Robot Motion Planning

Surya Prakash Thoguluva Kumaran Babu
Department of Mechanical and Aerospace Engineering
University of California San Diego
stk222@ucsd.edu

***Abstract*—The objective of this paper is to implement search-based and sampling-based motion planning algorithms to solve the optimal path in a given environment. Weighted A* and RRT algorithm were explored for this project.**

***Index Terms*—A*, RRT, Motion Planning.**

I. Introduction

Robot planning is an important area of research in the robotics domain. Given sensor data about the environment, the robot needs to know what it should do next to achieve a specific task. This is the exact objective of which planning and learning problems in robotics. Planning in robotics enables robots to operate in complex environments, handle uncertain situations, and accomplish tasks with a high degree of accuracy and precision. A subset of this is the robot path planning problem which can be solved in various ways. There has been a lot of research over the past few decades which tried to solve this problem. However, even today, although the high-level problem formulation is the same, we have different challenges. For example, the current auto-pilot control in some cars is still not 100% reliable and there is a lot of room for improvement.

In this paper, we have discussed the implementation of weighted A* search-based planning and RRT sampling-based planning for a 3D landscape and compared its performances and optimality.

II. Problem Formulation For Motion Planning

A. Given Map Description

We are given 7 sets of maps. The maps are described below:

- All the blocks and the boundaries are given as axis-aligned bounding rectangles.
- Each rectangle is described by a 9-dimensional vector, specifying its lower left corner $x_{min}, y_{min}, z_{min}$ and, its upper right corner $x_{max}, y_{max}, z_{max}$, and its RGB color (for visualization).
- A start point $x_s \in \mathbb{R}^3$ and goal point $x_\tau \in \mathbb{R}^3$ is given.

The objective is to find a path between the start node and to goal node without colliding with any obstacles.

B. Deterministic Shortest Path Problem

The problem of finding the optimal path can be treated as a deterministic short path problem in 3D space. The formulation is as follows:

- Given a graph with vertex set υ , edge set $\varepsilon \subseteq \upsilon \times \upsilon$ and the edge weights $C := \{c_{ij} \in \mathbb{R} \,|\, (i,j) \in \varepsilon\}$ where c_{ij} is the cost of going from vertex i to j or vertex j to i, find the shortest path from a start node s to end node τ.
- The edge cost in our case is the Euclidean distance between the two vertices.
- A path is defined as a sequence $i_{1:q} := (i_1, i_2, ...i_q)$ of nodes $i_k \in \upsilon$
- Two nodes are connected based on the motion model.
- Path length is the sum of edge weights along the path.

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

- All paths from $s \in \upsilon$ to $\tau \in \upsilon$ is defined as

$$P_{s,\tau} := \{i_{1:q} \,|\, i_k \in \upsilon, i_1 = s, i_q = \tau\}$$

- Objective for DSP is to find the minimum length from node s to node τ and the path:

$$dist(s,\tau) = \min_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}}$$

$$i^*_{1:q} = arg \min_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}}$$

- Value Function $V_t^F(x)$: It is the cumulative edge cost of starting at node s at time 0 and reaching node x at time t.
- Assumption is that there are no negative cycles in the graph. That is $J^{i_{1:q}} \geq 0$ for all $i_{1:q} \in P_{i,j}$ for all $i \in \upsilon$

The DSP problem can be solved by using label-correcting algorithms like A* or sampling based algorithms like RRT.

C. Label Correcting Algorithm Formulation

The DSP problem can be adapted to our problem by defining the vertices and edge costs accordingly.

- The set of all points within the bounding box is defined as $x, y, z \in \mathbb{R}^3$ which form the set of all vertices υ
- A edge ε is defined between a parent node $i \in \upsilon$ and child node $j \in \upsilon$. For every parent i there are 26 children node j which represents all the corners of a cube whose center is at node i.
- Stage cost c_{ij} is defined as the euclidean distance between node i and node j.
- Motion model: Each node j can only be reached by its parent node i with a stage cost c_{ij} if the node j is not within the obstacles.

- The value function is the label assigned to each node i is g_i. It is the estimate of the optimal cost to arrive from s to node i.

With the above modification, if given a finite state deterministic shortest path problem, the label correcting algorithm terminates with $g_i = dist(s, \tau)$. If there exists no shortest path from s to τ the algorithm terminates with $g_i = \infty$.

D. RRT Problem Formulation

The DSP problem can also be solved using a sampling-based approach. The main difference between search based and sampling based algorithms is graph construction. In search based the graph is constructed incrementally and all possible nodes are explored whereas in the sampling- approach, we construct the graph by sampling the free space. Typically a sampling based approach can be defined if we have the below functions implemented:

- Sample Free Space: Returns a sample node from free space
- Nearest: given a graph, return the node closest to the random node that is generated
- Near: Given a random node, return all the existing nodes in the graph.
- Collision Free: Given 2 points, check if the line segment joining them is collision-free
- Steer: Given points $x, y \in v$ and $\varepsilon > 0$, returns a point $z \in v$ that minimized $\|z - y\|$ while remaining within the ε distance from node x. This encodes the motion model.

$$Steer(x, y) := arg \min_{z:\|z-x\|\leq\varepsilon} \|z - y\|$$

Given the functions, sample to generate the graph and search the nodes for finding the shortest path.

III. Technical Approach For Search-Based Planning

The implementation of search-based planning using label-correcting algorithm like A* is mentioned below :

- Construct a graph with nodes for the search space
- Algorithm for collision detecting
- Selecting appropriate data structure for implementing the algorithm
- Checking optimality and improving time complexity

A. Graph Construction

Constructing the graph is one of the primary steps if we were to solve for a deterministic shortest path problem. The given space is a continuous 3D space that needs to be discretized to have finite states. In this project, the graph is constructed as the label-correcting algorithm is implemented. The graph nodes which we used in this project are as defined below:

- The entire 3D space is discretized into small cubes each of a desired resolution. The performance of the algorithm is checked for various resolutions.
- All the corners of the cube originating from node i are potential children nodes j of the parent.
- A node is inserted into the graph only if the collision detection function determines that the node is collision-free. If not, then that node will not be considered now or in the future.
- The edge cost from node i to node j is the euclidean distance between them.

B. Collision Detection

Since all the blocks in the environment are rectangular in nature, the collision checking basically can be reduced to see if the line segment connecting node i to node j intersects the planes of the 6 faces or not. The vector representation of the line in 3D is $X = B + t * D$, where B is a Tuple (x,y,z) of the first point and D is the direction of the line given by dx, dy, dz. If P1 has coordinates x_1, y_1, z_1 and P2 is another point with coordinates x_2, y_2, z_2, then D = P2 - P1 and B = P1. The parametric equation of a plane in 3D is given by $x + y + z = c$ where c is some constant. For axis-aligned bounding boxes, each face will only have either x or y, or z to be some constant. Substituting the coordinates of the line equations in the plane equation for each face and determining the parameter t, we could find the point of intersection of the line segment with the plane of the face. The algorithm is clearly defined below:

- Input: Node i, j coordinates, all obstacles bottom left and top right coordinates.
- Output: True if colliding else False
- Step 1: Iterate over all the obstacle blocks. For each blocks, do
- Step 2: Check if either node i or node j coordinates are inside the box. If yes return True else continue
- Step 3: Construct the line segment equation $X = B+t*D$ using node i coordinates as B and direction vector as defined above.
- Step 4: For all 6 faces of the box do
- Step 5: Find the parameter t by using the line and the plane equation of that face.
- If t is not between $0, 1$ then continue, as if t is beyond this range, then the line segment needs to be extended in order to intersect the plane. This is a case of no collision.
- If t is between $0, 1$ then find the other two coordinates using the determined parameter t.
- If these two coordinates like between the range of the corresponding two coordinates of the rectangle of that face, then it means that the line segment intersects that face. So return False, if not continue checking for the other faces and other blocks.

C. Data Structure for Label Correcting Algorithm

It is very important to choose appropriate data structures to efficiently execute the variants of label-correcting algorithms. The graph constructed is stored as a Python dictionary where the key is the coordinates of the node as it is always unique.

The value of a node is an object of a custom class that has the attributes mentioned below:

- Pqkey: The key of the node
- Coord: Coordinates of the node
- g Value: The cumulative edge cost required to come to this node from node s. This is initialized to ∞
- h Value: The heuristic of this node. In our case, the heuristic function is the Euclidean distance from this node to the goal node τ.
- Parent node: The key of the parent node is stored
- Parent action: The direction vector from the parent node to this node is stored.

The open list and closed list for A* algorithm are implemented by a priority queue data structure offered by the pqdict Python library. The key is the coordinates of the node and the value is the node object. The priority is assigned by a user-defined priority function which evaluates as the expression $g + \epsilon * h$ where g, h are obtained from the node object and ϵ is passed as an input to the A* algorithm.

D. *Label Correcting Algorithm*

The implemented A* class has many methods. The planning method is mentioned below.

- Input: Start node, Goal Node, Boundary box, Obstacle box
- Create the start node object and insert the key, and value pair into the graph and the open list.
- Loop through the openlist in the order of the priority of the elements in the queue until the openlist is empty
- Pop the element from the open list and insert it into closed list.
- Treating this element as the parent i, find all the children node j of this parent using the defined motion model and the direction vectors.
- For all the children node j
- If the node j falls with the boundary box, then continue with the loop and do nothing
- Else check for collision of a line segment of node i and node j using the defined collision function.
- If the collision is true, then continue with the loop without doing anything.
- Else this is a potential node. Check if this node is openlist by searching with the key.
- If the node is in open list and if $g_j > g_i + c_{ij}$ then update the priority of node j in the open list.
- Else if the node is in closed list, then continue the loop without doing anything.
- Else if all the above cases fail, then create the node object and set the label $g_j = g_i + c_{ij}$ and insert it into the open list.
- After all the children node of i is expanded, the termination condition is checked for early exit case.
- If the distance between the current node i and the goal node is below the resolution of the map then the entire loop terminates and comes out. The last node which met the termination condition is stored as the final node. This is done to retrieve the path.

Once the planning is executed, the getPath method is called. The pseudo-code is below:

- Initialise the start point to be the final node coordinates
- While the start point key exists in closed list do:
- Append the start point node's coordinate into a path list.
- Set the start point to be the parent of the start node and loop over
- Return path list.

IV. Technical Approach For Sampling Based Planning

In this project, we have explored the Rapidly Exploring Random Tree algorithm. Python motion planning project was used to implement RRT. The approach for RRT is given below:

- The input to the RRT functions are the boundary blocks, obstacle blocks, start point, and goal point.
- The hyperparameters like step length to construct the graph edges, number of maximum samples to run the algorithm, and length to check for collision.
- The RRT function was called with all the 9 maps and the results are computed and compared.

V. Discussion

A. *Effect of Varying Environment Resolution for A**

A few observations of using different environment resolution is mentioned

- As we increase the resolution, the time taken to compute the optimal path increases exponentially. This is due to the number of nodes growing in orders of 10 as we increase the resolution.
- As we increase the resolution, the path gets tighter and more optimal.
- As we increase the resolution, the termination condition takes the finish node much closer to the goal and hence we are nearer to the goal.

B. *Comparison of Different Environment*

Different environments had different path lengths and completion times. A brief summary of the observations is listed below:

- As we could see the maze map was the one which took the longest time. The reason could be because the heuristic was a bad underestimate of the distance from any node to the goal as there were many obstacles on the way.
- It is seen that in general, RRT has a high time with maps like Maze and Monza. It is due to the fact that we are using sampling-based approach the map has a lot of bug traps that need to be escaped.
- Cube was the fastest to converge in A*. This is because the heuristic accurately matched the actual estimate and hence at every step we proceeded in the right direction.

- Although the maze and Monza have the same path length, the computation time for maze is much larger than the monza. This is also due to the fact that the heuristic was more accurate to the actual distance and so the node expansion was not stuck in the local boundaries

C. *Path comparison between A* and RRT*

As can be seen from the figure, the A* produced optimal resolution complete for all the maps whereas RRT gives suboptimal path. This can be seen even in the single cube case.

D. *Time comparison between A* and RRT*

Regarding the time taken by A* to that of RRT, it is seen that RRT performs much better except monza map. This could be due to the bug traps that are present on the monza map.

E. *Effect of Probability to Connect in RRT*

Changing the probability to 0 made the program run into an infinite loop. This is because, until during the random draw the goal node appears, the algorithm will be running forever without termination. Different values were experimented with. If the p-value increases the time taken for complex maps like maze and monza also increases. This is due to the fact that now we more often check the goal node for connection. This also leads to more suboptimal paths in complex environments. The value p = 0.1 was found to be more optimal.

F. *Effect of Edge Length in RRT*

Setting different edge lengths for RRT had a significant impact on the results as mentioned below:

- Higher values of edge worked well and faster for simple maps like a cube.
- However higher values did not result in the termination of the RRT algorithm for complex maps like Maze and Monza. This is because longer edges collide more often on the obstacles than on shorter edges.
- Having shorter edges increased the computation time of the algorithm as now more nodes are inserted into the graph.

G. *Effect of Number of Sampling in RRT*

This is a critical parameter when the maps get complex. Having a higher number is better. For maps like maze and Monza, a value of 20000 was required to find an optimal path. This parameter also depends on other parameters. For example, if the resolution is very small then this number should be very large as it might take longer path to reach the goal and so we will need more samples.

VI. Results

The paths for all the maps for the A* are shown. The HTML file generated by RRT algorithm output is present in the code file in the folder RRT_output folder. A snapshot of the RRT implemented in the different environments is shown in the figures below. The table of comparison between the time taken for A* vs RRT for different maps is shown.

TABLE I
Results Comparison for A* and RRT

Map	Path A*	Time A*	Time RRT
Cube	7	0.14	0.07
Maze	79	62.2	60.26
Flappy bird	25	8.99	2.25
Monza	77	3.69	177.21
Window	26	22.52	1.42
Tower	32	22.86	9.48
Room	11	2.28	1.46

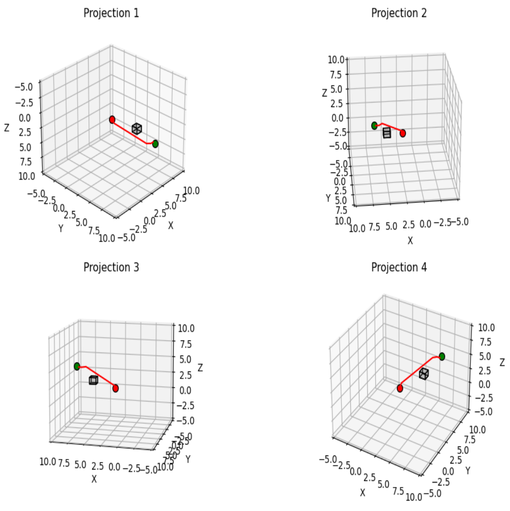

Fig. 1. Path in Cube Environment

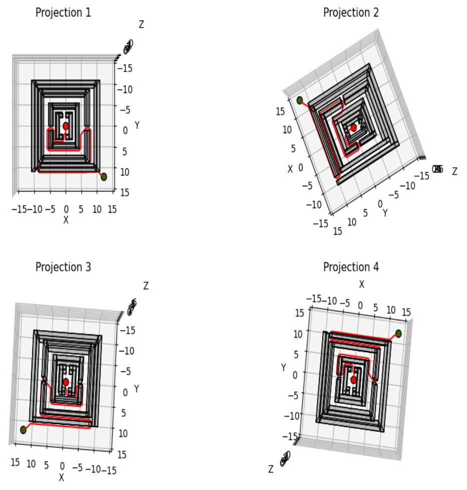

Fig. 2. Path in Maze Environment

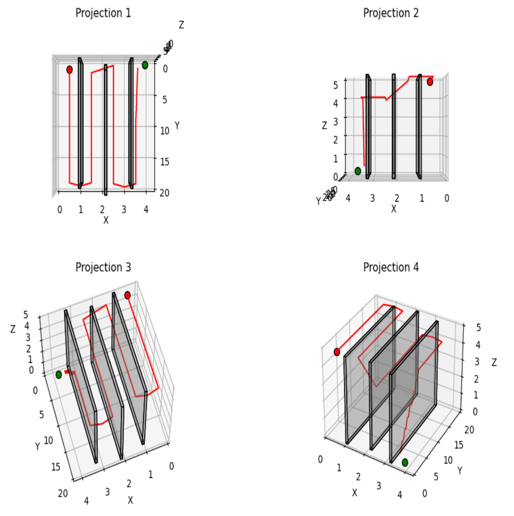

Fig. 3. Path in Monza Environment

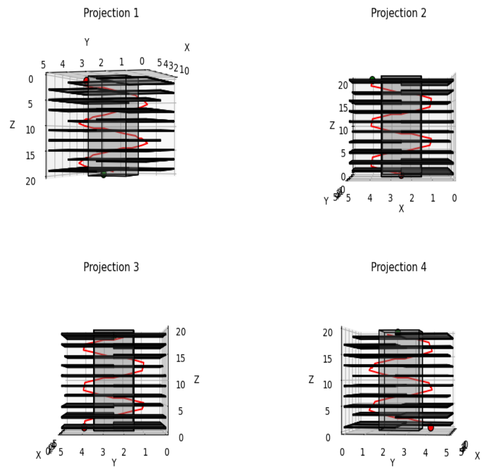

Fig. 5. Path in Tower Environment

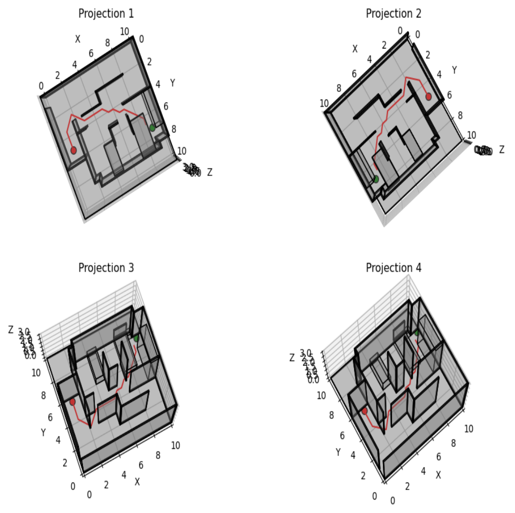

Fig. 4. Path in Room Environment

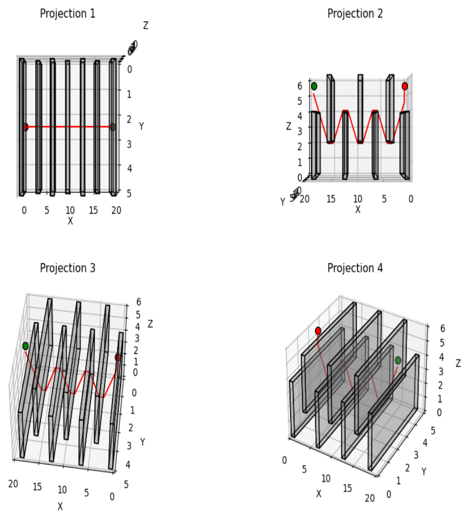

Fig. 6. Path in Flapp Bird Environment

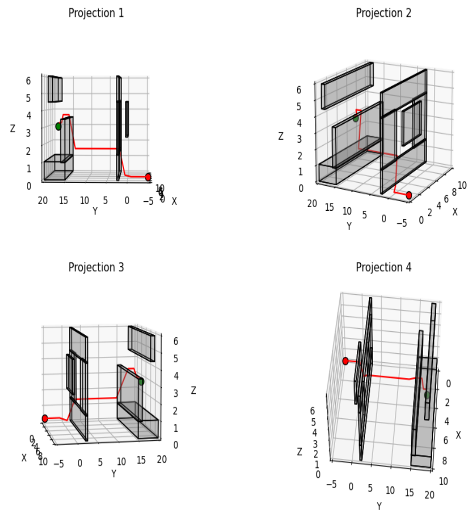

Fig. 7. Path in Window Environment

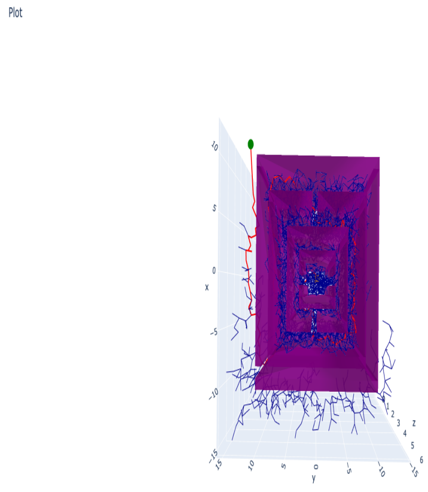

Fig. 9. RRT Path in Maze Environment

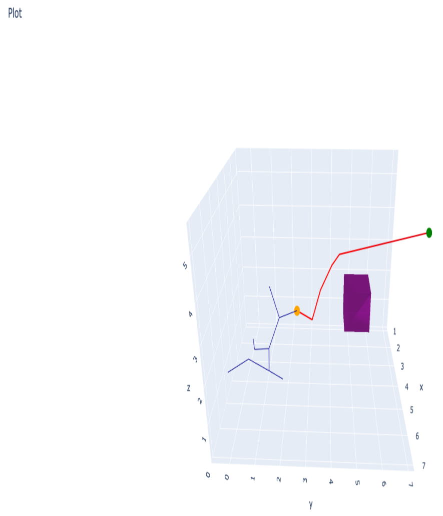

Fig. 8. RRT Path in Cube Environment

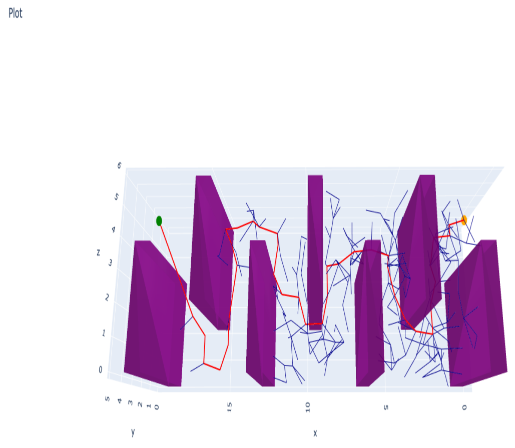

Fig. 10. RRT Path in Flappy bird Environment

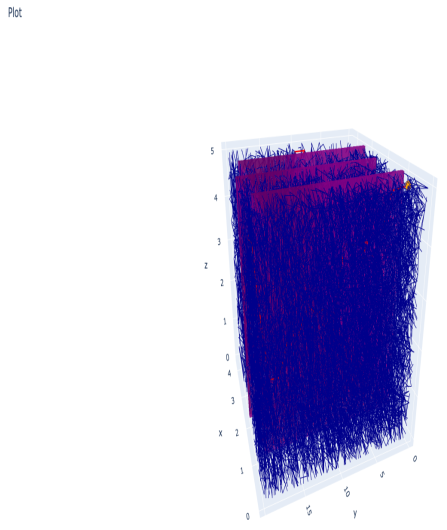

Fig. 11. RRT Path in Monza Environment

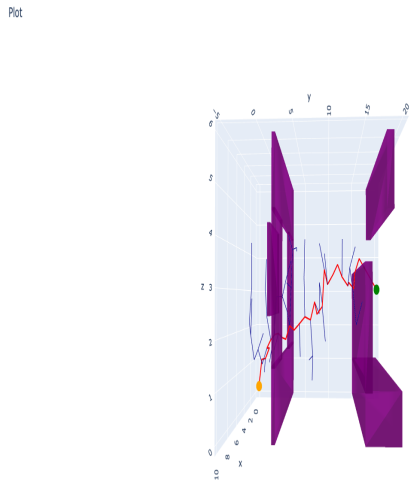

Fig. 13. RRT Path in Window Environment

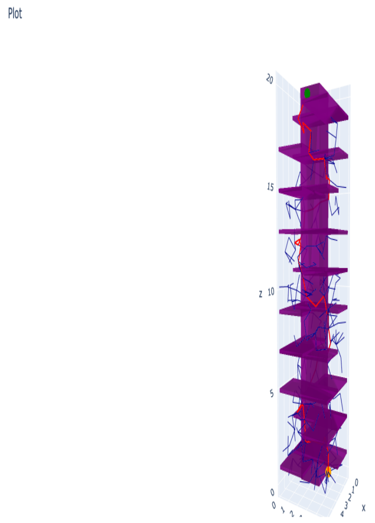

Fig. 12. RRT Path in Tower Environment

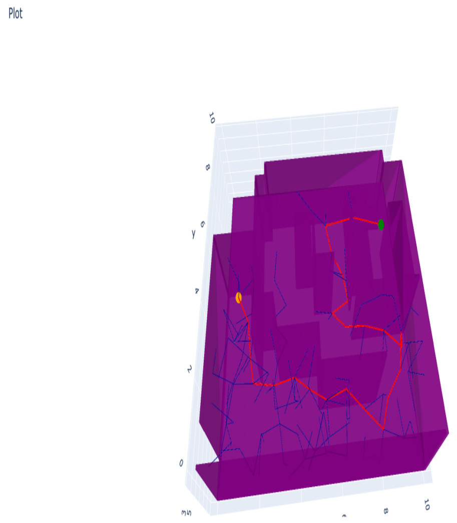

Fig. 14. RRT Path in Room Environment