ECE 276B: Planning & Learning in Robotics Project 2

Lin-Hsi Tsao (PID: A59015911)
Department of Electrical and Computer Engineering
University of California, San Diego
ltsao@ucsd.edu

Acknowledge: Wilson Liao (w4liao@ucsd.edu)

I. Introduction

Motion planning is important in the field of robotics. By doing so, the robot could reach the goal from the starting point. However, it is relatively harder to find an optimal path with less energy consumption and without any collision. For example, the label-correcting algorithm could ensure an optimal path but lead to poor time efficiency, which is not practical in a real-world scenario. To address this issue, faster alternatives like the Rapid-exploring Random Tree (RRT) have been proposed, which provide sub-optimal paths but significantly improve speed.

In this report, I introduced four kinds of motion planning algorithms, including weighted A*, RRT, RRT*, and RRT-Connect. The robot is tested in seven distinct environments. By conducting this analysis, we can gain insights into the performance of these algorithms concerning energy consumption, time efficiency, and other relevant metrics.

II. Problem Statement

In this section, the deterministic shortest path problem with obstacles could be defined as follows:

Given an environment which is a bounded space X with obstacles b_1, b_2, ..., $b_n \in B$, with the starting point s and the goal point τ, we try to find a path $f^*(t)$ satisfying

$$f^* = \underset{f}{argmin} \int_{t=0}^{t=T} C(f(t))dt \tag{1}$$

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

As for the discrete form, we try to find a path i_t satisfying

$$i^* = \underset{i}{argmin} \sum_{t=0}^{t=T} C(i_t) \tag{2}$$

For both the continuous form f^* and the discrete form i^*, the goal is to create a path with the smallest cost form $s \in X$ to $\tau \in X$. The cost function is defined as the euclidean distance between two points.

III. Technical Approach

In this section, the algorithm solving the problem would be expressed. The algorithm could be stated into three parts, which are search-based planning algorithm, sampling-based planning algorithm, and collision check.

A. Search-based Planning Algorithm

For the search-based planning algorithm, I implement the weighted A* algorithm. The weighted A* algorithm is based on the label-correcting algorithm with the heuristic function, which makes the algorithm converge faster. Algorithm 1 expresses the details of the weighted A* algorithm.

Algorithm 1 Weighted A* Algorithm

```
OPEN ← {s}, CLOSED ← {}, ε ≥ 1
g_s = 0, g_i = ∞ for all i ∈ X \{s}
while τ ∉ CLOSED do
    Remove i with the lowest f_i := g_i + ε h_i from OPEN
    Insert i into CLOSED
    for j ∈ Children(i) and j ∉ CLOSED do
        if g_j > (g_i + c_ij) then
            g_j ← (g_j + c_ij)
            Parent(j) ← i
            if j ∈ OPEN then
                Update priority of j
            else
                OPEN ← OPEN ⋃ {j}
            end if
        end if
    end for
end while
```

In this case, I choose octile distance as my heuristic function:

$$h_i = max_k|x_{\tau,k} - x_{i,k}| + (\sqrt{d} - 1)min_k|x_{\tau,k} - x_{i,k}| \tag{3}$$

where $x_{\tau,k}$ is the coordinate of the goal point in the k^{th} dimension, $x_{i,k}$ is the coordinate of the point i in the k^{th} dimension, and d is the number of dimensions. d equals to 3 in the assignment.

B. Sampling-based Planning Algorithm

For the sampling-based planning algorithm, I implement RRT, RRT*, and RRT-connect algorithm. Algorithm 2 expresses the details of the RRT algorithm.

Algorithm 2 Rapid-exploring Random Tree (RRT)

Initialize tree T with the start node
while goal not reached **do**
    Generate a random point q_{rand} in the configuration space
    Find the nearest node q_{near} in the tree T to q_{rand}
    Extend the tree T by adding a new node q_{new} towards q_{rand} from q_{near}
    if q_{new} is in collision **then**
        continue to the next iteration
    end if
    Add q_{new} as a child of q_{near} in T
    if q_{new} is close to the goal **then**
        Add the goal node to T and connect it to q_{new}
        return the path from start to goal
    end if
end while
return no path found

The RRT* and the RRT-connected algorithms are the extension of the RRT algorithm. As for the RRT* algorithm, it utilizes a cost function that considers the path length to guide the growth of the tree. Moreover, the RRT* algorithm introduces a rewiring mechanism, which allows for better exploration and the possibility of finding more optimal paths.

RRT-Connect is a variant of RRT that aims to efficiently connect the start and goal configurations. It maintains two separate RRT trees, one growing from the start configuration and the other from the goal configuration. RRT-Connect attempts to connect the two trees iteratively until a path between the start and goal configurations is found. Similar to RRT, RRT-Connect produces suboptimal paths, but it can provide fast exploration and relatively quick convergence to a feasible solution.

C. Collision-checking Algorithm

Given a path obtained from the planning algorithm, the robot could follow the path and reach the goal easily. However, it is likely that the path segment collides with the obstacles in the environment, which implies that there exists a path segment intersecting with the boundary of an obstacle. Therefore, a collision-checking algorithm is introduced to check whether the trajectory of the agent collides with an obstacle or not. Algorithm 3 shows the implementation details.

D. Comparison between different motion planning algorithms

Now, we compare the different motion planning algorithms in the aspect of optimality, completeness, memory efficiency, and time efficiency. The overall comparisons are shown in TABLE I.

Algorithm 3 Collision-checking Algorithm

Input starting point, end point, block boundary
Output collision or not
if starting point is inside block boundary **then**
    return True
end if
dir ← end point - starting point
for dim in [x, y, z] **do**
    if dir[dim] $>$ 0 **then**
        t ← (min(boundary[dim]) - staring point[dim]) / dir[dim]
    else
        t ← (max(boundary[dim]) - staring point[dim]) / dir[dim]
    end if
    if t $>$ 0 and t $<$ 1 **then**
        intersect point ← starting point + t · dir
        if intersect point intersects with the plane of the block **then**
            return True
        end if
    end if
end for
return False

Weighted A Algorithm:*

- **Optimality**: The weighted A* algorithm guarantees to find the optimal path when an admissible and consistent heuristic is used.
- **Completeness**: Weighted A* is complete, meaning it will find a solution if one exists, given a finite search space.
- **Memory Efficiency**: Weighted A* can consume significant memory as it needs to store all visited nodes in the open and closed lists.
- **Time Efficiency**: Weighted A* can have a moderate time efficiency, depending on the complexity of the search space and the efficiency of the heuristic function.

RRT (Rapid-exploring Random Tree):

- **Optimality**: RRT provides a sub-optimal path, which means it does not guarantee to find the shortest path.
- **Completeness**: RRT is probabilistically complete, meaning it has a high probability of finding a solution if one exists.
- **Memory Efficiency**: RRT has relatively low memory requirements as it only needs to store the generated tree structure.
- **Time Efficiency**: RRT is generally fast in terms of time efficiency.

RRT (Rapid-exploring Random Tree Star):*

- **Optimality**: RRT* can guarantee to find the optimal path, similar to the A* algorithm.
- **Completeness**: RRT* is also probabilistically complete, meaning it has a high probability of finding a solution if one exists.

TABLE I: Comparison of Path Planning Algorithms

Algorithm	Optimality	Completeness	Memory Efficiency	Time Efficiency
A*	Optimal	Complete	High	Moderate
RRT	Sub-optimal	Probabilistically Complete	Low	Fast
RRT*	Optimal	Probabilistically Complete	Moderate	Moderate
RRT-Connect	Sub-optimal	Probabilistically Complete	Low	Fast

- **Memory Efficiency**: RRT* has a moderate memory efficiency as it needs to store the generated tree structure.
- **Time Efficiency**: RRT* has a moderate time efficiency due to the additional rewiring steps compared to RRT.

RRT-Connect:

- **Optimality**: RRT-Connect provides a sub-optimal path, similar to RRT. It does not guarantee to find the shortest path.
- **Completeness**: RRT-Connect is probabilistically complete, meaning it has a high probability of finding a solution if one exists.
- **Memory Efficiency**: RRT-Connect has low memory requirements similar to RRT, as it only needs to store the generated tree structure.
- **Time Efficiency**: RRT-Connect is generally fast in terms of time efficiency.

IV. Results

A. Experimental settings

In the experiments, I discretize the environment with a resolution equaling 0.5 for the weighted A* searching algorithm. However, with the resolution equaling 0.5, the agent could not reach the goal in the flappy bird environment. Hence, I add the experiment with a resolution equaling 0.1 for the weighted A* algorithm in the flappy bird environment test.

For RRT series algorithms, I set the max steer distance to 0.05 to check for intersections with obstacles, set the max number of samples to 100,000 to ensure adequate samples covering the environment, and set the probability of checking for a connection to the goal to 0.1. The probability of checking would reduce the time to check whether there exists a path from the starting point to the goal.

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

B. Experimental results

The tables show the results of algorithms tested on different test cases. The comparisons include the time consumption, number of considered nodes, and path length. The x in A^*-x indicates the value of ϵ applied.

For most of the planning paths, the agent could reach the goal from the starting point. However, there are five cases that the agent could not successfully achieve the goal, which are

1) RRT-Connect in Maze
2) RRT-Connect in Flappy Bird
3) RRT-Connect in Monza
4) RRT-Connect in Tower
5) RRT-Connect in Room

The reason why the event happened is that the path would exceed the boundary of the environment.

C. Discussion

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

For the quality of computed paths, A*-1 is almost the best among all algorithms in all test cases. A*-5 generates slightly longer paths compared with A*-1. As for RRT-series algorithms, they tend to generate longer paths, which are not as optimal as the A* algorithms. Also, we could easily find that the path yielded by RRT series algorithms tend to be twisted.

For the run time of computing paths, we observed that the RRT series algorithm, which is based on sampling, is significantly faster than the A* algorithm, which is based on searching. A* algorithms with higher values of ϵ generally exhibit faster performance compared to those with smaller values, thanks to the ϵ-consistency property. On the other hand, RRT* tends to be slower than RRT and RRT-Connect due to the optimization process it incorporates.

In terms of the number of nodes considered, we observed that searching-based A* algorithms tend to consider a significantly larger number of nodes compared to the sampling-based RRT series algorithms. This increased node count results in higher memory and computational requirements, particularly for A* algorithms with smaller values of ϵ. However, using a larger value of ϵ can mitigate the search in local minima when an appropriate heuristic is employed.

TABLE II: Path length comparison between different algorithms in different test environments

Algorithm	Single Cube	Maze	Flappy Bird	Monza	Window	Tower	Room
A*-1	8	79	26 (res = 0.1)	40	30	34	12
A*-5	8	79	35 (res = 0.1)	40	31	35	12
RRT	9	110	42	109	31	45	23
RRT*	7	78	33	79	25	31	12
RRT-Connect	15	66	48	51	45	36	27

TABLE III: Run time (sec) comparison between different algorithms in different test environments

Algorithm	Single Cube	Maze	Flappy Bird	Monza	Window	Tower	Room
A*-1	4.72	17.44	162.66 (res = 0.1)	0.74	4.98	2.75	0.90
A*-5	5.56	22.15	222.35 (res = 0.1)	1.23	8.32	3.22	1.17
RRT	9.16	6.01	0.20	11.12	0.10	0.36	0.16
RRT*	0.01	87.16	5.21	28.79	0.85	18.84	0.99
RRT-Connect	0.04	2.27	0.59	0.28	0.09	0.69	0.15

TABLE IV: Numbner of considered nodes comparison between different algorithms in different test environments

Algorithm	Single Cube	Maze	Flappy Bird	Monza	Window	Tower	Room
A*-1	45	14,425	404,675 (res = 0.1)	3,231	9,719	2,384	852
A*-5	11	10,583	28,790 (res = 0.1)	977	75	465	86
RRT	7	10,881	810	42,244	258	1,310	458
RRT*	3	10,568	562	2,427	154	1,550	119
RRT-Connect	17	1,912	389	250	52	899	3,889

D. Visualization

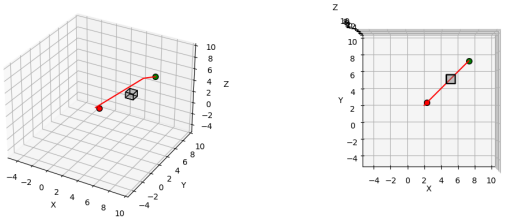

Fig. 1: Single Cube, A*-1

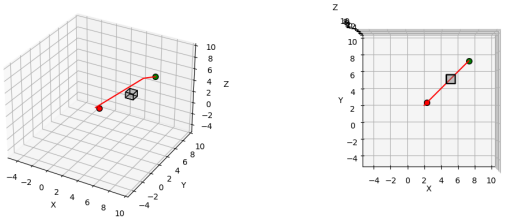

Fig. 2: Single Cube, A*-5

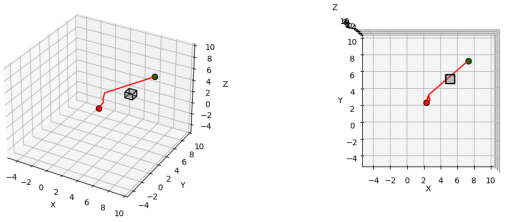

Fig. 3: Single Cube, RRT

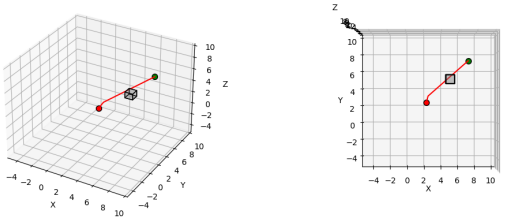

Fig. 4: Single Cube, RRT*

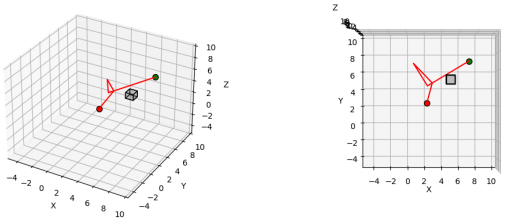

Fig. 5: Single Cube, RRT-CONNECT

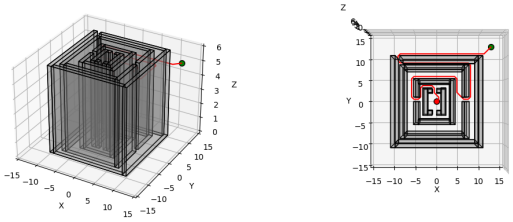

Fig. 6: Maze, A*-1

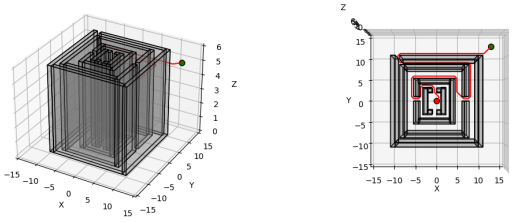

Fig. 7: Maze, A*-5

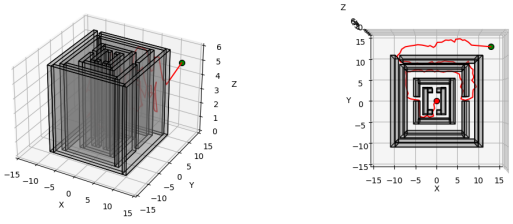

Fig. 8: Maze, RRT

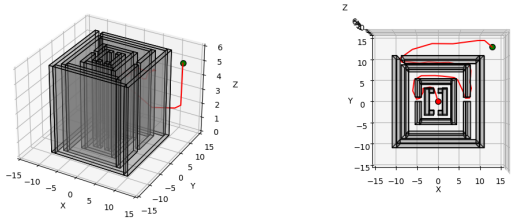

Fig. 9: Maze, RRT*

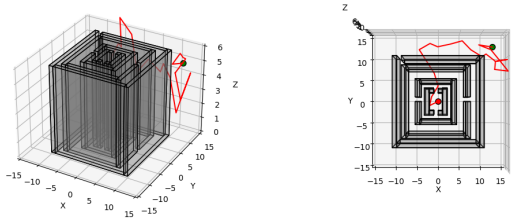

Fig. 10: Maze, RRT-CONNECT

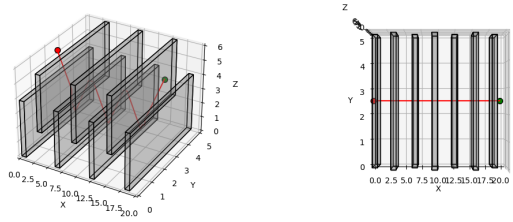

Fig. 11: Flappy Bird, A*-1

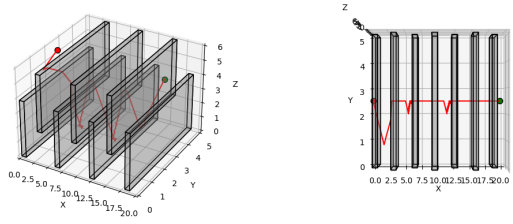

Fig. 12: Flappy Bird, A*-5

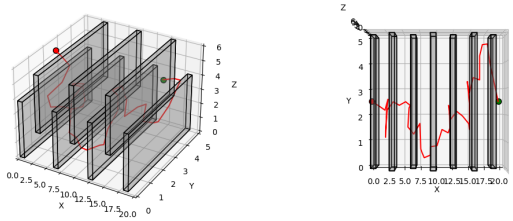

Fig. 13: Flappy Bird, RRT

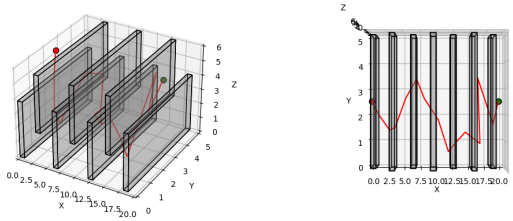

Fig. 14: Flappy Bird, RRT*

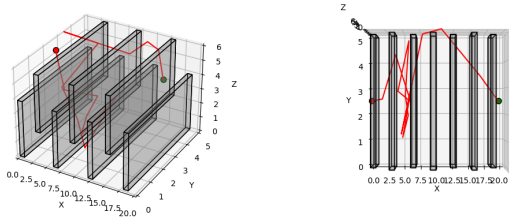

Fig. 15: Flappy Bird, RRT-CONNECT

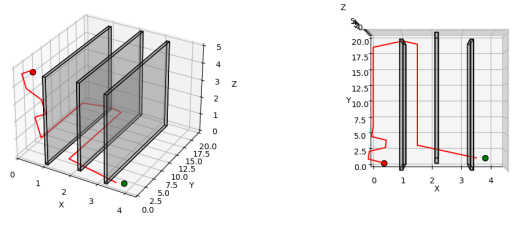

Fig. 16: Monza, A*-1

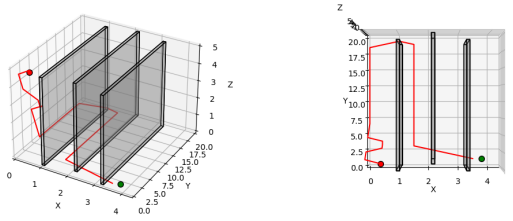

Fig. 17: Monza, A*-5

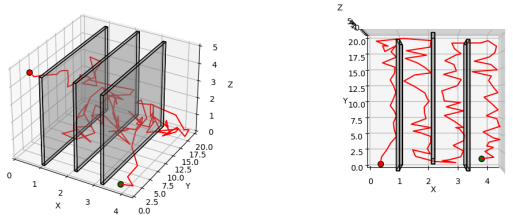

Fig. 18: Monza, RRT

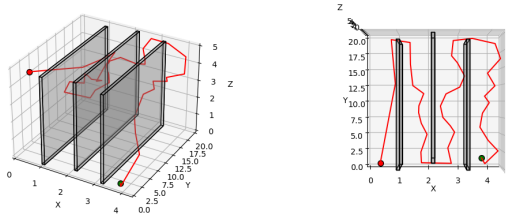

Fig. 19: Monza, RRT*

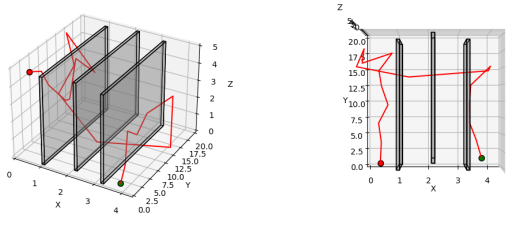

Fig. 20: Monza, RRT-CONNECT

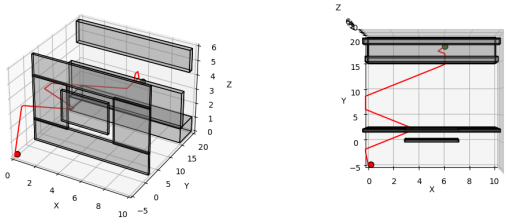

Fig. 21: Window, A*-1

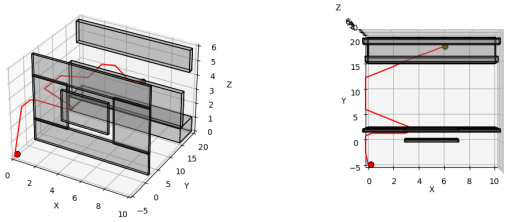

Fig. 22: Window, A*-5

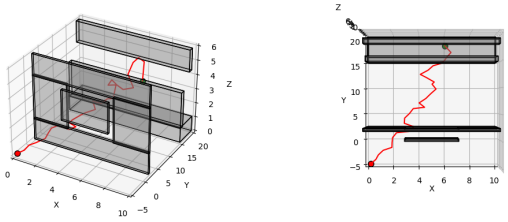

Fig. 23: Window, RRT

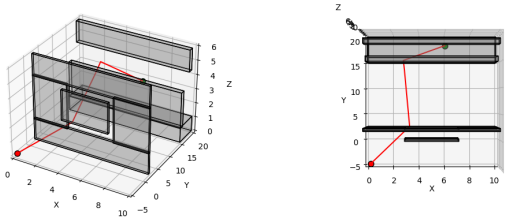

Fig. 24: Window, RRT*

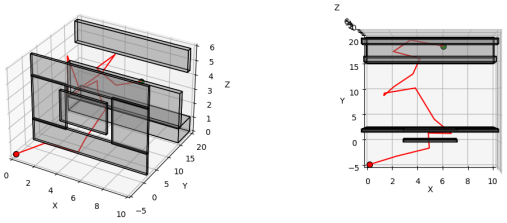

Fig. 25: Window, RRT-CONNECT

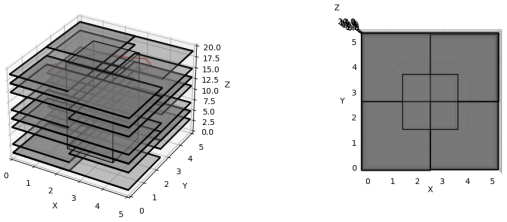

Fig. 26: Tower, A*-1

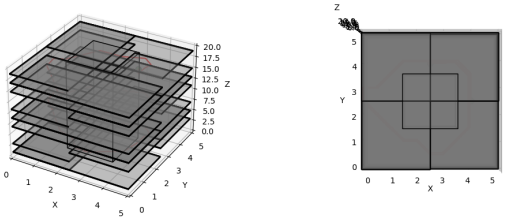

Fig. 27: Tower, A*-5

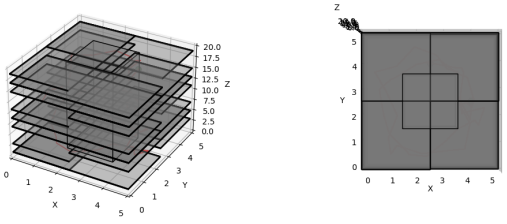

Fig. 28: Tower, RRT

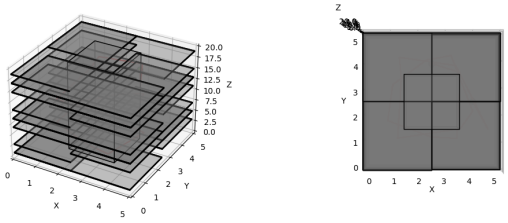

Fig. 29: Tower, RRT*

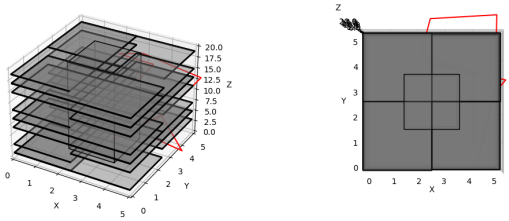

Fig. 30: Tower, RRT-CONNECT

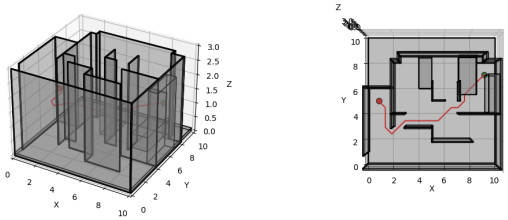

Fig. 31: Room, A*-1

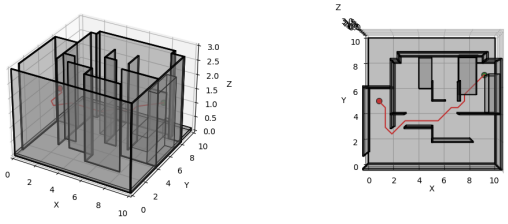

Fig. 32: Room, A*-5

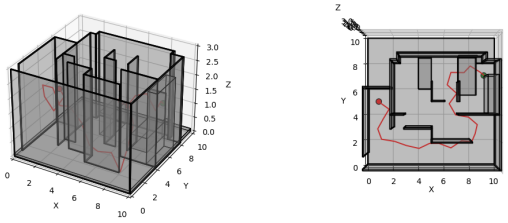

Fig. 33: Room, RRT

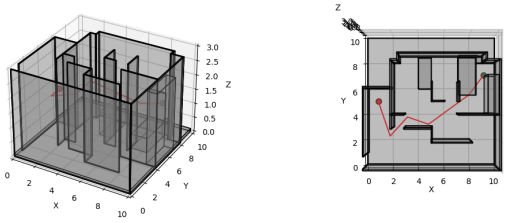

Fig. 34: Room, RRT*

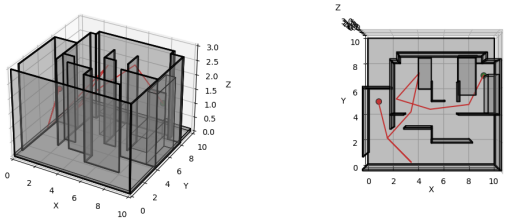

Fig. 35: Room, RRT-CONNECT