# ECE276B Project 2: Motion Planning

Wilson Liao
*Department of Electrical and Computer Engineering*
*University of California, San Diego*
w4liao@ucsd.edu

## I. Introduction

Motion planning, also known as path planning, is a critical problem in robotics. It involves determining a sequence of valid, collision-free motions for a robot or an agent to move from its current state to a desired goal state, while obeying constraints and optimizing certain objectives. Motion planning algorithms take into account the robot's kinematics, dynamics, and the environment in which it operates. The environment is typically represented as a map or a geometric representation, and the robot's state can include its position, orientation, and other relevant variables. Motion planning can be applied to various robotic systems, including autonomous vehicles, industrial manipulators, and humanoid robots. It plays a crucial role in enabling these systems to autonomously navigate and interact with their environment in a safe and efficient manner.

Algorithms for motion planning range from search-based approaches such as A* and Dijkstra's algorithm to sampling-based planners such as Rapidly Exploring Random Trees (RRT) and Probabilistic Roadmaps (PRM) and optimization-based planners (e.g., trajectory optimization using techniques like nonlinear programming or optimal control).

In this project, we compare weighted A*, RRT, RRT*, RRT-Connect algorithms on seven different environments with their advantages and disadvantages.

## II. Problem Formulation

### A. Deterministic shortest path problem

The motion planning problem in this project can be formed as a deterministic shortest path problem.

To implementing A* algorithm, we discretize the environment into 3-d occupancy grid, where we specify obstacles with 1 and free-space with 0. The state space $X$ is the reachable positions (grids with value 0) in the bounded environment. The control space $U$ is defined as actions that can reach to the neighboring positions in the gird including diagonal movements. $U$ can be written as follows:

$$U = \{(dx, dy, dz) \mid dx, dy, dz \in \{+1, 0, -1\}\} \setminus (0, 0, 0) \tag{1}$$

where there are total 26 actions that we can take in each position.

The motion model $f$ can be defined as:

$$x_{t+1} = f(x_t, u_t) = \begin{cases} x_t & \text{if } x_t = \tau \\ x_t + u_t & \text{if } (x_t + u_t \in X) \\ \text{not doing this action} & \text{otherwise} \end{cases} \tag{2}$$

where $\tau$ is the terminal node.

A path is a sequence defined as:

$$i_{1:q} = (i_i, i_2 ..., i_q),\ \forall i_k \in X \tag{3}$$

All paths from $x \in X$ to $\tau \in X$ can be defined as:

$$P_{s,\tau} = \{i_{1:q} \mid i_k \in X, i_1 = s, i_q = \tau\} \tag{4}$$

The path length is the sum of edge weights along the path:

$$J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}} \tag{5}$$

The objective of the problem is to find a path $i^*_{1:q}$ that has the min length from node $s$ to node $\tau$:

$$\textbf{dist}(s, \tau) = \min_{i^*_{1:q} \in P_{s,\tau}} J^{i^{1:q}} \tag{6}$$

$$i^*_{1:q} = \operatorname*{argmin}_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}} \tag{7}$$

## III. Technical Approach

### A. Collision detection

To detect whether our path collides with the obstacles in the environment, we check the path with each obstacle iteratively. We break the path into small segments (a step in the motion planning algorithms), which contains a start point and a end point, and we check whether each segment intersects with the obstacle. The algorithm we adopted can be broke down into three parts:

1) Check whether the start point is in the obstacle, if yes, then return true
2) Find the intersection of the segment and the obstacle on each axis
3) Check whether the intersections are in the obstacle, if at least one of them is in the obstacle, return true

The following is the algorithm we use in the project:

**Algorithm 1** Collision Detection

**Input**: s, $\tau$, min_bound, max_bound $\in R^3$
$dir = \tau - s$
**if** max_bound > s > min_bound **then**
    **return** True
**for** axis $\leftarrow$ 0 to 2 **do**
    **if** $abs(dir$[axis]$) > 0$ **then**
        **if** $dir$[axis] $> 0$ **then**
            $t =$ (min_bound[axis] $- s$[axis]) / $dir$[axis]
        **else**
            $t =$ (max_bound[axis] $- s$[axis]) / $dir$[axis]
    **if** $1 > t > 0$ **then**
        $pt = s + t \cdot dir$
        **for** axis$' \in \{0, 1, 2\}$ \axis **do**
            **if** max_bound[axis$'$] $>$ $pt$[axis$'$] $>$ min_bound[axis$'$] **then**
                **return** True
**return** False

where the min_bound and max_bound are the boundaries of the obstacle.

To check whether the path is collision-free, we check all the segments in the path with a obstacle on this algorithm, and repeat it on every obstacles.

### *B. Weighted A**

In the discretized environment, we defined the edge weights $c_{i_k,i_{k+1}}$ as the distance of the $i_k$ and $i_{k+1}$, which is the length of the control input it takes. Thus, $c_{i_k,i_{k+1}} \in \{1, \sqrt{2}, \sqrt{3}\}$. We adopt the weighted A* algorithm for the DSP problem, the following is the pseudo code of the algorithm.

**Algorithm 2** Weighted A* Algorithm

OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}, \epsilon \geq 1$
$g_s = 0$, $g_i = \infty$ for all $i \in X \setminus \{s\}$
**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**
            $g_j \leftarrow (g_j + c_{ij})$
            Parent($j$) $\leftarrow i$
            **if** $j \in$ OPEN **then**
                Update priority of $j$
            **else**
                OPEN $\leftarrow$ OPEN $\bigcup \{j\}$

where $g_i$ is the estimate of the optimal cost-to-arrive from $s$ to each visited $i \in X$; $h_i$ is the heuristic value from $i$ to $\tau$.

We choose Octile distance as our heuristic function:

$$h_i = \max_k |x_{\tau,k} - x_{i,k}| + (\sqrt{d} - 1) \min_k |x_{\tau,k} - x_{i,k}| \quad (8)$$

The OPEN list is implemented as a min heap to maintain the node with minimum key (f-value) on the top; while the CLOSE list is implemented as a set to store nodes that are popped out of OPEN list.

### *C. Sampling-based planning algorithm*

We implemented a search-based planning algorithm, Weighted A star, in the previous section; however, the search-based algorithms tend to be slow when the search space increases. On the other hand, sampling-based planning algorithm such as RRT can find a feasible path faster than search-based algorithms.

Here, we use the RRT, RRT* and RRT-Connect algorithms from Python motion planning library [1] to compare the results with those using A* algorithm.

The RRT is basically building a tree until the goal is part of it by doing the following steps:

1) Sample a free point $x_{rand}$
2) Find a nearest point $x_{nearest}$ in the current tree and $x_{rand}$
3) Find a new point $x_{new}$ which steers from $x_{nearest}$ towards $x_{rand}$ by a fixed distance $\epsilon$
4) If the segment from $x_{nearest}$ to $x_{new}$ is collision-free, insert $x_{new}$ into the tree
5) Repeat step 1-4 until a point is a distance $\epsilon$ from the goal is generated

The RRT-Connect is similar to RRT, but to build two trees, one starts from the start point, the other starts from the end point. By doing so, the algorithm can find a feasible path faster especially in environments with traps since it covers more sampling area. Besides, the RRT-Connect also attempts to connect the two trees at every iteration.

The RRT* is similar to RRT but it adds a rewire step in RRT to ensure the asymptotic optimality. The RRT* does the following steps:

1) Sample a free point $x_{rand}$
2) Find a nearest point $x_{nearest}$ in the current tree and $x_{rand}$
3) Find a new point $x_{new}$ which steers from $x_{nearest}$ towards $x_{rand}$ by a fixed distance $\epsilon$
4) Extend step: If the segment from $x_{nearest}$ to $x_{new}$ is collision-free, find all nodes within a neighborhood $N$
    a) Let $x_{nearest} = \text{argmin}_{x_{near} \in N} g(x_{near}) + c(x_{near}, x_{new})$ be the node in $N$ on the currently known shortest path from $x_s$ to $x_{new}$
    b) Add node $x_{new}$ and edge $(x_{nearest}, x_{new})$ to the tree
    c) update the label $x_{new}$ to $= g(x_{new}) = g(x_{nearest}) + c(x_{nearset}, x_{new})$
5) Rewire step: Check all nodes $x_{near} \in N$ to see if re-routing through $x_{new}$ reduces the path length
    a) If $g(x_{new}) + c(x_{new}, x_{near}) < g(x_{near})$, then remove the edge between $x_{near}$ and its parent and add a new edge between $x_{near}$ and $x_{new}$
6) Repeat step 1-5 until a point is a distance $\epsilon$ from the goal is generated

The extend step in RRT* only connects the best node in muliple near nodes; while the rewire step is basically doing label correcting for the nodes in the neighborhood.

## IV. Results

### A. Experiment setting

In the experiments, we discretize the environment with resolution equals 0.1 for A* searching.

For RRT series algorithms, we set max steer distance into 0.025 to avoid collisions, and set max number of samples into 50000 to ensure adequate samples covering the environment, especially maps with large constraints like maze and monza, and set probability of checking for a connection to goal to 0.1.

For RRT*, the max number of nearby branches to rewire is set to 32.

### B. Experiment result

The following tables are the result of different planning algorithms test on different test cases, where the x in $A^*$-x indicates the value of $\epsilon$ applied. RRT-Connect yields path exceed the boundary on monza map, thus the results remain blank.

| | single cube | maze | window | tower |
|---|---|---|---|---|
| A*-1 | **8.15** | **74.39** | 26.67 | **28.21** |
| A*-5 | **8.15** | 74.57 | 28.58 | 36.40 |
| RRT | 13.65 | 114.69 | 31.11 | 43.36 |
| RRT* | 8.77 | 78.29 | **24.57** | 32.80 |
| RRT-Connect | 13.00 | 98.37 | 32.02 | 46.03 |
| | flappy bird | room | monza | |
| A*-1 | **25.84** | **11.27** | **75.80** | |
| A*-5 | 31.63 | 11.73 | 76.04 | |
| RRT | 38.62 | 13.35 | 114.37 | |
| RRT* | 28.31 | 12.34 | 78.40 | |
| RRT-Connect | 46.01 | 24.26 | - | |

TABLE I: Path length of different test cases and algorithms

| | single cube | maze | window | tower |
|---|---|---|---|---|
| A*-1 | 0.6017 | 282.96 | 52.15 | 33.06 |
| A*-5 | 0.0179 | 224.14 | 0.21 | 2.86 |
| RRT | 0.0266 | 9.31 | 0.16 | 0.62 |
| RRT* | **0.0076** | 36.24 | 0.85 | 8.63 |
| RRT-Connect | 0.0383 | **5.95** | **0.06** | **0.55** |
| | flappy bird | room | monza | |
| A*-1 | 45.52 | 6.88 | 42.78 | |
| A*-5 | 0.72 | 0.65 | 32.12 | |
| RRT | **0.13** | **0.07** | 15.85 | |
| RRT* | 2.13 | 0.57 | **15.69** | |
| RRT-Connect | 0.22 | 0.08 | - | |

TABLE II: Run time (sec) of different test cases and algorithms

| | single cube | maze | window | tower |
|---|---|---|---|---|
| A*-1 | 3822 | 1385367 | 382211 | 270548 |
| A*-5 | 50 | 1079893 | 1369 | 21908 |
| RRT | 57 | 14198 | 477 | 1713 |
| RRT* | **7** | **9907** | 296 | **989** |
| RRT-Connect | 76 | 15640 | **202** | 1735 |
| | flappy bird | room | monza | |
| A*-1 | 344318 | 51819 | 357122 | |
| A*-5 | 5111 | 5074 | 272158 | |
| RRT | 394 | 270 | 48934 | |
| RRT* | **292** | **112** | **1847** | |
| RRT-Connect | 747 | 227 | - | |

TABLE III: Number of considered nodes of different test cases and algorithms

### C. Discussion

We discuss the performance of different planning algorithms on these aspects: (1) quality of computed paths, (2) run time for searching, (3) number of considered nodes.

For quality of computed paths, we found that RRT series tend to yield path that is not collision-free in difficult environment such as monza, even though we reduce the max steer distance lower than 0.1 (width of obstacles in monza map); while A* generates collision-free path in all test cases. In the aspect of path length, A*-1 is almost the best among all algorithms in all test cases. A*-5 yields slightly longer paths but is much faster. RRT and RRT-Connect yield way longer and twisted paths; while RRT* optimizes the path with the rewiring step and gets much shorter path similar to A*.

For run time of computing paths, we found that RRT series (sampling-based) algorithm is much faster than A* (searching-based) algorithm. A* using larger value of $\epsilon$ tends to be faster than those using smaller value due to the $\epsilon$-consistency. RRT* tends to be slower than RRT and RRT-Connect due to the optimization. Besides, RRT-Connect is even faster in some cases since it use bi-directional sampling. However, in highly constraints environment like maze and monza, RRT series consume much more time compare to other maps.

For number of considered nodes, we found that A* (searching-based) algorithms consider much more nodes than RRT series (sampling-based), which consumes more memory and computation, especially A* with smaller value of $\epsilon$, where using larger value of $\epsilon$ can reduce searching in local min given an appropriate heuristic. RRT* considers fewer nodes comparing to RRT and RRT-Connect since it performs the rewire step correcting the labels of neighboring nodes and update better edges, which can reduce the samples toward end point but consumes more computation in one iteration.

### D. Path visualization

The following figures are the motion planning path visualization of different planning algorithms on seven different environments.

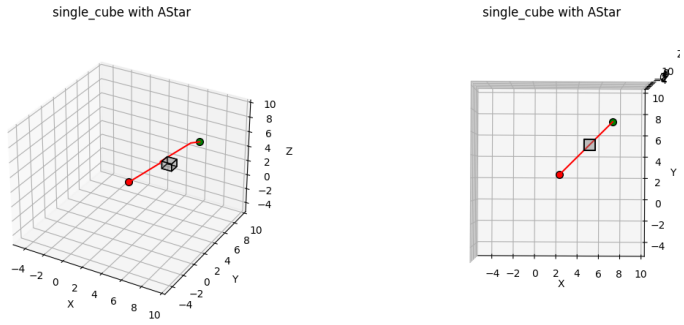


Fig. 1: Single Cube, A*-1

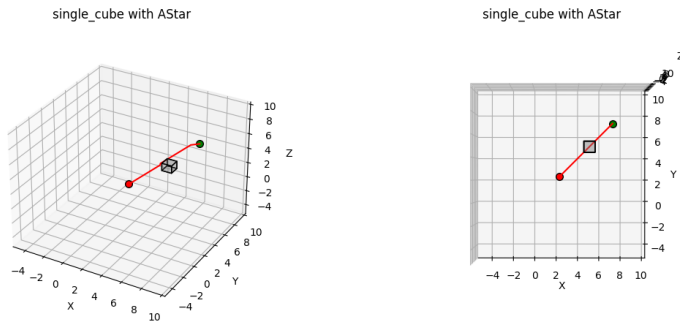


Fig. 2: Single Cube, A*-5

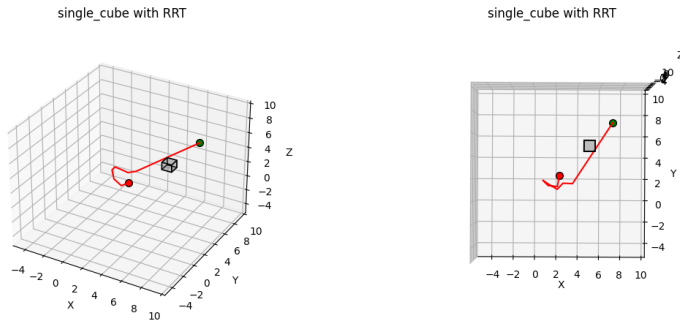


Fig. 3: Single Cube, RRT

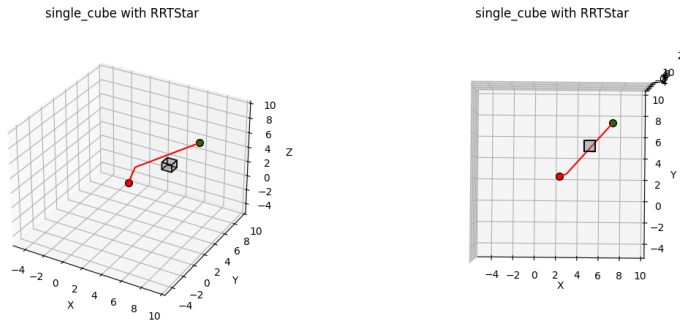


Fig. 4: Single Cube, RRT*

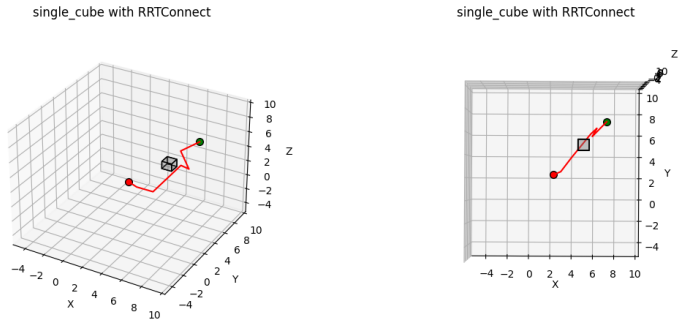


Fig. 5: Single Cube, RRT-Connect

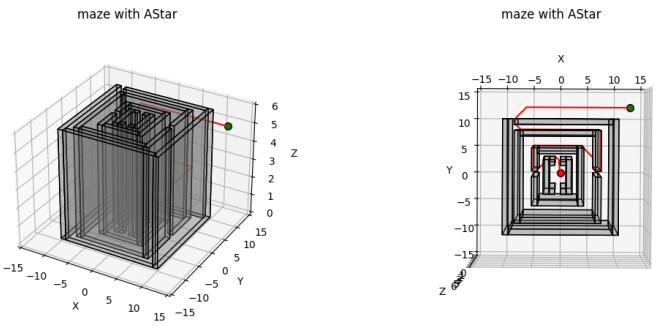


Fig. 6: Maze, A*-1

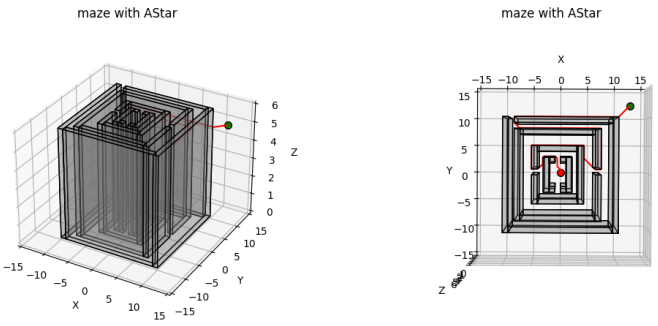


Fig. 7: Maze, A*-5

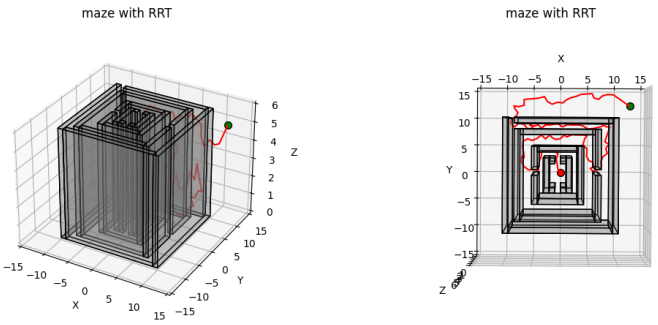


Fig. 8: Maze, RRT

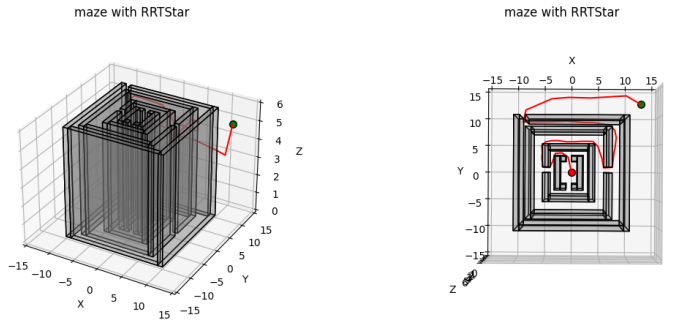


Fig. 9: Maze, RRT*

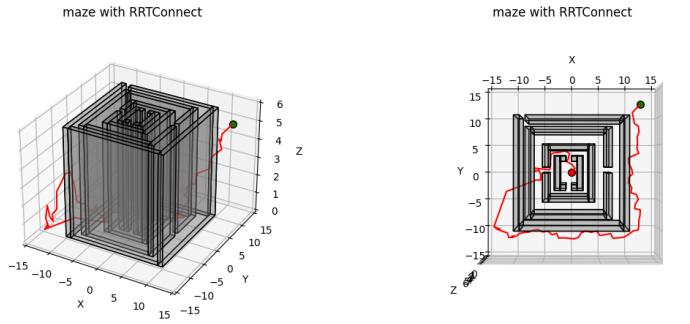


Fig. 10: Maze, RRT-Connect

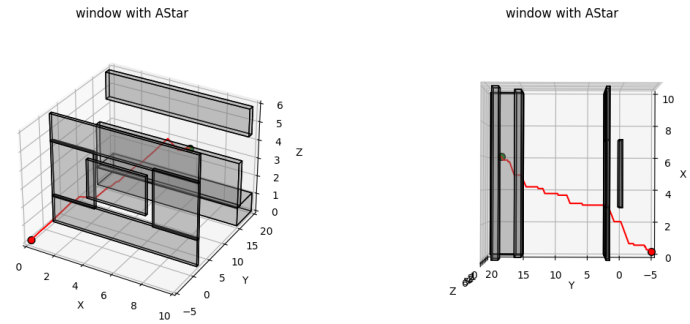


Fig. 11: Window, A*-1

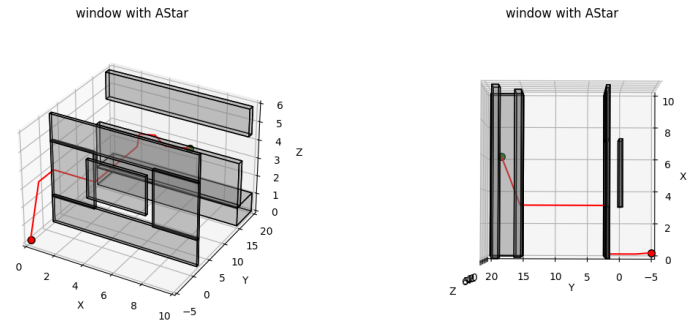


Fig. 12: Window, A*-5

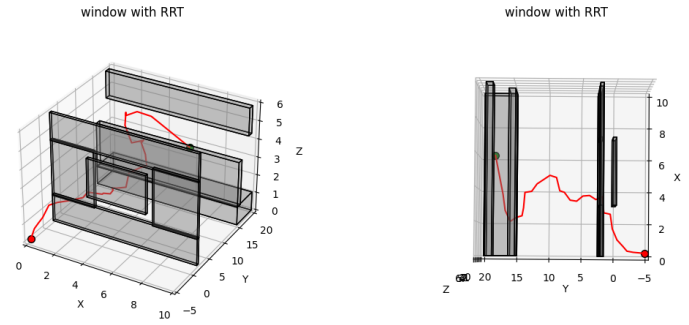


Fig. 13: Window, RRT

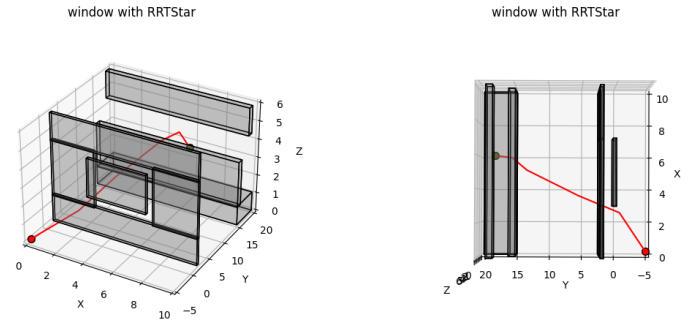


Fig. 14: Window, RRT*

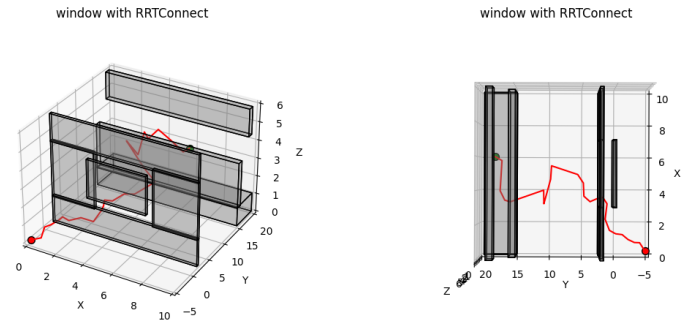


Fig. 15: Window, RRT-Connect

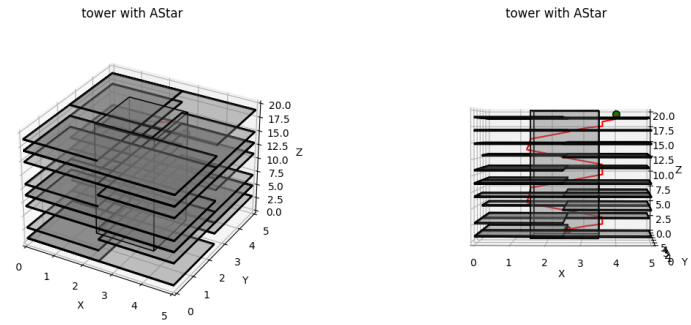


Fig. 16: Tower, A*-1

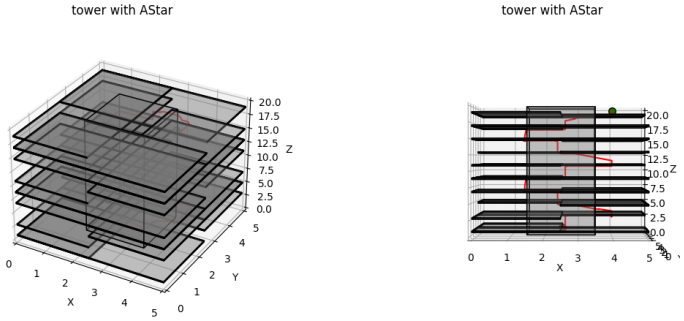


Fig. 17: Tower, A*-5

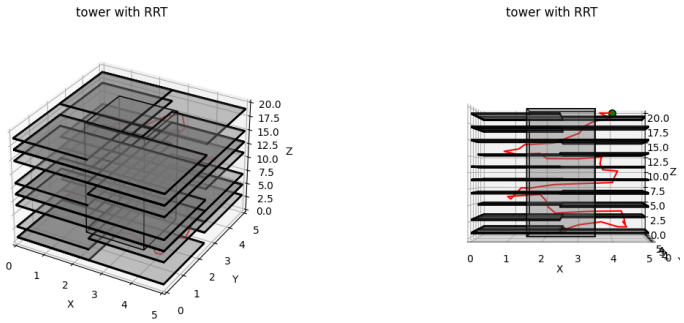


Fig. 18: Tower, RRT

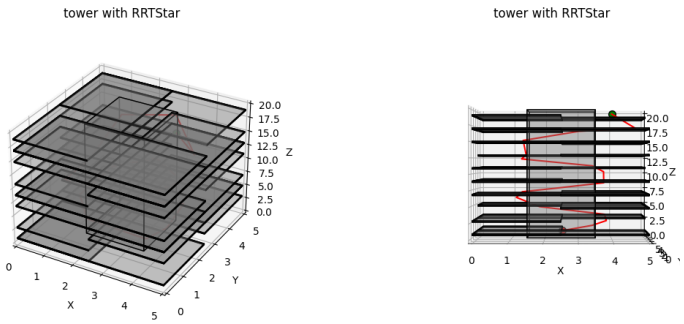


Fig. 19: Tower, RRT*

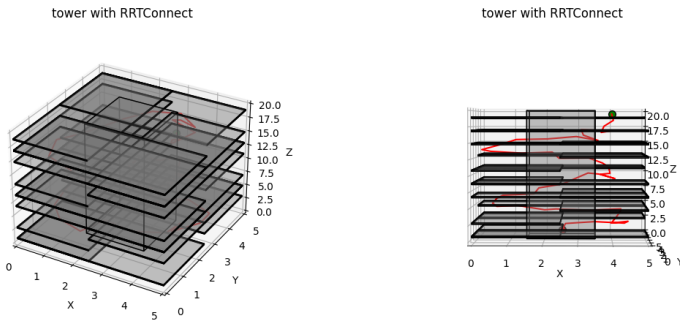


Fig. 20: Tower, RRT-Connect

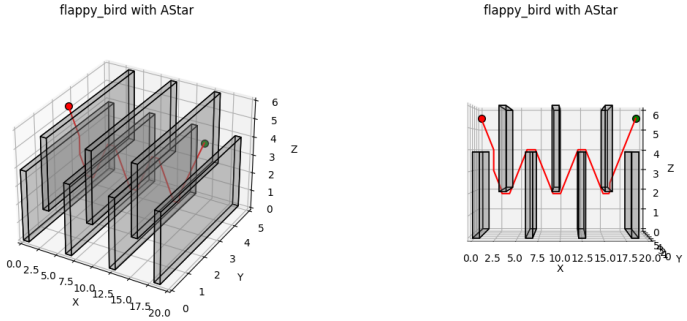


Fig. 21: Flappy Bird, A*-1

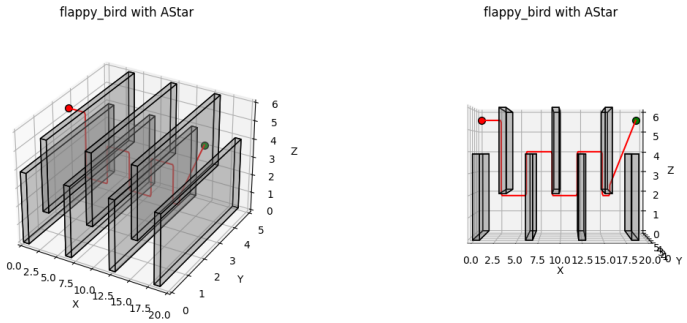


Fig. 22: Flappy Bird, A*-5

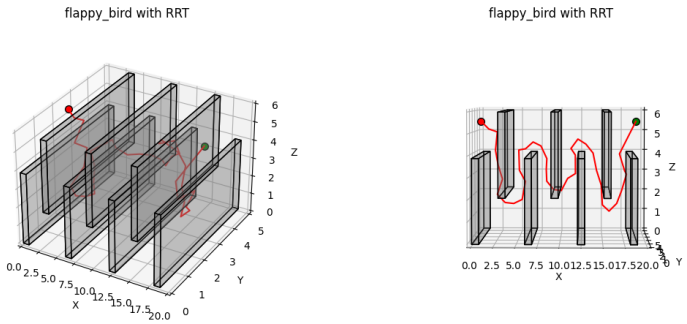


Fig. 23: Flappy Bird, RRT

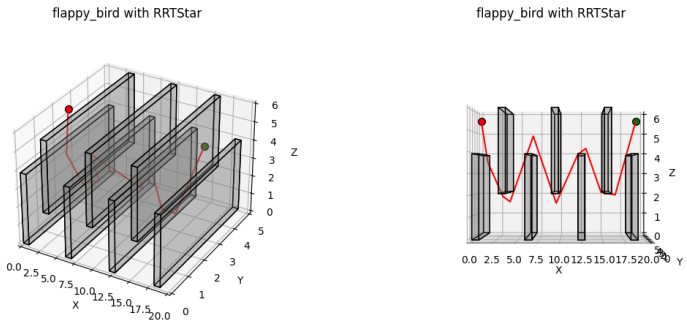


Fig. 24: Flappy Bird, RRT*

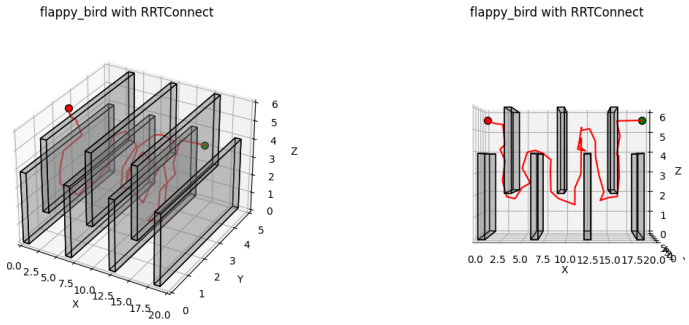


Fig. 25: Flappy Bird, RRT-Connect

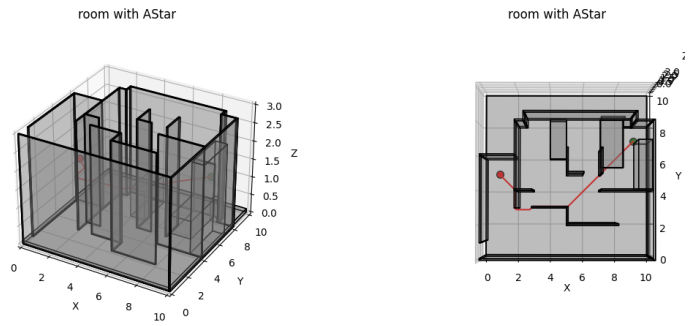


Fig. 26: Room, A*-1

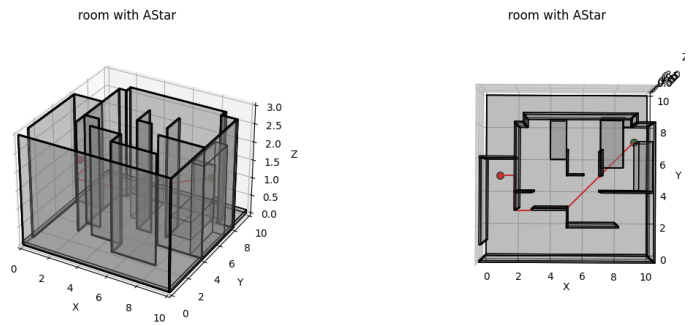


Fig. 27: Room, A*-5

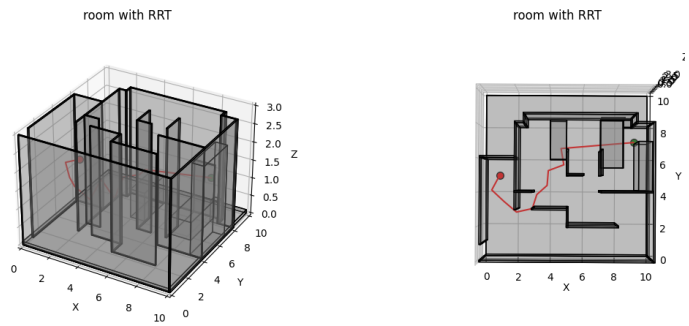


Fig. 28: Room, RRT

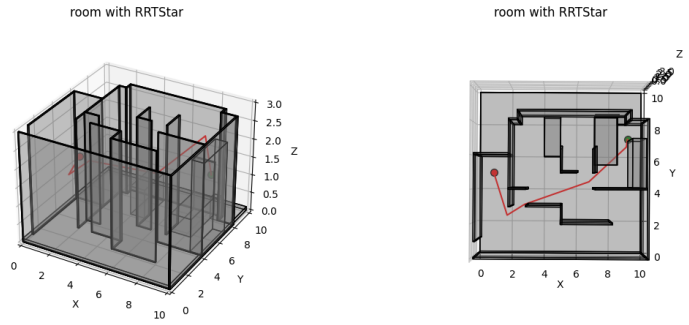


Fig. 29: Room, RRT*

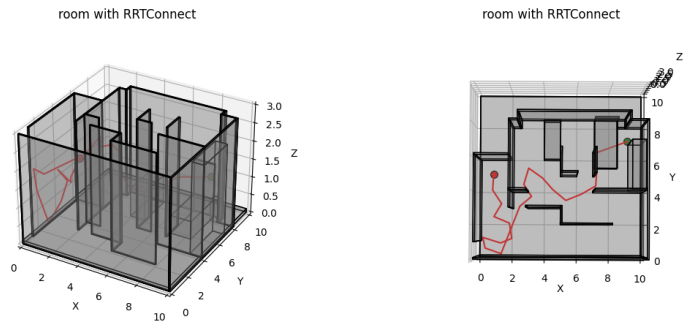


Fig. 30: Room, RRT-Connect

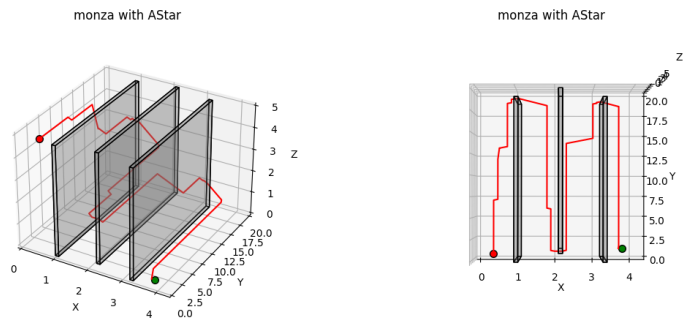


Fig. 31: Monza, A*-1

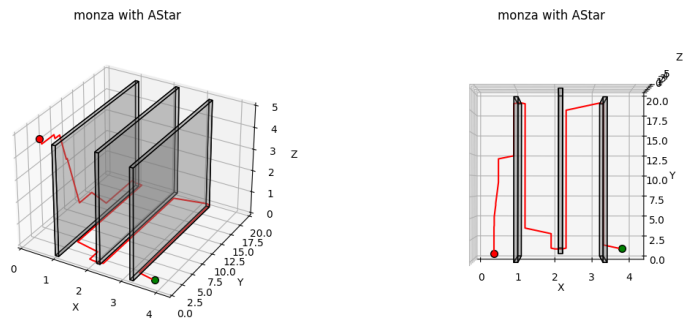


Fig. 32: Monza, A*-5

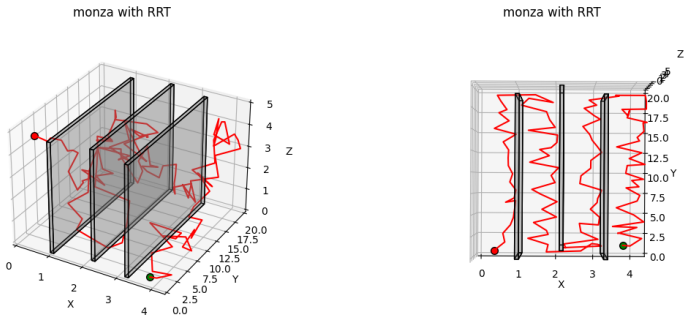


Fig. 33: Monza, RRT

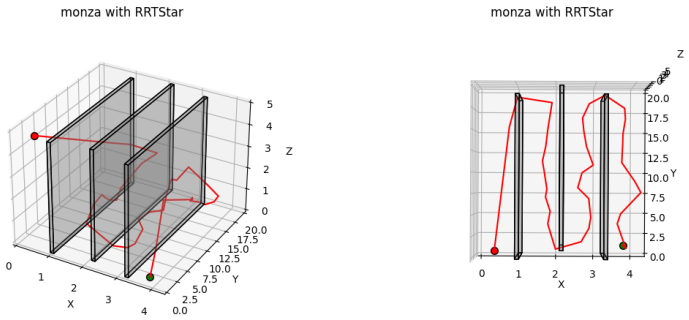


Fig. 34: Monza, RRT*

## V. Reference

[1] Python motion planning library
https://github.com/motion-planning/rrt-algorithms