ECE 276B Project 2
Motion Planning

Shiladitya Biswas
Dept. of Electrical and Computer Engineering
Universrity of California,San Diego
California, USA

I. Introduction

Motion or Path planning plays a major role in any robotic setup. For example in a house hold setup the robot should be able to navigate from one location to another to perform several household duties without colliding with any obstacles. Hence, for a given environment (i.e. creating a Map of the environment by using techniques discussed in ECE276A) the robot's software should be able to build or plan a path to go from one location to another. In motion planning our main objective is to find a feasible and minimum cost path from the starting point to the goal point. The cost here can be parameterized as time, distance or a combination of both. In this project, we are given a 3D environment (with obstacles in it) along the robots starting position and the goal position to reach to. We are asked to find the shortest path from the start position to the goal position. In robotics this can be formulated as a **Deterministic Shortest Path (DSP)** and there are primarily two ways in which this can be solved, (1) Search-Based Planning and (2) Sample based Planning. For this project, I implemented Weighted A-Star algorithm which is a Search-Based and motion Planning. Furthermore, Open Source Motion Planning library, OMPL was used to implement a Rapidly exploring Random Tree-Star (RRT-Star) algorithm i.e. a Sample Based Motion Planning. We are asked to compare the performance of both the planners for a give environment.

II. Problem Formulation

The motion planning problem can be formulated as follows. The complete environment can be thought of as a graph with infinite vertex space V and weighted edge space:

$$C := \{(i, j, c_{ij}) \in V \times V \times R \cup \{\infty\}\} \tag{1}$$

Here c_{ij} denotes the arc length or cost to go from vertex i to vertex j and each elements of V represents a location in the environment. Now, our objective is to find the shortest path from the starting location S to the goal location G, where $S, G \in V$. This problem is called the **Deterministic Shortest Path (DSP)** problem.
The DSP problem has the following terminologies:

I would like to thank Saurabh Mirani and Ayon Biswas for the helpful discussions on the project.

1) **Path:** A Path is defined as a sequence $i_{1:q} := (i_1, i_2....., i_q)$ of nodes $i_k \in V$.
2) **All Paths from S to G:**

$$I_{S:G} := (i_{1,q} | i_k \in V, i_1 = S, i_q = G) \tag{2}$$

3) **Path Length:** Sum of the arc lengths over a path $i_{1:q}$. Given as

$$J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}} \tag{3}$$

Mathematically speaking, the objective of a Motion Planning algorithm is to find a path,

$$i^*_{1:q} = \underset{i_{1:q} \in \mathbf{J}^{i_{S:G}}}{\arg\min}\ J^{i_{1:q}} \tag{4}$$

For this project, the $c_{i,j}$ function is the physical distance between the i^{th} and j^{th} vertex i.e. the physical distance between the i^{th} and j^{th} location points in 3D Space. Thus, the cost function is given by the ℓ_2 norm of the vector joining points i and j in 3D space.

$$c_{ij} = ||P_i - P_j||_2 \tag{5}$$

Where, $P_i = [x, y, z]$, and x, y, z are the X,Y and Z coordinate of point i.

III. Technical Approach

In this section, I have discussed my technical approach that I used in order to successfully implement the A* algorithm. I have also discussed the collision detection algorithm that I used to implement the RRT* Algorithm.

A. Environment Dicretization

In order to apply any search motion planner algorithm to the problem, we need to first construct a graph to properly define the environment. The empty spaces of the environment are valid states, while the spaces that have obstacles in them are invalid states. So, as per the given environment data I discretized the environment to form a 3D grid. Where the empty spaces have zero value and obstacle regions have infinite value. Added to this the ground plane, side walls and roof cells were also initialized with a infinite value. Likewise, the Start and goal positions of the robot were also dicretized / transformed to the grid world coordinates. This was done using the "numpy.ceil" function.

As we increase the resolution of the grid, we focus on a tiny volume (a set of voxels, 26 in number, to be exact) of the grid world at a given instant of time, this along with the A* algorithm enables us to automatically avoid obstacles in the grid. But if the grid resolution is kept high, s.t. a voxel's side length is more than the thickness of a wall, then there is high chance of path collision with the wall. On the other hand, high resolution implies more number of states, as a result the planning algorithm (esp. A* algorithm) becomes more computationally intensive. Hence grid resolution is an important parameter in this context.

B. Weighted A Algorithm*

In order to implement the weighted A* algorithm I first initialized a cost grid (say CG) i.e. a 3D matrix of the same size as the environment grid (built as discussed in the previous section, lets call it EG). Only the start cell location of the cost grid was initialized to zero (i.e. CG[Start]=0, CG[All other cells]=infinity). I defined two Priority Queue(PQ) dictionaries namely, OPEN and CLOSED. These PQ dictionaries store the values in the ascending order, s.t. upon poping an item from OPEN or CLOSED we get the item with the smallest value. I also defined a separate dictionary called PARENT, this was used to store the parent coordinates of a cell, as the algorithm proceeds. The OPEN list was initialized with the start location and its corresponding cost from the cost grid i.e. 0 (CG[Start]).

The weighted A* algorithm can be thought of as an informed A* algorithm. In the weighted A* algorithm, for a given child cell(j) we look at two things. Firstly, we calculate the cost of going from the parent cell(i) (c_{ij}) to the child cell(j) and secondly we calculate a heuristic function h_i defined as:

$$h_i = ||P_i - P_{goal}||_2 \tag{6}$$

where, P_i and P_j are the location of the child and goal node. When this heuristic function is multiplied by a constant ϵ, where $\epsilon >= 1$, it is called $\epsilon Heuristic$ A* algorithm.

After setting up and defining all the parameters and dictionaries as in the previous paragraph, the pseudo-code of the weighted A* algorithm can be summarized as in fig 1

1) Implementation:

1) For a given parent node "i", I first look into its 26 immediately neighbouring cells. These neighbouring cells are the children of "j"
2) Then I check if these children are valid states or not, i.e. they lie with in the bounds of the environment grid and they don't lie inside an obstacle/block. This is done by checking:

$$\begin{gathered}\textbf{IF} \\ 0 < j_x < \text{X dimension of EG} \\ 0 < j_y < \text{Y dimension of EG} \\ 0 < j_z < \text{Z dimension of EG} \\ \textbf{AND} \\ \text{EG[j]=0}\end{gathered} \tag{7}$$

Where, EG[j] is the value of the j^{th} point location in the environment grid. If j^{th} location is empty space the EG[j]=0, else if j^{th} location is an obstacle EG[j]= infinity. This way I filtered out the valid child cells/voxels. This also helped me to successfully avoid obstacles (given that I used a high grid resolution).

3) After getting the valid children cells (corresponding to parent cell "i"), I look at there cost value in the cost grid and update it (and also the parent of "j") as follows. For a given child cell "j" of parent "i".

$$\begin{gathered}\textbf{IF} \\ (CG[j] > CG[i] + c_{ij}) : \\ \textbf{THEN} \\ CG[j] = CG[i] + c_{ij} \\ PARENT[j] = i\end{gathered} \tag{8}$$

4) Once this is done I check if "j" is present in the OPEN dictionary. If "j" is in present in OPEN then I updated the priority of "j" as follows:

$$\begin{gathered}\textbf{IF } (\text{"}j\text{"} \in OPEN) : \textbf{THEN} \\ \text{Update value of "j" } = CG[j] + \epsilon h_j (\epsilon >= 1) \\ \textbf{ELSE} \\ \text{Add node "j" to OPEN list with value=} \\ CG[j] + \epsilon h_j (\epsilon >= 1)\end{gathered} \tag{9}$$

5) Steps 1 to 4 were repeated for all the children of i.
6) Next we remove the item "k" with the smallest $CG[k] + \in h_k$ value from the OPEN list. We insert this new node "k" into our CLOSED Dictionary. This new node becomes our new parent and we repeat all the steps from 1 to 5 with this new parent "k".
7) Steps 1 to 6 are repeated until we find the goal node (G) in our CLOSED Dictionary.
8) Once the whole loop from step 1-7 are complete, we back trace the PARENT dictionary to get the least cost path to go from the starting node to the goal node.

```
1: OPEN ← {s}, CLOSED ← {}, ε ≥ 1
2: g_s = 0, g_i = ∞ for all i ∈ V \ {s}
3: while τ ∉ CLOSED do
4:     Remove i with smallest f_i := g_i + ε h_i from OPEN
5:     Insert i into CLOSED
6:     for j ∈ Children(i) and j ∉ CLOSED do
7:         if g_j > (g_i + c_ij) then
8:             g_j ← (g_i + c_ij)
9:             Parent(j) ← i
10:            if j ∈ OPEN then
11:                Update priority of j
12:            else
13:                OPEN ← OPEN ∪{j}
```

Fig. 1. Weighted A* Algorithm with $\epsilon Heuristic$

C. Rapidly exploring Random Tree star(RRT)*

From the numerous sampling based search algorithms present in OMPL library I chose to use the Rapidly exploring

Random tree Star (RRT*) algorithm with an objective function of reducing the path total path length from Start to Goal. A sampling based planning algorithm plans by generating a sparse sample-based graph of the free configuration space of the robot (Here the empty spaces in the environment). Rapidly exploring Random Tree (RRT) algorithm is a type of sample based search algorithm in which a tree is constructed from a random sample of the valid states with the starting point as the root of the tree. The tree grows until it contains the goal position state. The RRT algorithm is well suited for single-shot planning between a single pair of start and goal node.

The RRT* algorithm is an extension of the RRT algorithm in which we go one step further in rewiring the tree to ensure asymptotic optimally of the path. Briefly speaking, RRT* records the distance each vertex has traveled relative to its parent vertex (i.e. the cost of the vertex). After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper cost than the proximal node is found, the cheaper node replaces the proximal node. Secondly, after a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. Hence the path becomes a lot more smoother.

D. Collision Detection

The collision detection algorithm is very plane and simple. Let S_1 and S_2 be two states (here points in 3D space) we have to check if the line segment connecting S_1 and S_2 is collision free or not. If we have the collision then the straight path motion between S_1 and S_2 is invalid else the motion is valid and we can expand the tree.

To do this I first created axis aligned Bounding box for obstacles in the environment, using its boundary coordinates i.e. $(x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$ using the aabb.create_from_bounds function from Pyrr library . Next, from the two given states (between whom the motion validity is to be checked), we build a ray. This is done using the ray.create_from_line from the Pyrr library. Next I used the ray_intersect_aabb(ray,aabb) function from the Pyrr library to check for intersection of the ray and the AABB box. If we have get an intersection point, then the motion is invalid else the motion is valid.

It is observed that in general the RRT* algorithm visits lesser states as compared to the A* Algorithm.

IV. Results and Discussions

The output of the both the planners for all the 7 environments/maps are shown in figs. 2,3,4,5,6 and 7 and 8. Table I shows the performance comparison of both Search-based planner (A* Algorithm) and Sample Based Planner(RRT* Algorithm). For the A* algorithm ϵ value was taken to be 10 and grid resolution that worked perfectly for all the maps was 0.1.

TABLE I
Performance comparison of A* and RRT* algorithm

Map	RRT* Algorithm		A* Algorithm	
Path Length	Nodes visited	Plan time(S)	Nodes visited	Plan time(S)
Maze	4854	761.897	11012	763.89
Path Length	76.132		80	
Single Cube	12	1.011	32	1.0001
Path Length	7.883		8	
Flappy Bird	188	1.002	10098	25.376
Path Length	30.668		32	
Window	22	1.001	2467	0.466
Path Length	24.43		27	
Tower	172	2.01	6343	7.011
Path Length	32.377		27	
Room	67	10.01	2124	5.212
Path Length	10.83		13	
Monza	36560	1543.2	3527	155.93
Path Length	73.401		76	

It is observed that in general the RRT* algorithm visits much lesser nodes as compared to the A* algorithm. Except for cases where the environment is very bad. For example, in the Monza map the robot has to travel from one end of the room to another and pass through extreme ends of the walls. since the RRT* is a sample based algorithm, it has less sense of direction as compared to the weighted A* algorithm. In the Monza Map the A* algorithm works faster since it has a heuristic function in it. This heuristic function provides a sense of direction (the algorithm quickly gets to know how far a child node is from the goal, the node that are nearer to the goal position are expanded more as compared to the node far from the goal, hence for really complicated environment A* algorithm should perform well). On the other hand the RRT* algorthm is more computationally feasible and is hence suitable for mostly stochastic maps, where the goal is immobile As can be seen in fig 8 the A* path is mostly straight as compared to the RRT* generated path below. In such an environment, the RRT* algorithm has to sample a lot and then accordingly find the path. This is evident from table I, the A* algo visits roughly 10 times lesser nodes as compared to RRT* algo.

Other than that, in most of the environment the RRT* algorithm performs much better and faster in comparison to the A* algorithm. The RRT* algorithm visits lesser nodes as compared to A*. The length of the shortest path we get are almost same for both the algorithms. It is also noted that the RRT* path generated is very random in nature, while the A* algo shortest path always tries to stay in a plane.

A. Effect of ϵ on A algorithm performance*

Keeping the grid resolution and map same. I varied the value of ϵ from 1 to 90. The behaviour is depicted in graphical form. It is observed that all the 3 quantities i.e. Total Planning time, Shortest path length and number of visited node converges to a constant values as ϵ value is increased. This is in fact true. The more the value of ϵ the more we believe the heuristic function, as a result we visit lesser number of nodes, which in turn reduces the planning time of the algorithm. Although

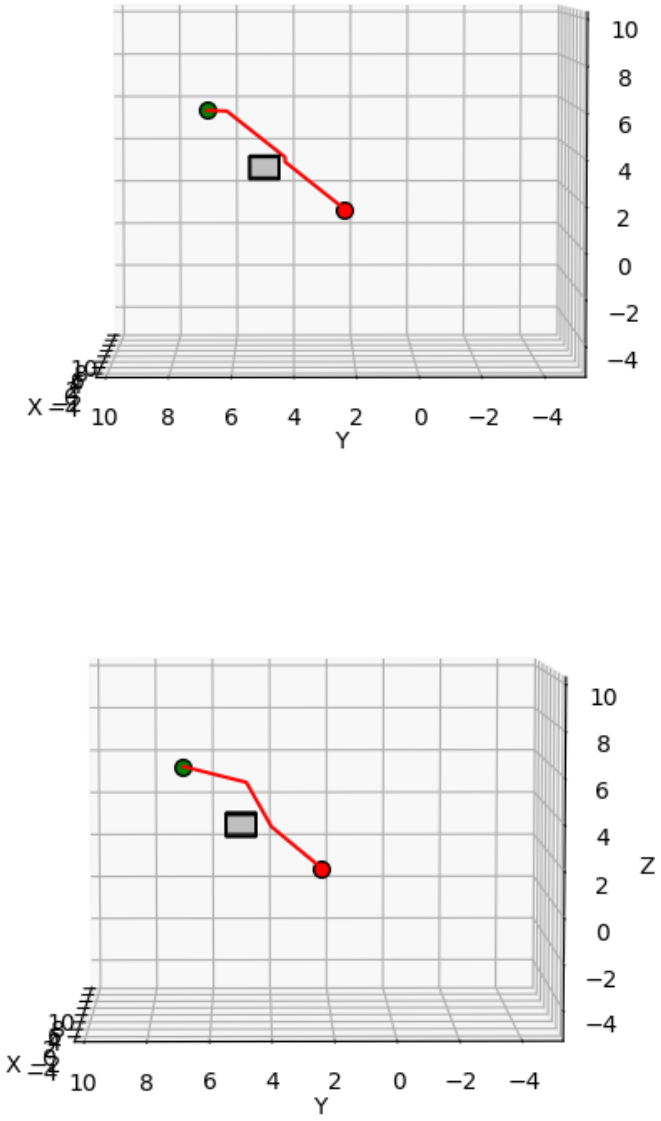


Fig. 2. Single Cube map (Top: A* o/p, Bottom: RRT* o/p)

there is a variation in the length of the shortest path, but still its variation is not that drastic. The slight variation arises due to the fact that I have dicreetized the map.

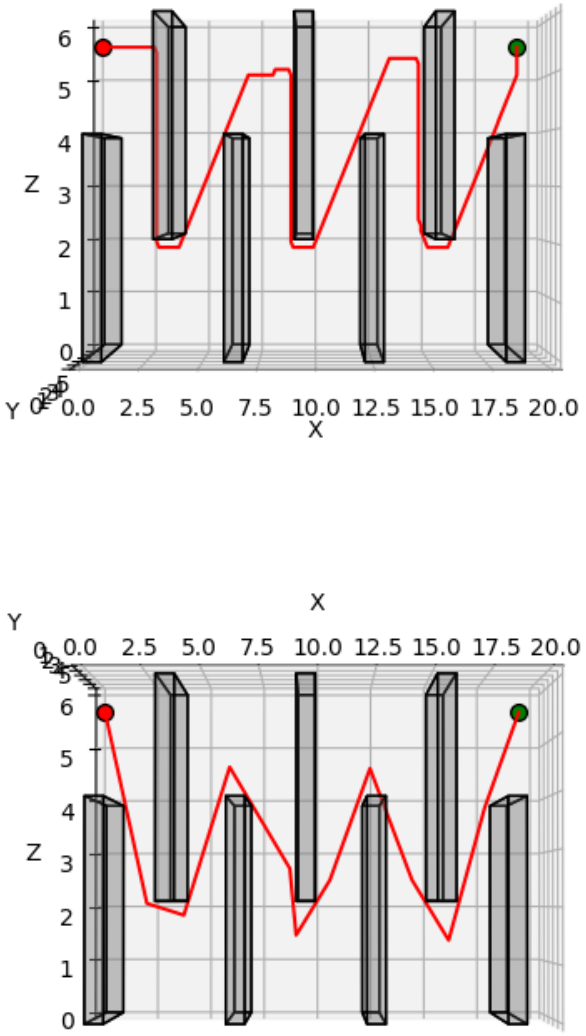
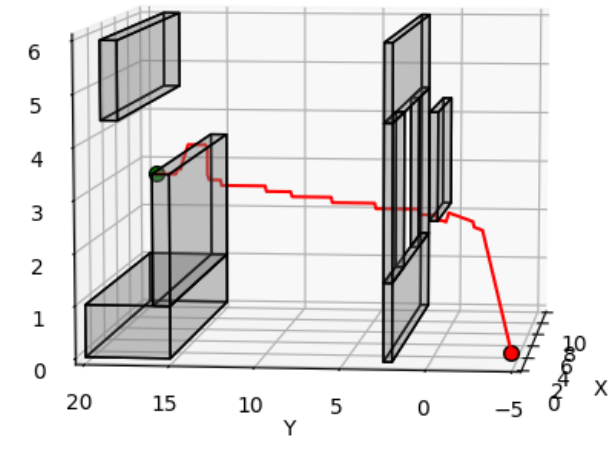


Fig. 3. Flappy_Bird map (Top: A* o/p, Bottom: RRT* o/p)



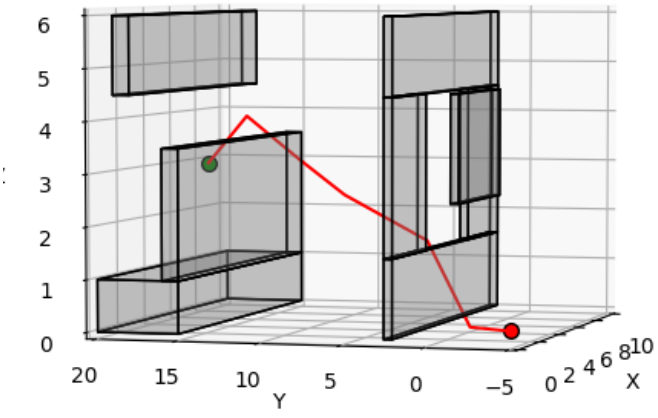
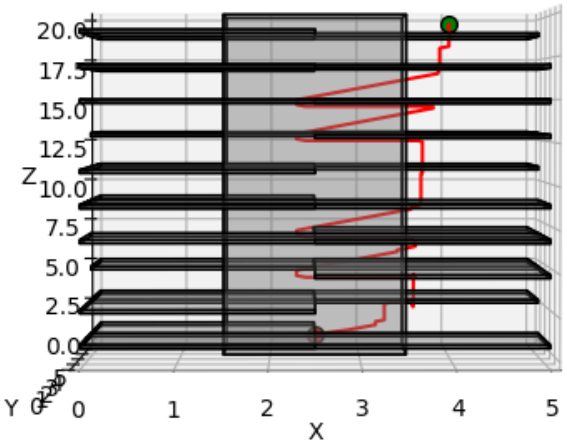


Fig. 4. Window map (Top: A* o/p, Bottom: RRT* o/p)



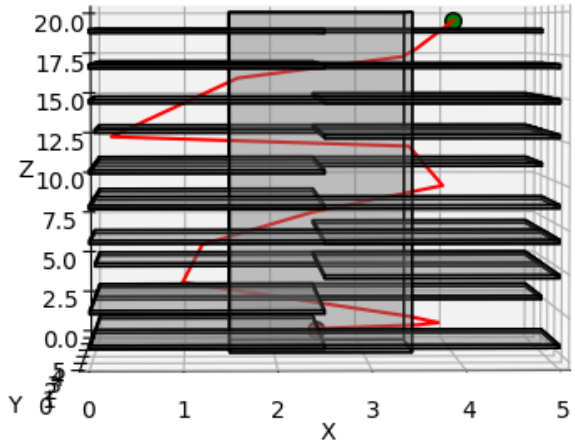
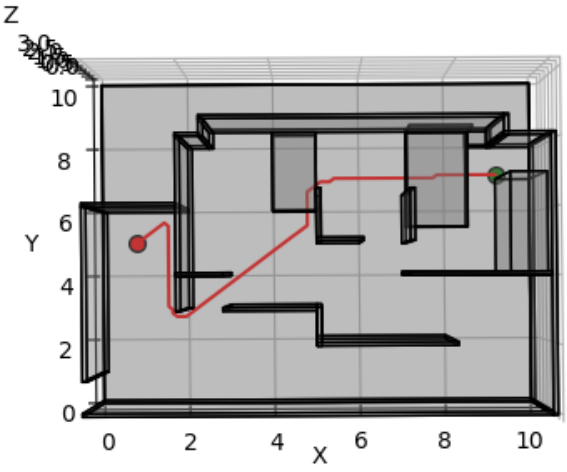


Fig. 5. Tower map (Top: A* o/p, Bottom: RRT* o/p)



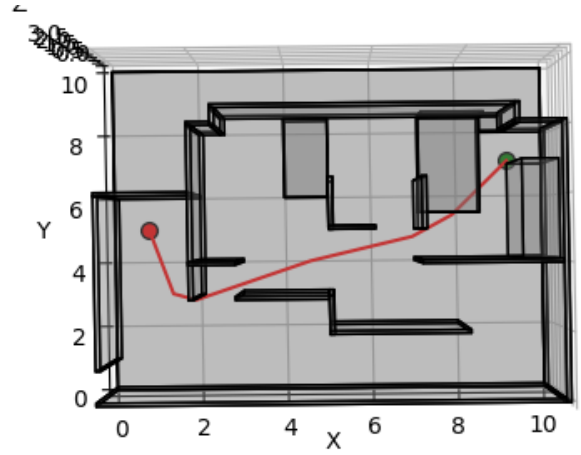
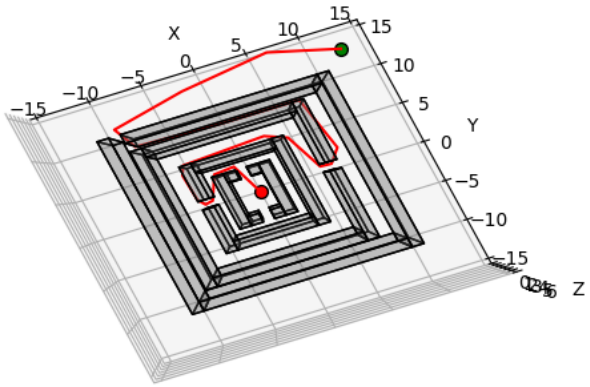


Fig. 6. Room map (Top: A* o/p, Bottom: RRT* o/p)



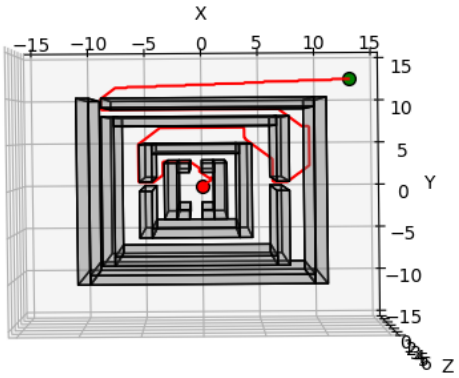


Fig. 7. Maze map (Top: A* o/p, Bottom: RRT* o/p)

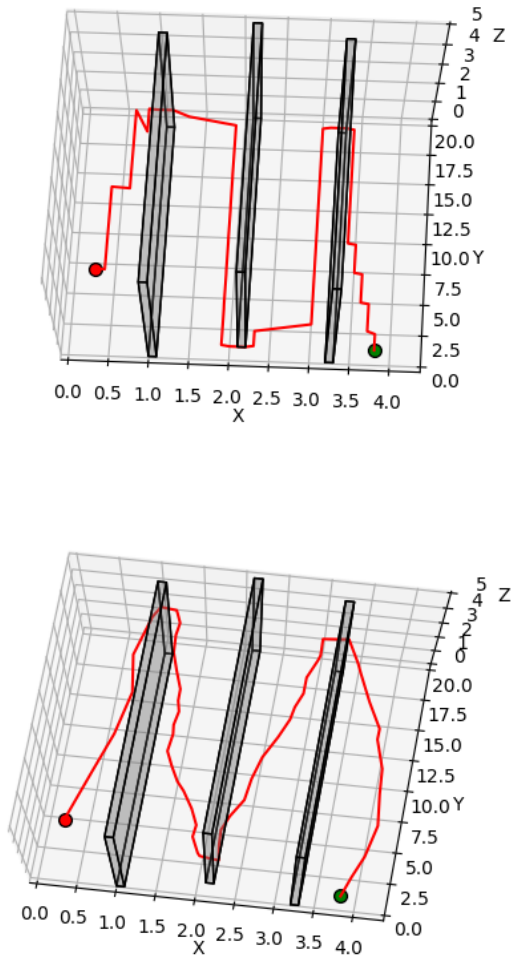


Fig. 8. Monza map (Top: A* o/p, Bottom: RRT* o/p)

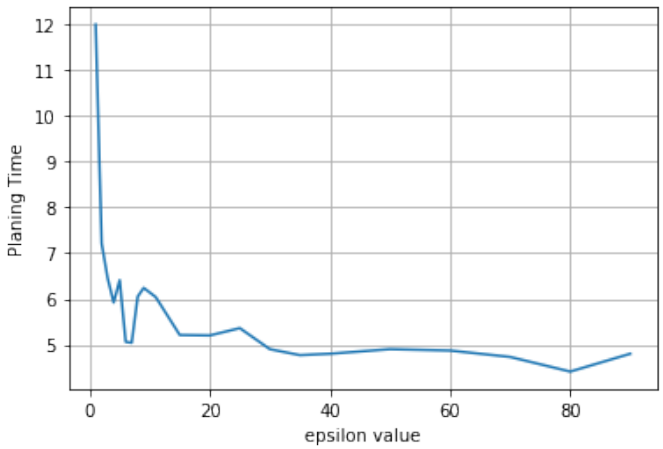


Fig. 9. Planning Time variation w.r.t ϵ value

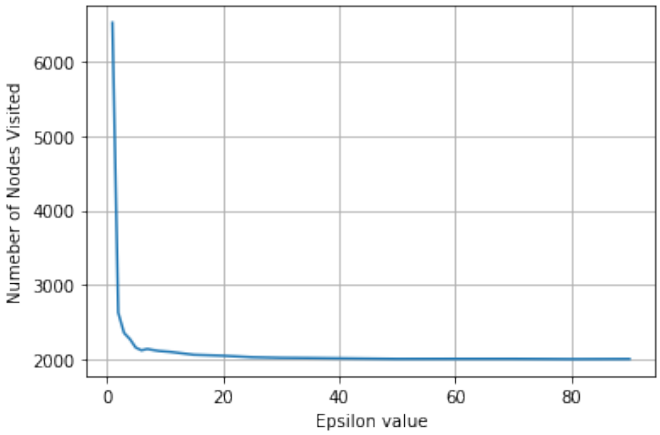


Fig. 10. No of nodes visited variation w.r.t ϵ value

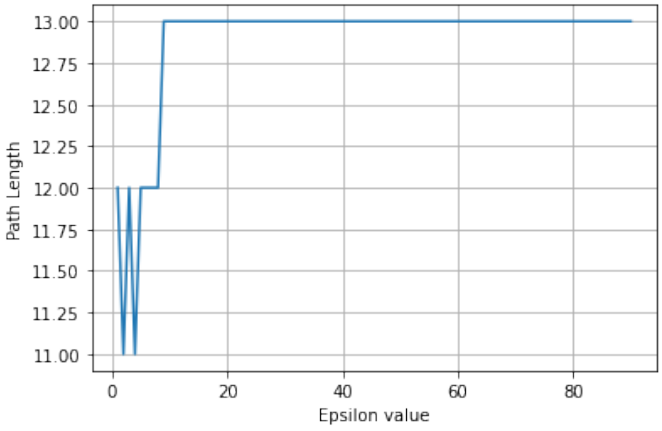


Fig. 11. Shortest path length w.r.t ϵ value