Project Report

All the code can be found on:
https://github.com/JingHHj/Resume/tree/main/ECE276B/PR2/starter_code

1. Introduction

In this project, we are required to implement motion planning algorithms in a 3D Euclidean space. To help our agent find the shortest path from the start to the goal, stay inside of the boundary and avoid all the obstacles. There are three parts in total. In the first, a search based motion planning was implemented (A* search). While in the last part, we were doing sampling-based motion planning algorithm, which is the RRT.

2. Problem Statement

Overall, this project can be generated as a deterministic shortest path (DSP) problem. In a DSP problem, there is a graph composed by a vertex set $\mathcal{V}$ which is the state space. The vertices are connected by edges which has a label c_{ij} on it, defined as the cost it takes to get from node i to j. Also there will be a start s and a goal τ . When the agent get from a node 1 to another node k and path through limited nodes it can form a path which is a sequence of nodes $i_{1:k} = (i_1, i_2, \dots, i_k)$. The length of the path is defined as the sum of the cost of all the edges that the agent has been through: $J^{i_{i:k}} = \sum_1^{k-1} c_{i_k, i_{k+1}}$. Our objective is to find the shortest path from the start to the goal which in this case a shortest path can be defined as the path with the lowest overall cost. Also there is a important assumption that needs to be mention: There are no negative cycles in the graph, i.e. $J^{i_{i:k}} > 0$ for all vertex $i \in \mathcal{V}$.

3. Technical Approach

1) Search Based Algorithm

   In this part, we implemented a A* search for the project. Before we got into the core part of the algorithm, there's two things need to be done: collision checking and discretization

   i. Collision Checking

      Basically, in this part we need to check for every two points, if the line segment in between pass through any of the obstacles in the map. It is a bit tricky since this is a 3D case. So, we decided to turn into a 2D problem. We projected every line segment and every block onto every plane (x, y, z). If on all three plain, the line is intersected with the projection of the box, there is a collision between this line segment and the that block. Eventually, every line segment will check will all the blocks and do it for all the segment in a path. If and only if there is zero collision occurs, we can say that there is no collision.

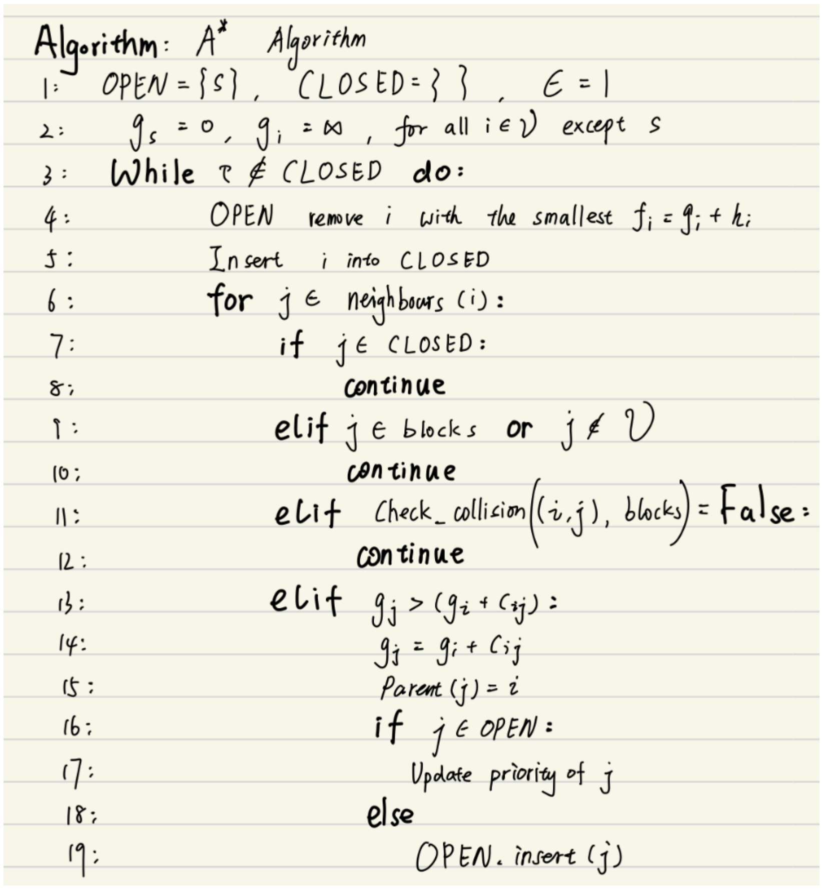
During the 2D collision check, we chose to use a function called intersection from python library shapely to accomplish it. It could return the length of the line segment, if we input a line and a square. If the length of the line is 0, it means there is no intersection

ii. Discretization

In this project, we were given a continuous space, in which we certainly cannot implement A* search on it. So, we need a function to discretize the whole space in to cells, and convert the result back after we do the A* search. Taking considerate of the time we our algorithm is going to take, we se the size of the cell into 0.5 meters. Also, since we are using cells, it means the coordinate represent the center of the cell instead of left down corner, which means we need to take consider that when we convert the path back to meters after the computation.

iii. A*

Basically, the A* search algorithm works as the pseudocode below:



```
Algorithm: A* Algorithm
1:  OPEN = {s}, CLOSED = { }, $\epsilon = 1$
2:      $g_s = 0$, $g_i = \infty$, for all $i \in V$ except s
3:    While $\tau \notin$ CLOSED do:
4:            OPEN remove i with the smallest $f_i = g_i + h_i$
5:            Insert i into CLOSED
6:            for $j \in$ neighbours(i):
7:                if $j \in$ CLOSED:
8:                    continue
9:                elif $j \in$ blocks or $j \notin V$
10:                   continue
11:               elif Check_collision((i, j), blocks) = False:
12:                   continue
13:               elif $g_j > (g_i + c_{ij})$:
14:                   $g_j = g_i + c_{ij}$
15:                   Parent(j) = i
16:                   if $j \in$ OPEN:
17:                       Update priority of j
18:                   else
19:                       OPEN.insert(j)
```

OPEN: A list that keep all the nodes of the frontier which are ready to be expand. In this case we were using pqdict from python library pqdict. It is in the form of dictionary and can automatically find the item with the lowest value (can be self-define).

CLOSED: A list that keep all the nodes that already been expanded which have the correct g-values and will not be expand again.

s: The starting node

τ: The goal node

c_{ij}: The cost that is it going to take from the current node to the next node. In this project, there will be three possible cases, 1, $\sqrt{2}$ and $\sqrt{3}$. Since we allow diagonal move

h_i: The heuristic of the node

g_i: Estimate of optimal cost-to-arrive from s to i

$\mathcal{V}$: The state space, in this case is all the possible nodes in the free space.

ϵ: The weight for weighted A* star. It is used to accelerate the computation when it is bigger than 1. In this case, we just set $\epsilon = 1$.

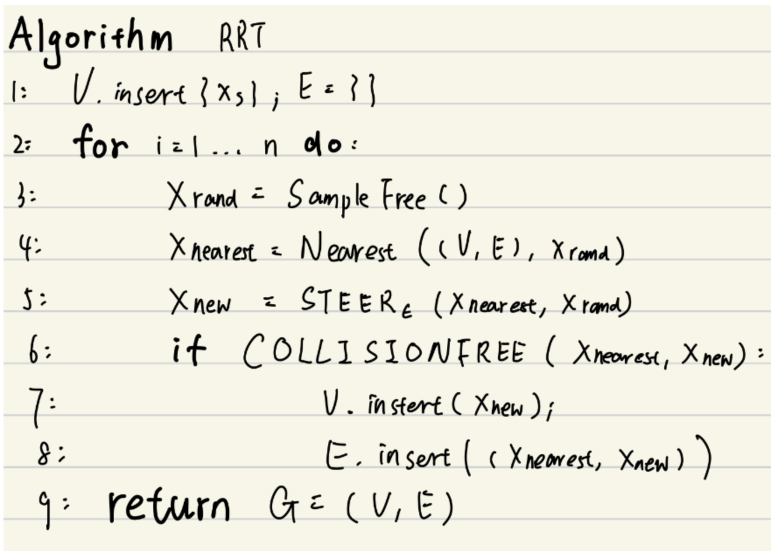
f_i: The criteria for our priority queue, defined as $f_i = g_i + h_i$. We will always remove the node with the lowest f_i in the queue.

One of the most important things for designing a A* search is to chose the heuristic function. We need it to be admissible and better to be consistent. We've been tested the Manhattan distance ($h_i = \left\| x_i - x_\tau \right\|_1 = \sum_k \left| x_{i,k} - x_{\tau,k} \right|$ which is inadmissible when can move diagonally), the diagonal distance ($h_i = \left\| x_i - x_\tau \right\|_1 = max_k \left| x_{i,k} - x_{\tau,k} \right|$) and the Euclidean distance ($h_i = \left\| x_i - x_\tau \right\|_2$). Eventually we narrowed it down to using the Euclidean distance

2) Sampling Based Algorithm

In the sampling base algorithm, the search space is continuous. Instead of discretizing the whole map at the beginning before we search, we will be doing sampling and searching at the same time. In this case, it can guarantee to find the shortest a path within shorter time.

In our code we are using Randomly Exploring Random Tree (RRT) algorithm. The way it construct a map is by sample a new configuration from the free space: $x_{rand} \in C_{free}$, find the nearest neighbor $x_{nearest}$ in graph G , connect them if straight line is collision-free. The algorithm is defined as follow:



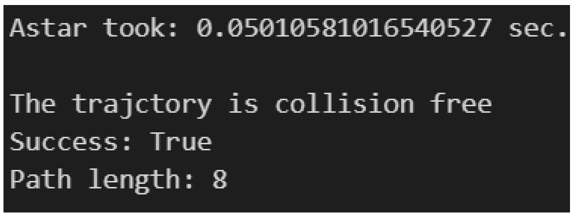
```
Algorithm RRT
1: V.insert{$x_s$}; E = {}
2: for i=1... n do:
3:         $X_{rand}$ = SampleFree()
4:         $X_{nearest}$ = Nearest((V, E), $X_{rand}$)
5:         $X_{new}$ = $STEER_\epsilon$($X_{nearest}$, $X_{rand}$)
6:         if COLLISIONFREE($X_{nearest}$, $X_{new}$):
7:                 V.instert($X_{new}$);
8:                 E.insert(($X_{nearest}$, $X_{new}$))
9: return G=(V, E)
```

G: Graph. Defined as G = (V, E) in which V are the set of all the vertices and E are the set of all the edges that connect vertices in V.
C_{free}: The free space
C : The configuration spaces
SampleFree: A function that returns a Sample from free space
Nearest: A function that returns the nearest vertex $x_{nearest}$ from a given one x
$STEER_{\epsilon}$: A function that returns a point z within distance of ϵ from x, when given two points x and y
$COLLISIONFREE$: A function that returns a bool that shows whether the line segment between the given points x and y are collision free or not

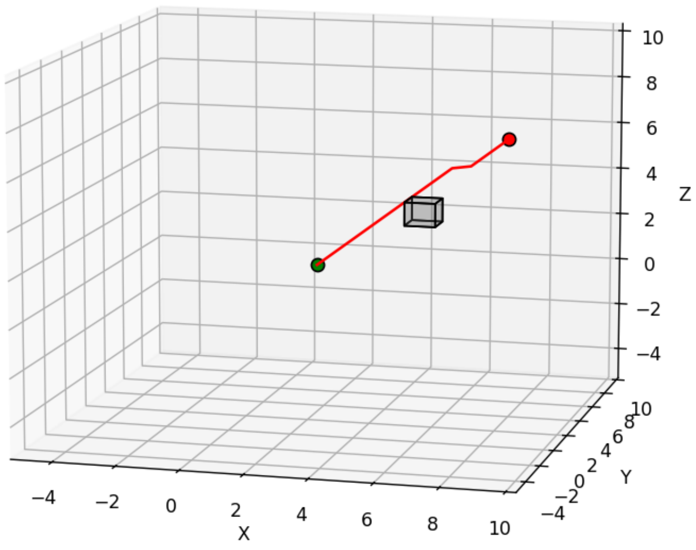
4. Results

1) Search Based
In this case we are running a A* algorithm, the weight is 1 and the resolution of the cell is 0.5 meters. The following are the results

i. Single Cube Case
Info:

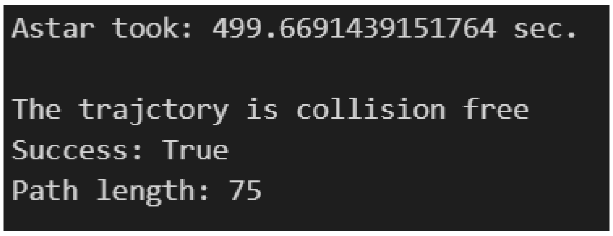


```
Astar took: 0.05010581016540527 sec.

The trajctory is collision free
Success: True
Path length: 8
```

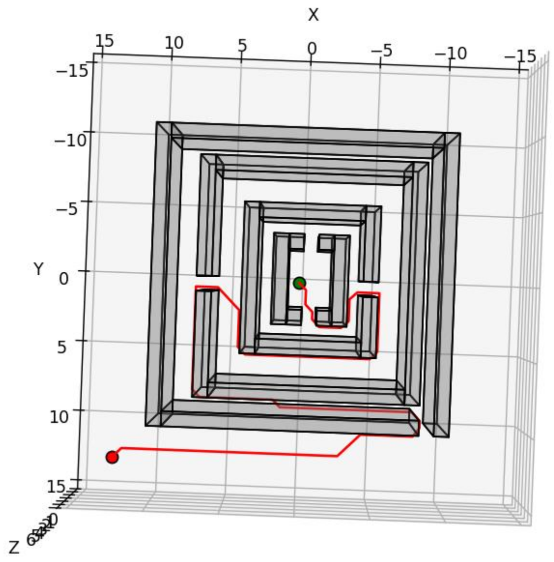


According to the information and the plot above we can, see that the A* works perfectly on this map and it is quick.

ii. Maze Case
Info:



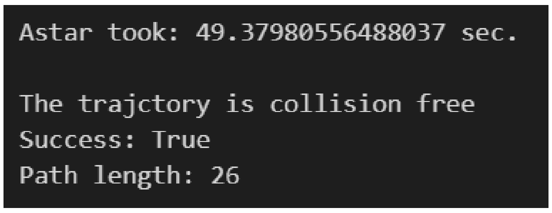
```
Astar took: 499.6691439151764 sec.

The trajctory is collision free
Success: True
Path length: 75
```



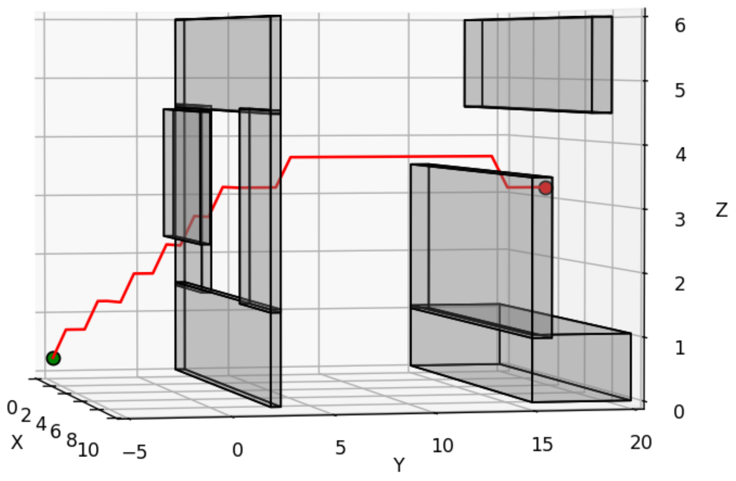
According to the information and the plot above we can, see that the A* works perfectly on this map. Even though its slow, but take consider the complexity, it is still impressive.

iii. Window Case

Info:

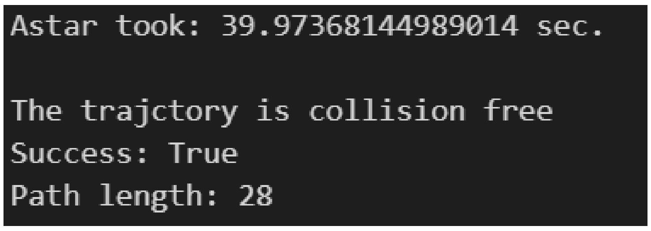


```
Astar took: 49.37980556488037 sec.

The trajctory is collision free
Success: True
Path length: 26
```

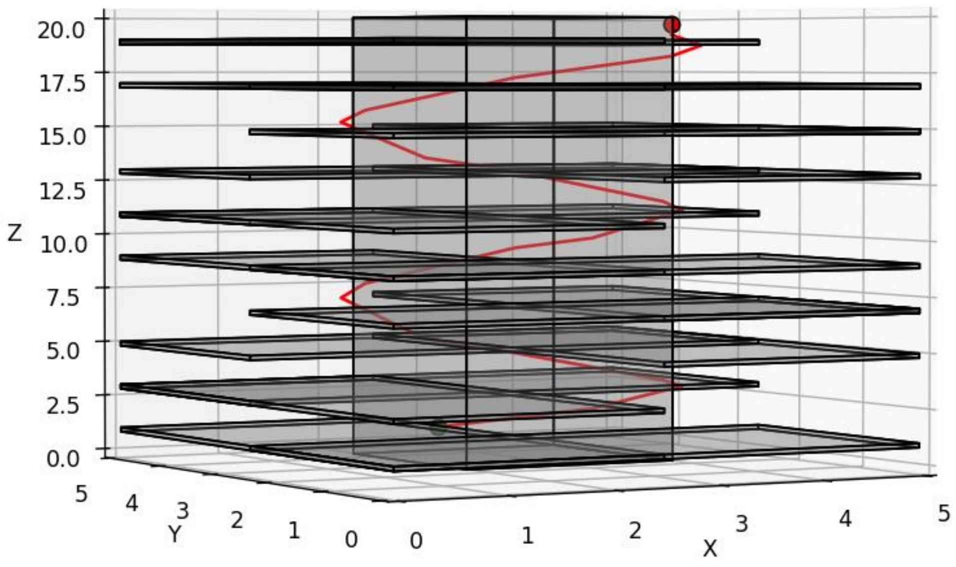


According to the information and the plot above we can, see that the A* works perfectly on this map. We can say it's fast can find the path correctly

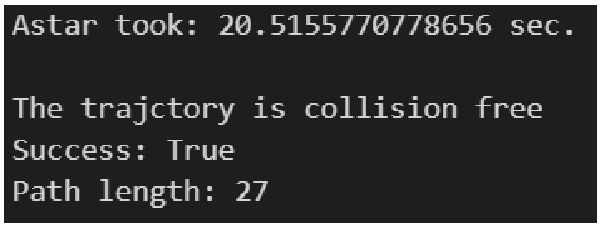
iv. Tower Case

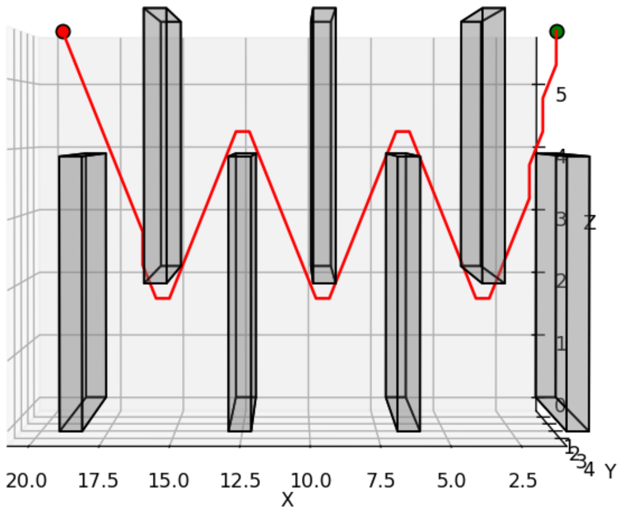


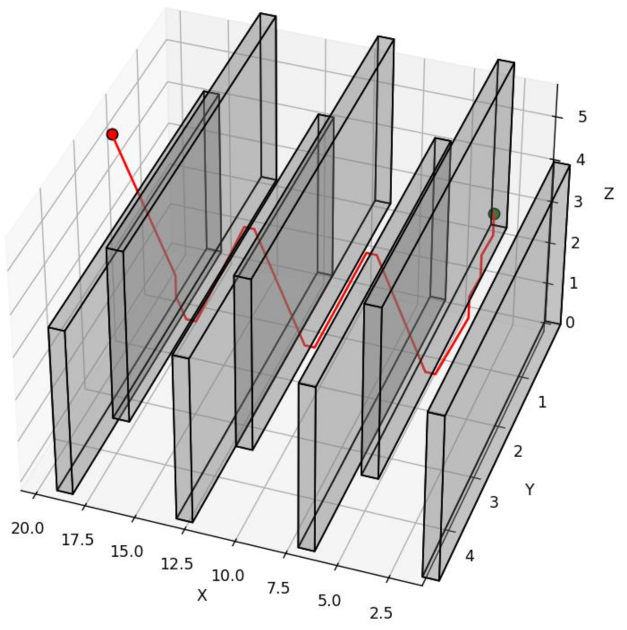
```
Astar took: 39.97368144989014 sec.

The trajctory is collision free
Success: True
Path length: 28
```



v. Flappy Bird Case
Info:

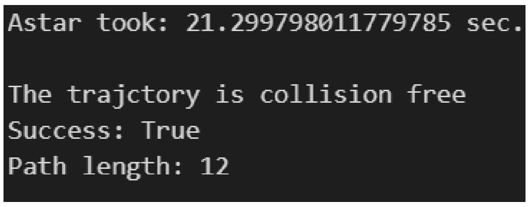

```
Astar took: 20.5155770778656 sec.

The trajctory is collision free
Success: True
Path length: 27
```



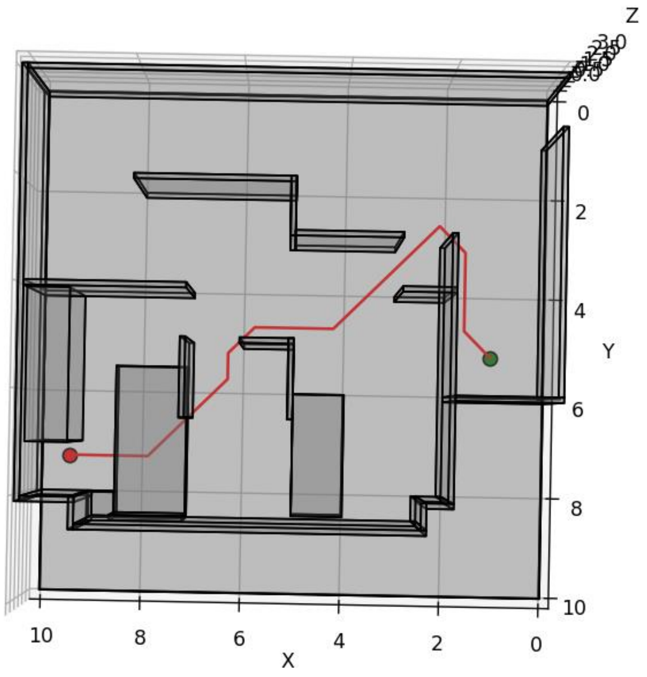


According to the information and the plot above we can, see that the A* works perfectly on this map. We can say it's fast can correctly find the goal.

vi. Room Case

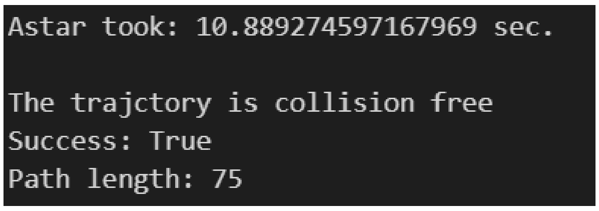


```
Astar took: 21.299798011779785 sec.

The trajctory is collision free
Success: True
Path length: 12
```

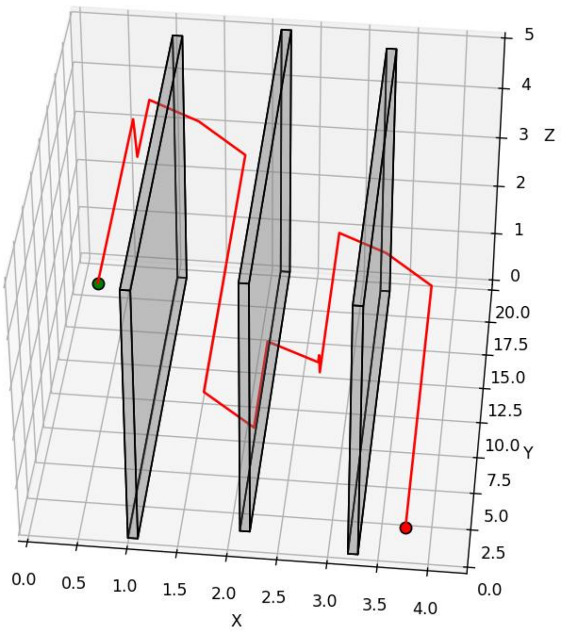


vii. Monza Case

Info：



```
Astar took: 10.889274597167969 sec.

The trajctory is collision free
Success: True
Path length: 75
```



viii. Summary

As we can see here, all the cases can find the path from the start to the goal which prove the reliability of our algorithm.

Also, we found something interesting things during the debugging. At first, there were some cases that works, while other doesn't work due to the OPEN list was empty even before the expansion reach the goal. Eventually, we got the problem solved and we believe it was cause via discretization. Before the problem was solved, everything (goal, start, blocks and boundary) was put into the meters2cells function and being discretized. However, that was not a good choice. Since there could be some case that there is collision in the cells coordinates but no collision meters coordinates or vice versa. Therefore, it was improved by only discretize the goal and the start, while left the blocks and the boundary in meters coordinates. When it's time to do collision checking, the node would be converted into meters.

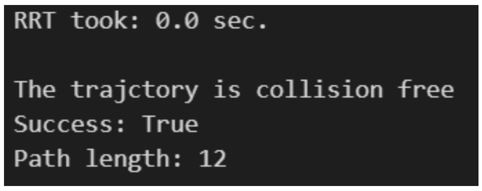
Also, there were some cases that the algorithm found the goal, but were straight penetrating through the obstacles. It was believed due to the wrong way of collision checking. In the last part, the idea of how it is implemented was already discussed. However, there was one special case wasn't being considered of at first. When the line segment is parallel to one of the axes, its projection to that plane is a point which means the length is 0. In this case, no matter where this line segment was, it would be regard as no collision with all the blocks in our old algorithm. Therefore, the modification we made is that if this case happens, only the projection in two other plane is taking in consideration. Eventually, the new algorithm managed to find the right path.

2) Sampling Based

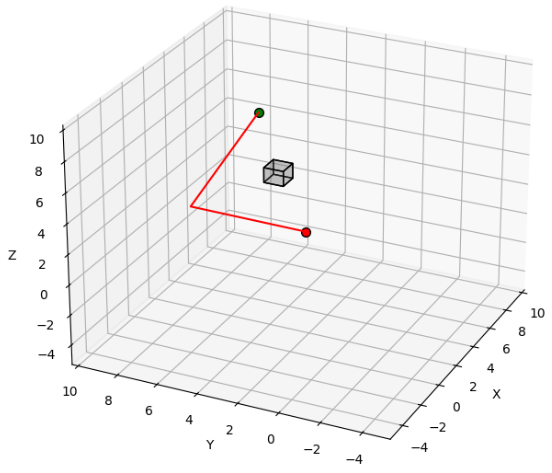
Here we are implementing the RRT algorithm by using a existing library from: https://github.com/motion-planning/rrt-algorithms.

i. Single Cube Case

Info:

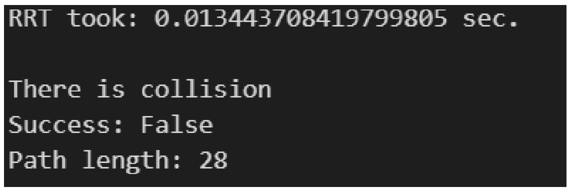


```
RRT took: 0.0 sec.

The trajctory is collision free
Success: True
Path length: 12
```

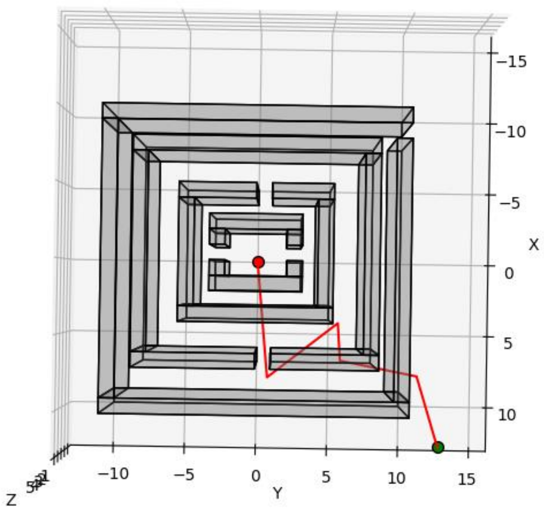


ii. Maze Case

Info:

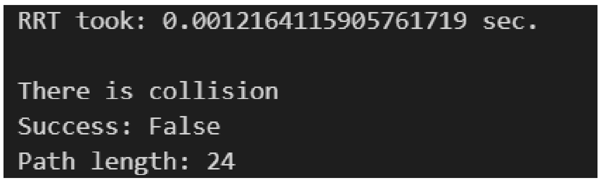


```
RRT took: 0.013443708419799805 sec.

There is collision
Success: False
Path length: 28
```

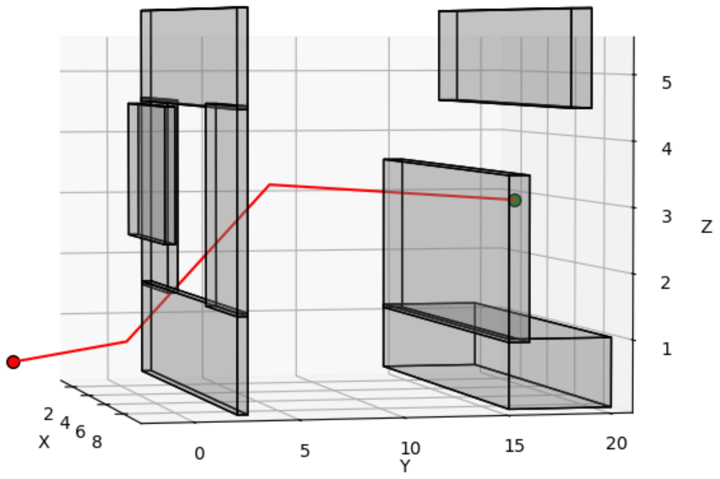


iii. Window Case

Info:

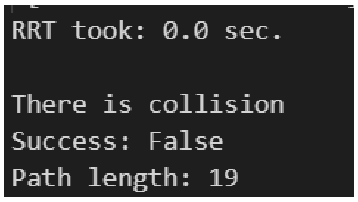


```
RRT took: 0.0012164115905761719 sec.

There is collision
Success: False
Path length: 24
```

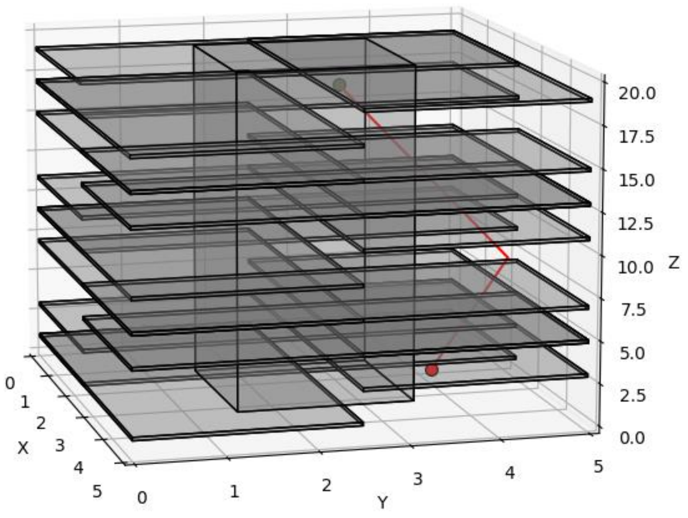


iv. Tower Case

Info:

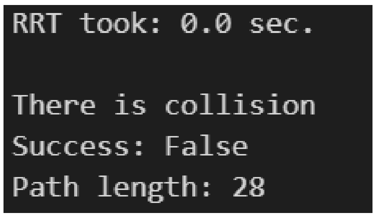


```
RRT took: 0.0 sec.

There is collision
Success: False
Path length: 19
```

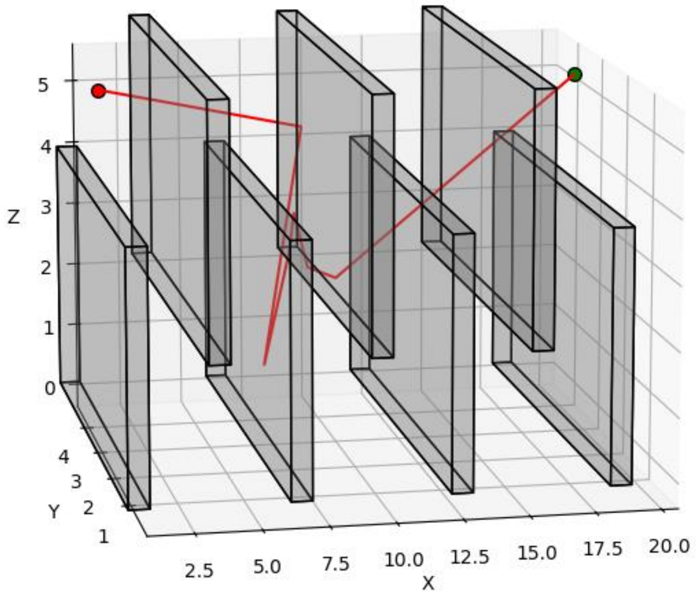


v. Flappy Bird Case

Info:

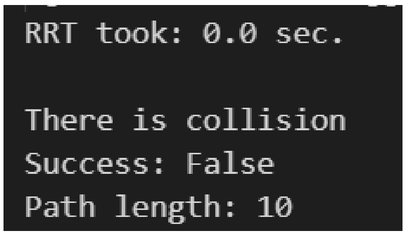


```
RRT took: 0.0 sec.

There is collision
Success: False
Path length: 28
```

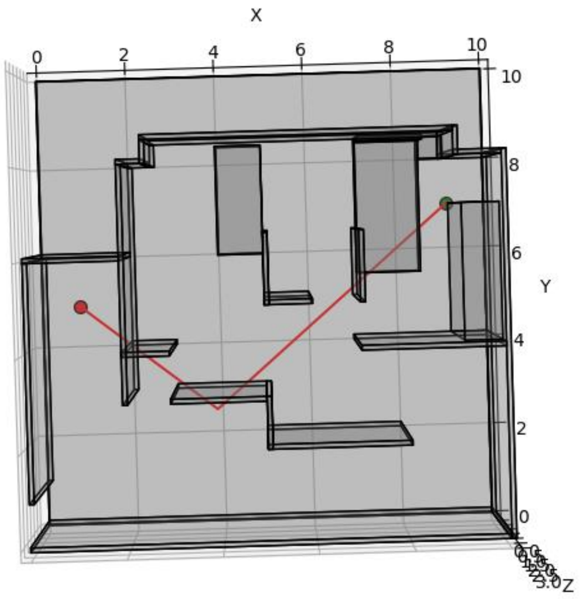


vi. Room Case

Info:

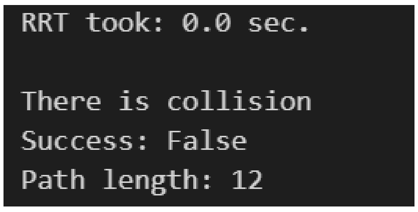


```
RRT took: 0.0 sec.

There is collision
Success: False
Path length: 10
```

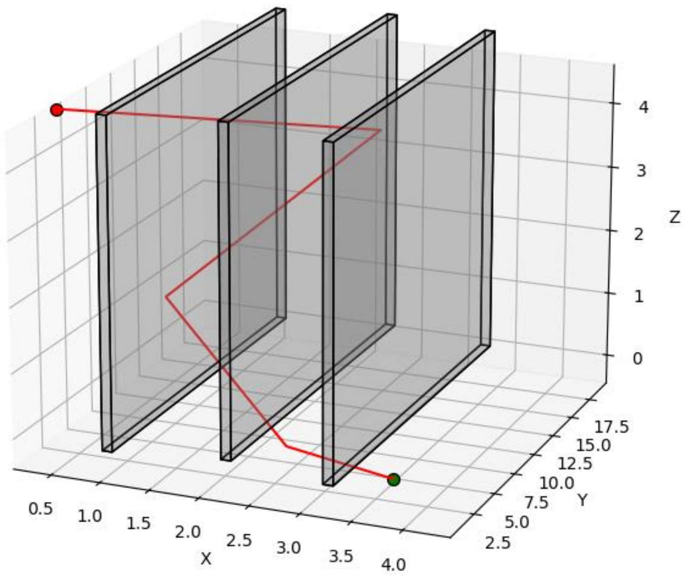


vii. Monza Case

Info:



```
RRT took: 0.0 sec.

There is collision
Success: False
Path length: 12
```



viii. Summary

According to the result we have demonstrated, we can say that RRT can find a path in extremely fast way. However, the it cannot guarantee it is the shortest and it is collision free. Since we did not have a lot of time to tune our algorithm, we can even see that most of the cases were end up having an incorrect path with collision. We tried to change the r (length of smallest edge to check for intersection with obstacles), max_samples (max number of samples to take before timing out) and the q (length of tree edges). But did not end having a better combination that can help generate better paths. While only in the first case which is the single cube, it is collision free. This prove the idea that RRT algorithm can be efficient in simple cases.

Also, our first idea was to try the OMPL (open motion planning library). However, it was kind of hard to install it on windows system and end up stuck on the part of even installing vcpkg (which is used to install OMPL). While it did not end there. We also test it out on our Linux system (Ubuntu 24.04). However, there were some problems to our Ubuntu system which we could not even ran the VScode on it.