ECE 276B Project 2: Motion Planning

1st Pranav Maddireddy
ECE 276B
University of California, San Diego
pmaddire@ucsd.edu

***Abstract*—This paper explores the implementation of weighted A* and RRT* to perform Search and Sample based motion planning.**

I. Introduction

One of the most important concepts in Robotics is motion planning. Motion planning is the method that your robot uses to identify optimal paths and what control sequences will get them there. There are many algorithms that perform motion planning effectively, the methods that this paper will focus on are *weighted A**, and *Rapidly-exploring random trees (RRT)*. A* uses heuristics to perform label correction on the set of states belonging to the environment. These states are usually discretized using patterns like square or triangle grids where each node is a vertex of the shapes. The weighted A* algorithm takes an input ϵ that prioritizes node expansion in the direction of the goal. Higher ϵ corresponds with a more aggressive expansion path towards the goal. This can be helpful in an open space but could become a slower process if it runs into objects in the environment, so its important to strike a balance with the value of ϵ. A* guarantees finding the optimal path within finite time, however weighted A* ($\epsilon > 1$) can only guarantee a suboptimal path, but can be faster. A* and weighted A* are popular because of their optimality guarantees but can become slow and computationally heavy due to needing a discretized graph of the environment. Instead of using a computationally heavy search based planning algorithm like A* we can use a sample based method like RRT. RRT randomly samples nodes from the environment and adds them to the graph if they are valid points and a path to the rest of the tree is valid. It keeps doing this until the tree reaches a node close enough to the goal and then goes straight to the goal. This approach allows us to not have to consider all the discretized states and rather just handle a subspace of the environment through the sampled nodes. This method requires less computation but doesn't have any optimality guarantees in finite time but does provide an asymptotic optimal path guarantee in infinite time. We will see how both of these methods perform on a set of different environments.

II. Problem Statement

A. Part a

The First part of the project is to design an algorithm that checks for collisions between line segments and objects in the environment including the border of the environment. I need to design a collision checker function that will take two nodes as inputs and evaluate if the path between them causes a collision with environment objects or the border. This function will be used to validate the optimal control path from my planning algorithms as well as assist in choosing paths within the algorithms.

1) Part b: For part b I need to implement a search-based planning algorithm. I choose to implement weighted A* because it has optimality guarantees and has levers I can use to optimize for efficiency. To implement A* I need to define a heuristic function, defined a planner with some movement rules, and create the main node expansion loop that keeps track of open and closed lists. Then I need to validate the algorithm for different ϵ weights and valid path outputs. This algorithm must expand states in order of lowest $f_i = g_i + \epsilon h_1$ and correct the g labels of each state. The final g labels and path will be ϵ suboptimal (path cost $\leq g_\tau \leq \epsilon dist^*(s, \tau)$ where dist* is the optimal path).

B. Part c

For part c I need to implement a sample-based planning method. I'm choosing to use to implement RRT* using The Open Motion Planning Library (OMPL). This library has a variety of optimized planning algorithms made in C++. RRT* requires a random node sampling method, a method that finds the nearest map node to a sampled node, a method that steers a path from a map node to a new sampled node, a collision detection method and the main loop that puts these together. The RRT* algorithm randomly samples nodes and attempts to add them to the graph by checking if the path between the nearest graph node and the sampled node is collision free and then adding it. We can also include an ϵ term such that $\epsilon < \inf$ means that the algorithm adds a node somewhere between the nearest node and sampled node instead. This allows the map to take smaller steps more often. RRT* differentiates itself from RRT by checking if it can improve its node path connections at each step, meaning the algorithm attempts to rewire the nodes near the newest node to help find a more optimal path.

III. Technical Approach

A. Part a

I started my collision detection function by discretizing the path between the nodes I am checking. I do this by creating three different vectors, one for each coordinate dimension (x,y,z) that evenly sample points along the path. Then I check

if any of these points lie beyond the border of the environment by comparing the maximum and minimum x,y, and z values of the environment to the discretized path. We know that all the objects in the environment are rectangular prisms that are defined by their lower left and upper right vertices. I use these coordinates to check if any point along my discretized path is on or inside any of the objects. If no collisions are detected, we get a False output and we get a True output when a point along the path is discovered to be on our out of the boundary or on or in an object.

B. Part b

For my search based approach I implemented Weighted A*. This algorithm describes a label-correcting method that uses a

Algorithm 1 Weighted A* Algorithm

OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}, \epsilon \geq 1$
$g_s = 0, g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
while $\tau \notin$ CLOSED **do**
  Remove i with smallest $f_i := g_i + \epsilon h_i$ from OPEN
  Insert i into CLOSED
  for $j \in$ Children(i) and $j \notin$ CLOSED **do**
    if $g_j > (g_i + c_{ij})$ **then**
      $g_j \leftarrow (g_i + c_{ij})$
      Parent$(j) \leftarrow i$
      if $j \in$ OPEN **then**
        Update priority of j in OPEN
      else
        OPEN $\leftarrow$ OPEN $\cup \{j\}$

heuristic to help interpret which nodes are the most promising and prioritize expanding those nodes first. This allows us to find an optimal or suboptimal path quicker and with less node expansions than other label-correcting approaches. I implemented this algorithm by creating a ANode class that creates an object that holds data values for each node, an Environment class that finds successor nodes, checks if we reached the goal, and calculates my heuristic values (euclidean norm in this case), and the main Astar class that hods the open list priority queue, closed list, performs node expansion, and updates the graph and g values of each discovered node.

```
class ANode(object):
    def __init__(self, key, coordinates):
        self.key = key
        self.coordinates = coordinates
        self.g = math.inf
        self.h = 0.0
        self.parent = None
        self.parent_action = None
        self.is_open = False
        self.is_closed = False

    def __lt__(self, other):
        return self.g < other.g
```

```
class Environment:
    def isGoal(self, node):
        return True

    def getSuccessors(self, node):
        return successor_list, cost_list,
        action_list

    def getHeuristic(self, node):
        return 0.0
```

Astar class includes:

```
from pqdict import pqdict

def aStar(start_coordinates, Env,
                  epsilon = 1.0):
    current = ANode(tuple(start_coordinates),
              start_coordinates)
    current.g = 0.0
    current.h = Env.getHeuristic(current)

    Graph[current.key] = current
    OPEN = pqdict()

    while True:
        if Env.isGoal(current.coordinates):
            return recoverPath(current, Env)

        current.is_closed = True
        updateData(current, Graph, OPEN, Env,
                   epsilon)

        if not OPEN:
            return # If OPEN is empty,
                     no path is found

        # remove the element with smallest f
          value
        current = OPEN.popitem()[1][1]
```

```
def updateData(current, Graph, OPEN, Env):
    successor_list, cost_list, action_list =
                    Env.getSuccessors(current)
    for s_coord, s_cost, s_action in
        zip(successor_list, cost_list,
                action_list):
        s_key = tuple(s_coord)
        if s_key not in Graph:
            Graph[s_key] = ANode(s_key,
                                 s_coord)
            Graph[s_key].h =
                    Env.getHeuristic(s_coord)
        child = Graph[s_key]

        tentative_g = current.g + s_cost
        if( tentative_g < child.g ):
            child.parent,
                child.parent_action = current,
                                      s_action
            child.g = tentative_g

            fval = tentative_g +
                   epsilon*child.h
            if child.is_open:
                OPEN[s_key] = (fval, child)
                OPEN.heapify(s_key)
            elif child.is_closed and
                        reopen_nodes:
                OPEN[s_key] = (fval, child)
                child.is_open,
                child.is_closed = True, False
            else:  # new node, add to heap
                OPEN[s_key] = (fval, child)
                child.is_open = True
```

Typically, for search based algorithms we need to have a discretized representation of our environment. To tackle the heavy computational challenges that this introduces, I implemented a method known as "lazy discretization" which only defines nodes in the environment once they are discovered. My environment class discovers new nodes when looking for successors through my 26 direction based movement rule and adds them to the graph if they are valid (not in an object our out of bounds). I also included a precision variable that controls the precision (distance between nodes) of my map discretization.

C. Part c

I used the OMPL library to implement my RRT* solution in a Linux environment. To do this, I went through the documentation and set up an OMPL class. This class defines the environment boundaries, implements a similar collision function to part a, and specifies parameters like validity checking resolution and maximum planning time. Then I replaced my A* planner with my RRT* planner in the main function to test it.

```
class OMPLPlanner:
    def __init__(self, boundary, blocks):
        # Initialize planning environment
        self.boundary = convert_to_float(boundary)
        self.blocks = convert_to_float(blocks)

        # Setup 3D planning space
        self.space = create_3D_space()
        set_space_bounds(self.space, boundary)

        # Configure planner settings
        self.planner = setup_basic_planner
                                (self.space)
        set_collision_checker(self.planner,
                self.isStateValid)
        set_collision_resolution(0.05)

    def isStateValid(self, state):
        # Check if state is collision-free
        x,y,z = get_position(state)
        for block in self.blocks:
            if position_inside_block(x,y,z,
                                     block):
                return False
        return True

    def plan(self, start, goal,
                    planning_time):
        # Validate input states
        if not self.isStateValid(start) or
                not self.isStateValid(goal):
            warn("Invalid start/goal")
            return straight_line(start, goal)

        # Configure planner
        set_start_and_goal(start, goal)
        planner = create_RRT_star()
        set_planner_parameters(planner)

        # Attempt to find path
        if solve(planner, planning_time):
            path = get_solution_path()
            smooth_path = interpolate_path(path)
            return smooth_path if validate_path
            (smooth_path) else None
        else:
            return None

    def check_path_collision(self, path):
        # Verify entire path is collision-free
        for point in path:
            if not self.isStateValid(point):
                return True
        return False
```

IV. Results

TABLE I
WEIGHTED A* RESULTS FOR DIFFERENT PRECISION AND ϵ VALUES

Environment	Precision	ϵ	Node expansions	Path Length
Single Cube	0.4	1	8474	8.305
Single Cube	0.4	3	227	8.305
Single Cube	0.4	10	218	8.305
Single Cube	0.7	1	921	8.513
Single Cube	0.7	3	135	8.547
Single Cube	0.7	10	135	8.547
Single Cube	1	1	203	8.715
Single Cube	1	3	97	8.811
Single Cube	1	10	97	8.811
Window	0.4	1	37875	27.685
Window	0.4	3	1333	27.685
Window	0.4	10	748	28.309
Window	0.7	1	8299	28.282
Window	0.7	3	398	28.810
Window	0.7	10	386	29.524
Window	1	1	1215	29.11
Window	1	3	265	30.035
Window	1	10	269	31.500

From these two tests in Table 1 we can see that the best parameters for path length and efficiency are precision = 0.7 and epsilon = 3.

TABLE II
WEIGHTED A* RESULTS FOR ALL ENVIRONMENTS WITH PRECISION = 0.7 AND ϵ = 0.3

Environment	Node expansions	Path Length
Single Cube	135	8.547
Maze	23266	77.310
Flappy Bird	3408	28.338
Pillars	15845	31.501
Window	398	28.810
Tower	6225	29.595
Room	1151	12.071

A. Path Quality:

Tables II and III show that both weighted A* and RRT* find paths of similar lengths. In many a few instances RRT* finds better paths than weighted A* due to having an asymptotic optimality guarantee as shown in table IV. Paths created from RRT* monotonically decrease as planning time goes to infinity and converges to the optimal path while weighted A* only has ϵ sub optimal path guarantees. As seen from the images, weighted A* tends to generate sharper paths while RRT* paths have more curvature and may be easier to navigate in real application.

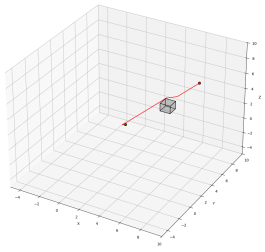

Fig. 1. Single cube with $\epsilon = 3$ and precision = 0.7

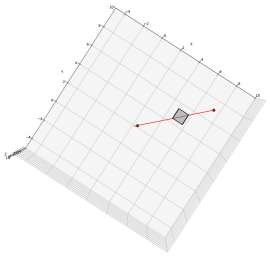

Fig. 2. Single cube with $\epsilon = 3$ and precision = 0.7

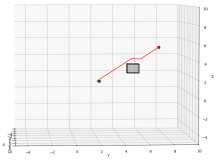

Fig. 3. Single cube with $\epsilon = 3$ and precision = 0.7

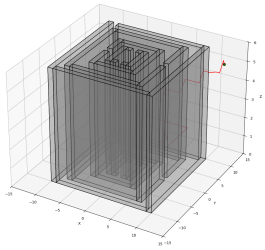

Fig. 4. Maze with $\epsilon = 3$ and precision = 0.7

B. Number of Considered Nodes

Tables II and III show that RRT* finds an initial path while considering much fewer nodes than weighted A* but these paths are significantly longer. To achieve paths better than or closer weighted A* RRT* requires many more nodes to be sampled than Weighted A* expands.

C. Effects of different parameters

Table I shows the results of different ϵ and precision values on A* in two different environments. Lower precision values find shorter paths but require expanding many more nodes while higher epsilon values require expanding less nodes but loosen the optimality guarantees. To strike a balance, I chose to go with $\epsilon = 3$, and precision = 0.7. Tables IV and V show the effects of planning time and expansion length on RRT*. RRT*'s asymptotic optimality guarantee gives us diminishing rewards for increasing planning time so I decided to go with 15 seconds. The expansion length defines how fast our graph can expand. Trying to expand too quickly will cause many invalid expansion in dense environment as shown in table V. I tested this on the most dense environment we had and decided to choose an expansion length=2 as a result of table V.

V. Discussion

I didn't run into many problems when building my collision checker and Weighted A* algorithm. One thing I noticed is that my run time dramatically decreased when I reduced the amount of points I check on the path in my collision detection which makes me thing that most of my algorithm's run time is used up there. One way to improve my weighted A* would be to make a faster collision detection. For part c, I ran into some issues with OMPL because I'm using a windows system , but once I had the code setup in a Linux environment it wasn't very difficult. I wanted to try different node sampling methods but I was unable to do it because I kept getting C++ wrapper errors. The OMPL library allows you to build a python wrapper for planning algorithms done in C++, this makes the runtime very quick but also means the formatting has to be very specific, so its difficult to edit some parts of the process.

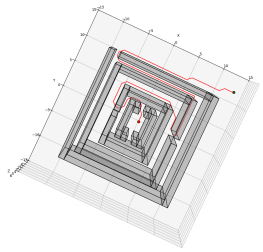

Fig. 5. Maze with $\epsilon = 3$ and precision = 0.7

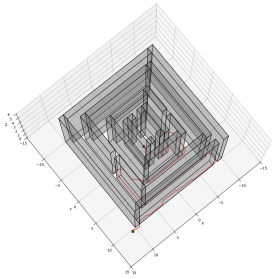

Fig. 6. Maze with $\epsilon = 3$ and precision = 0.7

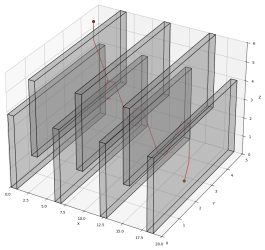

Fig. 7. Flappy Bird with $\epsilon = 3$ and precision = 0.7

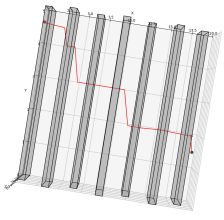

Fig. 8. Flappy Bird with $\epsilon = 3$ and precision = 0.7

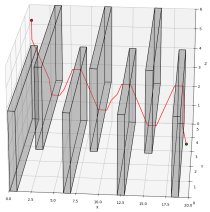

Fig. 9. Flappy Bird with $\epsilon = 3$ and precision = 0.7

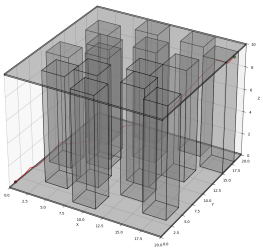

Fig. 10. Pillars with $\epsilon = 3$ and precision = 0.7

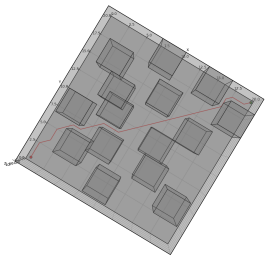

Fig. 11. Pillars with $\epsilon = 3$ and precision = 0.7

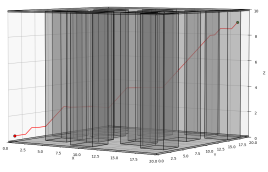

Fig. 12. Pillars with $\epsilon = 3$ and precision = 0.7

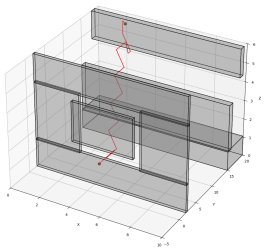

Fig. 13. Window with $\epsilon = 3$ and precision = 0.7

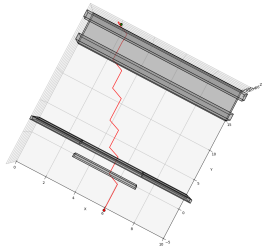

Fig. 14. Window with $\epsilon = 3$ and precision = 0.7

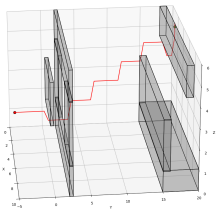

Fig. 15. Window with $\epsilon = 3$ and precision = 0.7

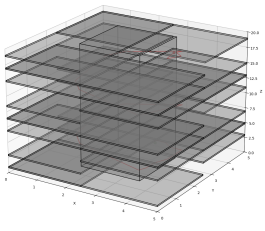

Fig. 16. Tower with $\epsilon = 3$ and precision = 0.7

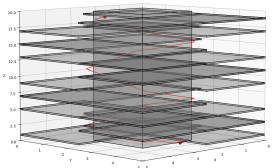

Fig. 17. Tower with $\epsilon = 3$ and precision = 0.7

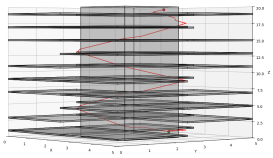

Fig. 18. Tower with $\epsilon = 3$ and precision = 0.7

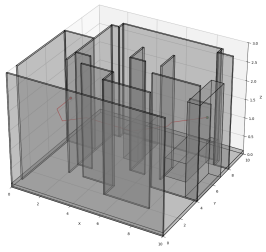

Fig. 19. Room with $\epsilon = 3$ and precision = 0.7

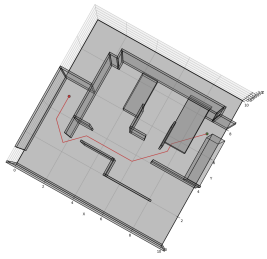

Fig. 20. Room with $\epsilon = 3$ and precision = 0.7

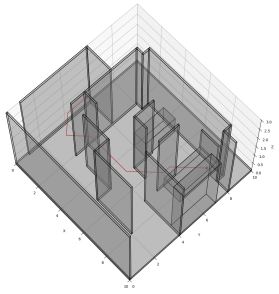

Fig. 21. Room with $\epsilon = 3$ and precision = 0.7

TABLE III
RRT* RESULTS FOR PLANNING TIME = 15 SECONDS AND EXPANSION LENGTH = 2

Env.	First path nodes.	First path length	Plan. time nodes	Plan. time path len.
Single cube	89	11.55	30237	7.891
Maze	4650	84.89	20491	75.637
Flappy Bird	434	36.46	20853	28.420
Pillars	845	36.89	19687	28.734
Window	168	30.81	21232	25.866
Tower	390	37.28	8153	29.747
Room	84	21.86	10706	11.050

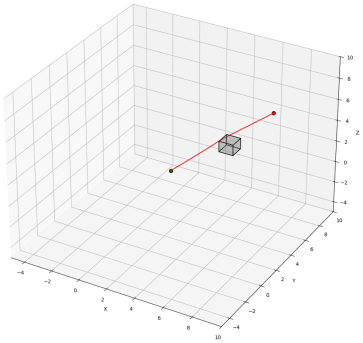

Fig. 22. RRT* Single Cube with Planning time = 15s, and expansion length = 2

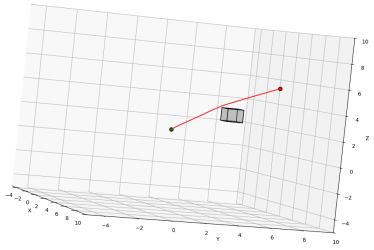

Fig. 23. RRT* Single Cube with Planning time = 15s

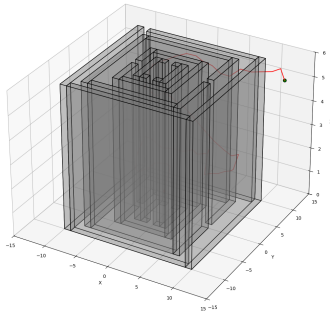

Fig. 24. RRT* Maze Cube with Planning time = 15s, and expansion length = 2

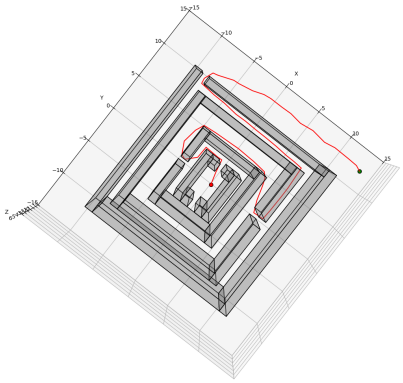

Fig. 25. RRT* Maze Cube with Planning time = 15s, and expansion length = 2

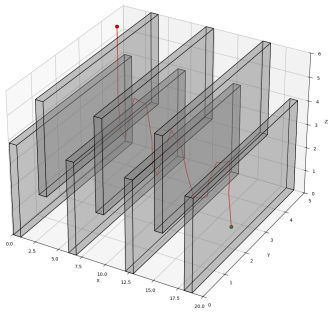

Fig. 26. RRT* Flappy Bird with Planning time = 15s, and expansion length = 2

TABLE IV
RRT* ASYMPTOTIC OPTIMALITY

Env.	Initial path	10s	15s	20s	200s
Maze	84.89	76.501	75.637	75.119	73.327

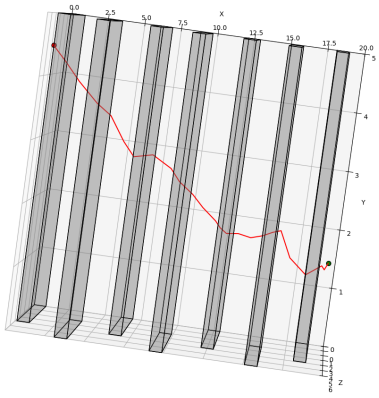

Fig. 27. RRT* Flappy Bird with Planning time = 15s, and expansion length = 2

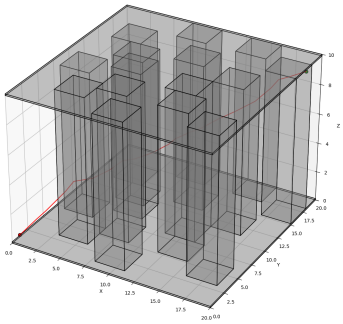

Fig. 28. RRT* Pillars with Planning time = 15s, and expansion length = 2

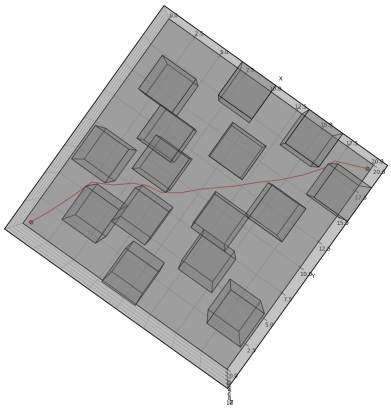

Fig. 29. RRT* Pillars with Planning time = 15s, and expansion length = 2

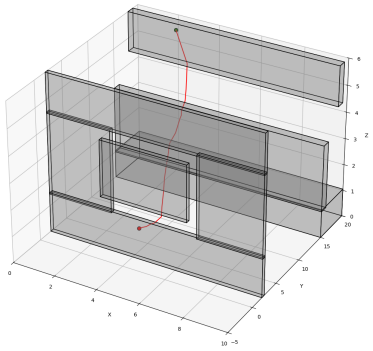

Fig. 30. RRT* Window with Planning time = 15s, and expansion length = 2

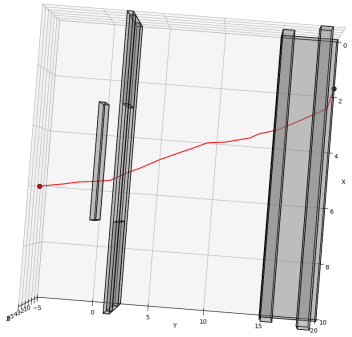

Fig. 31. RRT* Window with Planning time = 15s, and expansion length = 2

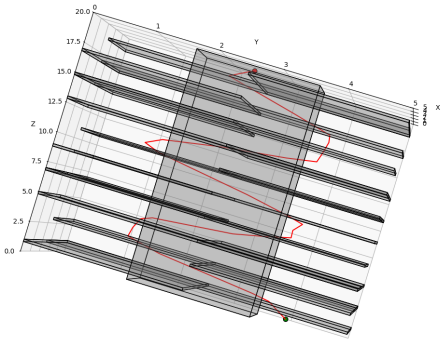

Fig. 32. RRT* Tower with Planning time = 15s, and expansion length = 2

TABLE V
RRT* EXPANSION PATH LENGTH VS EXPANSION LENGTH WITH PLANNING TIME = 10S

Env.	**0.5**	**1**	**1.5**	**2**	**2.5**
Maze	113.022	81.904	76.957	76.296	76.617

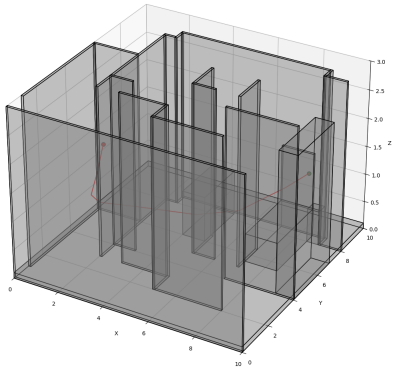

Fig. 33. RRT* Room with Planning time = 15s, and expansion length = 2

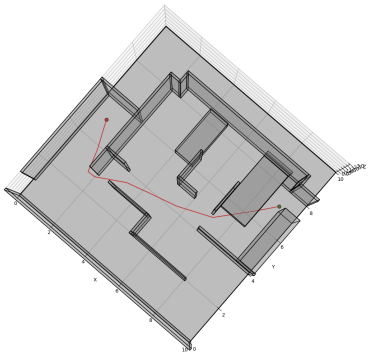

Fig. 34. RRT* Room with Planning time = 15s, and expansion length = 2