# 3D Motion Planning based on A* and RRT

Yuxiang Kang
*Department of Mechanical and Aerospace Engineering*
*University of California, San Diego*
La Jolla, U.S.A
yuk007@ucsd.edu

***Abstract*—This project considers motion planning in 3D Euclidean space. An agent was tasked to move towards the known goal while staying inside the map boundary and avoiding collision with a set of rectangular obstacles. A* and RRT algorithms were implemented in this process for motion planning. Slab method was implemented for ray/AABB collision detection.**

***Index terms*: 3D motion planning, A*, Slab method, RRT**

## I. Introduction

In robotics, motion planning is an important topic as it participates in generating the robot's control policy. A motion planning algorithm should be able to generate a proper control policy for an agent to achieve a required path, given a known environment and certain constraints. This control policy should consider both effectiveness and efficiency. The problem of finding this control policy in a given environment is also called as Deterministic Shortest Path (DSP) problem.

A* algorithm is a graph traversal and path search algorithm commonly used in solving DSP problem. Its main thought is to maintain a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration of main loop, A* determines the node to expand, by finding the node with least expected full cost from start to goal. To expand a node, A* removes it from OPEN list, add its neighboring nodes into OPEN list, and put the expanded node into CLOSED list to avoid repeating search. After the goal node has been expanded, we can reverse along its parenting relationship to find a complete path.

Another commonly used algorithm in motion planning is Rapidly exploring random tree (RRT). RRT is a sample-based planning technique. Its main thought is to construct a tree from random samples with root of start node, and grow until it contains a path to goal node.

Another important aspect in 3D motion planning is collision detection. In this project, we are mainly facing the collision between ray and axis-aligned bounding boxes (AABBs), for which Slab method is a widely used approach, shown in **Fig 1**.

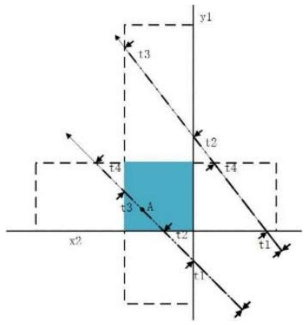

**Figure 1**: Slab method example in 2D environment

## II. Problem Statement

In this specific project, the environment is a 3-D Euclidean space described by a rectangular outer boundary and a set of rectangular obstacle blocks. Each rectangle is geometrically well defined by the coordinates of its lower left corner and its upper right corner. The states of start and goal $\boldsymbol{x}_s, \boldsymbol{x}_\tau \in \mathbb{R}^3$ are also known.

In the process of solving the DSP problem with A* algorithm, we defined the DSP problem as:

- $\mathcal{X}$: State space, defined as
$$\mathcal{X} = \{\boldsymbol{x} | \boldsymbol{x} = [x, y, z]\}$$
- $\mathcal{U}$: Control space, which is used to find the neighboring nodes of a known node. The elements in $\mathcal{U}$ are vectors from a cube's center to 26 points on its surface, shown in **Fig 2**.

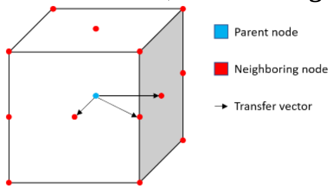

**Figure 2**: Transfer vectors from a node to its neighboring node

- $c_{ij}$: Stage cost. The cost $c_{ij}$ to transfer from node $i$ to node $j$ is set to be the Euclidean distance between the two nodes:
$$c_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$$
- $g_i$: Accumulative cost defined the total cost accumulated along the path from start to current node $i$.
- $h_i$: Heuristic function, defined as the Euclidean distance between the node and goal:
$$h_i = \|\boldsymbol{x}_i - \boldsymbol{x}_\tau\|_2$$
- $f_i$: Estimated cost, defined as:
$$f_i = g_i + h_i$$

## III. Technical approach

After fully defined each element of the DSP problem, it is able for us to implement to implement planning algorithms and generate complete paths from start to goal.

### *A. Collision check*

To check the collision between path $(i, j)$ and a block, we implemented slab method. On each axis, we first determine the near and far slabs, calculate the difference of coordinates between node and slabs, then normalize the difference to ratio. Among three ratio pairs, if the smallest one of far slab ratio is smaller than the largest one of near slab ratio, then we can determine that the ray has no intersections with this block.

### *B. A* algorithm*

As heuristic function, which is Euclidean distance, is only 1-consistent, we implemented unweighted A* algorithm to maintain an optimal path. The algorithm is shown in **Algorithm 1**.

**Algorithm 1** A* Algorithm

```
1: OPEN ← {$\boldsymbol{x}_s$},CLOSED ← {}
2: $g_s = 0, g_i = \infty$ for all $i \in \mathcal{X} \backslash \{\boldsymbol{x}_s\}$
3: while $\|\boldsymbol{x}_i - \boldsymbol{x}_\tau\|_2 > \sigma$ do:
4:      Remove $i$ with smallest $f_i = g_i + h_i$ from OPEN
5:      Insert $i$ into CLOSED
6:      for $j \in$ Children$(i)$ and $j \notin$ CLOSED:
7:          check validity of $(i, j)$
8:          if $g_j > (g_i + c_{ij})$:
9:              $g_j \leftarrow (g_i + c_{ij})$
10:             Parent$(j)$ ← $i$
11:             if $j \in$ OPEN:
12:                 Update priority of $j$
13:             else:
14:                 OPEN← OPEN ∪ {$j$}
```

This Algorithm starts from start node and extend neighboring nodes one node at a time until distance from the currently picked node to goal is within tolerated error $\sigma$, which could ensure the algorithm to reach a termination when the motion model is discrete.

Before we try to update a node's cost and parenting relationship, we checked the node's validity, including whether this node is inside an obstacle and whether the path from the its parent node intersect with a block. With the collision check, we can avoid generating shorter yet flawed route.

For optimality, because we defined the heuristic function as Euclidean distance, which is consistent, the algorithm should technically generate optimal results.

For efficiency, because we just store the nodes that have been considered and proved to be valid in GRAPH, wo do not need to create grid map for the entire space. As a result, this algorithm has better memory efficiency than A* in a complete 3D grid map. As we used "pqdict" to find the node with smallest estimated cost, this algorithm would have good time efficiency.

### *C. RRT algorithm*

When implementing RRT algorithm, we used the "rrt" package in Python motion planning library. Aside from modifying the input data type to match the package, we set the length of smallest edge to check for intersection with obstacles as 0.05 so that the path wouldn't penetrate thin walls. We also increased the max number of samples to take before timing out as the scale of maps is large with respect to the scale of tree edges and some of the maps have complex obstacles.

## IV. Results

### *A. Paths generated by A**

The generated paths for 7 known maps and key values in the searching process with implementation of A* algorithm are shown in **Fig 3~9** and **Chart 1**.

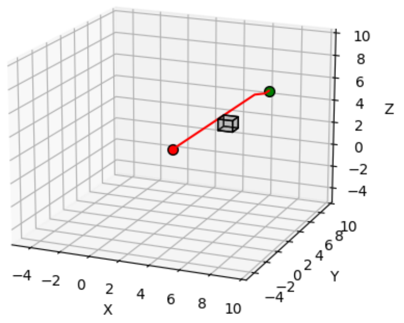

**Figure 3**: Visualized path for map "single_cube" with A*

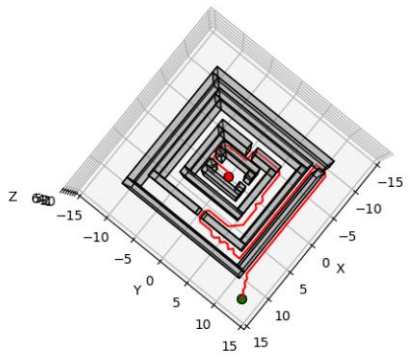

**Figure 4**: Visualized path for map “maze” with A*

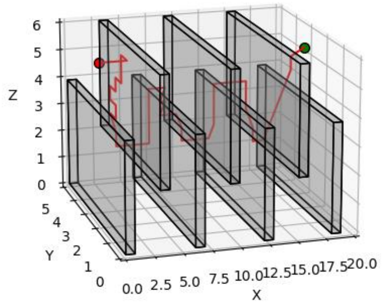

**Figure 5**: Visualized path for map “flappy_bird” with A*

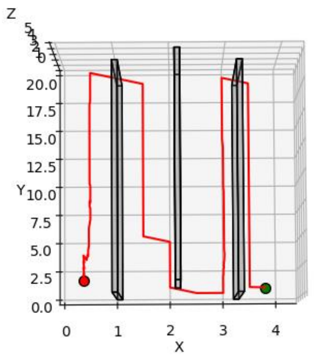

**Figure 6**: Visualized path for map “monza” with A*

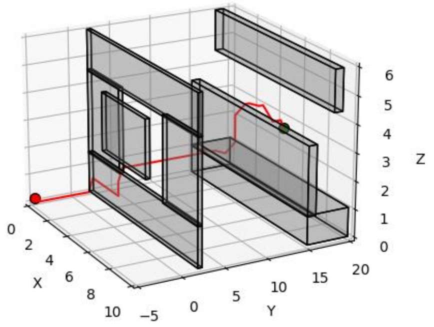

**Figure 7**: Visualized path for map “window” with A*

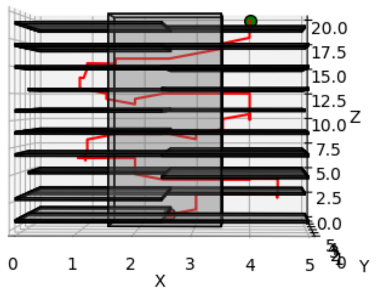

**Figure 8**: Visualized path for map “tower” with A*

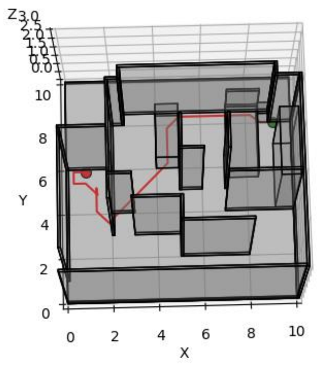

**Figure 9**: Visualized path for map “room” with A*

| Map | Path length | Number of nodes considered | Planning time/s |
|---|---|---|---|
| single_cube | 8.0 | 310 | 0.02 |
| maze | 90.0 | 58588 | 18.11 |
| flappy_bird | 35.6 | 30210 | 3.65 |
| monza | 87.9 | 19833 | 2.61 |
| window | 30.5 | 10292 | 1.22 |
| tower | 44.6 | 19067 | 5.68 |
| room | 13.9 | 1993 | 0.54 |

**Chart 1**: Key values in A* planning process

***B. Paths generated by RRT***

The generated paths for 7 known maps and key values in the searching process with implementation of RRT algorithm are shown in **Fig 10~16** and **Chart 2**.

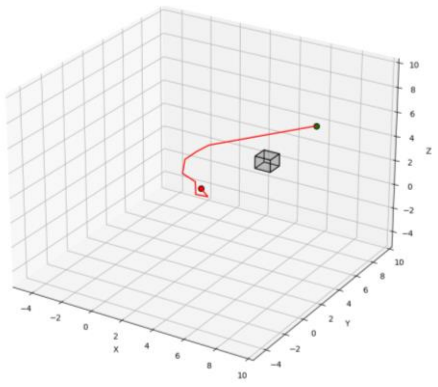

**Figure 10**: Visualized path for map “single_cube” with RRT

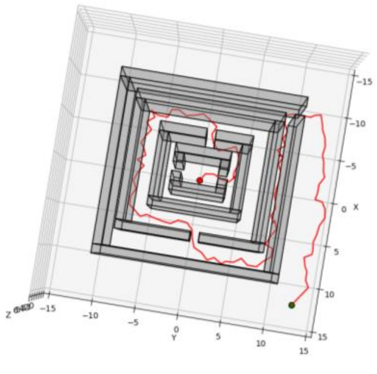

**Figure 11**: Visualized path for map “maze” with RRT

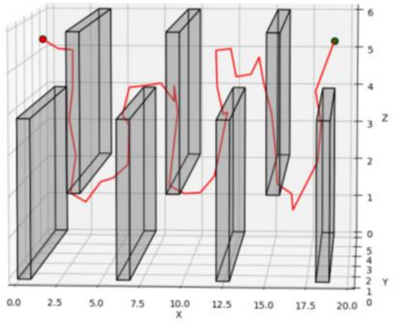

**Figure 12**: Visualized path for map “flappy_bird” with RRT

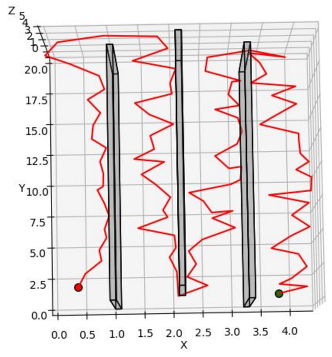

**Figure 13**: Visualized path for map “monza” with RRT

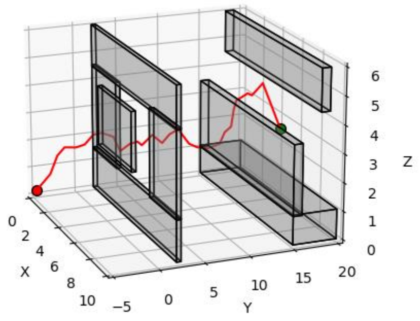

**Figure 14**: Visualized path for map “window” with RRT

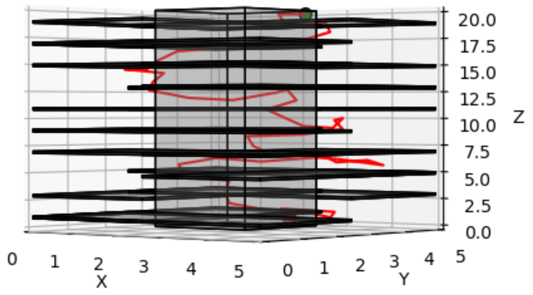

**Figure 15**: Visualized path for map “tower” with RRT

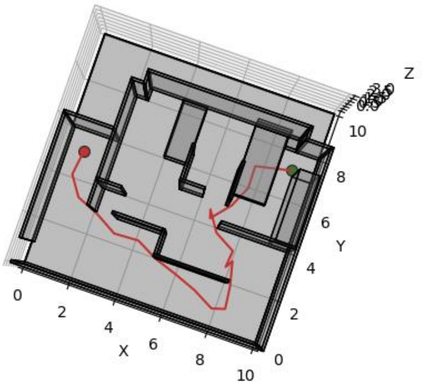

**Figure 16**: Visualized path for map “room” with RRT

| Map | Path length | Number of nodes considered | Planning time/s |
|---|---|---|---|
| single_cube | 13.2 | 31 | 0.01 |
| maze | 117.1 | 11221 | 7.31 |
| flappy_bird | 38.6 | 523 | 0.17 |
| monza | 108.9 | 51407 | 17.23 |
| window | 37.1 | 335 | 0.12 |
| tower | 39.1 | 1582 | 0.55 |
| room | 15.5 | 168 | 0.05 |

**Chart 2**: Key values in RRT planning process

## *C. Discussion*

The motion planning with implementation of A* algorithm and RRT algorithm both worked successfully in all 7 maps, and generated reasonable paths in an acceptable time. To better understand the advantages and disadvantages of each algorithm, we compared the generated paths in the aspects below:

(1) Path length

As we are trying to solve a DSP problem, the length of the generated path is important. **Fig 17** shows the path lengths of each planning process:

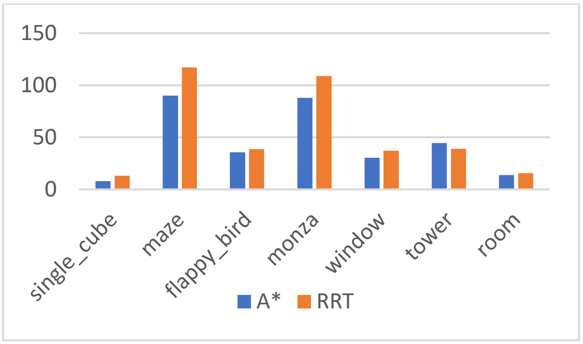

**Figure 17**: Path length from A* and RRT algorithm

According to **Fig 17**, we can see the path generated by A* algorithm is basically shorter than those generated by RRT algorithm, which means A* has better optimality, especially in the maps with long straight distance like “monza”.

(2) Planning time

The planning time for each map is shown in **Fig 18**. We can see the planning time needed in A* is generally longer than in RRT. However, like the path length, RRT shows less efficiency when dealing with the maps with long straight distance.

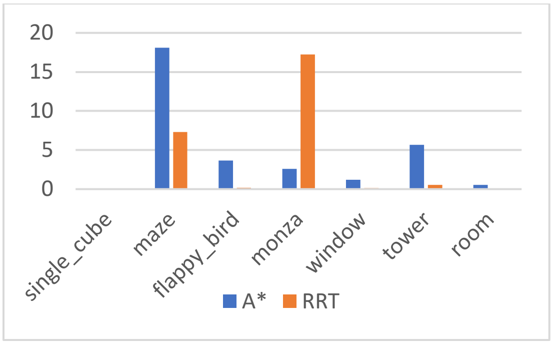

**Figure 18**: Planning time from A* and RRT algorithm

3) Number of nodes considered

When conducting motion planning in large map, it is important to keep the number of nodes considered in a reasonable scale, as it has a strong influence in the program’s memory efficiency. The number of nodes for each map is shown in **Fig 19**, where the numbers are normalized to ratio of the node number in A* to the node number in RRT. We can see that RRT algorithm has a higher time efficiency than A* in most of maps.

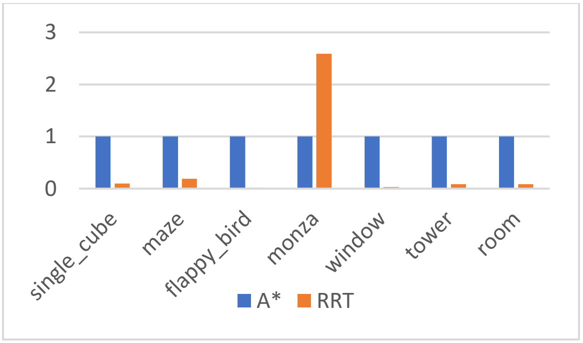

**Figure 19**: Number of nodes considered in A* and RRT algorithm

Generally speaking, as a search-based motion planning algorithm, A* is likely to generate a better optimized path, especially in environments with complex obstacles. However, as a sample-based motion planning algorithm, RRT has better time and memory efficiency in most maps. So, when we are dealing with motion planning in a relatively simply environment, we could first try RRT to generated an acceptable path. When we are conducting motion planning in a complex environment or care more about the path’s optimality, search-based algorithms like A* is a good choice.