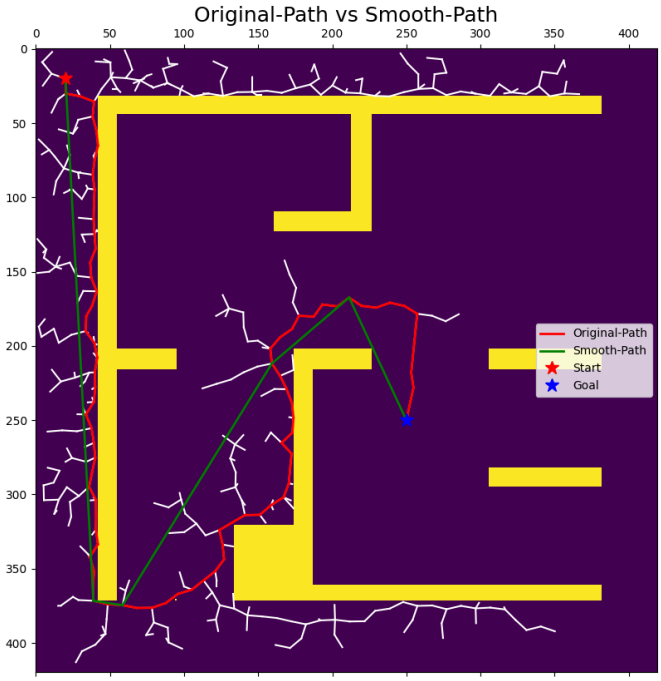


1.4. Map7: **max_iter** = 3000 **dq** = 10 **p** = 0.2 **start** = (20,20) **goal**= (250,250)

Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124



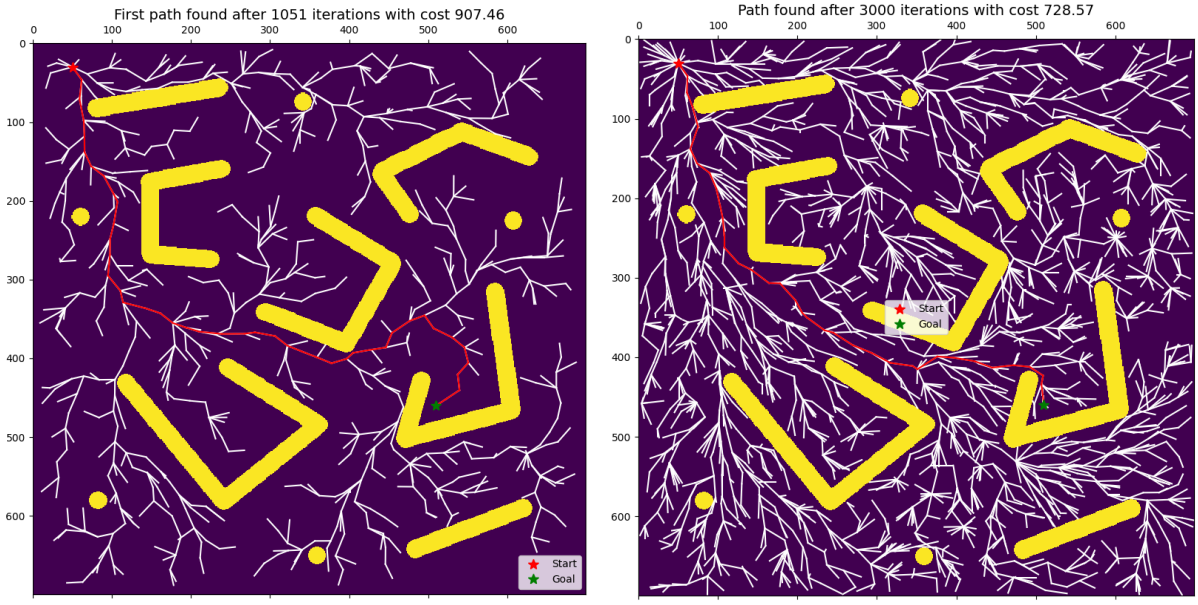




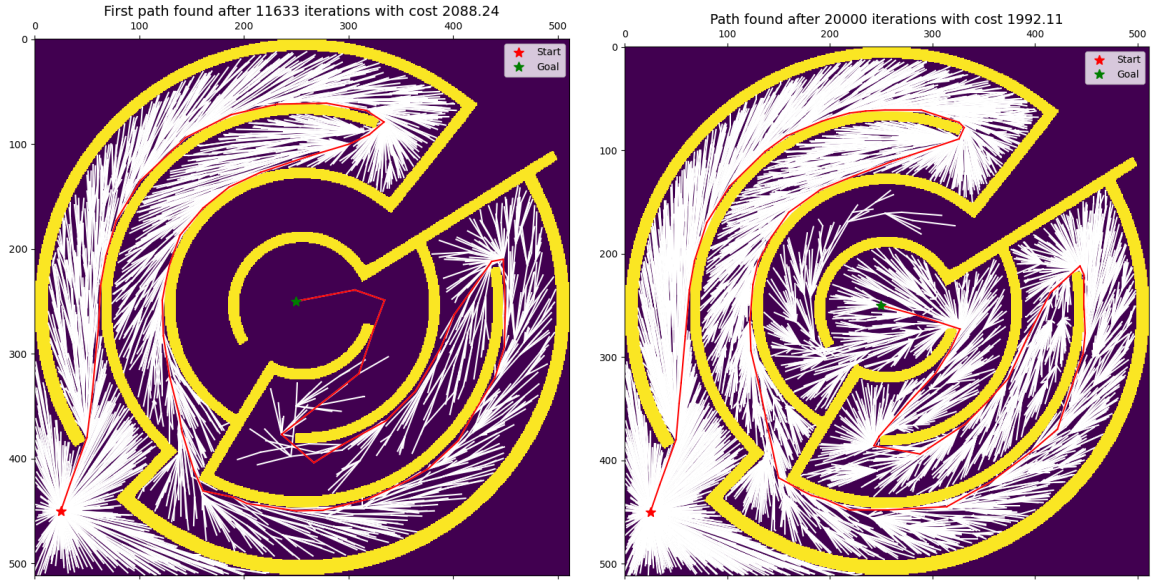
2. RRT*

2.1 Map4: **max_iter** = 3000, **dq** = 20, **p** = 0.2, **max_search_radius** = 30, **start**=(30,50), **goal**= (460,510)

Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124

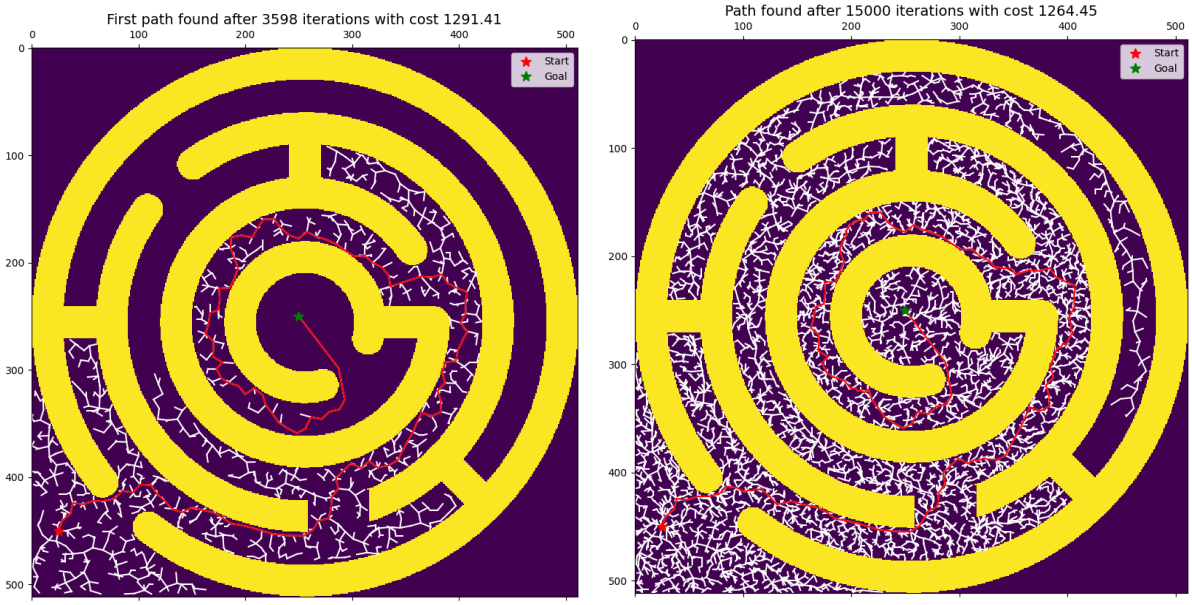


2.2. Map5: **max_iter** = 20000, **dq** = 30, **p** = 0.3, **max_search_radius** = 50, **start**=(450,25), **goal**= (250,250)



2.3. Map6: **max_iter** = 15000, **dq** = 10, **p** = 0.2, **max_search_radius** = 5, **start**=(450,25), **goal**= (250,250)

Lab 4 - Sampling Based algorithms - Mazen Elgabalawy u1999109, Solomon Chibuzo Nwafor u1999124



2.4. Map7: **max_iter** = 3000, **dq** = 10, **p** = 0.2, **max_search_radius** = 5, **start**=(20,20), **goal**= (250,250)