1 Introduction

In the field of robotics, how to determine the optimal path to the destination is a critical problem because an optimal path can help to reduce energy consumption of the robot, bring down the risk of damage and so on. Label correcting algorithm is one of many algorithm designed for path planning. It can guarantee an optimal path. However, in practice, not only energy but also time is precious. In complicated cases, LCA and other methods based on it is still too slow to balance the time efficiency and other metrics. Therefore, some faster methods, like Rapid-exploring Random Tree, are proposed to boost the speed by providing a sub-optimal path.

In this report, I implemented six algorithms, collision detection based on Minkowski difference and winding number, A*, RRT, RRT*, RRT-Connect and RRT*-Connect, to compare them with seven different environments and discussed their advantages and disadvantages.

2 Problem Statement

The deterministic shortest path problem with obstacles is as follows.

In an environment which is a bounded space $\mathcal{X}$ with several obstacles, $b_1, b_2, ..., b_n \in \mathcal{B}$, with the specified start position s and the goal position τ, try to find the path $f^*(t)$ that

$$f^*(0) = s \tag{1}$$
$$f^*(T) = \tau \tag{2}$$
$$f^*(t) \notin \mathcal{B}\ \forall t \in [0, T] \tag{3}$$
$$f^* = \underset{f}{\operatorname{argmin}} \int_0^T l\left(f(t)\right)\ dt \tag{4}$$

where $l(\cdot)$ is the cost function.

The discrete form of this problem is:

try to find the optimal path $i^*_{1:T} = \underset{i_{1:T}}{\operatorname{argmin}}\ J^{i_{1:T}}$ from all paths from $s \in \mathcal{X}$ to $\tau \in \mathcal{X}$:

$$\mathbb{I}_{s,\tau} = \{i_{i_{1:T}} | i_k \in \mathcal{X}, i_k \notin \mathcal{B}, i_1 = s, i_T = \tau\} \tag{5}$$

Notice that T is not a constant. It can be any poitive real number.

In this report, I defined the path length as the cost function. The path length is given by (4) and its discrete form is defined as the sum of arc lengths over the path

$$J^{i_{1:T}} = \sum_{k=1}^{T-1} c_{i_k, i_{k+1}} \tag{6}$$

3 Technical Approach

3.1 Collision Detection

3.1.1 2D: minkowski difference

Minkowski difference is well-known for detecting the interference of two objects. To understand Minkowski difference, we need to introduce Minkowski sum at first.

Given two sets A and B which represent two objects, the Minkowski sum is

$$A \oplus B = \{a + b \,|a \in A, b \in B\} \tag{7}$$

Similarly, the Minkowski difference is defined as

$$A \ominus B = \{a - b \,|a \in A, b \in B\} \tag{8}$$

We can regard $-b$ as the mirror of b with respect to the origin.

From figure (1) and (2), we can see that when the line intersects with the rectangle, the Minkowski difference contains the origin.

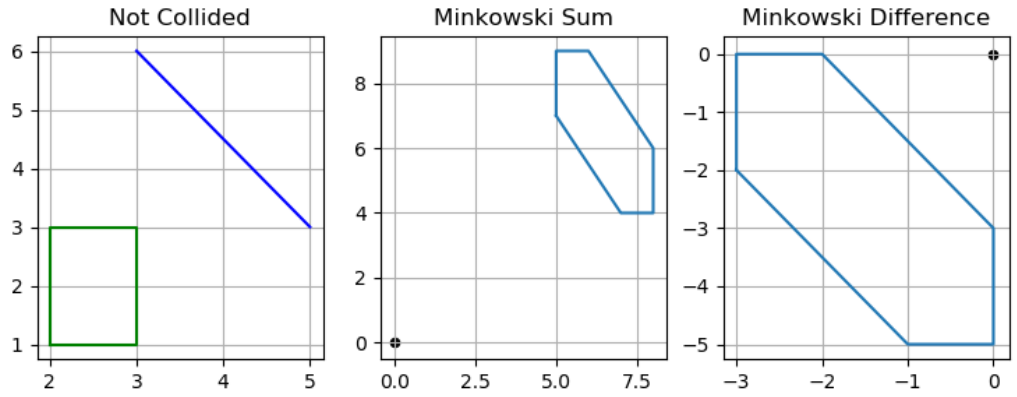

Figure 1: 2D Case of No Collision

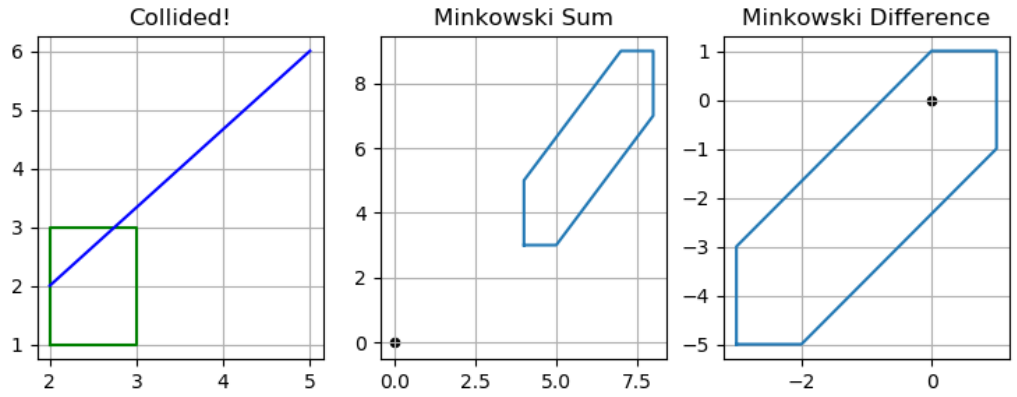

Figure 2: 2D Case of Collision

3.1.2 3D: projection to 2D

In 3D, we can easily split the problem into three 2D collision detection problems by projecting the objects to three planes, XY, YZ, ZX. From figure (3, 4, 5), we can see that only when all the projection results show collision does the box collides with the line segment.

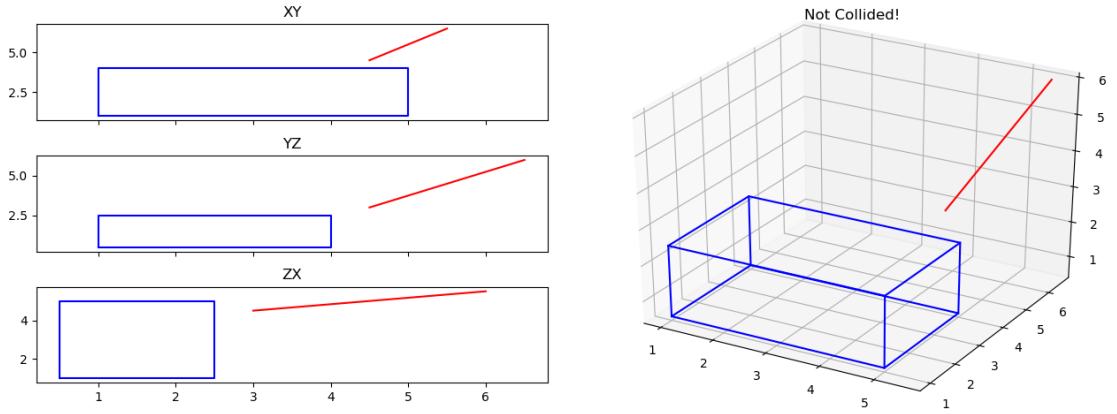

Figure 3: 3D Case 1 of No Collision

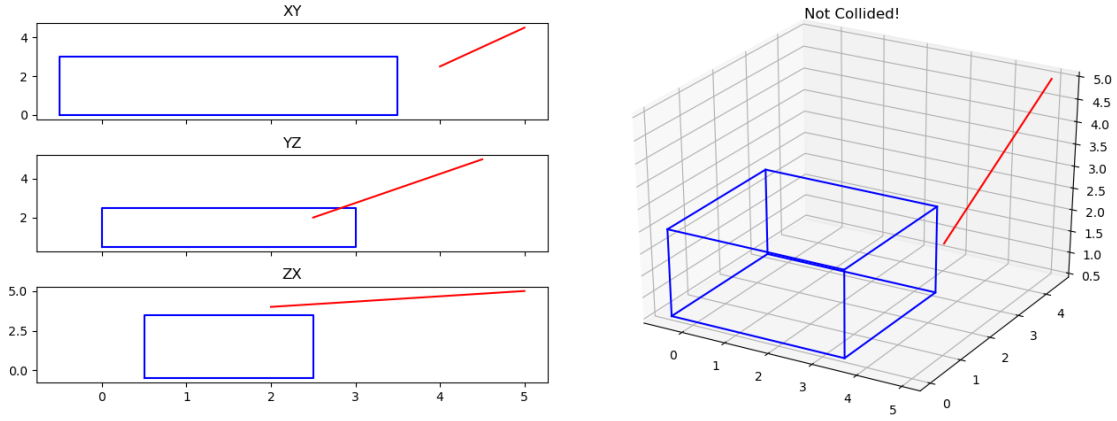

Figure 4: 3D Case 2 of No Collision

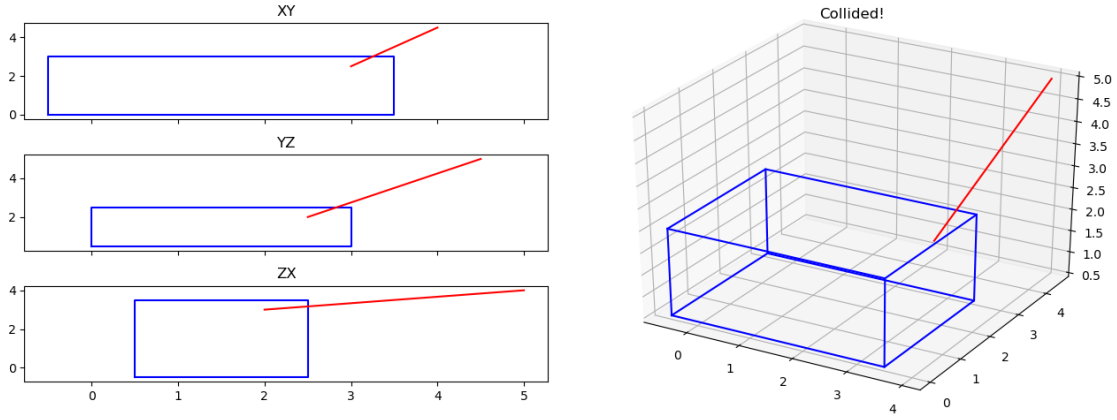

Figure 5: 3D Case of Collision

3.2 Winding Number

To detect whether the Minkowski difference contains the origin, we can use winding number. When the winding number is 0, the origin is on the edge of the polygon or outside the polygon. Otherwise, the origin is inside the polygon. The method to calculate the winding number is as follows.

Algorithm: Winding Number

INPUT: point $P = (x, y)$ and vertices of a polygon V
OUTPUT: the winding number W
W $\leftarrow 0$
for every edge of the polygon
    v_i, v_{i+1} are two ends of the edge
    $(x_1, y_1) \leftarrow v_i$, $(x_2, y_2) \leftarrow v_{i+1}$
    if $y_1 \leq y$
        if $y_2 > y$ and ISLEFT(v_i, v_{i+1}, P)
            W += 1
        else if $y_2 \leq y$ and ISRIGHT(v_i, v_{i+1}, P)
            W -= 1
RETURN W

FUNCTION ISLEFT(p1, p2, p)
    RETURN $[p_1p_2 \times p_1p]_z > 0$
FUNCTION ISRIGHT(p1, p2, p)
    RETURN $[p_1p_2 \times p_1p]_z < 0$

3.3 A*

A* is based on Label Correcting Algorithm. In this report, I implemented the A* with ϵ-consistency. Compared with LCA, the differences are

- Use OPEN and CLOSE set.
- Usd f values which combine g value and ϵ-consistent heuristic to set the priorities of elements in the OPEN set.

3.4 RRT, RRT*, RRT-Connect, RRT*-Connect

A* is a search-based algorithm, which may be slow when the space to search is too huge. RRT, RRT*, RRT-Connect, RRT*-Connect all aims to find out a feasible path as fast as possible.

RRT is the basic one. And RRT* tries to rewire existing samples to optimize the path simultaneously. However, we find that how fast the tree of samples expand in the space depends on how wide the tree distributes in the space. Therefore, RRT-Connect uses two trees, one from the start, the other from the goal, to speed up the expansion of sampled area so that an availabel path can be found faster. But its path is almost as long as RRT's because there is not rewiring. Therefore, we can combine RRT* and RRT-Connect to get RRT*-Connect.

My implementation of RRT, RRT*, RRT-Connect and RRT*-Connect is based on rrt-algorithm. But rrt-algorithm has serious problem of collision detection. It detects collision in a discrete way, which is possible to ignore some collisions. Also, there are some mistakes such as checking whether the goal is reached with a probability instead of every cycle. And the code structure is a little messy. For example, the part of checking if goal is reached and generating the path includes several functions that can be simplify. Therefore, I took its idea of building a RRTBase and then building other RRT algoritms based on it and implemented RRT, RRT*, RRT-Connect and RRT*-Connect with my own collision detection.

4 Results

4.1 Performance

In the experiment, A* searches the space discretized with presision of 0.5 and RRT series algorithms search in the continuous space. The max steer distance is 0.5 and the probability of using the goal as the sample point is 0.1. For RRT*, to guarantee the performance overtime, the maxmium number of samples to rewire in each cycle is 10. The maximum number of samples to used for RRT series in every test is 50000. For every planner and every test case, the result is an average of 5 experiments.

Because A* searches in a discrete space of 0.5 precision, its path is sub-optimal as shown in table (2). But its result is almost the best among the five planners. A* can guarantee a relatively optimal path although it takes a long time in complicated cases. A* with 5-consistency is much faster than A* with 1-consistency and its path length is still almost the same as A* with 1-consistency.

Compared with A*, RRT and RRT* may fail in difficult environment even they are allowed to sample a lot. RRT series are much faster than A* in most cases. RRT-Connect makes it even faster with bi-directional sampling. And from table (4), we can find that RRT-Connect requires the least samples to find the path, which is helpful when the memory resource is limited. However, RRT and RRT-Connect both give longer path because they don't optimize their sample connections online. RRT* solves this problem by rewiring neighbors of the new sample in every cycle, but it becomes much slower.

I tried to get both the bonus of RRT* and RRT-Connect. So, I built RRT*-Connect and tested it. The results show that RRT*-Connect can be very fast in most cases and still give a relatively short path. However, in difficlut cases like maze and monza, RRT*-Connect can become very slow. For example, RRT*-Connect spent 239 seconds in maze environment.

Overall, A* with 5-consistency is the best one that balances the trade off between path optimality and planning speed. In real application, it is a very good strategy to do path planning with A* starting from a high ϵ and then optimize the result with a lower ϵ if there is time left.

	single cube	maze	window	tower	flappy bird	room	monza
A*-1	1.00	1.00	1.00	1.00	1.00	1.00	1.00
A*-5	1.00	1.00	1.00	1.00	1.00	1.00	1.00
RRT	1.00	1.00	1.00	1.00	1.00	1.00	0.40
RRT*	1.00	1.00	1.00	1.00	1.00	1.00	0.40
RRT-Connect	1.00	1.00	1.00	1.00	1.00	1.00	1.00
RRT*-Connect	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 1: Success Rate of Different Test Cases and Planners

	single cube	maze	window	tower	flappy bird	room	monza
A*-1	8.3228	79.2922	27.0644	32.8003	25.2990	12.0711	78.0345
A*-5	8.3228	81.4596	27.2772	39.3595	29.7635	12.6569	78.4054
RRT	10.1618	120.7628	31.1166	43.8659	39.1512	19.5159	107.3766
RRT*	8.5647	79.7764	24.8592	31.2900	28.7028	14.8443	76.7844
RRT-Connect	8.2989	121.6182	30.9931	45.4622	42.0410	13.8632	105.9050
RRT*-Connect	8.1787	88.5241	26.4305	33.4949	29.2203	14.8110	77.5948

Table 2: Path Length of Different Test Cases and Planners

	single cube	maze	window	tower	flappy bird	room	monza
A*-1	0.2478	102.0771	30.3749	27.0703	19.9399	3.8537	11.1336
A*-5	0.0298	74.0040	0.3849	3.1811	3.2664	1.0540	8.0914
RRT	0.0403	40.0786	0.2529	2.2224	0.4834	0.4695	34.0580
RRT*	0.1721	170.0572	0.8659	10.8379	2.4077	2.1930	243.9128
RRT-Connect	0.0033	36.9188	0.1047	1.9129	0.3925	0.0883	15.2249
RRT*-Connect	0.0044	238.9776	0.4052	7.1060	1.1278	1.2086	57.4051

Table 3: Used Time of Different Test Cases and Planners

	single cube	maze	window	tower	flappy bird	room	monza
A*-1	437	9992	5624	2684	3822	566	3892
A*-5	190	7210	518	920	1379	279	2887
RRT	114	14450	347	1347	535	260	42736
RRT*	126	14118	224	1244	560	247	43482
RRT-Connect	20	21418	137	831	386	63	15461
RRT*-Connect	21	24899	135	823	318	164	15468

Table 4: Used Samples of Different Test Cases and Planners

4.2 Result Visualization

The view of looking from top shows that the path crosses obstacles because of occlusion. And from these visualization results, we can see that A* always give the most smooth path.

4.2.1 Single Cube

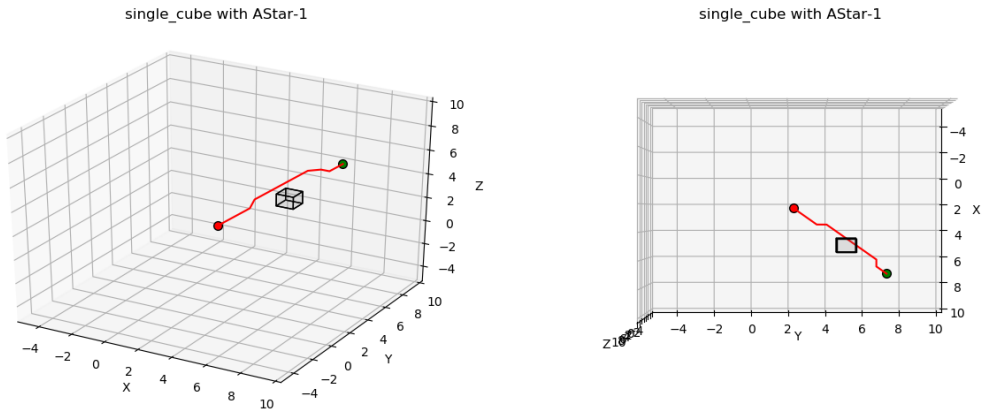

Figure 6: Single Cube, A*, $\epsilon = 1$

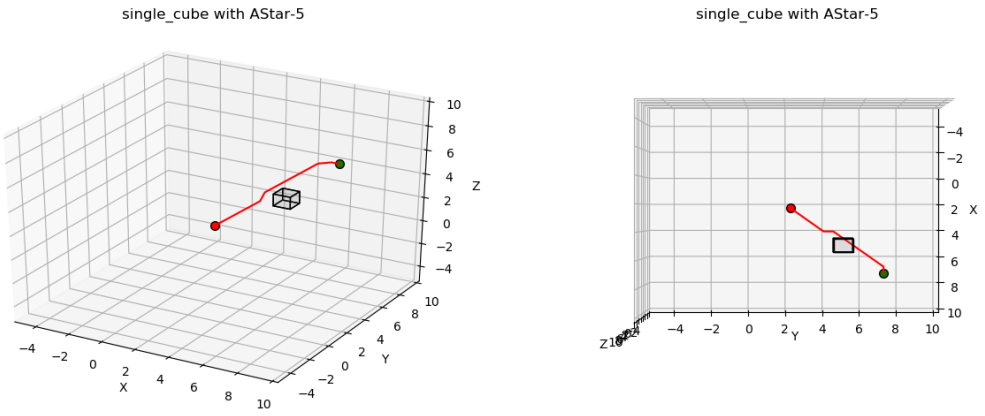

Figure 7: Single Cube, A*, $\epsilon = 5$

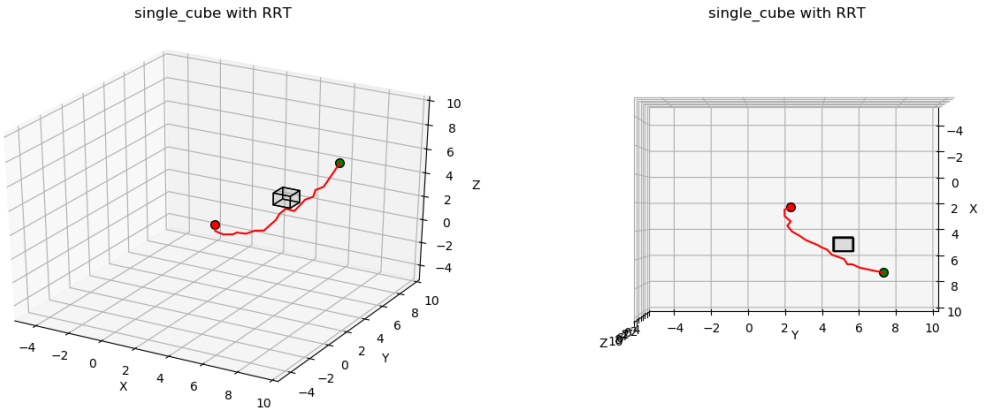

Figure 8: Single Cube, RRT

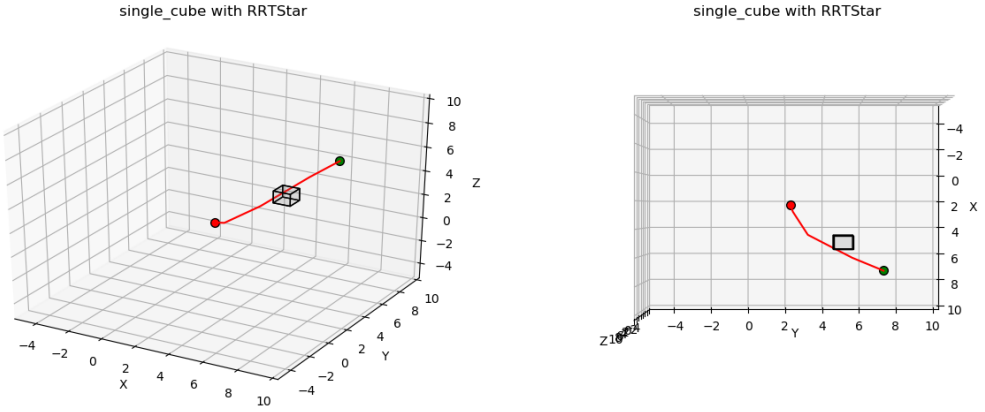

Figure 9: Single Cube, RRT*

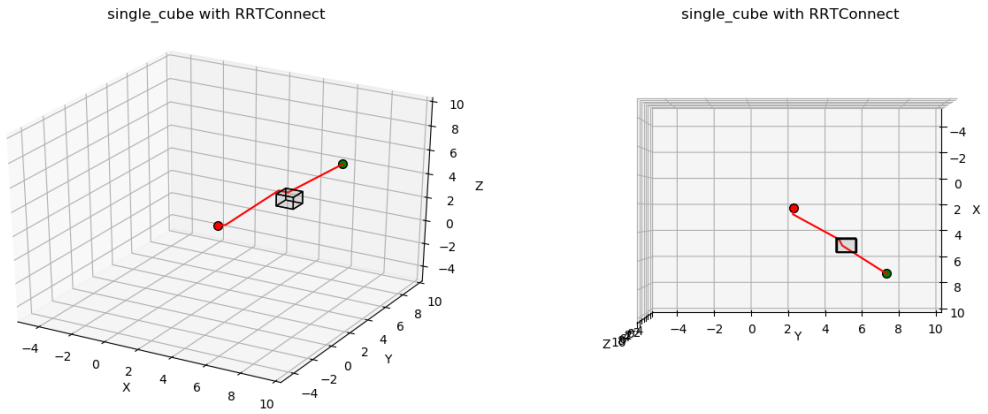

Figure 10: Single Cube, RRT-Connect

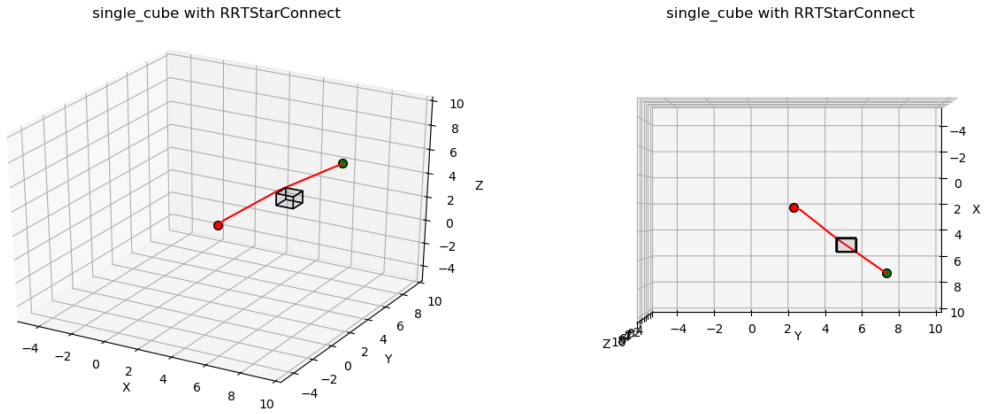

Figure 11: Single Cube, RRT*-Connect

4.2.2 Maze

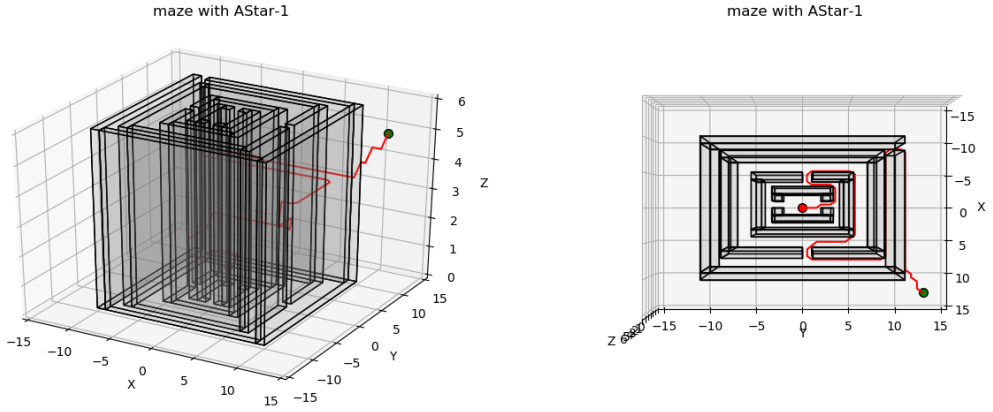

Figure 12: Maze, A*, $\epsilon = 1$

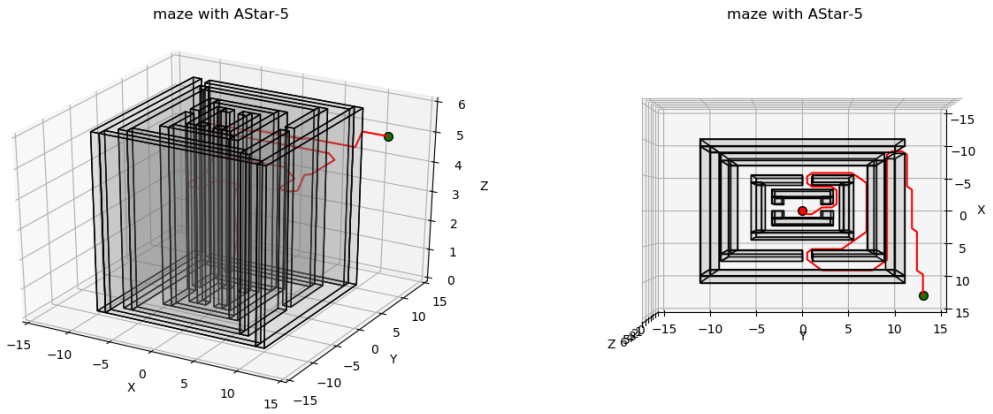

Figure 13: Maze, A*, $\epsilon = 5$

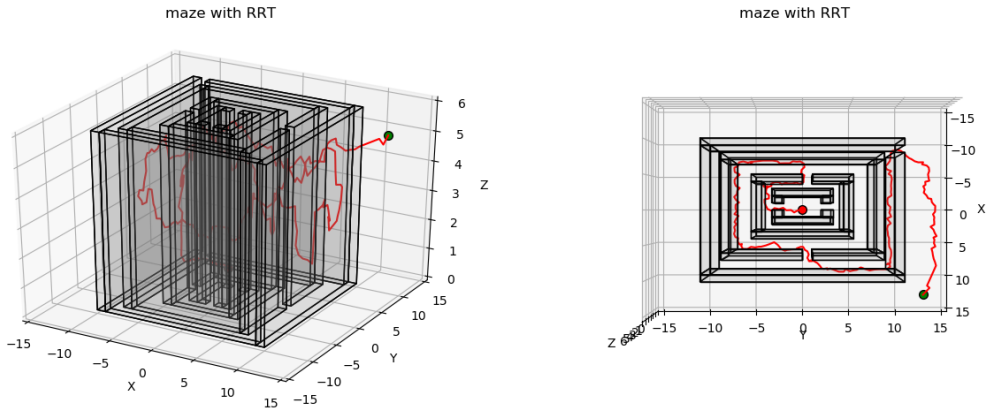

Figure 14: Maze, RRT

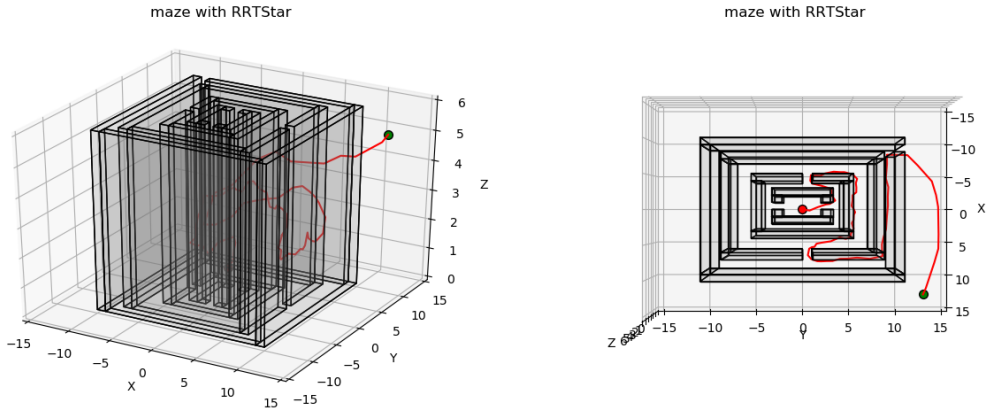

Figure 15: Maze, RRT*

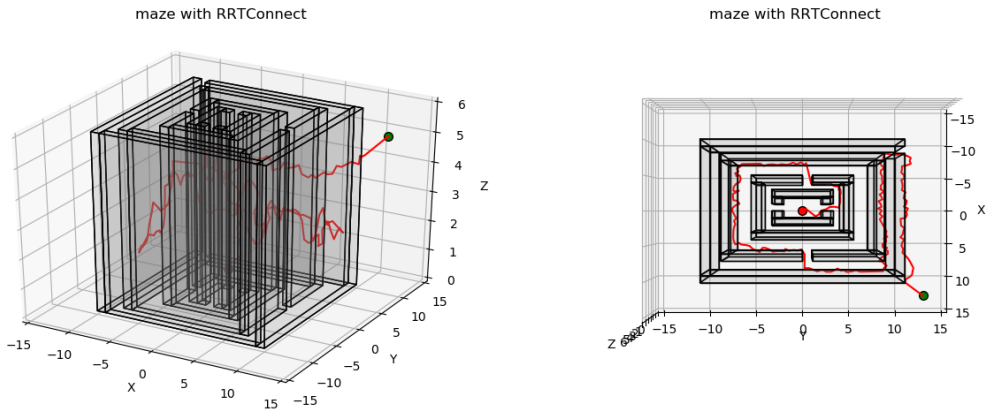

Figure 16: Maze, RRT-Connect

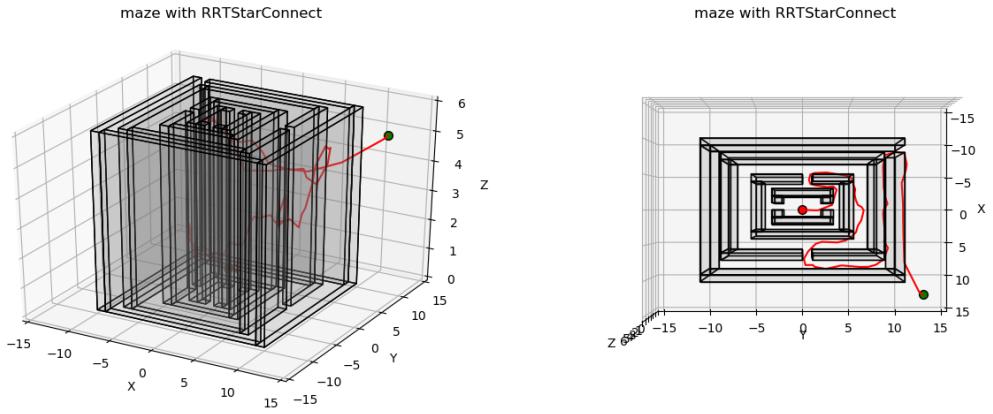

Figure 17: Maze, RRT*-Connect

4.2.3 Window

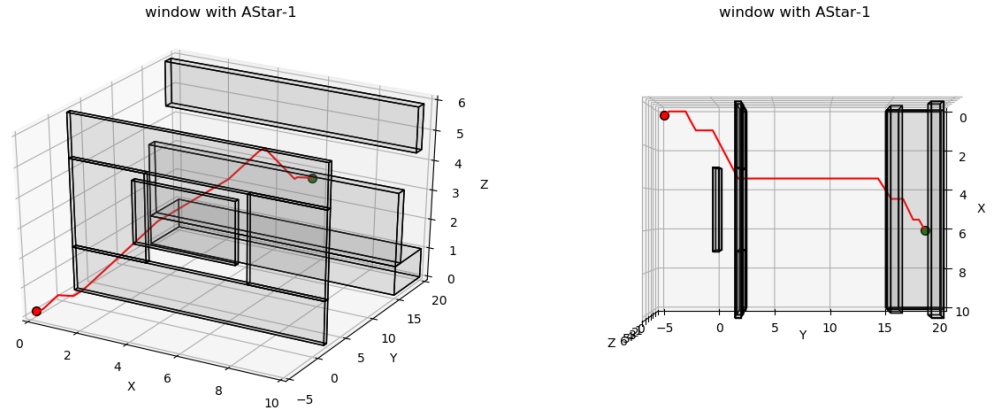

Figure 18: Window, A*, $\epsilon = 1$

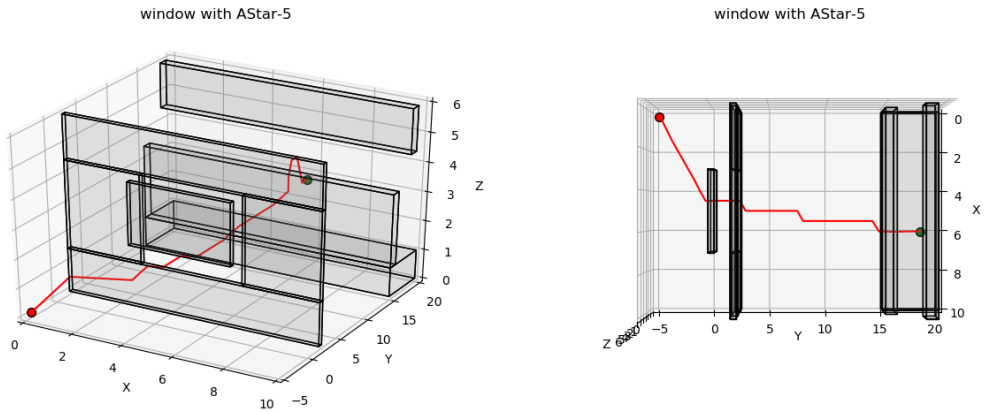

Figure 19: Window, A*, $\epsilon = 5$

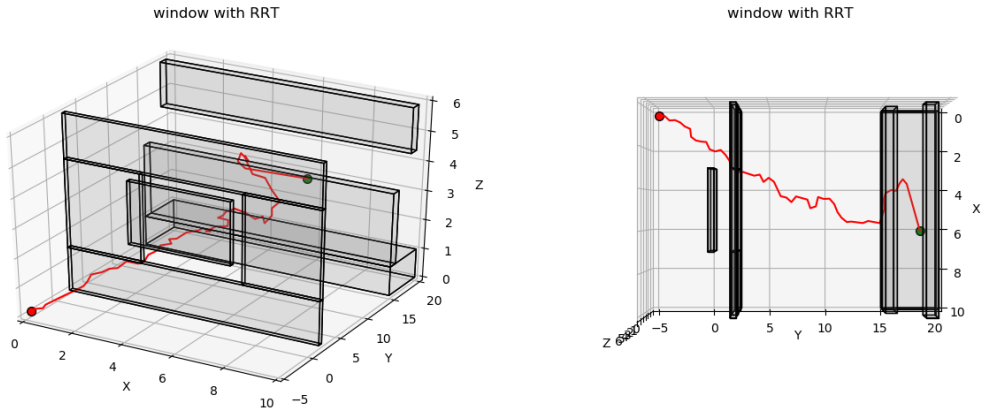

Figure 20: Window, RRT

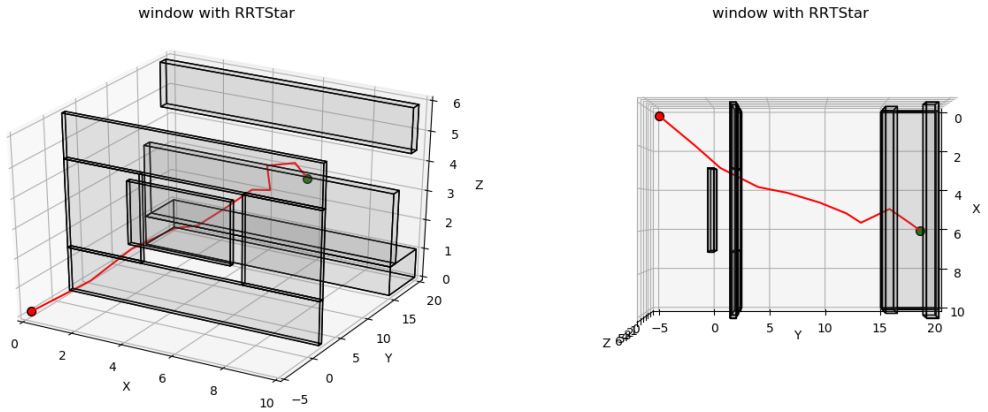

Figure 21: Window, RRT*

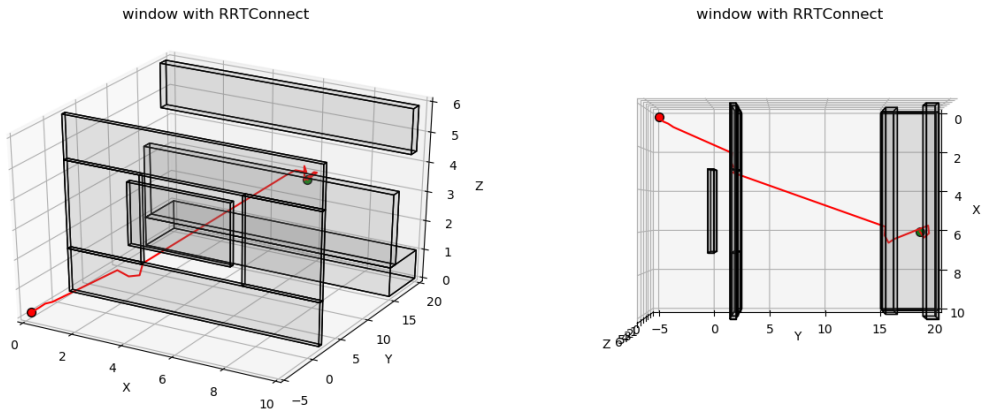

Figure 22: Window, RRT-Connect

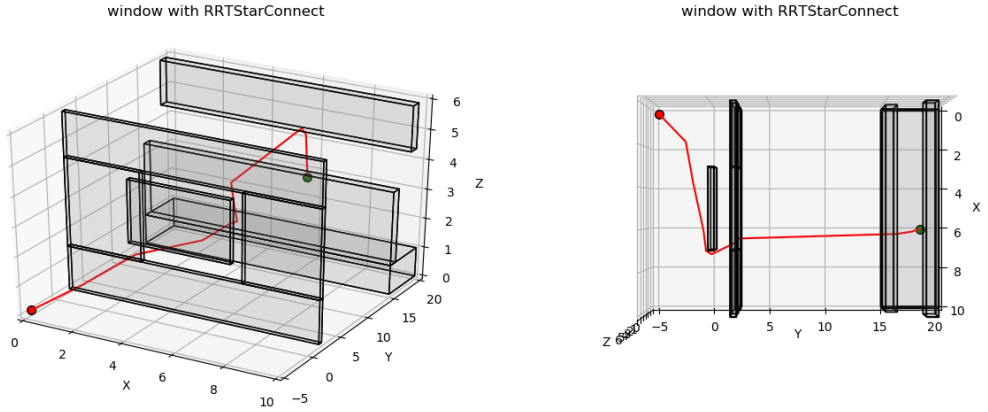

Figure 23: Window, RRT*-Connect

4.2.4 Tower

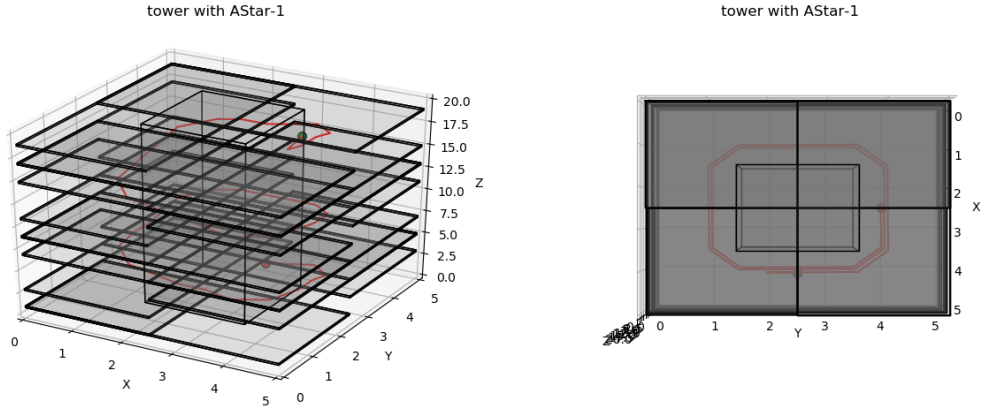

Figure 24: Tower, A*, $\epsilon = 1$

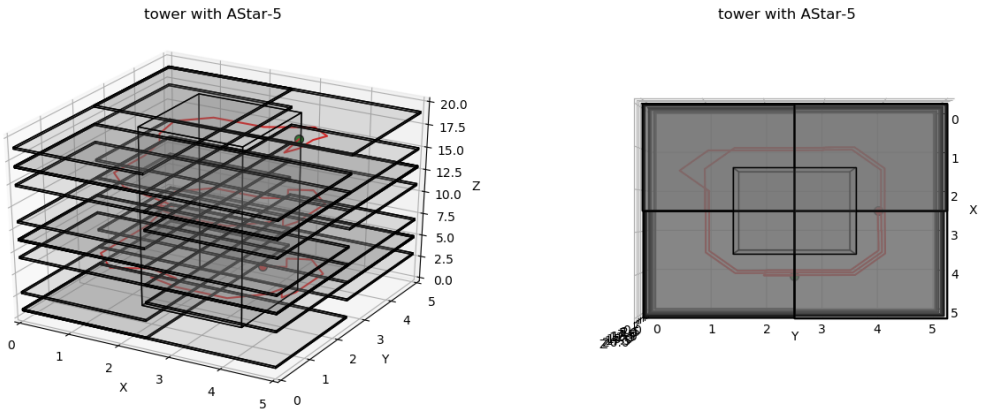

Figure 25: Tower, A*, $\epsilon = 5$

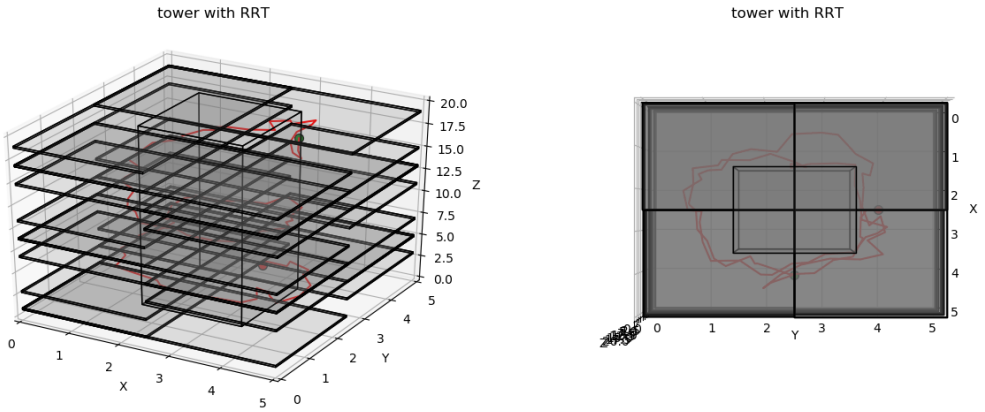

Figure 26: Tower, RRT

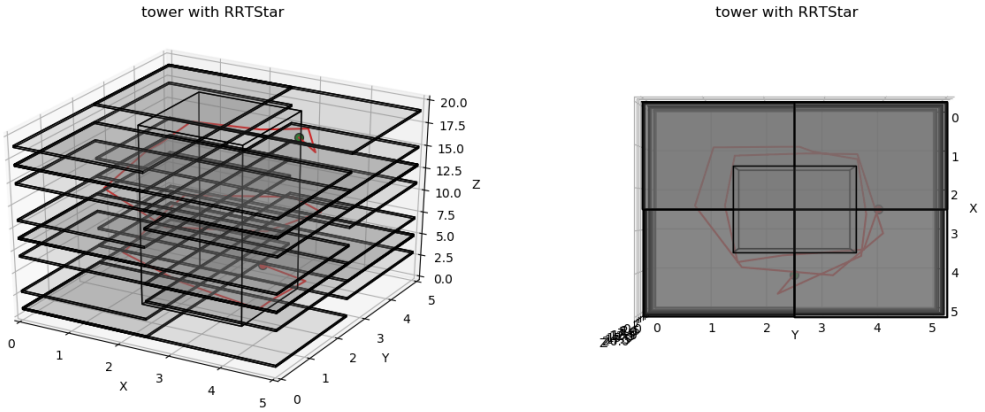

Figure 27: Tower, RRT*

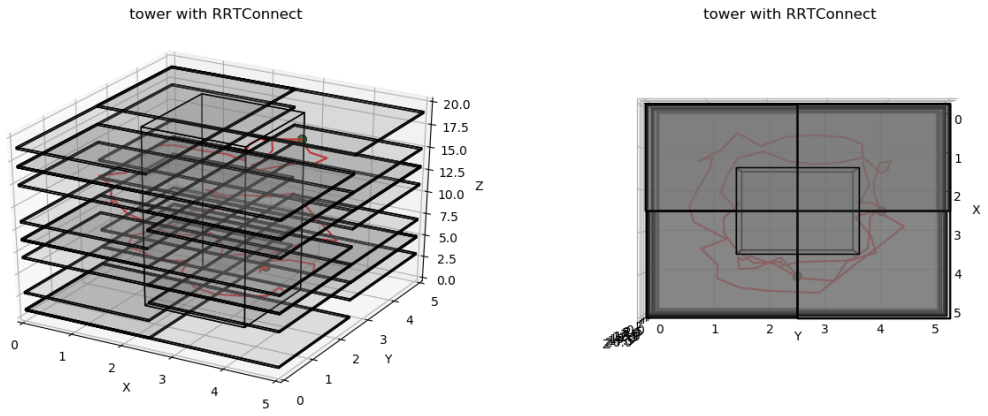

Figure 28: Tower, RRT-Connect

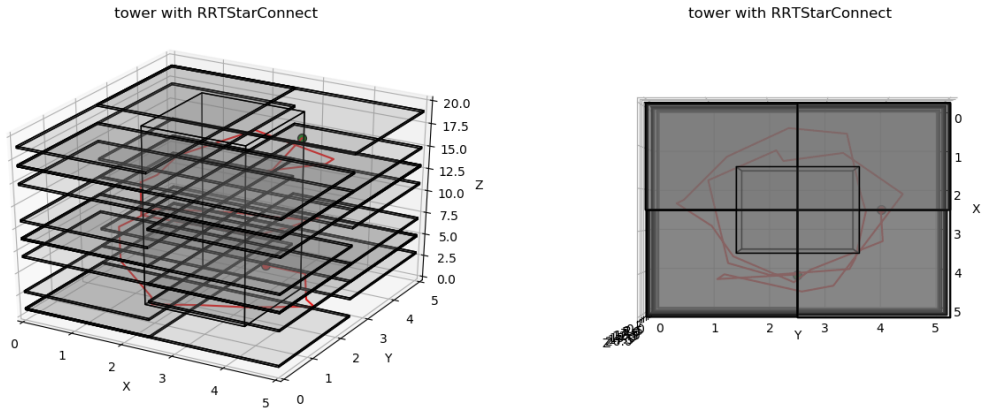

Figure 29: Tower, RRT*-Connect

4.2.5 Flappy Bird

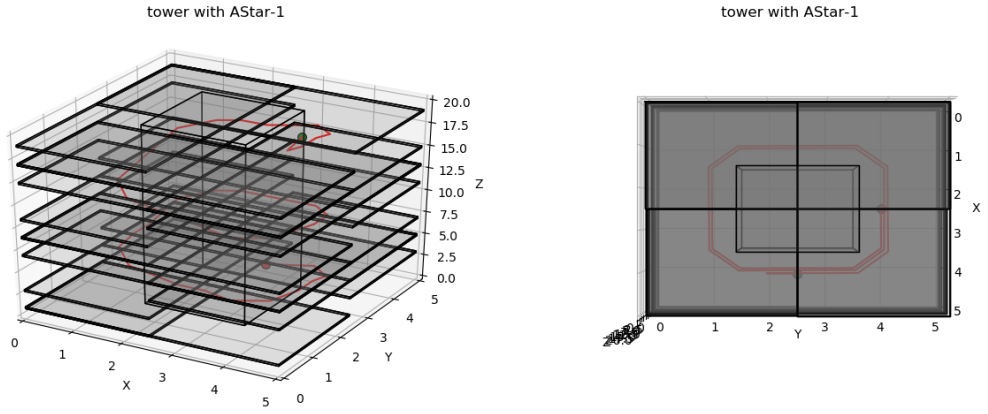

Figure 30: Tower, A*, $\epsilon = 1$

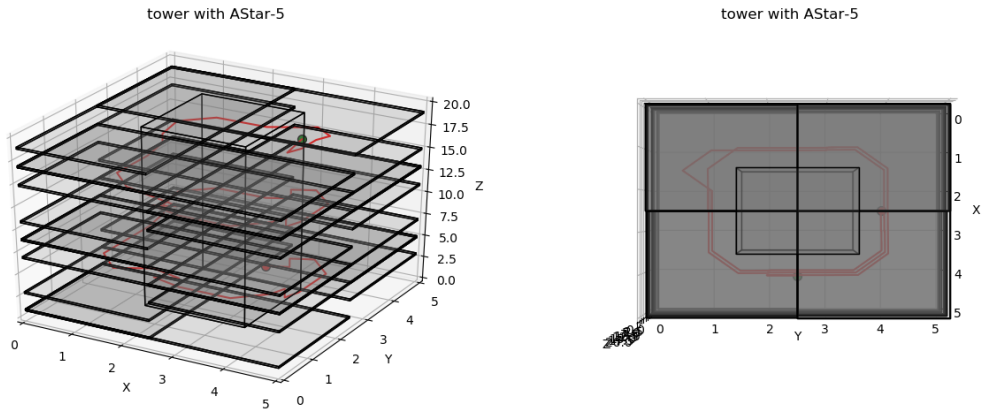

Figure 31: Tower, A*, $\epsilon = 5$

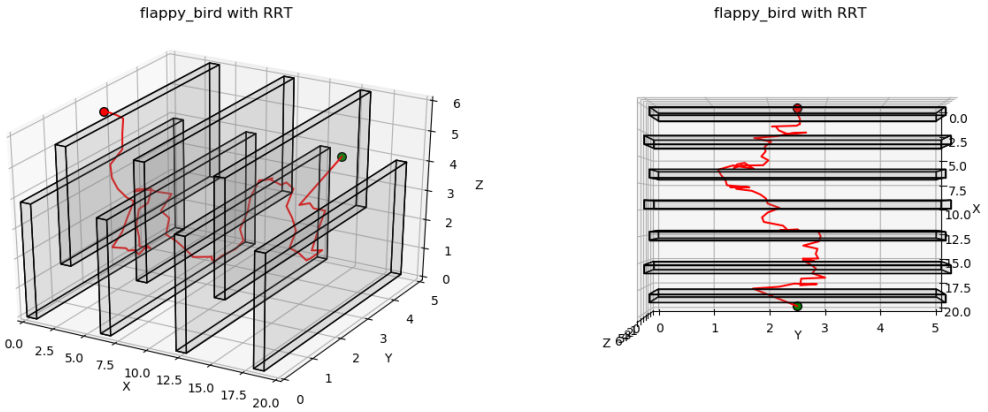

Figure 32: Flappy Bird, RRT

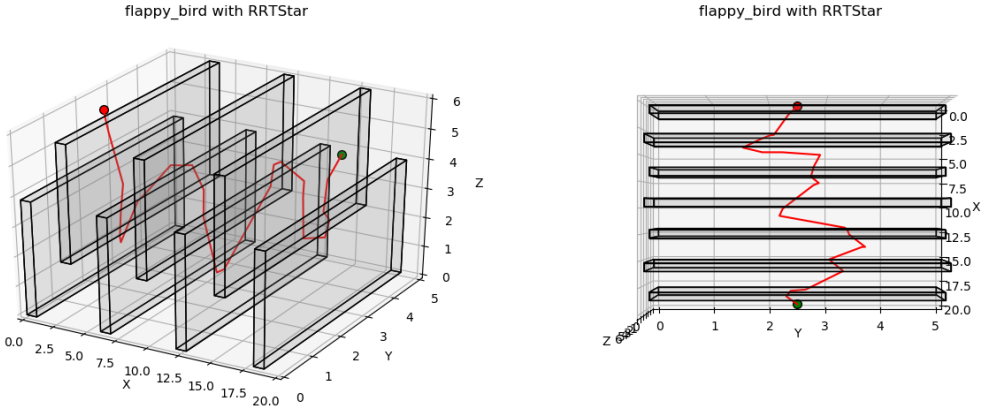

Figure 33: Flappy Bird, RRT*

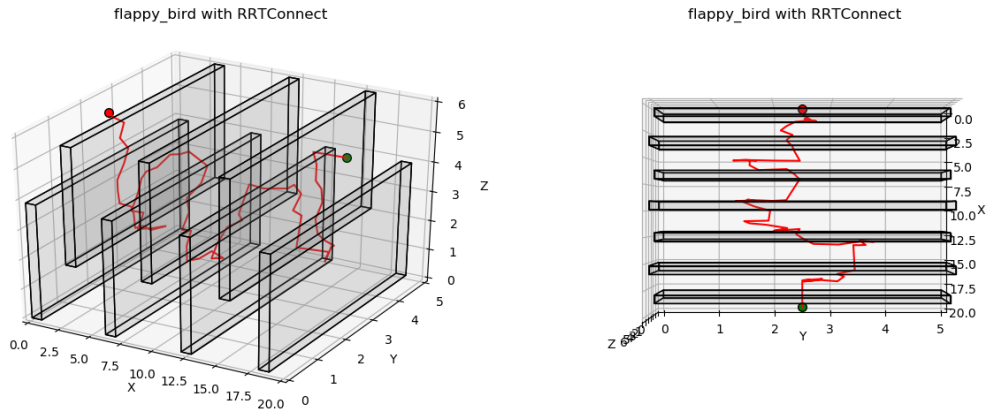

Figure 34: Flappy Bird, RRT-Connect

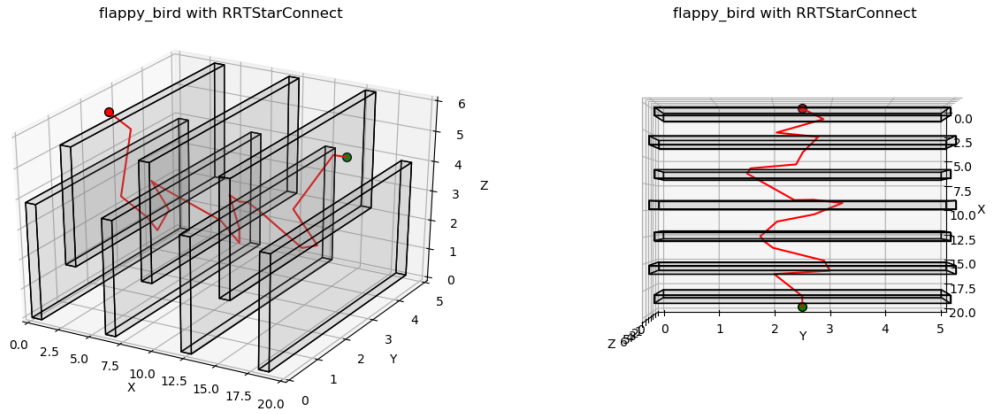

Figure 35: Flappy Bird, RRT*-Connect

4.2.6 Room

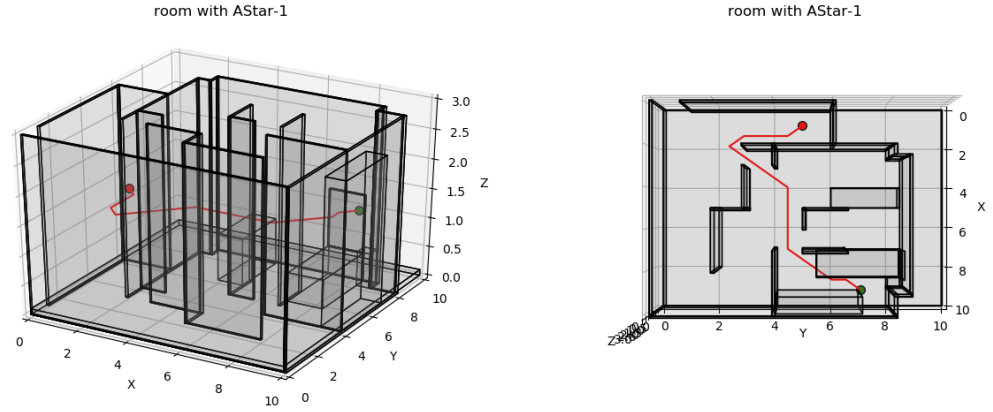

Figure 36: Room, A*, $\epsilon = 1$

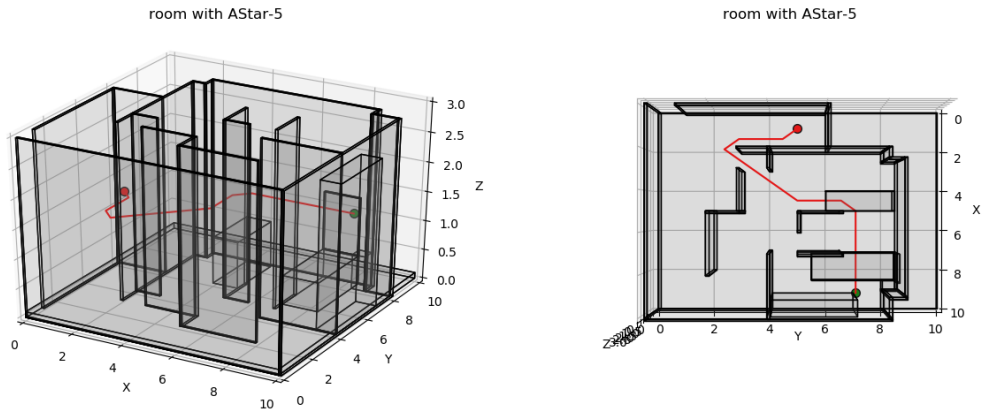

Figure 37: Room, A*, $\epsilon = 5$

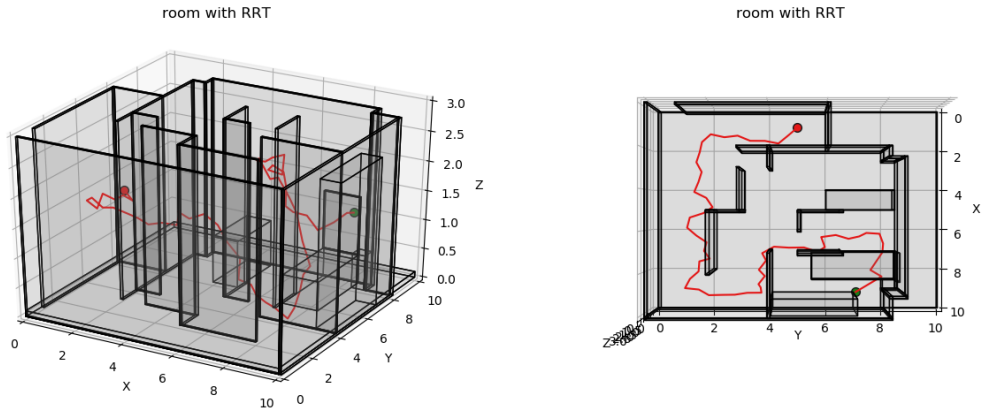

Figure 38: Room, RRT

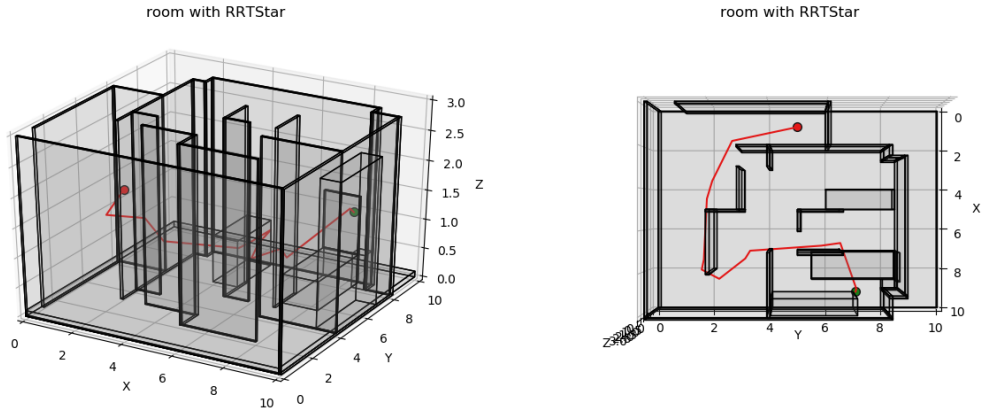

Figure 39: Room, RRT*

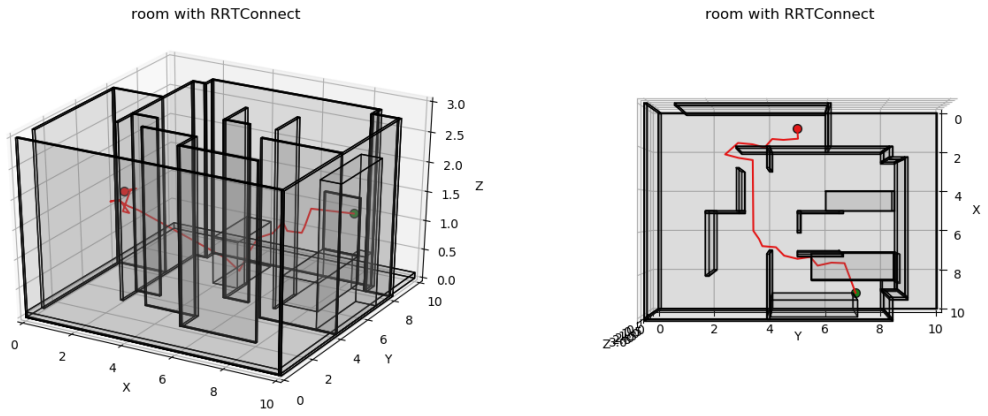

Figure 40: Room, RRT-Connect

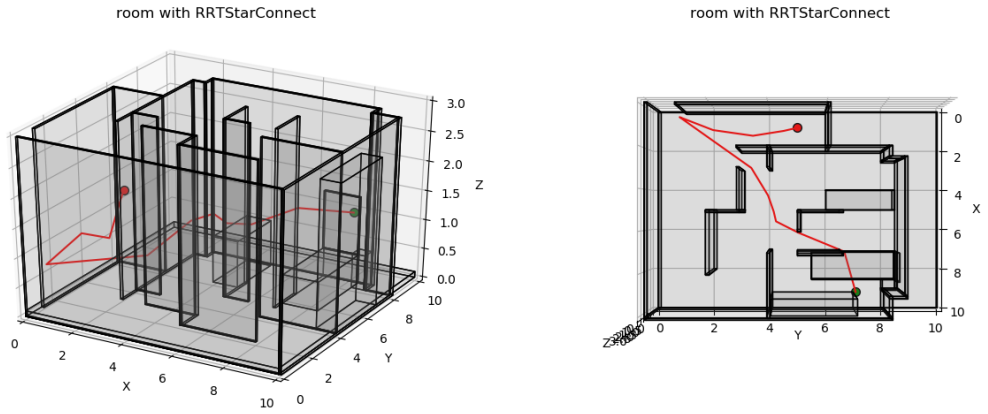

Figure 41: Room, RRT*-Connect

4.2.7 Monza

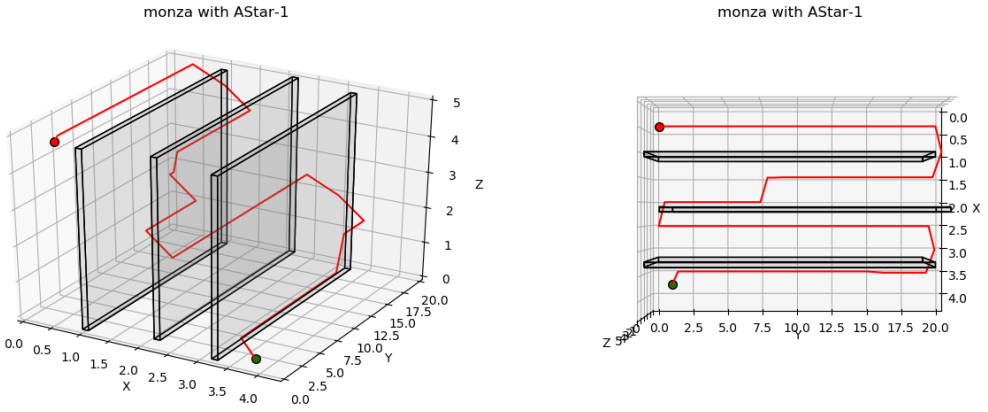

Figure 42: Monza, A*, $\epsilon = 1$

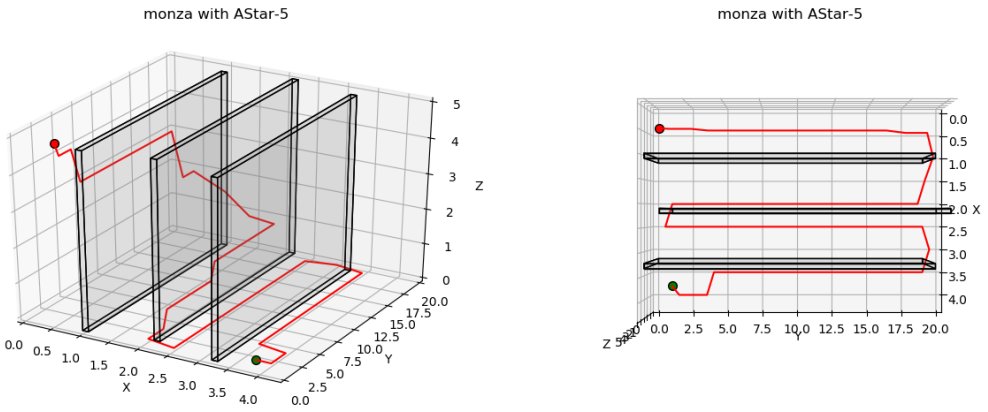

Figure 43: Monza, A*, $\epsilon = 5$

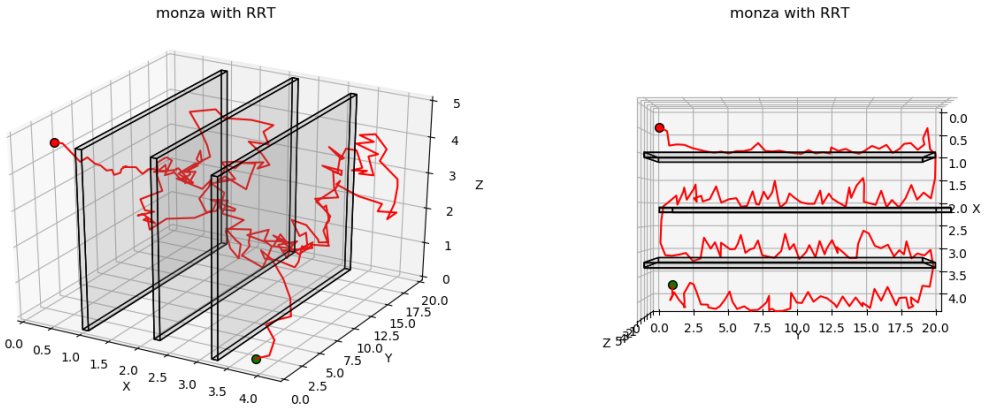

Figure 44: Monza, RRT

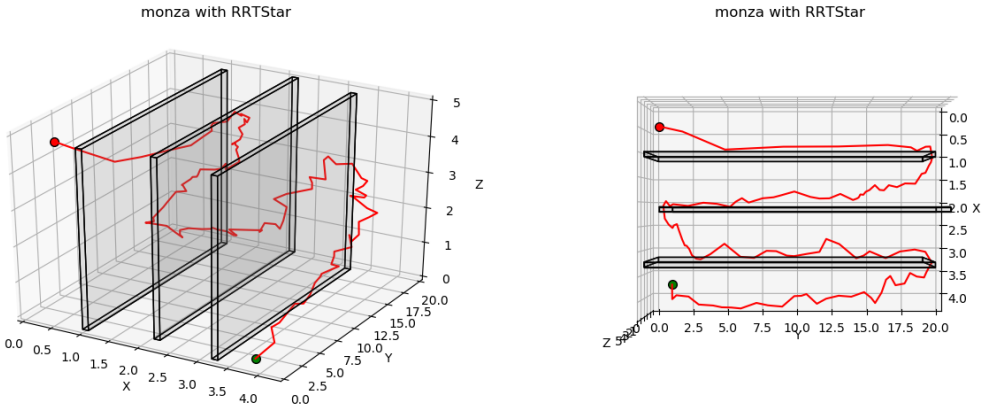

Figure 45: Monza, RRT*

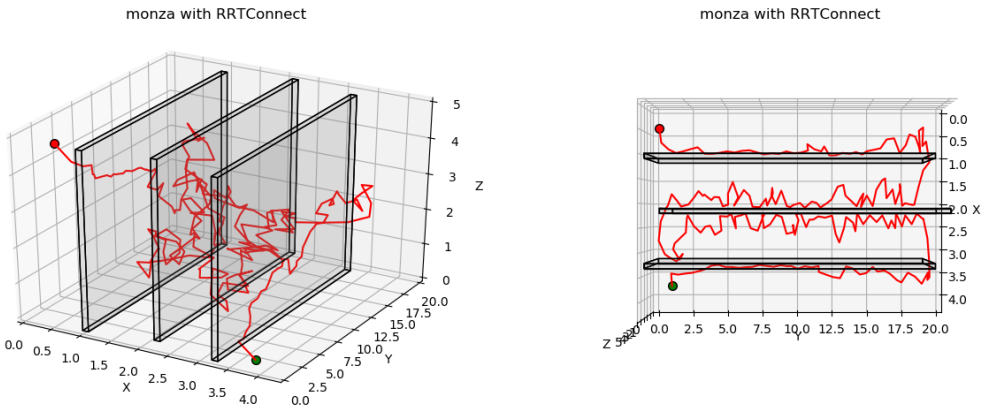

Figure 46: Monza, RRT-Connect

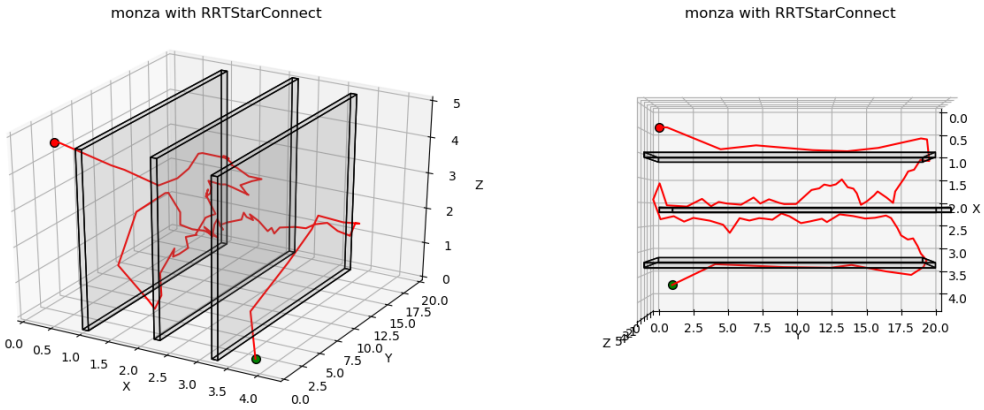

Figure 47: Monza, RRT*-Connect