Project 2: Motion Planning

Behrad Rabiei
Dept. of Electrical and Computer Engineering
University of California San Diego
Email: brabiei@ucsd.edu

***Abstract*—This paper is the report for Project 2 of ECE 276B (Planning and Learning in Robotics). The overall objective of this assignment is to find valid paths from a starting point to an endpoint in a variety of environments. The project requires you to 1) implement a collision checker to ensure your path does not collide with any obstacles, 2) implement a search-based algorithm to find a path, and 3) implement a sampling-based algorithm to find a path.**

I. Introduction

Autonomous robotic navigation represents a cornerstone of modern robotics, crucial for enhancing the autonomy and efficiency of robots in complex and dynamic environments. The development of robust navigation systems enables robots to perform a wide range of tasks, from industrial automation and logistics to autonomous vehicles and space exploration. By leveraging advanced sensors, algorithms, and learning techniques, these systems aim to improve safety, reduce human labor, and expand the capabilities of robots beyond traditional boundaries. The motivation behind advancing autonomous navigation is not only to increase operational efficiency but also to innovate in areas where human access is dangerous or impractical.

A. Search Based Path Planning

Search-based path planning is a method used in robotics and other fields to navigate a device or agent from one point to another while avoiding obstacles. This technique involves mapping out an environment into a grid or graph, where each point or node represents a possible state or position the robot can occupy. The planner then searches through these nodes, using algorithms like Dijkstra's or A* (A-star), to find the most efficient route from the starting point to the destination. The chosen path must avoid any areas that represent barriers or hazards, ensuring the robot moves safely and effectively. This method is particularly useful in structured environments where the layout and obstacles are clearly defined, making it easier to plan and predict movements.

B. Sampling Based Path Planning

Sample-based path planning is a strategy commonly used in robotics to navigate complex or high-dimensional spaces where traditional grid-based search methods might be impractical. Unlike search-based methods that evaluate a fixed set of points, sample-based planning randomly generates points (or "samples") throughout the navigation area. These samples are then connected to form a roadmap of possible paths that the robot can take. Algorithms like Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM) are popular in this approach. The planner progressively builds a network that explores the space, avoiding obstacles by only retaining the connections that lead through safe, navigable paths. This method is particularly advantageous in large or unstructured environments, offering a flexible and often more computationally efficient way to find viable routes from start to finish.

C. Project Objectives

The overall objective of this project is to develop and evaluate motion planning algorithms in 3-D environments characterized by rectangular obstacles and boundaries. The project is structured into distinct parts aimed at enhancing both the reliability and efficiency of motion planning strategies. Initially, we are tasked with implementing or utilizing a collision-checking algorithm that can assess the safety of paths in 3-D space, specifically checking collisions between line segments and axis-aligned bounding boxes. Subsequently, we are required to devise our own search-based planning algorithm, applying enhanced methods such as weighted A* or jump point search, with an emphasis on incorporating efficient collision detection. Finally, the project offers a choice between creating a sampling-based planning algorithm, such as RRT, or utilizing and learning from established motion planning libraries like OMPL or Python's motion-planning library.

II. Problem Formulation

In this section, we will properly formulate the problems of path planning and collision detection.

A. Path Planning in a 3D Environment

Given:

- A 3D space denoted as $\mathbb{R}^3$.
- A starting point $s \in \mathbb{R}^3$.
- An endpoint $e \in \mathbb{R}^3$.
- An environment boundary represented as a bounding box $B \subset \mathbb{R}^3$.
- A set of axis-aligned bounding box obstacles $\{O_1, O_2, \ldots, O_n\}$ where each O_i is a subset of B and represents an area that cannot be traversed.

Objective: Find a path $P : [0, 1] \to \mathbb{R}^3$ such that:

- $P(0) = s$ (path starts at s)
- $P(1) = e$ (path ends at e)

- $P(t) \in B$ for all $t \in [0,1]$ (path remains within the environment boundary)
- $P(t) \notin O_i$ for all $t \in [0,1]$ and for all i (path avoids all obstacles)

Mathematical Formulation:

1) **Continuity Constraint**: P should be a continuous function to ensure the path is feasible for traversal.
2) **Boundary Constraint**:

$$\forall t \in [0,1], \quad P(t) \in B$$

3) **Obstacle Avoidance Constraint**:

$$\forall t \in [0,1], \forall i \in \{1,2,\ldots,n\}, \quad P(t) \notin O_i$$

4) **Path Optimality** (optional, based on specific requirements like shortest path, minimum energy, etc.):

$$\min \int_0^1 \| \frac{dP}{dt} \| \, dt$$

This could represent minimizing the length of the path, although different metrics might be used depending on the application (e.g., time, energy).

B. Collision Detection between Line Segments and AABBs in 3D Space

We want to know whether a line segment intersects any given axis-aligned bounding boxes (AABBs) in a continuous 3D space, which is a critical aspect of evaluating the safety of a path planning algorithm.

- An **axis-aligned bounding box** (AABB) in 3D is characterized by its minimum corner $\mathbf{min} = (x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and maximum corner $\mathbf{max} = (x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$.
- A **line segment** in 3D is specified by its two endpoints $\mathbf{p}_1 = (x_1, y_1, z_1)$ and $\mathbf{p}_2 = (x_2, y_2, z_2)$.

The conditions under which the line segment defined by $\mathbf{p}_1$ and $\mathbf{p}_2$ intersects with one or more AABBs within the defined 3D space are:

1) Identify all t in the parameterization $\mathbf{p}(t) = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$ for $t \in [0,1]$ that satisfy intersection conditions with any AABB.
2) For each AABB, ascertain whether there exists a t such that the segment $\mathbf{p}(t)$ lies within the bounding limits of the AABB along all three dimensions x, y, and z.
3) The criteria for intersection must ensure that the line segment touches or passes through the interior of any AABB without merely touching the boundary unless explicitly considered a collision.

III. Technical Approach

A. Pybullet's Collision Detection

For the task of collision detection, we decided to go with the PyBullet library. In PyBullet, the `rayTest()` function implements ray-based collision detection by defining a ray with starting (`fromPosition`) and ending points (`toPosition`). Utilizing the physics engine's broadphase and narrowphase detection mechanisms, `rayTest()` efficiently identifies potential intersections between the ray and objects within the simulation environment. During the broadphase, axis-aligned bounding boxes (AABBs) quickly discard non-intersecting entities, while the narrowphase conducts detailed intersection tests against the collision meshes of potentially colliding objects. This dual-phase approach optimizes the computational overhead, making `rayTest()` suitable for applications requiring rapid and accurate collision queries, such as sensor simulation or interactive environments. The function returns detailed hit information, including the object, hit fraction, intersection point, and normal at the hit location, facilitating various applications in robotics, virtual reality, and game development.

B. A-Star Algorithm

The A* algorithm seeks to find the shortest path from a start node to a goal node in a graph by minimizing the cost function $f(n) = g(n) + h(n)$, where:

- $g(n)$ represents the actual cost from the start node to any node n.
- $h(n)$ is a heuristic function estimating the lowest cost from node n to the goal.

Heuristic Function: The heuristic function $h(n)$, implemented as `heuristic(node, goal)`, uses the Euclidean distance to estimate the cost from the current node to the goal. It is defined mathematically by:

$$h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2 + (z_n - z_g)^2}$$

where (x_n, y_n, z_n) are the coordinates of node n and (x_g, y_g, z_g) are the coordinates of the goal. We chose 1 for the weight of the heuristic.

Motion Possibilities: The function `possible_motions(current_node, goal, boundary)` determines valid movements from the current node. The movements include left, right, up, down, forward, and backward, with each movement adjusted by a predefined amount, checking for collisions and boundary constraints.The final movement amount we chose was 0.3.

Collision and Boundary Checks: Collisions and boundary checks ensure that a node does not violate spatial constraints.

Node Expansion: Each node expansion checks against the open and closed lists, ensuring that the node with the least $f(n)$ value is selected for expansion. Nodes are generated and evaluated based on:

- Updating $g(n)$ for the movement cost to the node.
- Ensuring that new nodes do not revisit previously closed nodes.
- Comparing nodes in the open list to possibly replace with a node having a lower $g(n)$.

Path Reconstruction: Once the goal is reached, the path is reconstructed by backtracking from the goal node to the start node using parent pointers. The process continues until the start node is reached, yielding the complete path from start to goal.

Algorithm Termination: The algorithm terminates when the goal node is placed into the closed list, indicating that the shortest path has been found, or when there are no more nodes to explore, indicating no available path.

C. Bi-Directional RRT Algorithm

The Bi-Directional Rapidly-exploring Random Trees (Bi-Directional RRT) algorithm is an extension of the standard RRT algorithm designed to efficiently find paths in high-dimensional spaces by simultaneously building two trees, one from the start and the other from the goal. This method often reduces the time to find a path compared to the traditional RRT.

Algorithm Description: Two trees, T_{start} and T_{goal}, are initiated respectively from the start node x_{start} and the goal node x_{goal}. The algorithm iteratively expands these trees towards each other by generating random nodes in the space, and connecting them to the nearest node in each tree, attempting to bridge the gap between the two trees.

Mathematical Formulation: Let V_{start} and V_{goal} represent the set of vertices in the trees T_{start} and T_{goal} respectively. At each iteration, a random node x_{rand} is sampled from the search space X.

The nearest nodes x_{near}^{start} and x_{near}^{goal} in each tree are determined by:

$$x_{near}^{start} = \underset{x \in V_{start}}{\operatorname{argmin}}\ d(x, x_{rand}), \quad x_{near}^{goal} = \underset{x \in V_{goal}}{\operatorname{argmin}}\ d(x, x_{rand})$$

where $d(a, b)$ denotes a distance metric in the space.

New nodes x_{new}^{start} and x_{new}^{goal} are then created by extending from x_{near}^{start} and x_{near}^{goal} towards x_{rand} by a fixed step size Δq, constrained by the system's dynamics and collision constraints.

Hyperparameters:

- **Step Size** Δq: Defines how far each new node is from its nearest node. Affects the smoothness and speed of tree expansion. In our case we chose 0.3 as the step size.
- **Goal Bias** γ: A parameter that determines the frequency at which x_{rand} is set to x_{goal} instead of being sampled randomly. This biases the tree expansion towards the goal.
- **Maximum Iterations** N: The maximum number of iterations the algorithm runs, serving as a stopping condition. We set this to 500K.

Termination: The algorithm terminates when a path is found connecting T_{start} and T_{goal}, or the number of iterations exceeds N.

Performance: The performance of Bi-Directional RRT is highly dependent on the tuning of its hyperparameters, especially in complex environments where the path to the goal is constrained by obstacles.

IV. Results

Across the board, we see that Bi-Directional RRT finds shorter paths to the goal compared to the A* results (refer to figures 22 and 23). With that being said, the BDRRT takes significantly longer to find a valid path in environments with many obstacles and tight spaces to navigate through. In terms of the smoothness of the trajectory, we observe that the A* algorithm provides smoother paths in complex environments (refer to figures 10, 11, 16, and 17), while BDRRT yields easier paths to execute in simpler environments (refer to figures 1, 2, 13, 14, 19, and 20). Regarding the number of samples generated, if the environment is simple, then BDRRT needs to generate very few samples. Conversely, if the environment is complex, the algorithm needs to produce many samples, explaining the differences in runtimes (refer to figures 3 and 15 vs. 6 and 12). Note: all results are included at the end of the report.

V. Conclusion

We successfully achieved the overarching goal of developing and evaluating motion planning algorithms within 3-D environments filled with rectangular obstacles and boundaries. Through the utilization of refined collision-checking algorithms, we ensured the safety and viability of paths, effectively assessing collisions between line segments and axis-aligned bounding boxes. Our deployment of enhanced search-based planning algorithms like weighted A* demonstrated significant improvements in path smoothness and efficiency, particularly in complex environments. Moreover, our exploration into sampling-based planning algorithms, such as Bi-Directional RRT, and the incorporation of established libraries like OMPL or Python's motion-planning library, highlighted the adaptability and speed of these methods in various scenarios. The comparative analysis between A* and Bi-Directional RRT provided insightful data on their respective strengths, showcasing A*'s capability for path finding in complex environments and Bi-Directional RRT's proficiency in faster planning times in simple settings. This project underscored the critical importance of algorithm selection based on specific environmental challenges.

References

[1] https://natanaso.github.io/ece276b/

[2] https://github.com/motion-planning/rrt-algorithms

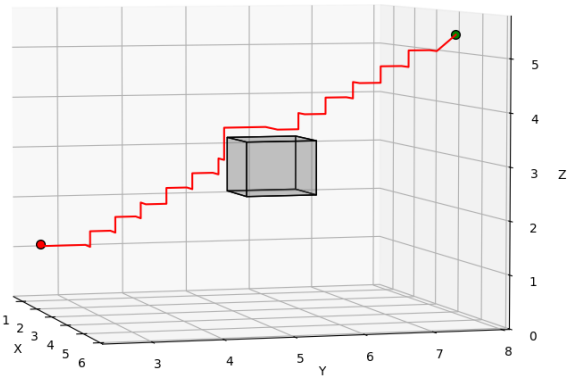

Fig. 1. Single Cube A*

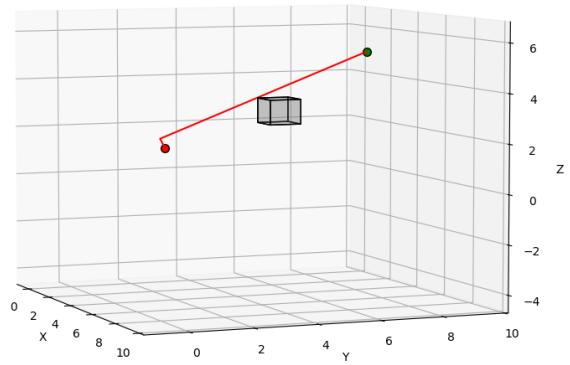

Fig. 2. Single Cube Bi-Directional RRT Path

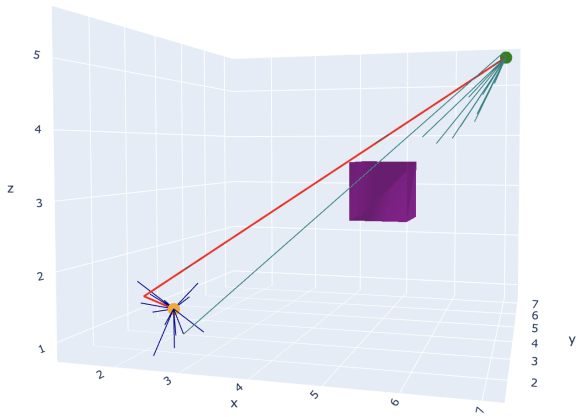

Fig. 3. Single Cube Bi-Directional RRT Tree

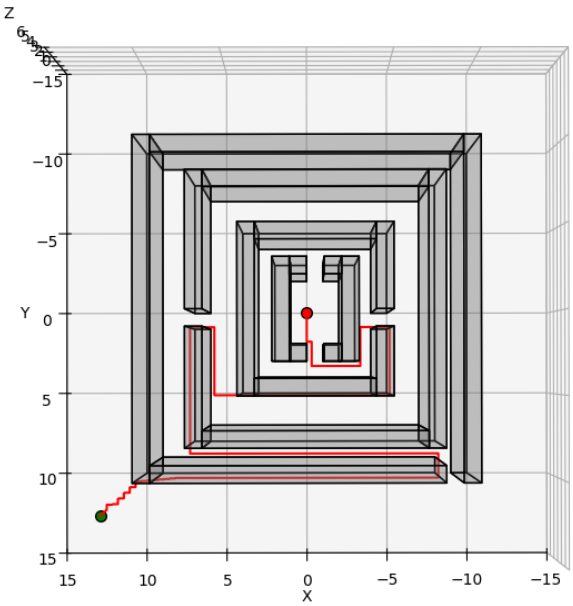

Fig. 4. Maze A*

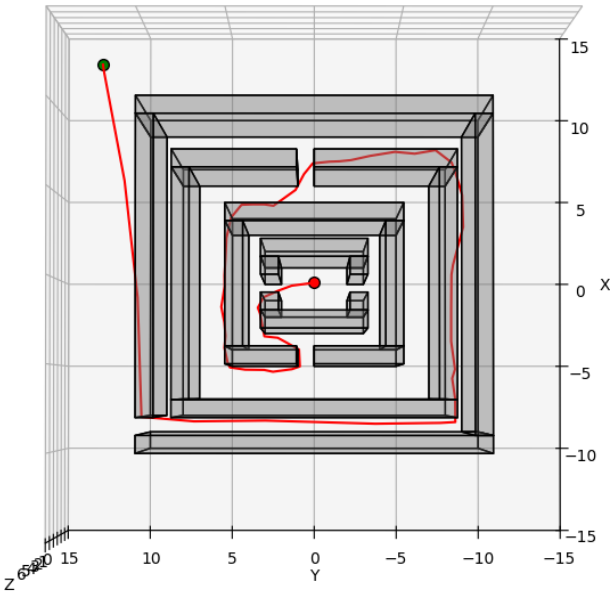

Fig. 5. Maze Bi-Directional RRT Path

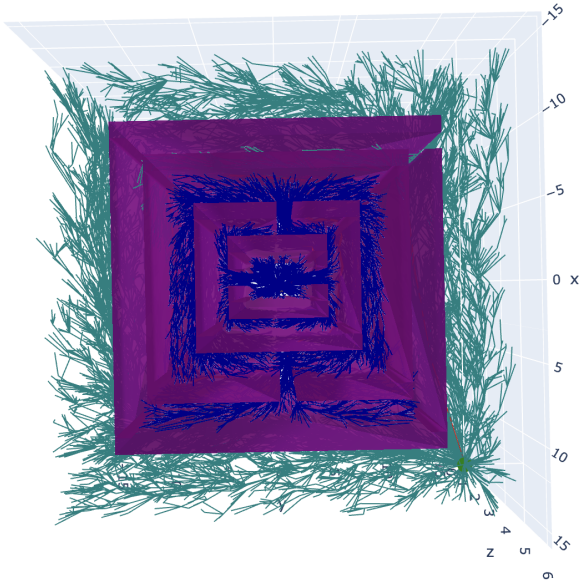

Fig. 6. Maze Bi-Directional RRT Tree

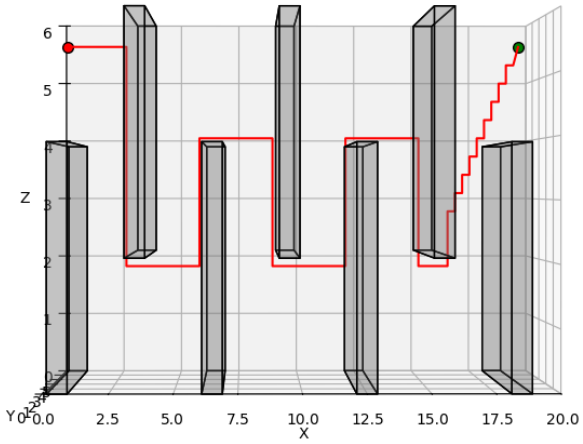

Fig. 7. Flappy Bird A*

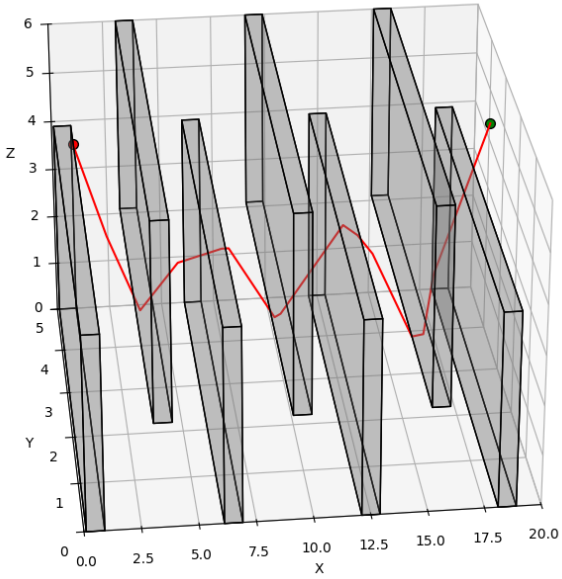

Fig. 8. Flappy Bird Bi-Directional RRT Path

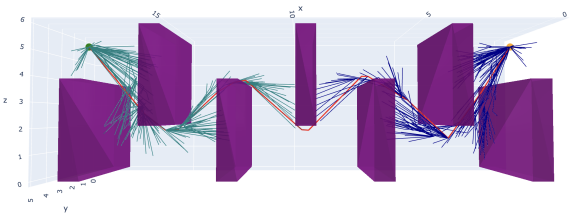

Fig. 9. Flappy Bird Bi-Directional RRT Tree

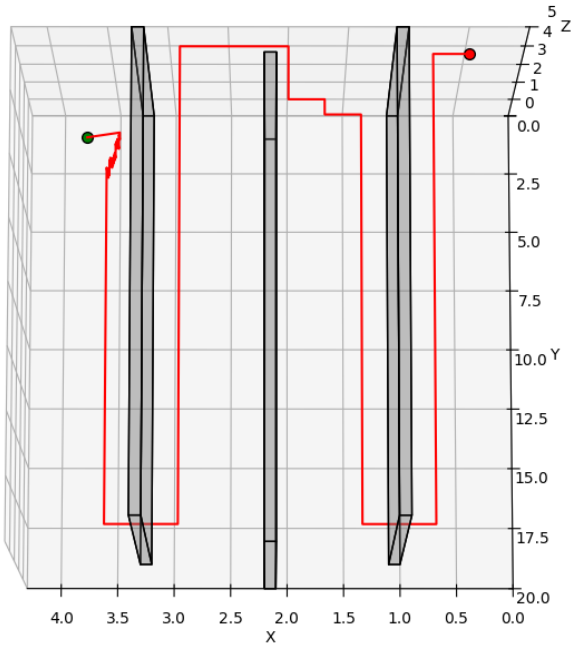

Fig. 10. Monza A*

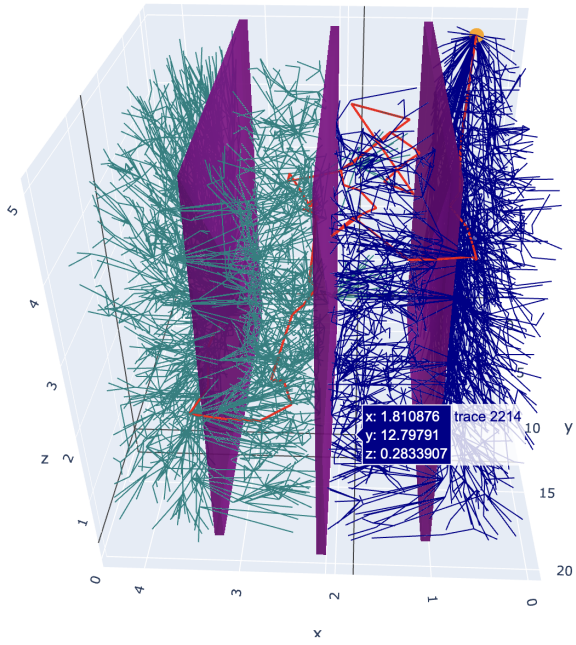

Fig. 12. Monza Bi-Directional RRT Tree

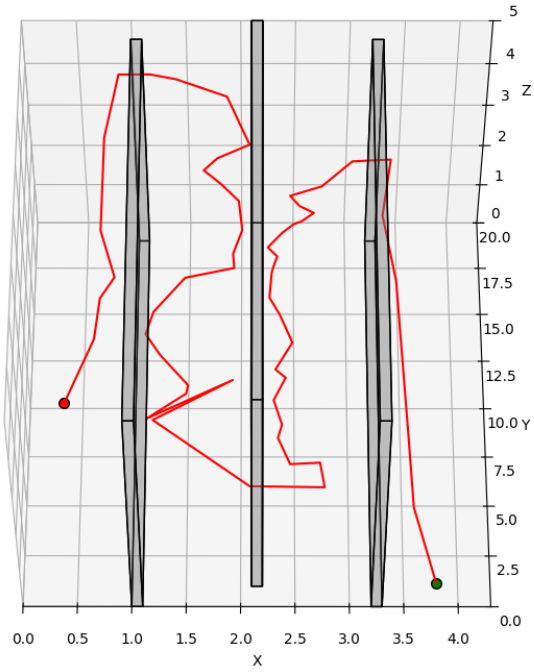

Fig. 11. Monza Bi-Directional RRT Path

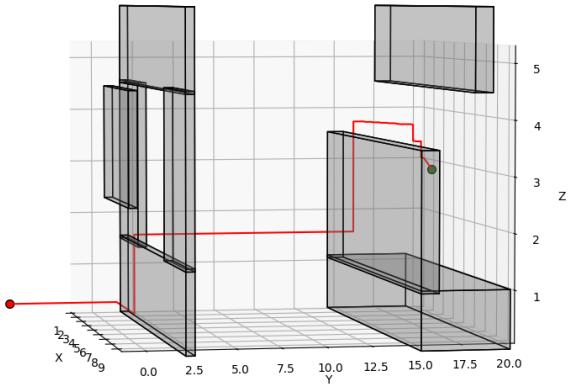

Fig. 13. Window A*

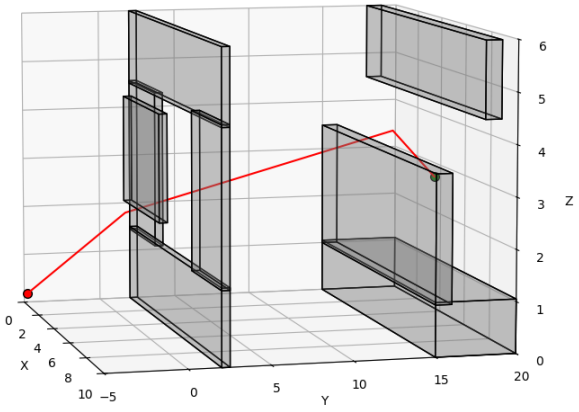

Fig. 14. Window Bi-Directional RRT Path

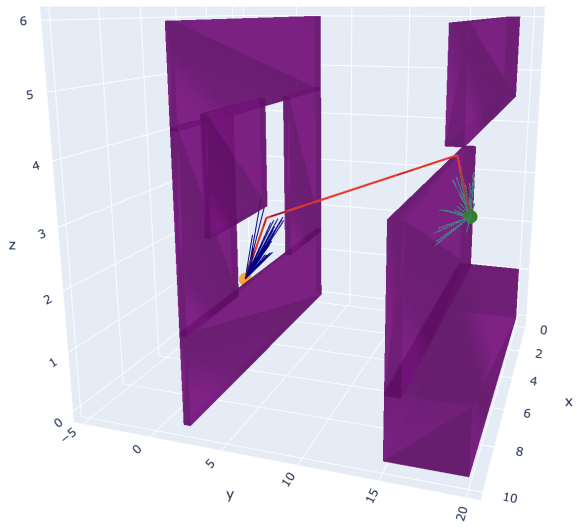

Fig. 15. Window Bi-Directional RRT Tree

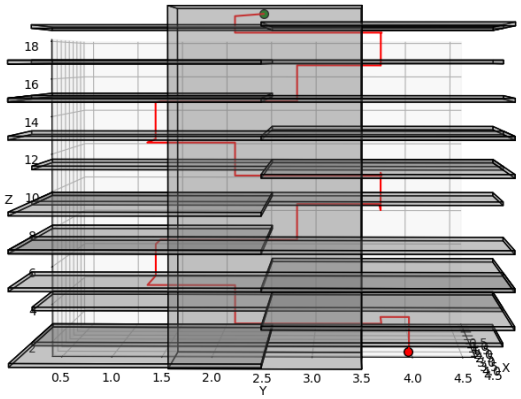

Fig. 16. Tower A*

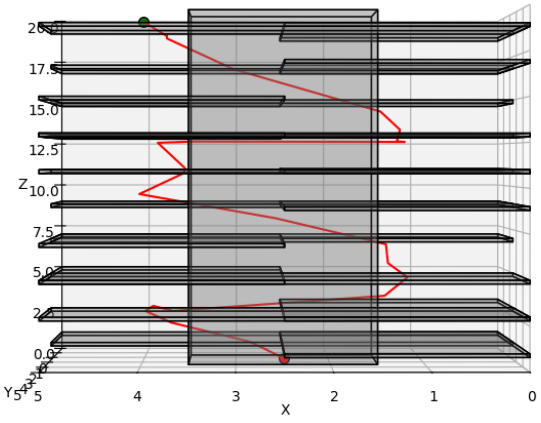

Fig. 17. Tower Bi-Directional RRT Path

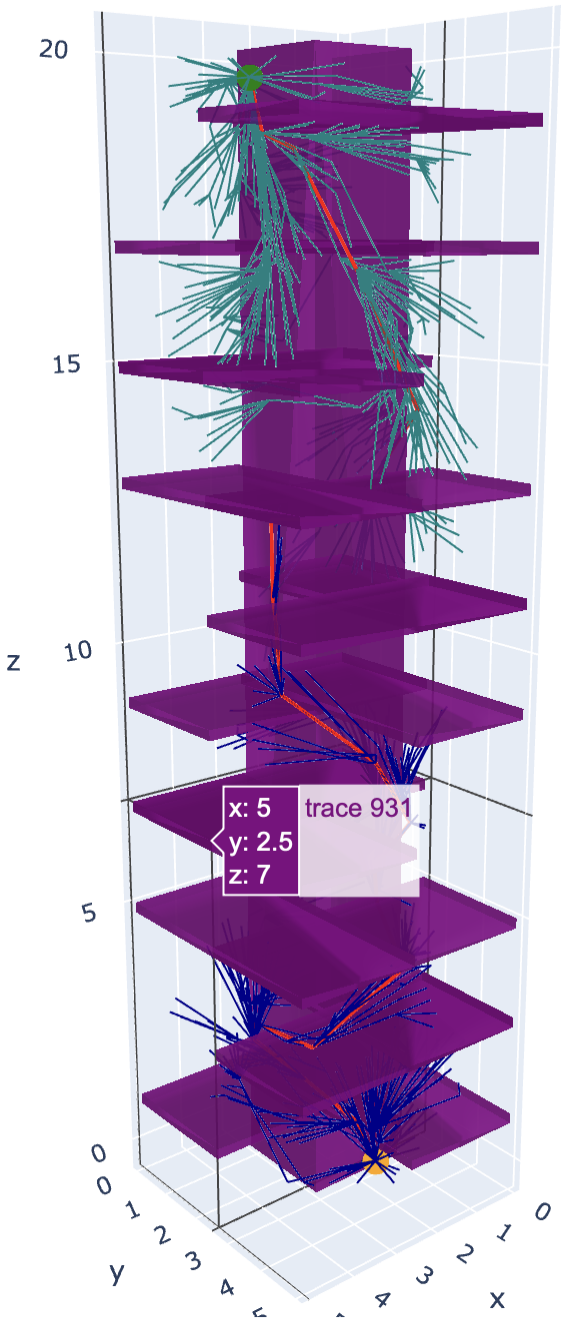

Fig. 18. Tower Bi-Directional RRT Tree

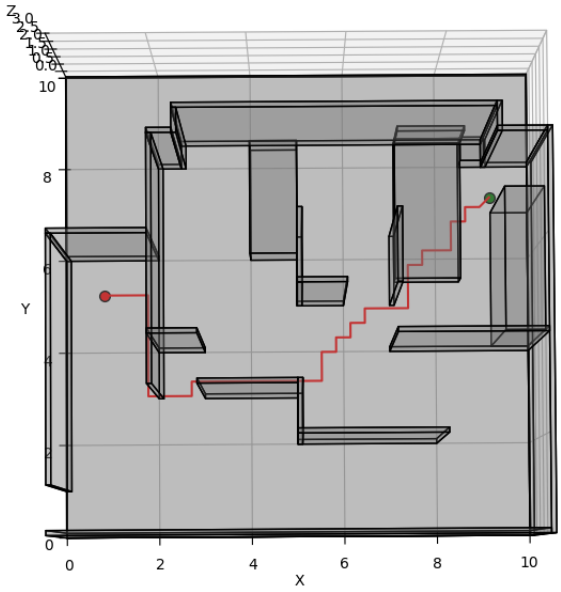

Fig. 19. Room A*

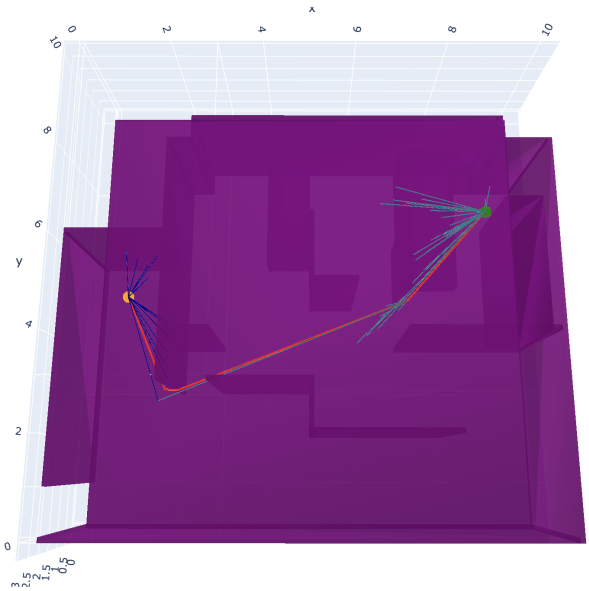

Fig. 21. Room Bi-Directional RRT Tree

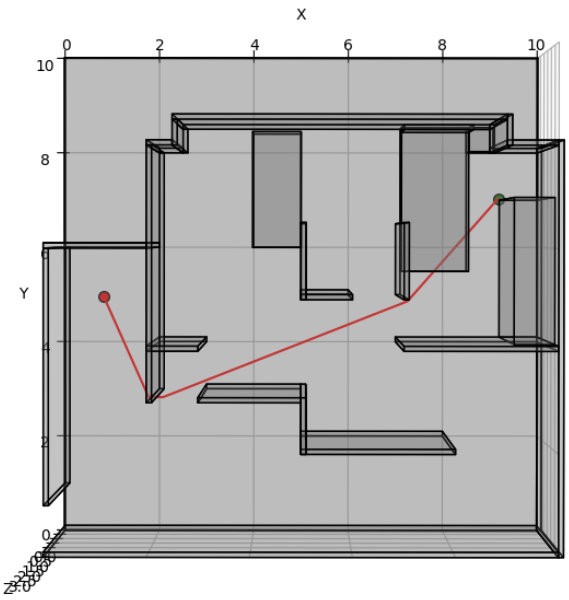

Fig. 20. Room Bi-Directional RRT Path

```
Running single cube test...

Planning took: 2.5154128074645996 sec.

Success: True
Path length: 13


Running maze test...

Planning took: 10.42668104171753 sec.

Success: True
Path length: 81


Running flappy bird test...

Planning took: 0.6909499168395996 sec.

Success: True
Path length: 33


Running monza test...

Planning took: 0.7793200016021729 sec.

Success: True
Path length: 81


Running window test...

Planning took: 4.764715909957886 sec.

Success: True
Path length: 32


Running tower test...

Planning took: 0.5541300773620605 sec.

Success: True
Path length: 41


Running room test...

Planning took: 0.21724200248718262 sec.

Success: True
Path length: 14
```

Fig. 22. A* Length and Runtime

```
Running single cube test...

Can connect to goal
Planning took: 0.2968120574951172 sec.

Success: True
Path length: 8


Running maze test...

Can connect to goal
Planning took: 287.63212418556213 sec.

Success: True
Path length: 75


Running flappy bird test...

Can connect to goal
Planning took: 5.196721076965332 sec.

Success: True
Path length: 27


Running monza test...

Can connect to goal
Planning took: 52.1704580783844 sec.

Success: True
Path length: 80


Running window test...

Can connect to goal
Planning took: 1.8916311264038086 sec.

Success: True
Path length: 24


Running tower test...

Can connect to goal
Planning took: 15.801533937454224 sec.

Success: True
Path length: 28


Running room test...

Can connect to goal
Planning took: 0.939990758895874 sec.

Success: True
Path length: 11
```

Fig. 23. Bi-Directional RRT Length and Runtime

CONTENTS