ECE276B-P2

Zhenyu Wu | PID: A69030822

May 23, 2025

1 Introduction

In this project, we will need to solve a motion planning problem in 3-D Euclidean Space. Our goal is to plan a collision free path $\{x_s, x_1, x_2 ... x_\tau\}$ from start position $x_s \in R^3$ to goal position $x_\tau \in R^3$. To solve this problem, we will use 2 types of motion planning methods:

1. Search-based Method
2. Sample-based Method

For the Search-based Method, we need to firstly build a discretized graph $G = (V, E)$ for the environment and perform graph search on G. For the Sample-based Method, we directly sample points $x \in R^3$ in free space C_{free} of the environment to construct the graph $G = (V, E)$.

2 Problem Statement

In this problem, we have a 3-D configuration space. We are given obstacles

$$C_{\text{obs}} = \{x_1^{\text{obs}}, x_2^{\text{obs}}, \dots, x_n^{\text{obs}}\},$$

as well as start state $x_s \in C_{\text{free}}$ and goal state $x_\tau \in C_{\text{free}}$.

The feasible path can be expressed as a continuous function

$$\rho : [0,1] \to C_{\text{free}} \quad \text{where} \quad \rho(0) = x_s,\ \rho(1) = x_\tau,$$

and the set of all feasible paths is denoted as $\mathcal{P}_{s,\tau}$.

Given a cost function $J : \mathcal{P} \to R$, we will find a feasible path ρ^* such that

$$J(\rho^*) = \min_{\rho \in \mathcal{P}_{s,\tau}} J(\rho).$$

To apply Search-based Method and Sampling-based Method, we translate this motion planning problem to a DSP (Discrete Search Problem) problem.

In the DSP problem, we use vertices to represent states in free space:

$$V = \{x_i \in C_{\text{free}} \mid i = 1, 2, \dots, n\}.$$

For each two vertices $u, v \in V$, we connect them with edge $e = (u, v)$ if the line segment from u to v lies entirely in C_{free}:

$$e = (u, v) \in E \quad \text{iff} \quad \text{Line}(u, v) \subset C_{\text{free}}.$$

Each edge $e = (u, v)$ is associated with a non-negative cost $l(e) \geq 0$.

A feasible path in graph $G = (V, E)$ from start to goal is a sequence of vertices

$$\rho = \{x_s, x_1, x_2, \dots, x_\tau \mid x \in C_{\text{free}}\}$$

such that each consecutive pair $(x_i, x_{i+1}) \in E$, and the set of all such paths is denoted as $\mathcal{P}_{s,\tau}$.

Each path has total cost

$$l(\rho) = \sum_{i=s}^{\tau-1} l(x_i, x_{i+1}),$$

where $l(x_i, x_{i+1})$ is the edge cost associated with edge $(x_i, x_{i+1}) \in E$.

Given $x_s \in V$ and $x_\tau \in V$, our objective is to find an optimal path ρ^* such that

$$l(\rho^*) = \min_{\rho \in \mathcal{P}_{s,\tau}} l(\rho).$$

3 Collision Checking

To determine whether a line segment intersects with an axis-aligned bounding box (AABB), I apply the method found at `https://tavianator.com/2022/ray_box_boundary.html`. The AABB is defined by its lower and upper bounds in each coordinate dimension:

$$B = [x_{\min}, y_{\min}, z_{\min}, x_{\max}, y_{\max}, z_{\max}].$$

The line segment is represented in parametric form as:

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0), \quad t \in [0, 1],$$

where $\mathbf{p}_0$ and $\mathbf{p}_1$ are the segment's endpoints.

The idea is to compute the range of t values for which $\mathbf{p}(t)$ lies within the AABB in all three coordinate directions. For each axis $i \in \{x, y, z\}$, we compute the values of t at which the segment enters and exits the slab bounded by $[\min_i, \max_i]$, where:

$$t_1^{(i)} = \frac{\min_i - p_0^{(i)}}{p_1^{(i)} - p_0^{(i)}}, \quad t_2^{(i)} = \frac{\max_i - p_0^{(i)}}{p_1^{(i)} - p_0^{(i)}}.$$

We define:

$$t_{\text{enter}}^{(i)} = \min(t_1^{(i)}, t_2^{(i)}), \quad t_{\text{exit}}^{(i)} = \max(t_1^{(i)}, t_2^{(i)}).$$

The final intersection interval across all three dimensions is:

$$t_{\min} = \max_i \left(t_{\text{enter}}^{(i)}\right), \quad t_{\max} = \min_i \left(t_{\text{exit}}^{(i)}\right).$$

The line segment intersects the AABB if:

$$t_{\min} \leq t_{\max}$$

To determine whether a path $\rho = \{x_0, x_1, \ldots, x_n\}$ is collision-free, we decompose it into a sequence of line segments:

$$(x_0, x_1),\ (x_1, x_2),\ \ldots,\ (x_{n-1}, x_n).$$

For each segment (x_i, x_{i+1}), we check whether it intersects with any obstacle in a given set of axis-aligned bounding boxes (AABBs), denoted as:

$$\mathcal{C}_{\text{obs}} = \{B_1, B_2, \ldots, B_m\}.$$

We define a collision checking function $\texttt{check\_collision}(x_i, x_{i+1}, B_j)$ that returns `True` if the segment intersects block B_j.

Then, the path ρ is considered collision-free if:

$$\forall i \in \{0, \ldots, n-1\},\ \forall j \in \{1, \ldots, m\}, \quad \texttt{check\_collision}(x_i, x_{i+1}, B_j) = \texttt{False}.$$

Otherwise, we detect a collision at the first segment (x_i, x_{i+1}) and terminate the check early.

4 Search-based Method

For the Search-based Method, I implemented the standard A* algorithm according to the lecture slides.

4.1 Graph Building

In standard implementation, we need to build the entire graph $G = (V, E)$ before running A*. In my implementation, I found that the time and memory usage for building entire graph by systematically descritize the space is highly inefficient. Therefore, in my implementation, I choose to build the graph dynamically during expansion. Specifically, when the node $i \in V$ with the lowest cost-to-go $f(i)$ is popped from the queue Q, we perform the following steps:

1. Generate all neighbor states $j \in \mathcal{N}(i)$ using 26-connected offsets in 3D space. This means that for each node $i = (x, y, z)$, we consider all grid points of the form:

   $$(x + \Delta x,\ y + \Delta y,\ z + \Delta z), \quad \text{where } \Delta x, \Delta y, \Delta z \in \{-r, 0, r\},$$

   with r being the map resolution. We exclude the trivial offset $(0, 0, 0)$, which corresponds to the node itself. This results in $3 \times 3 \times 3 - 1 = 26$ neighbors.

2. For each candidate neighbor j, check if it lies within the boundary and is collision-free (i.e., the edge (i, j) lies entirely in C_{free}).

3. If j passes the feasibility checks, create a new node if it does not exist, compute its cost $g(j)$, and update its parent and priority in the OPEN list if a better path is found.

4. We also include a special case: if the current node is within resolution distance from the goal x_τ, we attempt to directly connect to the goal, provided the edge is collision-free.

4.2 Heuristic

In my implementation, I use the Euclidean distance in 3D space as the heuristic function:

$$h(x) = \|x - x_\tau\|_2,$$

where x_τ is the goal state.

This heuristic is both **admissible** and **consistent**, since in a 3D Euclidean configuration space:

- Euclidean distance never overestimates the actual shortest path cost (admissibility).
- It satisfies the triangle inequality, i.e., $h(x) \leq c(x, x') + h(x')$, making it consistent.

4.3 Completeness and Optimality

The A* algorithm is both **complete** and **optimal** under standard conditions.

Optimality: In our implementation, we use a heuristic function that is both **admissible** and **consistent**. Specifically, the Euclidean distance in 3D configuration space. This guarantees that A* will return an optimal solution (i.e., the lowest-cost path), if one exists, when $\epsilon = 1$.

Epsilon-suboptimality: By introducing a parameter $\epsilon \geq 1$ to scale the heuristic:

$$f(x) = g(x) + \epsilon \cdot h(x).$$

When $\epsilon > 1$, the algorithm becomes **ϵ-suboptimal**, meaning it returns a path whose cost is at most ϵ times the optimal cost. This tradeoff allows for faster planning at the expense of guaranteed optimality.

Completeness: A* is **resolution complete**, meaning that it is guaranteed to find a solution if one exists within the discretized state space That is, if there exists a collision-free path composed entirely of valid transitions between discrete grid points (as defined by the resolution), A* will eventually find it. Resolution completeness does not guarantee finding a solution if one exists in the continuous space but cannot be represented at the given discretization level.

4.4 Comparison with Standard Graph-based A*

In standard implementations of A*, the entire configuration space is discretized, and the full graph $G = (V, E)$ is constructed before the search begins. While this approach can be efficient for small, low-dimensional maps, it becomes increasingly infeasible in high-dimensional or large-scale environments.

Advantages of Dynamic Graph Construction. In our implementation, we adopt a dynamic graph construction strategy. Instead of building the full graph, we generate neighbors of a node on demand during node expansion. This approach offers several advantages:

- **Memory Efficiency:** We avoid storing the full set of vertices and edges, significantly reducing memory usage. Only the visited nodes and their computed neighbors are maintained.
- **Lazy Evaluation:** Edges and transitions are evaluated only when needed, which reduces unnecessary collision checking and boundary validation in unexplored regions.
- **Scalability:** For sparse environments or large maps where most regions are irrelevant to the optimal path, dynamic graph expansion avoids redundant computation.

Limitations and Scalability Considerations. While the dynamic approach improves memory usage, it does not reduce the overall search complexity. In fact, due to repeated runtime evaluations (e.g., collision checking for each generated neighbor), the per-node expansion cost is higher compared to pre-built graphs.

Moreover, in very large-scale environments, discretizing the entire configuration space with sufficient resolution is itself impractical due to:

- The exponential growth of the number of nodes with finer resolution.
- The inability to store or precompute all possible transitions.
- The high cost of real-time geometric operations (e.g., obstacle checking) at each expansion.

For large maps, this implies that A* with dynamic graph construction remains computationally expensive. If the environment is static, a more efficient strategy is to construct the graph once offline and store it permanently. In such cases, a traditional graph-based A* can reuse the precomputed structure and perform planning efficiently by simply querying the stored graph data.

5 Sampling-based Method

For the sampling-based method, I directly apply the **RRT*** planner from the OMPL library. To ensure feasibility of sampled paths, I define both a state validity checker and a motion validator as follows:

- **State Validity Checker:** A state $x \in R^3$ is valid if it lies within the configuration space boundary and is not inside any obstacle, i.e., $x \in C_{\text{free}}$.
- **Motion Validator:** A motion between two states (u, v) is valid if the straight-line segment connecting them lies entirely within the free space. This is implemented using the AABB-based collision checker described earlier.

I fix certain RRT* parameters and allow others to be user-defined:

- **Fixed parameters:**

  - Goal bias: 0.05
  - Tree pruning threshold: 0.1
  - K-nearest rewiring: `False`

- **User-specified parameters:**
  - Search range (i.e., maximum edge extension distance)
  - Maximum planning time

I use OMPL's built-in path interpolation to smooth the final trajectory before extracting waypoints for post-processing or visualization.

6 Result

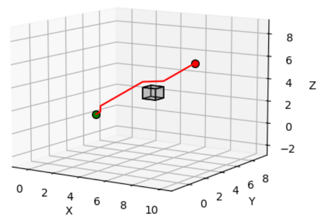
6.1 A*

In the A* experiment, I tried following set-ups

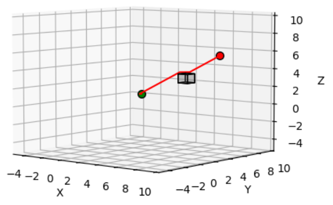
1. Map Resolution: 0.5, Eps: 1.0
2. Map Resolution: 0.2, Eps: 1.0
3. Map Resolution: 0.2, Eps: 5.0

6.1.1 Path Visualization

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

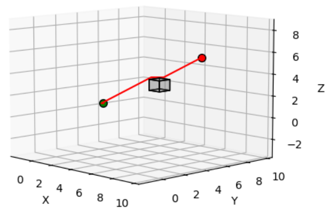
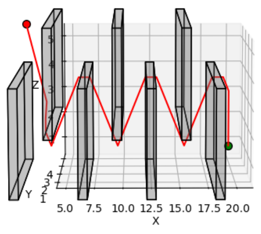
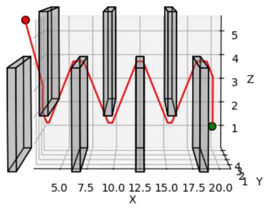


Figure 1: **A* Cube**

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

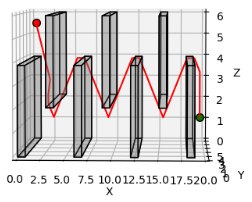
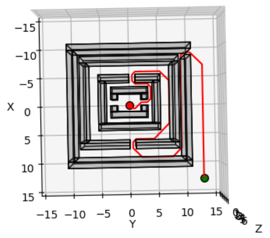
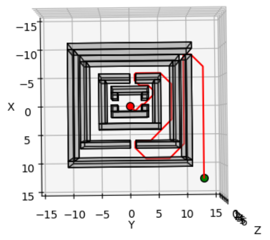


Figure 2: **A* Bird**

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

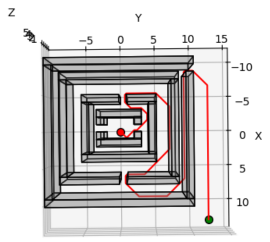
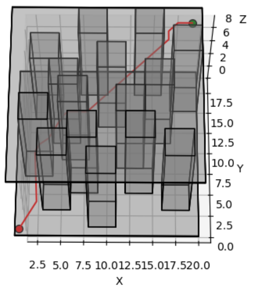
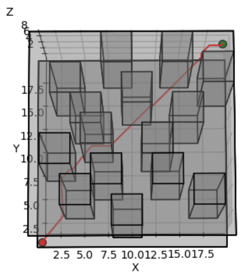


Figure 3: **A* Maze**

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

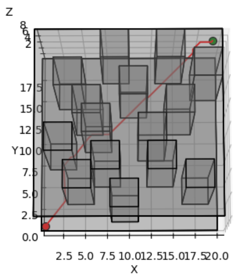
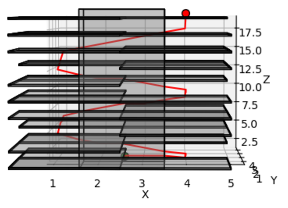
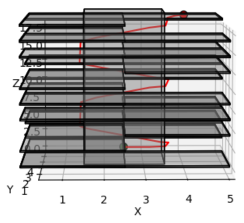


Figure 4: **A* Pillars**

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

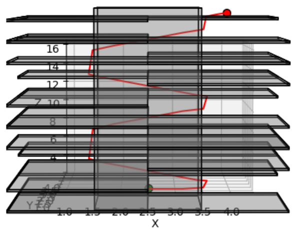
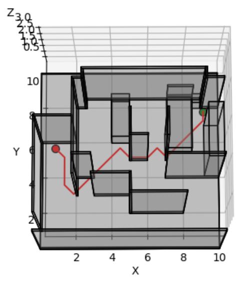
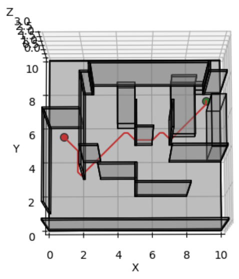


Figure 5: **A* Tower**

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

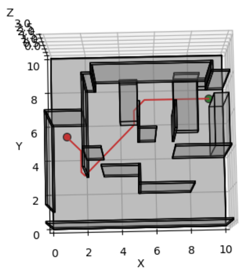
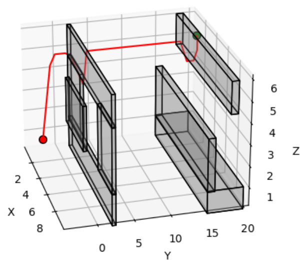
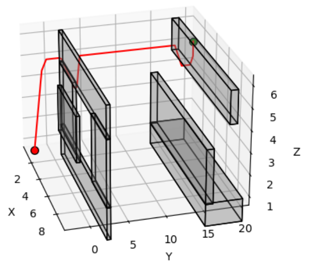


Figure 6: **A* Room**

a Map Resolution: 0.5, Eps: 1.0



b Map Resolution: 0.2, Eps: 1.0



c Map Resolution: 0.2, Eps: 5.0

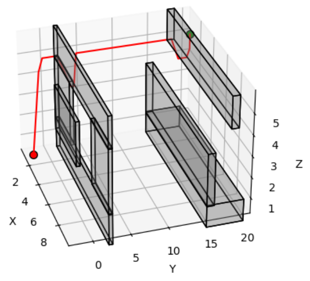


Figure 7: **A* Window**

6.1.2 Path Planning Result

Note that the Planning Time may vary across devices

Table 1: A* performance under Map Resolution = 0.5 and Epsilon = 1.0

Environment	Expanded Nodes	Time (s)	Path Length
Single Cube	551	0.2	8.32
Maze	15,694	17.0	81.56
Flappy Bird	4,820	2.5	29.63
Pillars	9,487	10.1	31.93
Window	10,819	7.3	30.17
Tower	2,823	3.2	32.85
Room	962	1.1	12.90

Table 2: A* performance under Map Resolution = 0.2 and Epsilon = 1.0

Environment	**Expanded Nodes**	**Time (s)**	**Path Length**
Single Cube	38,947	10.6	8.26
Maze	301,558	346.6	79.51
Flappy Bird	56,366	29.5	29.89
Pillars	275,521	298.4	31.32
Window	175,970	116.7	29.79
Tower	46,156	65.6	28.63
Room	20,942	28.9	12.13

Table 3: A* performance under Map Resolution = 0.2 and Epsilon = 5.0

Environment	**Expanded Nodes**	**Time (s)**	**Path Length**
Single Cube	31	0.0	8.26
Maze	132,785	166.9	79.51
Flappy Bird	40,196	22.8	29.89
Pillars	1,050	1.1	31.32
Window	63,400	47.1	29.79
Tower	19,668	29.5	28.63
Room	2,588	3.4	11.82

Comparison between Map Resolution = 0.5 and 0.2 (Epsilon = 1.0)

- **Expanded Nodes:** Across all environments, reducing the resolution from 0.5 to 0.2 leads to a significant increase in the number of expanded nodes. This is expected because lower resolution results in a finer discretization of the configuration space, creating more possible states and increasing the branching factor at each node. For example, in the Maze environment, expanded nodes increase from 15,694 to 301,558.

- **Time:** The increased number of states at lower resolution directly contributes to a dramatic increase in computation time. For instance, in the Pillars environment, planning time jumps from 10.1 seconds (res=0.5) to 298.4 seconds (res=0.2). This reflects the additional cost of maintaining and searching a much larger OPEN list.

- **Path Length:** The path length slightly improves at lower resolution. This is because the finer grid provides more precise control over the agent's movements, allowing it to follow paths that are closer to the true geometric optimum. For example, in the Tower environment, the path length reduces from 32.85 to 28.63.

While decreasing the map resolution allows for more accurate and possibly shorter paths, it greatly increases both memory consumption and runtime. This trade-off highlights the importance of selecting an appropriate resolution based on task requirements and computational constraints.

Comparison between Epsilon = 1.0 and Epsilon = 5.0 (Map Resolution = 0.2)

- **Expanded Nodes:** The number of expanded nodes is significantly reduced under $\epsilon = 5.0$. For example, in the Maze environment, expansion drops from 301,558 nodes to just 132,785. This is because a larger ϵ increases the weight of the heuristic, making the search more greedy and goal-directed, thereby reducing exploration.

- **Time:** The reduced number of node expansions directly translates into much faster planning times. In the Single Cube test, the time drops from 10.6 seconds to nearly 0.0 seconds. In more complex environments like Pillars and Room, speedups are also substantial (e.g., 298.4s $\rightarrow$ 1.1s in Pillars).

- **Path Length:** Although $\epsilon = 5.0$ breaks the optimality guarantee, the resulting paths are often surprisingly close to optimal. For instance, the path lengths in all environments differ only slightly compared to the $\epsilon = 1.0$ case (e.g., 29.89 vs. 29.89 in Flappy Bird; 31.32 vs. 31.32 in Pillars). This indicates that the inflated heuristic can still guide the search toward near-optimal paths in structured or well-behaved environments.

Using a larger ϵ introduces bounded ϵ-suboptimality, but can drastically reduce planning time and memory usage. In many cases, especially where the heuristic is informative and the search space is not highly deceptive, the path quality remains very close to optimal despite the theoretical relaxation.

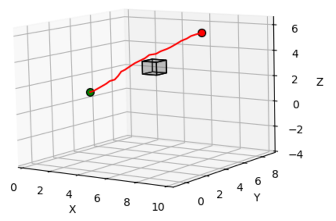
6.2 RRT*

In the RRT* experiment, I tried following set-ups

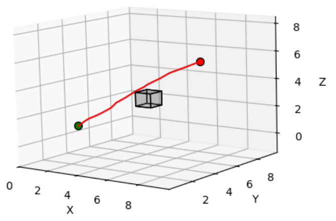
1. Search Range: 0.5
2. Search Range: 1.0
3. Search Range: 3.0

6.2.1 Path Visualization

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

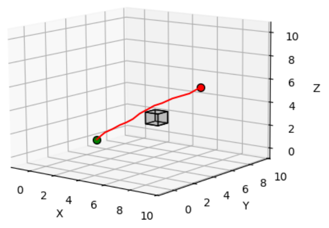
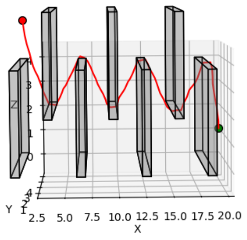
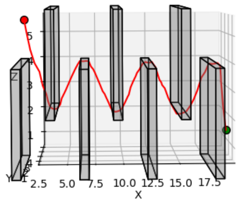


Figure 8: **RRT* Cube**

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

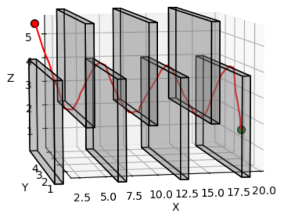
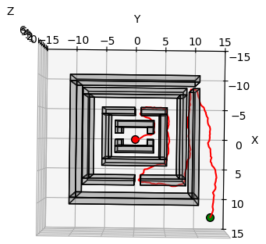
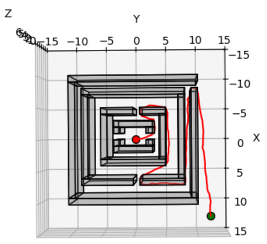


Figure 9: **RRT* Brid**

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

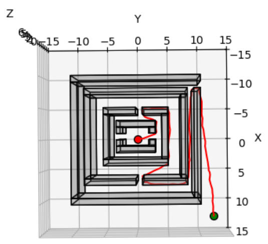
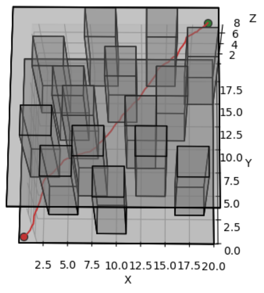
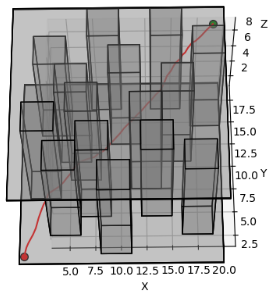


Figure 10: **RRT* Maze**

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

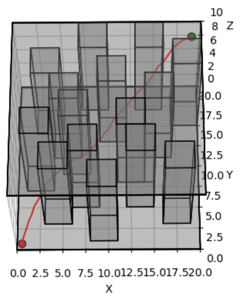
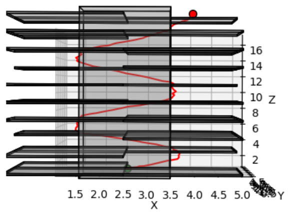
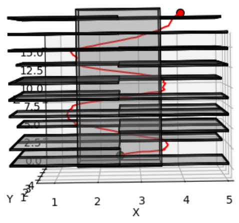


Figure 11: **RRT* Pillars**

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

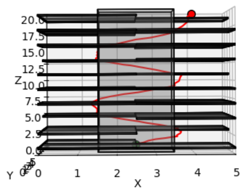
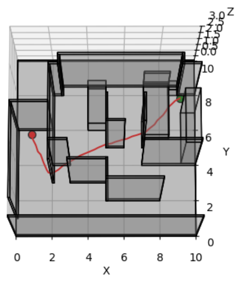
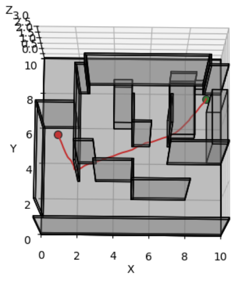


Figure 12: **RRT* Tower**

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

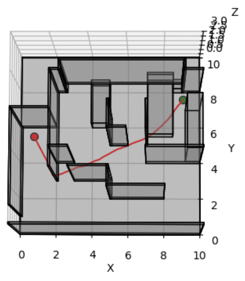
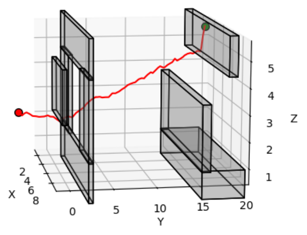
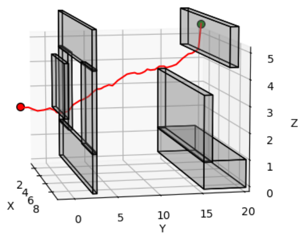


Figure 13: **RRT* Room**

a Search Range: 0.5



b Search Range: 1.0



c Search Range: 3.0

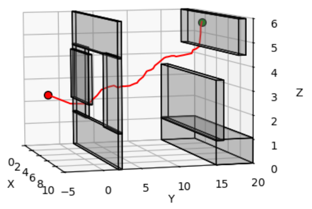


Figure 14: **RRT* Window**

6.3 Path Planning Result

Table 4: RRT* performance with Search Range = 0.5 (Time Limit = 60s)

Environment	**Time (s)**	**Created Nodes**	**Rewire Checks**	**Path Length**
Single Cube	60.1	463,347	5,925,221	8.12
Maze	60.1	169,956	1,607,685	78.46
Flappy Bird	60.2	211,272	8,569,602	26.95
Pillars	60.1	151,598	1,810,316	30.32
Window	60.1	217,387	6,661,693	26.17
Tower	60.1	161,490	6,325,126	27.58
Room	60.1	186,221	8,950,397	10.76

Table 5: RRT* performance with Search Range = 1.0 (Time Limit = 60s)

Environment	Time (s)	Created Nodes	Rewire Checks	Path Length
Single Cube	60.2	507,927	20,305,376	7.99
Maze	60.2	181,524	6,810,272	73.25
Flappy Bird	60.2	245,905	10,867,581	26.80
Pillars	60.2	205,140	9,899,100	28.79
Window	60.2	270,212	10,877,642	25.87
Tower	60.1	188,091	8,082,704	27.45
Room	60.2	232,315	12,086,357	10.69

Table 6: RRT* performance with Search Range = 3.0 (Time Limit = 60s)

Environment	Time (s)	Created Nodes	Rewire Checks	Path Length
Single Cube	60.2	541,304	22,406,411	8.02
Maze	60.2	211,962	9,095,051	72.74
Flappy Bird	60.2	387,344	18,099,219	26.70
Pillars	60.1	220,824	10,965,779	28.56
Window	60.2	393,198	16,786,499	25.94
Tower	60.2	210,569	9,196,944	27.60
Room	60.1	238,277	12,464,628	10.66

Comparison across Search Range (RRT*)

- **Path Length:** In most environments, the final path length improves (i.e., becomes shorter) as the search range increases from 0.5 to 1.0. This is because a larger range allows the planner to explore longer connections and potentially bypass local obstacles or narrow passages more efficiently. However, from 1.0 to 3.0, the improvement in path length is marginal. For instance, in the Maze environment, path length goes from 78.46 (range=0.5) to 73.25 (range=1.0), then slightly improves to 72.74 (range=3.0).
- **Created Nodes:** The number of created states generally increases with the search range. For example, in the Single Cube environment, created nodes go from 463k (range=0.5) to 507k (range=1.0), then to 541k (range=3.0).
- **Rewire Attempts:** Rewire attempts grow significantly with increased search range. This is expected because a larger range increases the number of neighbor candidates to be considered for rewiring during each iteration. For example, the number of rewire checks in Window increases from 6.7 million (range=0.5) to 10.9 million (range=1.0) and then to 16.8 million (range=3.0).

Increasing the search range improves path quality up to a point, but also leads to higher computation in terms of state generation and rewiring. A moderate value (e.g., 1.0) seems a good balance between exploration and refinement.