UCSD Spring 2021 ECE276B: Motion Planning in 3D Search vs Sampling Based Method

Steven Wong
Machine Learning and Data Science
University of California, San Diego
San Diego, CA 92093
scw039@ucsd.edu

May 17, 2021

1 Introduction

In this project, we look at motion planning in 3D using search and sampling based methods. Given an environment with obstacles, our objective is to use these two methods to find a path from the start node to the goal node. These nodes are represented as three dimensional coordinate points. This is an important problem as it can be extended to robotic motion planning - how a robot can safely traverse a map from point A to point B without colliding with any obstacles. Our search based method, the weighted A* Algorithm, will be rigorously compared to a sampling based method, the rapidly exploring random tree star (RRT*) Algorithm.

2 Problem Statement

We will formulate the 3D motion planning problem as a deterministic shortest path (DSP) problem using the following definitions. The motion model $f(x_i)$ is obtained by adding a vector $d \in \mathbb{Z}^3$ to x_i where:

$$d \in \{(a,\ b,\ c)^T \mid a,\ b,\ c \in \{-res,\ 0,\ res\}\} \backslash \{(0,\ 0,\ 0)^T\} \tag{1}$$

d here is the control space - i.e. all possible candidates to move (26 possibilities), while res defines the resolution - i.e. how far we want our agent to move. Thus the state space $\mathcal{X}$ includes all reachable points given by the motion model provided we do not collide with an obstacle and stay within the boundary. This can be written formally as:

$$\mathcal{X} \in \{(x,\ y,\ z) \mid \begin{bmatrix} x_{min} < x < x_{max} \\ y_{min} < y < y_{max} \\ z_{min} < z < z_{max} \\ (x,\ y,\ z) \notin obstacle \\ (x,\ y,\ z)\ from\ f \end{bmatrix}\} \tag{2}$$

The initial state x_0 is the starting coordinate $(x_{start}, y_{start}, z_{start})$. Obstacles are axis aligned bounding boxes (AABB) defined by its bottom left corner (smallest coordinates) and its top right corner (largest coordinates). In our implementation, the cost to go from one node to another is the euclidean distance between the two. This insures no negative cycles in the graph, as the distance function returns only positive costs.

$$l(x_i, x_j) = ||x_i - x_j||_2 \tag{3}$$

The terminal cost is defined as follows:

$$\mathbf{q}(x) = \begin{cases} 0 & \text{x is at the goal} \\ \infty & \text{x is not at the goal} \end{cases} \tag{4}$$

The objective is to find the path of least cost between the start node and the goal node. This is simply the path in which the sum of edge weights are a minimum. Our planning horizon T will be variant to how many iterations we need in our Algorithm, with a smart termination in the end. This method will be described in the next section. Now that we have defined a state space, control space, motion model, a movement cost, and a planning horizon - ($\mathcal{X}$, d, f, l, T, $\mathbf{q}$), we have defined the terms needed to solve a DSP problem.

3 Technical Approach

3.1 Search Based Planning

We will use the weighted A* algorithm to solve the 3D motion planning problem. We must first define the shortest cost to arrival found so far- g, and the underestimated cost to goal heuristic function- h. The shortest cost from start to the node of interest i is initially defined as:

$$g = \begin{cases} g_{start} = 0 \\ g_i = \infty \end{cases} \tag{5}$$

The heuristic function, defined as the underestimated cost from node of interest i to the goal τ can be calculated using a few steps.

1. find how many x,y,z steps one must take to get from node i to τ by calculating:

$$\Delta x, \Delta y, \Delta z = |(x_i,\ y_i,\ z_i) - (x_\tau,\ z_\tau,\ z_\tau)| \tag{6}$$

2. Sort the $\Delta x, \Delta y, \Delta z$ in increasing order: $(\Delta x, \Delta y, \Delta z) \longrightarrow (a, b, c)$
3. Solve for h:

$$h = \sqrt{res * 3} * a + \sqrt{res * 2} * b + \sqrt{res} * c \tag{7}$$

This heuristic function is a guaranteed underestimate of the cost of a shortest path from i to τ, making it admissible, satisfying the heuristic requirement for weighted A* to work properly. We define weight term ϵ, which measures how much we trust the heuristic.
Next we define obstacle avoidance and detection. We deploy an altered, vectorized, python version of Tavianator's 2015 Code. This function takes in node i and neighbor node j, and returns True if the line segment between the two nodes intersects an AABB. In addition, we check to see if the neighbor node j leaves the boundaries of the state space. To recover the path of minimal cost back, we keep track of the parents of every node using a python dictionary *Parent*. The weighted A* algorithm follows the pseudocode below.

Algorithm 1: Weighted A* Algorithm

Result: Returns shortest path found
Inputs:
All DSP variables defined in problem statement section ($\mathcal{X}$, d, f, T, l)
Initialization:
$OPEN$ {start}, $CLOSED$ {}, $\epsilon = 2$
$g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{X} \backslash \{start\}$
while $||x_i - \tau||_2 \geq 0.1$ **do**
  Remove i with smallest $f_i = g_i + \epsilon h_i$ from $OPEN$
  Insert i into $CLOSED$
  for $j \in$ *Neighbors(i) and* $j \notin$ *CLOSED* **do**
    if *No Collision* **then**
      if $g_i + l(x_i, x_j) < g_j$ **then**
        $g_j \longleftarrow g_i + l(x_i, x_j)$
        $Parent(j) \longleftarrow i$
        if $j \in OPEN$ **then**
          $f_j = g_j + \epsilon h_j$
        end
        else
          $OPEN \longleftarrow OPEN \cup \{j\}$
        end
      end
    end
  end
end

Our stopping criteria is simply if the expanded node is sufficiently close enough to the goal, and there are no obstacles in the way, we directly connect to the goal.
This algorithm can scale to large maps in efficient time as we use a discritized space (26 connected grid) of the continuous environment. We can change how fine grain we want our trajectory to be based off the res variable. Unfortunately, because of the discritization, our algorithm is sub-optimal - i.e. it will not return the path of least cost given reasonable res, but rather something extremely close. This is the trade off we make for time efficiency. With the 26 connected grid we gain a much faster (up to 100x) implementation than a 26 connect sphere. A 26 connected sphere would search through most of continuous space, making the algorithm more optimal but much slower. We saw it fit to trade out the 26 connected sphere for time efficiency. In specfic, for a binary heap, sparse graph implementation, our time complexity is O($|\mathcal{V}|$log$|\mathcal{V}|$), where $|\mathcal{V}|$ is the number of nodes. In addition, the memory efficiency of the 26 connected grid is favorable as we expand less nodes in continuous space - i.e. number of nodes to be stored is O($|\mathcal{V}|$). As $\mathcal{V}$ gets increasingly larger, weighted A* will run out of memory, and the time complexity increases. The weighted A* algorithm is complete because it will return a feasible path if one exists within finite time.

3.2 Sampling Based Planning

We compare our search based method to the RRT* algorithm using code from The Python motion planning library. In RRT*, random samples in continuous space are picked to create an expanding graph. RRT* incorporates a extend and rewire step, in which we edit the graph based on label correcting methods similar to A*. This algorithm is also complete and asymptotically optimal.

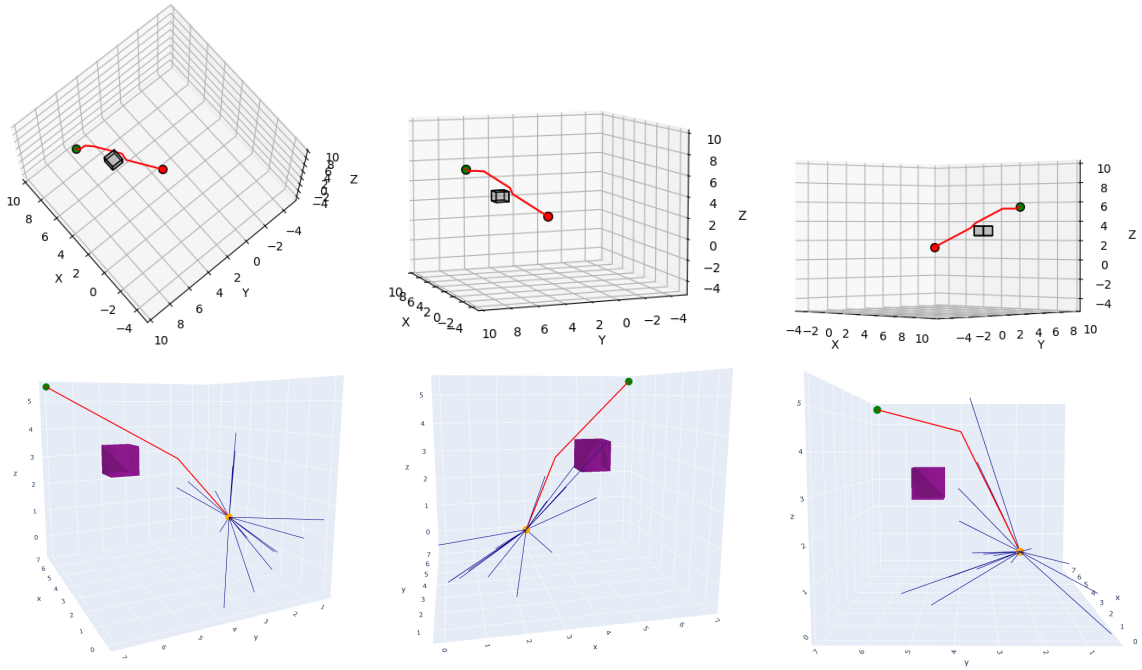
Parameters specified in the code are the maximum iterations before timeout, which was set to 15000, the length of tree edges, set to 1, and the length of the smallest edge r to check for intersection with obstacles, set to 0.1. The collision checking method implemented by the Python motion planning library involves taking samples of points with resolution $r = 0.1$ along the line segment, and checking if any of those points lie inside an obstacle. This is a method that can be used for non-AABB obstacles, but since all obstacles in our environment are AABB, we can come up with the closed form solution described in the previous section for a faster implementation. Finally, our rewire count is 32, meaning that we are taking 32 nearest nodes around us for rewire considerations.

4 Results

Here we compare the performance of the weighted A* algorithm to the RRT* algorithm in 7 different environments. For these tests, the weighted A* algorithm uses a $res = 0.5$, $\epsilon = 2$. The parameters for RRT* are the same as the ones mentioned in the previous section. Here the top row of images are the results of the weighted A* algorithm, and the bottom row of images are the results for the RRT* algorithm.

4.1 Single Cube

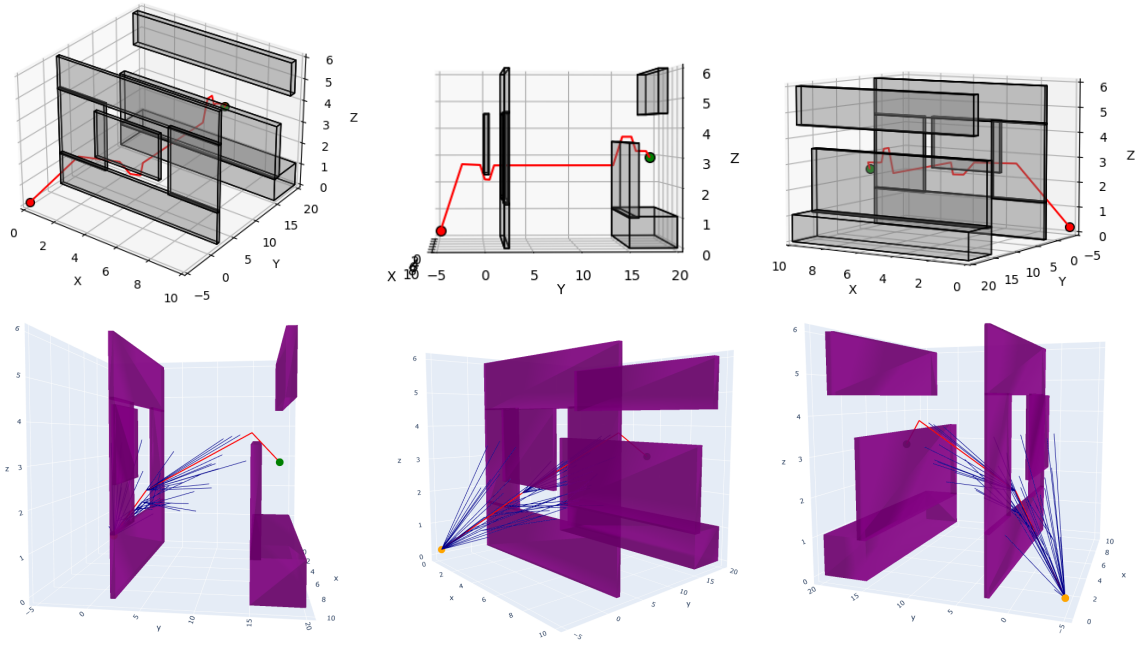
In this environment a single cube stands in the way from start to goal.



RRT* has less turns and thus a visually more optimal path. As it is a more basic environment, the discrepancies between the two methods is hard to determine.

4.2 Window

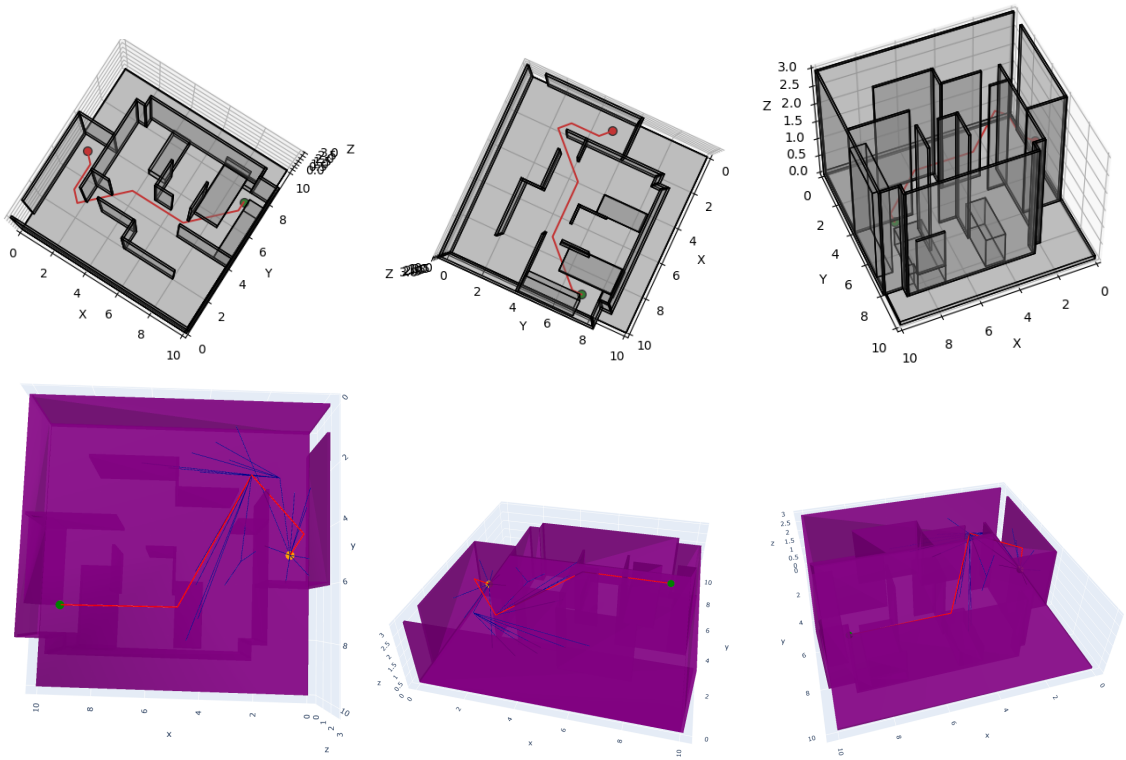
In this environment we navigate through a window to the goal.



Here again RRT* has a better quality path, turning very few times. Notice the blue lines on the bottom row of images. These blue lines represent the expanded nodes in the RRT*. Comparing this to the expanded nodes in Appendix A for weighted A*, we can see that the weighted A* algorithm expands in the direction of the goal. This means that the weighted A* algorithm is more efficiency in expansion (executes in less time).

4.3 Room

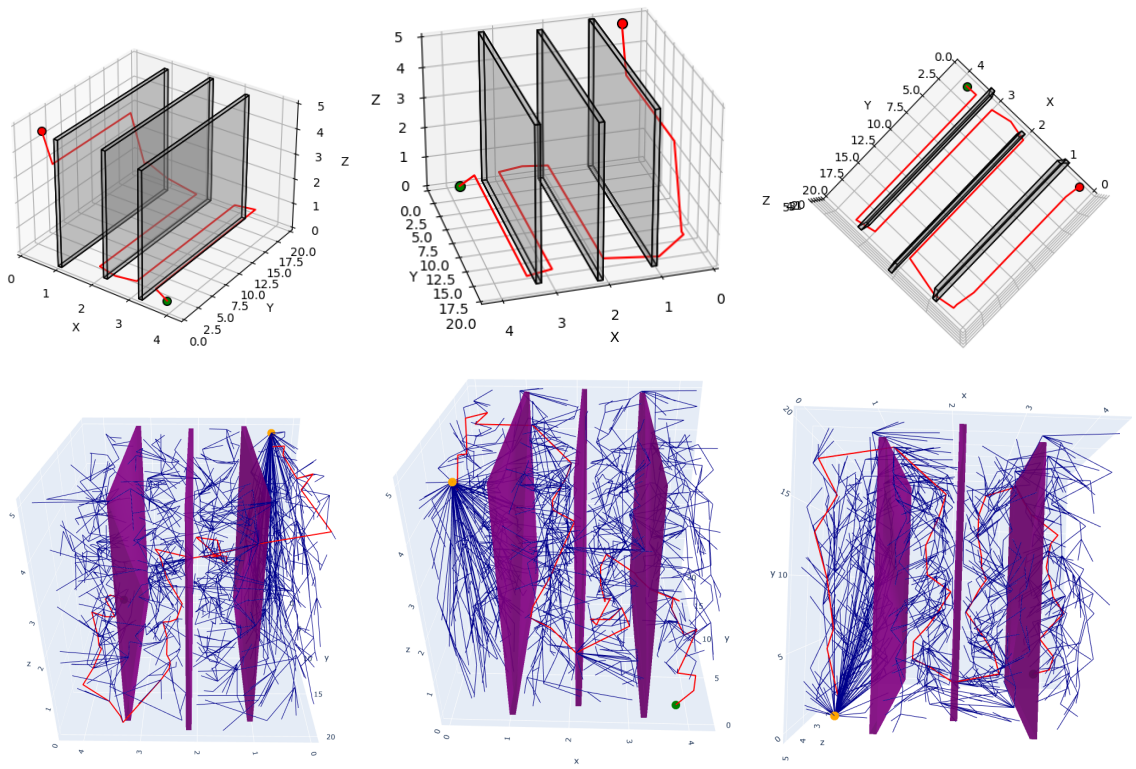
In this environment we navigate through a room to the goal.



The quality of both paths look similar in this case, but the weighted A* algorithm executed the program in a much shorter time. We can see this by looking at the expanded nodes in Appendix A.

4.4 Monza

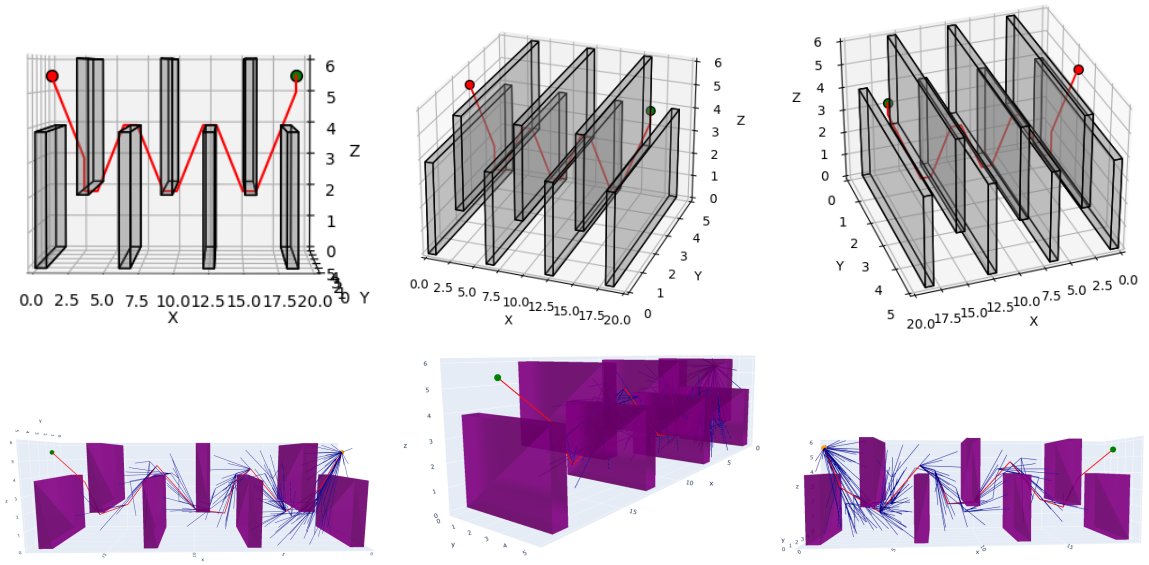
In this environment, we navigate around walls to reach the goal.



Here the RRT* algorithm does not do as well as weighted A* in terms of path quality. The path in RRT* is rather back and forth, while weighted A* produces straight lines, increasing its optimality. By comparing the expanded nodes through Appendix A, we can see that weighted A* expands a more uniform set of points (26 connected grid expansion), while the RRT* samples a large number of random points, leading to poor results, and slower execution time. Also note, $r = 0.05$ so the path does not phase through obstacles.

4.5 Flappy Bird

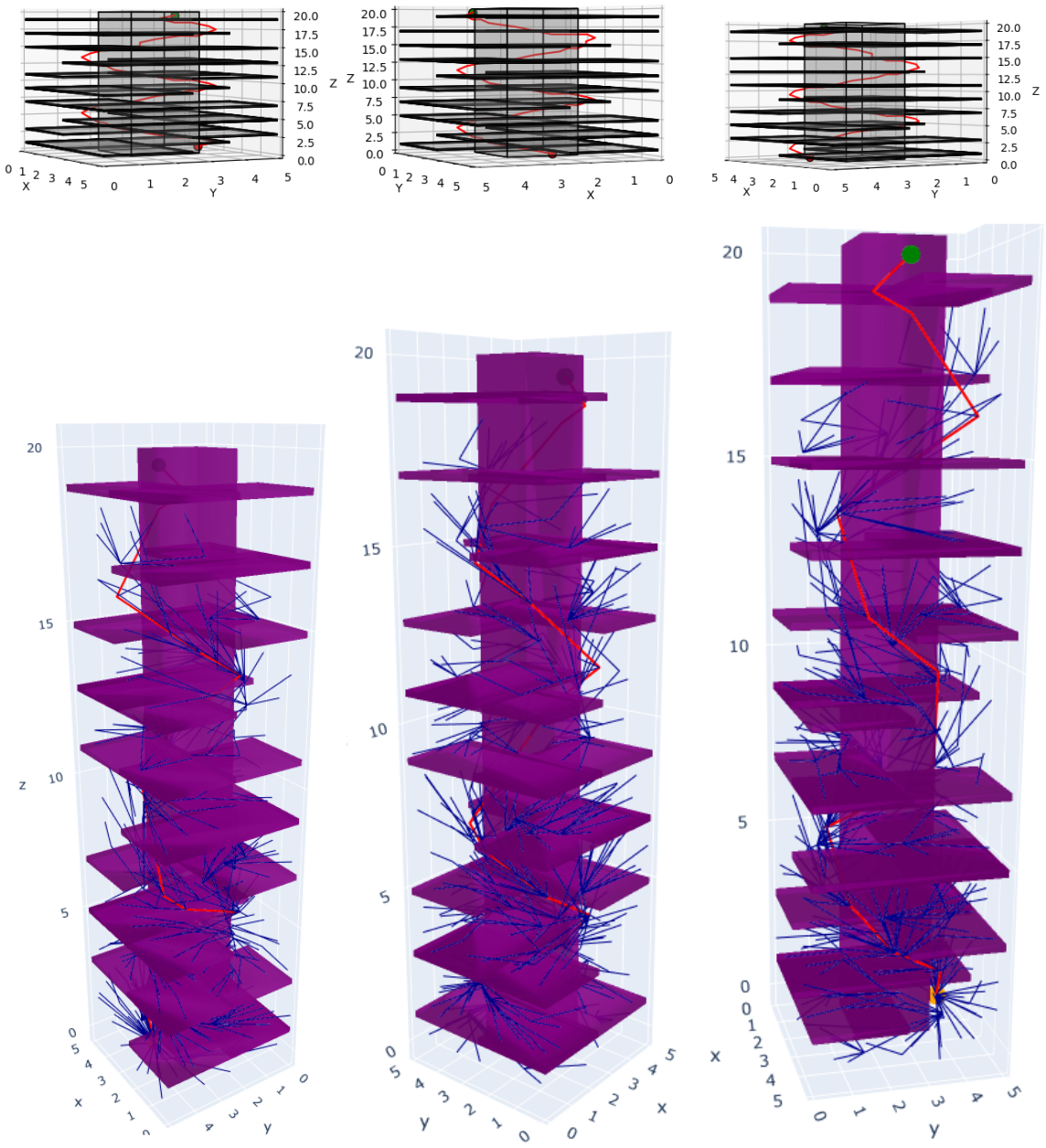
In this environment, we navigate over and under walls to reach the goal.



Here, using a verticality logic with Monza, we can see why weighted A* does a better job at finding a more optimal path than RRT*. The randomness in a complex environment seems to deteriorate the quality of the path. In addition, weighted A* was much quicker to compute this path, as there were less nodes to expand, and the collision checking was optimized. A node comparison can be seen by looking at Appendix A.

4.6 Tower

In this environment, we navigate through a tower to reach the goal.

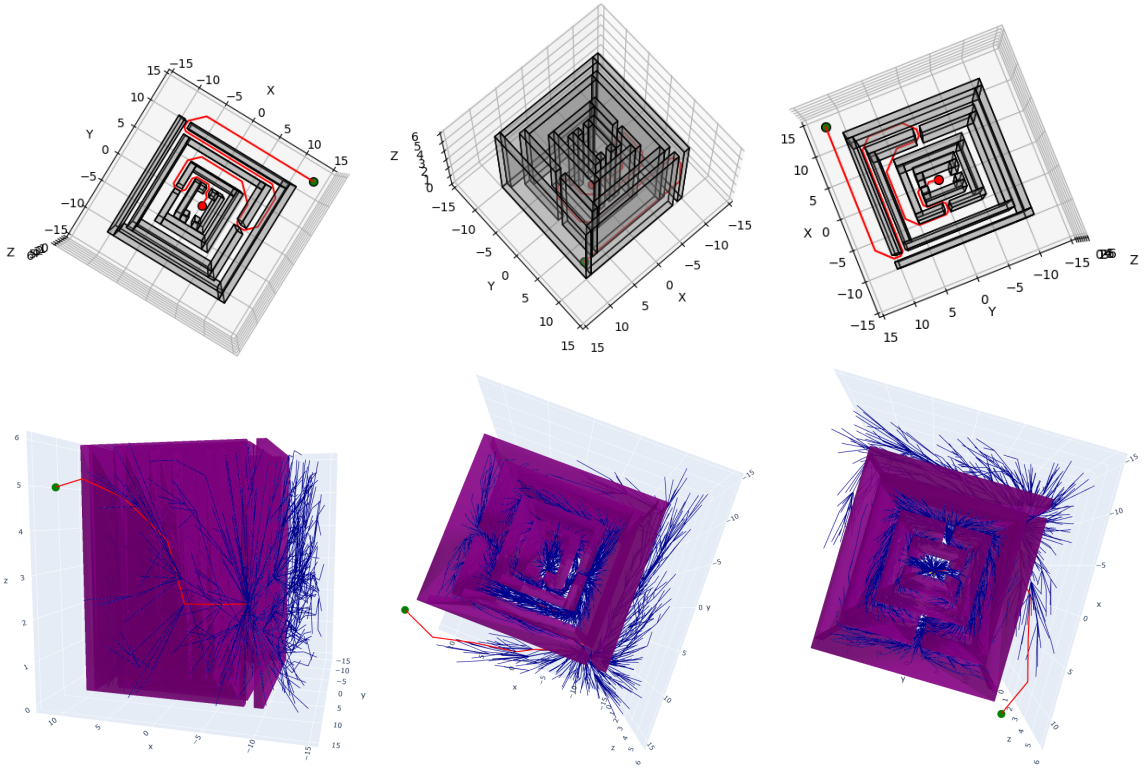


Here RRT* and weighted A* achieve similar results, with roughly the same path length. It is important to note that each time RRT* is run, the path length changes, and thus one can only use

the average length of each trial as a metric for comparison. The execution time of weighted A* once again is much faster. The expanded nodes can be seen in Appendix A.

4.7 Maze

In this environment, we navigate through a maze to reach the goal.



Here, although RRT* finds a very feasible path, the execution time is incredibly long. One can imagine that randomly picking spots in a maze to explore is not as systematic as the weighted A* algorithm. Looking at Appendix A, one can observe that the systematic nodes expanded by weighted A* allow for a faster implementation.

4.8 Path Length Chart

Here we compare the path length between weighted A* and RRT*.

Path Length		
MAP	**A***	**RRT***
Single Cube	**8**	8.08
Window	27	**24.69**
Room	**12**	13.2
Monza	**78**	78.01
Flappy Bird	**25**	27.43
Tower	33	**31.09**
Maze	81	**76.37**

4.9 Number of Expanded Nodes

Here we compare the number of nodes expanded/considered between weighted A* and RRT*.

Number of Considered Nodes		
MAP	**A***	**RRT***
Single Cube	**13**	19
Window	**54**	236
Room	**86**	111
Monza	**1951**	2674
Flappy Bird	734	**406**
Tower	1071	**977**
Maze	**5852**	12055

4.10 Parameter Tuning

Parameter tuning varies on the environment that we are currently solving. In this paper, we will go over one example with parameter tuning. Observe the case of Flappy Bird. We will tune the parameters of resolution and heuristic function in weighted A*, and the resolution and rewire count for RRT*.

4.10.1 Weighted A* Parameters

Changing Resolution:
Notice as the resolution increases, we are unable to find the optimal path as we can only move in a *res* direction in the 26 connected grid, leading to worse results.

Changing ϵ:

Once can notice that when $\epsilon = 0$, you can notice a deviation in the path at the end. We speculate this is because the number of expanded states increases as we do not trust our heuristic function as much with such a low ϵ. When we trust our heuristic too much, notice we also do not produce desirable results, because the expanded nodes trust the heuristic too much. As ϵ increases, the time of execution decreases, but the path becomes less optimal.

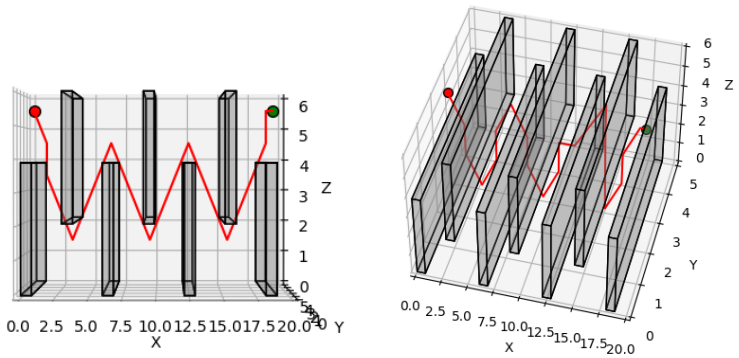


Figure 1: Left: $res = 1$ Right: $res = 1.5$

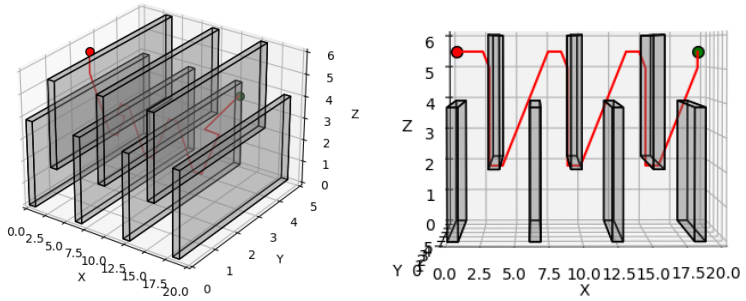


Figure 2: Left: $\epsilon = 0$ Right: $\epsilon = 20$

Changing Heuristic:
Changing the heuristic to the euclidean distance:

$$h = ||(x, y, z) - (x_{goal}, y_{goal}, z_{goal})||_2 \tag{8}$$

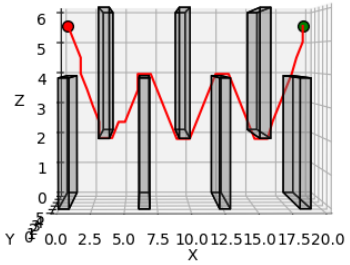


Figure 3: Euclidean Heuristic

Notice it is not as efficient as the heuristic we implemented because our original heuristic is a closer

estimate of the optimal cost than that of the euclidean distance.

4.10.2 RRT* Parameters

Changing Resolution:

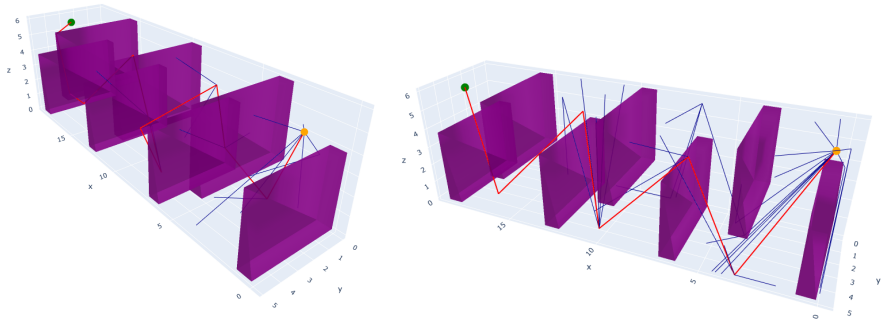


Figure 4: Left: $res = 4$ Right: $res = 8$

Notice for the same reasons as why increasing res is worse in weighted A*, increasing the res in RRT* is not optimal as well

Changing Rewire Count

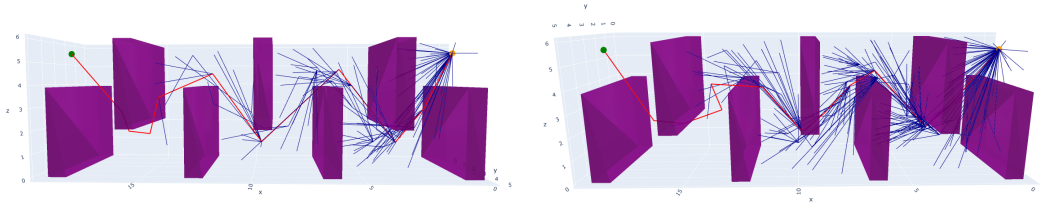


Figure 5: Left: 64 rewire Right: 128 rewire

Notice that rewiring to 64 produces good results, but increasing it 128 makes it worse. This is because once one considers that many nearest nodes for rewiring, there is a chance we rewire the wrong ones, leading to suboptimal results.

5 Discussion

The two methods weighted A* and RRT* are comparable in performance- they produce meaningful results with close to optimal paths. However, the RRT* algorithm is much slower. This can be due to the case that the collision checking implemented by the Python motion planning library is sampling based. As mentioned before this method is used to account for non-AABB obstacles. Since our weighted A* collision checker only accounts for AABBs, we are able to vectorize a closed form solution to the problem. Another theory is the randomness of the RRT*. Choosing more bad samples will increase the execution time. (i.e. we see that on average RRT* expands more nodes). Because of these reasons, it is not meaningful to include a time comparison, as there are more than one confounding variables that can be affecting it. To make it better, we can implement our vectorized closed form collision detector for the RRT*, and then compare the times. Doing this, we can then see a more conclusive result on time complexity. In addition, we can improve the weighted A* algorithm by simply playing with the hyperparameters, decreasing the res to achieve more optimal results while also balancing the amount of time it takes to execute. Overall, both search and sampling based methods show promising results for optimal pathing, but weighted A* runs at an incredibly faster rate.

6 Appendix A

Blue points show expanded nodes in the weighted A* algorithm. These are to be compared to the blue lines in RRT* plots.