ECE 276B Project 2: Motion Planning

1st Yifan Wu
UCSD ECE
yiw084@ucsd.edu

***Abstract*—This project will focus on comparing the performance of search-based and sampling-based motion planning algorithms in 3-D Euclidean space.**

I. Introduction

Motion planning is a fundamental problem in robotics. Finding a path for an agent that it can move along this path from agent's initial configuration to goal configuration without colliding with any static obstacles or other agents in the environment. One solution to these type of problems is to apply dynamic programming algorithm. Design and formulate the problem as a Markov Decision Process (MDP), the goal of the MDP problem is to reach the goal state with lowest cost.

Motion planning in robotics can also be formulate as a deterministic shortest path (DSP) problem with continuous state and control spaces in a known environment. When the robot body is not a point, the problem is also known as the Piano Movers Problem.

The objective of motion planing is to find a feasible and cost-minimal path from the current configuration of the robot (start position) to a goal position or region with consideration of cost (distance, time, energy, risk, etc.) and constraints (environment constraints, kinematics/dynamics of the robot).

There are several motion planning approaches:

1) Exact algorithms
2) Search-based planning algorithms
3) Sampling-based planning algorithms

Since Exact algorithm is computationally expensive and unsuitable for high-dimensional spaces. In this project, we are going to compare the performance of search-based and sampling-based motion planning algorithms and find the feasible of agent in 3D Euclidean space.

II. Problem Statement

A. Environment

We are going test the performance of search-based and sampling-based motion planning algorithms in continuous 3-D Euclidean space.

A set of 3-D environments described by a rectangular outer boundary with respect to x-axis, y-axis, z-axis are provided. Each boundary is specified with $[X_{min}, X_{max}]$, $[Y_{min}, Y_{max}]$, and $[Z_{min}, Z_{max}] \in R$

In each environment, there exists a set of rectangular obstacle blocks. Let $i \in Z^+$ denoted the label of each block. Each rectangle is described by a 9 dimensional vector, specifying its lower left corner (xi_{min}, yi_{min}, zi_{min}), its upper right corner (xi_{max}, yi_{max}, zi_{max}), and its RGB color, where xi_{min}, yi_{min}, zi_{min}, xi_{max}, yi_{max}, $zi_{max} \in R$, and RGB value in range [0, 255].

The start $x_s \in R^3$ and goal $x_\tau \in R^3$ coordinates are also specified for each of the available environments. A sample environment is shown in Figure 1.

B. Configuration Space

In our case, we defined Configuration space $\mathcal{C}$ as a set of 3-tuple or a 3D tensor (i, j, k) to represent the space inside the outer boundary, where

$$(i,j,k) = \begin{cases} X_{min} < i < X_{max} \\ Y_{min} < j < Y_{max} \\ Z_{min} < k < Z_{max} \end{cases} \in \mathbb{R}^3 \tag{1}$$

Similarly, define the Obstacle space $\mathcal{C}_{obs}$ as a set of 3D tensor$(i_{obs}, j_{obs}, k_{obs})$ represent the space occupied by the set of blocks in each specified environment, where

$$(i_{obs}, j_{obs}, k_{obs}) = \begin{cases} i_{obs} \in [xi_{min}, xi_{max}] \\ j_{obs} \in [yi_{min}, yi_{max}] \\ k_{obs} \in [zi_{min}, zi_{max}] \end{cases} \in \mathbb{R}^3 \tag{2}$$

The Free space $\mathcal{C}_{free}$ is also defined as a set of 3D tensor $(i_{free}, j_{free}, k_{free})$ represent the space not occupied by the set of blocks in each specified environment and inside the Configuration Space. Initial state $x_s \in \mathcal{C}_{free}$, and Goal state $x_\tau \in \mathcal{C}_{free}$

$$\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs} \in \mathbb{R}^3 \tag{3}$$

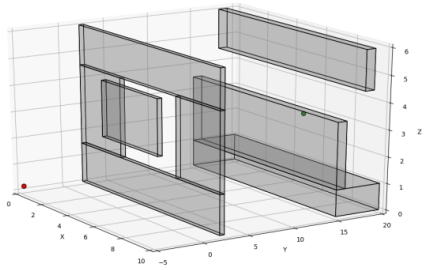


Fig. 1. *A sample environment showing the obstacles in gray, start position in red, and goal position in green.*

C. Deterministic Shortest Path

1) DSP Formulation in Search-based Planning: Generates a graph by systematically discretizing $\mathcal{C}_{free}$. Once a graph is constructed (via cell decomposition, skeletonization, lattice, etc.), we are able to formulate the problem as follow:

- Assume there are no negative cycles in the graph i.e., $\mathcal{J}^{i_{1:q}} \geq 0$ for all $i_{1:q} \in \mathcal{P}_{i,j}$ and all $i \in \mathcal{V}$
- Path: a sequence $i_{1:q} := (i_1, i_2, ..., i_q)$ of nodes $i_k \in \mathcal{V}$
- All paths from $s \in \mathcal{V}$ to $\tau \in \mathcal{V}$:

$$\mathcal{P}_{s,\tau} := \{i_{i:q} | i_k \in \mathcal{V}, i_1 = s, i_q = \tau\} \tag{4}$$

- Path length: sum of edge weights along the path

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

- Motion Planning Objective: find a path that has the min length form node s to node τ such that

$$dist(s, \tau) = min_{i_{1:q} \in \mathcal{P}_{s,\tau}} \mathcal{J}^{i_{1:q}} \tag{6}$$

2) DSP Formulation in Sampling-based Planning: Generates a sparse sample-based graph in $\mathcal{C}_{free}$, we are able to formulate the problem as follow:

- Initial state $x_s \in \mathcal{C}_{free}$, and Goal state $x_\tau \in \mathcal{C}_{free}$
- Path: a continuous function $\rho : [0, 1] \to \mathcal{C}_{free}$. Denoted set of all path as P.
- Feasible path: a continuous function $\rho : [0, 1] \to \mathcal{C}_{free}$ such that $\rho(0) = x_s$ and $\rho(1) = x_\tau$. Denoted set of all feasible paths as $P_{s,\tau}$.
- Motion Planning Objective: Given a path planning problem (C_{free}, x_s, x_τ) and a cost function $\mathcal{J} : P \to \mathbb{R}_{\geq 0}$, find a feasible path ρ^* such that:

$$\mathcal{J}(\rho^*) = \min_{\rho \in P_{s,\tau}} \mathcal{J}(\rho) \tag{7}$$

- Report failure if no such path exists

III. Technical Approach

There are several motion planning approaches:

1) Exact algorithms
2) Search-based planning algorithms
3) Sampling-based planning algorithms

Since Exact algorithm is computationally expensive and unsuitable for high-dimensional spaces. In this project, we are going to compare the performance of search-based and sampling-based motion planning algorithms and find the feasible of agent in 3D Euclidean space.

A. Motion Planning as Graph Search Problem

Motion planning as a deterministic shortest path problem on a graph:

1) Decide:
   a) Pre-compute the C-Space (e.g., inflate the obstacles with the robot radius)
   b) Perform collision checking on the fly
2) Construct a graph representing the planning problem
3) Search the graph for a (close-to) optimal path

B. Graph Construction

In this project, we use cell decomposition to decompose the free space into simple cells and represent its connectivity by the adjacency graph of these cells.

Since our C-space is in continuous 3-D Euclidean space, we discretize the space into discrete 3D Grid with resolution 0.1. $\mathcal{X}$ is a discrete set of states of cells denoted as (i, j, k), where i is location on x-axis of 3D gird, j is location on y-axis of 3D gird and k is location on z-axis of 3D gird. Using this 3D grid or 3D tensor as a discrete representation of Configuration space C, where value of each entry can be either 0 or 1 representing free space or obstacle.

$$\mathcal{X}(i, j, k) = \begin{cases} 0 = \text{Free} \in \mathcal{C}_{free} \\ 1 = \text{Occupied} \in \mathcal{C}_{obs} \end{cases} \tag{8}$$

Therefore, we pre-compute the C-Space and encode the environment info in to the 3D Grid.

A 26-connected grid is construct over the 3-dimensional integer lattice. Each node i has coordinates $x_i \in Z^3$, where x_i is a 3-dimensional vector whose entries are integer. Node i has 26 neighbors j with coordinates x_j obtained by adding a vector $d \in Z^3$ to x_i, where d is an element of the set:

$$\left\{(a, b, c)^\top | a, b, c \in \{-1, 0, 1\}\right\} \backslash \left\{(0, 0, 0)^\top\right\} \tag{9}$$

Define the cost function as Euclidean Distance, we exclude the case that agent stay in the same state to ensure there are no negative cycles in graph.

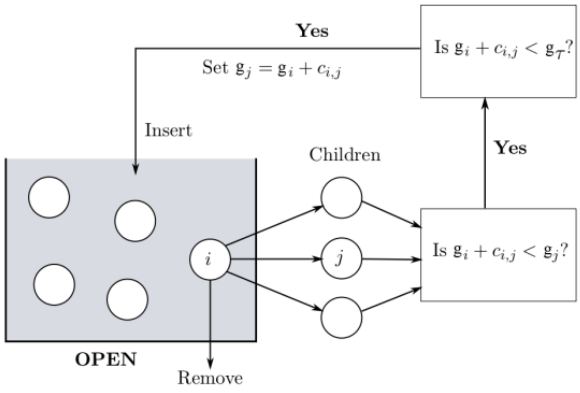


Fig. 2. *Label Correcting Algorithm*

C. Label Correcting Methods for the DSP Problem

The forward DPA computes the shortest paths from the star s to all nodes. Many nodes are not part of the shortest path from s to τ. We apply label correcting (LC) algorithms for the DSP problem so that we do not necessarily visit every node of the graph.

- **Label** g_i: estimate of the optimal cost from s to each visited node $i \in \mathcal{V}$
- Each time g_i is reduced, the labels g_j of the children of i are corrected: $g_j = g_i + c_{ij}$

- **OPEN**: set of nodes that can potentially be part of the shortest path to τ

If there exists at least one finite cost path from s to τ, then the Label Correcting (LC) algorithm terminates with $g_\tau = dist(s, \tau)$, the shortest path length from s to τ. Otherwise, the LC algorithm terminates with $g_\tau = \infty$

D. A* Algorithm

The A* algorithm is a modification to the LC algorithm in which the requirement for admission to OPEN is strengthened:

$$\textbf{from } g_i + c_{ij} < g_\tau \textbf{ to } g_i + c_{ij} + h_j < g_\tau \tag{10}$$

where h_j is a positive lower bound on the optimal cost from node j to τ known as a heuristic function:

$$0 \leq h_j \leq \textbf{dist}(j, \tau) \tag{11}$$

There are 3 kinds of states:

- **CLOSED**: set of states that have already been expanded
- **OPEN**: set of candidates for expansion
- **Unexplored**: the rest of the states

E. Heuristic Function

A heuristic function h_i is constructed using special knowledge about the problem. It tells A* an estimate of the minimum cost from any vertex to the goal. Heuristic must be admissible for the A* Algorithm to work correctly. It is admissible if:

$$h_i \leq \textbf{dist}(j, \tau) \text{for all } i \in \mathcal{V} \tag{12}$$

A heuristic may be consistent to make the A* algorithm more efficient. It is consistent if:

- $h_\tau = 0$
- $h_i \leq c_{ij} + h_{ij}$ for all $i \neq \tau$
- $j \in$ Children(i)

A heuristic can be ϵ-Consistent if:

- $h_\tau = 0$
- $h_i \leq \epsilon c_{ij} + h_{ij}$ for all $i \neq \tau$
- $j \in$ Children(i)
- $\epsilon \geq 1$

F. Choice of Heuristic Function

In this project we implement 4 common choice of heuristic function. In Grid-base planning, let $x_i \in \mathbb{R}^3$ be the position of node i:

1) Manhattan distance: $h_i := ||x_\tau - x_i||$
2) Euclidean distance: $h_i := ||x_\tau - x_i||_2$
3) Diagonal distance: $h_i := ||x_\tau - x_i||_\infty$
4) Octile distance: $h_i := ||x_\tau - x_i||_\infty + ||x_\tau - x_i||_{-\infty}$

Since we are using 26-connectivity grid and Euclidean distance as cost, we ensure option 1 to 3 are admissible. According to triangle inequality, using Octile distance as heuristic might not be admissible. Hence, for Octile distance, we modify it into:

$$h_i := ||x_\tau - x_i||_\infty + 0.7 \cdot ||x_\tau - x_i||_{-\infty} \tag{13}$$

in order to make it admissible.

Algorithm 2 Weighted A* Algorithm

```
1:  OPEN ← {s}, CLOSED ← {}, ε ≥ 1
2:  g_s = 0, g_i = ∞ for all i ∈ V \ {s}
3:  while τ ∉ CLOSED do                          ▷ τ not expanded yet
4:      Remove i with smallest f_i := g_i + ε h_i from OPEN   ▷ means g_i + ε h_i < g_τ
5:      Insert i into CLOSED
6:      for j ∈ Children(i) and j ∉ CLOSED do
7:          if g_j > (g_i + c_ij) then
8:              g_j ← (g_i + c_ij)
9:              Parent(j) ← i                    expand state i:
10:             if j ∈ OPEN then                 ◦ try to decrease g_j using path from s to i
11:                 Update priority of j
12:             else
13:                 OPEN ← OPEN ∪{j}
```

Fig. 3. *Weighted A* Algorithm*

G. A* Algorithm with an ϵ-consistent Heuristic

A difference form Dijkstra's Algorithm is that, we are not removing the node with current lowest cost, but removing the node with current lowest cost plus epsilon time the estimation of the cost to go, f_i:

$$f_i := g_i + \epsilon h_i \tag{14}$$

1) A* algorithm will assign cost of zero to our initial node s and infinity (∞) for cost all other nodes.
2) Create a Parent Dictionary to keep track of the parent node of child node with lowest cost for reconstruct the shortest path
3) Initialize a empty Indexed priority queue (PQ) called OPEN to stores (node, estimated cost f_i) pair sorted by minimum cost order. Insert (s, 0) to the OPEN in order to kick start the algorithm
4) Initialize a empty Indexed priority queue (PQ) called CLOSED.
5) While goal node not in CLOSED or OPEN is not empty, remove the next most promising (node, f_i) pair. and insert this node to CLOSE
6) Once the node is in CLOSED, it will never be visited again.
7) For the current node, consider all of its children (26-connectivity) and calculate cost through the current node.
8) Compare the new cost to the current assigned value. If the new cost is smaller, then assign new f_i value to the node and insert a new (node, f_i) pair into OPEN. if the node is already in OPEN update it's index and f_i.
9) If goal node in CLOSED, break the iteration and the algorithm has finished.
10) If OPEN is empty, FAIL.

The Algorithm of weighted A* is shown in figure 3

H. Weighted A* Algorithms Analysis

1) Completeness: a planning algorithm is called complete if it:

- Returns a feasible solution, if one exists,
- Returns FAIL in finite time, otherwise.

A* terminates in a finite number of iterations if $\mathcal{V}$ is finite or if $c_{ij} > 0$ for $i, j \in \mathcal{V}$ and the degree of each node $i \in \mathcal{V}$ is finite. In this case, our state is discrete and finite, there is no

negative cycle and each node i has at most 26 children. We ensure it terminate in a finite number of iteration.

2) Optimality:

- A planning algorithm is optimal if it returns a path with shortest length J^* among all possible paths from start to goal
- A planning algorithm is ϵ-suboptimal if it returns a path with length $J \leq \epsilon J$ for $\epsilon \geq 1$ where J^* is the optimal length

The optimality of A* depends on the heuristics.

- If A* uses a consistent heuristic, then it is guaranteed to return an optimal path to goal
- If A* uses an admissible but inconsistent heuristic, then it is guaranteed to return an optimal path as long as closed states are re-opened
- If A* uses an ϵ-consistent heuristic, then it is guaranteed to return an ϵ-suboptimal path with cost $\textbf{dist}(s,\tau) \leq g_\tau \leq \epsilon\ \textbf{dist}(s,\tau)$

Since we are using weighted A* Algorithm, we are inflating the consistency. By multiply ϵ to heuristic, we will no longer get the optimal path, but we are able to expand fewer nodes. Using Manhattan distance, Euclidean distance, Diagonal distance and Octile distance as heuristic function in weighted A*. These are underestimations of the cost in most complex obstacle environment. Therefore, we ensure the heuristic is admissible and ϵ-consistent to obtain ϵ-suboptimal path. Note that, although we use Euclidean distance as cost, the true cost from node to goal may not be a direct line in complex environments. Since Euclidean distance allows any direction of movement, A* could found paths on the grid but agent is not allow to move to that position directly.

In our implementation of weighted A*, it expands states in the order of $f_i = g_i + \epsilon h_i$. If the heuristic is not consistent it is not guaranteed to expand no more states than A*

3) Memory: A* does provably minimum number of expansions to find the optimal solution but this might require an infeasible amount of memory. It is still the case for weighted A*

4) Time Efficiency: As mention above, as the heuristic is consistent, A* performs the minimal number of state expansion to guarantee Opimality. On the other hand, weighted A* is ϵ-suboptimal. It trades optimality for speed. Weighted A* checks fewer node so that the path cannot be optimal, but it's much faster than A*. The common time complexity for A* is $O(|V|log|V|)$.

I. Sampling-based Planning

Sampling-based Planning generates a sparse sample-based graph in C_{free}. By searching the graph for a path, it guarantees that the probability of finding one if it existes approaches 1 as the number of iteration approaches ∞. It provides asymptotic suboptimality bounds on the solution. Also, it can be used in high-dimensional planning which can be faster and requires less memory than search-based planning in many domains.

1) Primitive Procedures for Sampling-based Motion Planning:

- SAMPLE: returns iid samples from C
- SAMPLEFREE: returns iid samples from C_{free}
- NEAREST: given a graph $G = (V, E)$ with $V \subset C$ and point $x \in C$ returns a vertex $v \in V$ that is closest to x:

$$NEAREST((V, E), x) := \arg\min_v ||x - v|| \quad (15)$$

- NEAR: given a graph $G = (V, E)$ with $V \subset C$ and point $x \in C$, returns the vertices in V that are within a distance r from x:

$$NEAR((V, E), x, r) := v \in V|\ ||x - v|| \leq r \quad (16)$$

- STEER: given points x, $y \in C$ and $\epsilon > 0$, returns apoint $z \in C$ that minimizes $||z - y||$ whiles remaining within ϵ from x:

$$STEER_\epsilon(x, y) := argmin_{z:||z-x||\leq\epsilon}||z - y|| \quad (17)$$

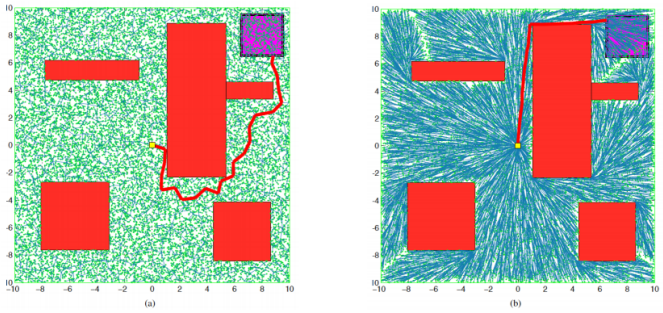
- COLLISIONFREE: given points x, $y \in C$, return TRUE if the line segment between x and y lies in C_{free} and FALSE otherwise.

J. Rapidly Exploring Random Tree (RRT)

A tree constructed from random samples with root x_s. The tree is grown until it contains a path to x_τ . RRTs are well-suited for single-shot planning between a single pair of x_s and x_τ which is exactly the case in our project. RRT can be implemented in the original workspace (need to do collision checking) or in configuration space.

The procedure of RRT: starting with an initial configuration x_s build a graph until the goal configuration x_τ is part of it.

- Sample a new configuration,x_{rand}, find the nearest neighbor $x_{nearest}$ in G and connect them
- if $x_{nearest}$ lies on an existing edge then split the edge
- If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by a collision detection algorithm.
- Occasionally (e.g., every 100 iterations) add the goal configuration x_τ and see if it gets connected to the tree



(a) RRT (b) RRT*

Fig. 4. *RRT vs RRT**

K. RRT*

RRT is probabilistically complete: the probability that a feasible path will be found if one exists, approaches 1 exponentially as the number of samples approaches infinity. However, RRT is not optimal. The probability that RRT converges to an optimal solution, as the number of samples approaches infinity, is zero.

On the other hand, RRT* combines RRT plus rewiring of the tree to ensure asymptotic optimality. It contains two steps: extend and rewire. The extend step is similar to RRT. The rewire step utilize the concept of label correcting. The performance difference is shown in Figure 4

1) RRT: Extend Step:*

- Generate a new potential node x_{new} identically to RRT
- Instead of finding the closest node in the tree, find all nodes within a neighborhood N of radius, $min(r^*, \epsilon)$ where

$$r^* > 2(1+\frac{1}{d})^{\frac{1}{d}} \cdot (\frac{Vol(C_{free})}{Vol(Unitd-ball)})^{\frac{1}{d}} \cdot (\frac{log|V|}{|V}|)^{\frac{1}{d}} \tag{18}$$

- Let $x_{nearest} = argmin_{x_{near} \in N} g(x_{near}) + c(x_{near}, x_{new})$ be the node in N on the currently known shortest path from x_s to x_{new}
- $V \leftarrow V \cup \{x_{new}\}$
- $E \leftarrow E \cup \{x_{nearest}, x_{new}\}$
- Set the label of x_{new} to:

$$g(x_{new}) = g(x_{nearest}) + c(x_{nearest}, x_{new}) \tag{19}$$

2) RRT: Rewire Step:*

- Check all nodes $x_{near} \in N$ to see if re-routing through x_{new} reduces the path length
- if $g(x_{new}) + c(x_{new}, x_{near}) < g(x_{near})$, then remove the edge between x_{near} and its parent and add a new edge between x_{near} and x_{new}

As shown in Figure 4, both RRT and RRT* have same nodes in the tree. Only the edge connections are different. After applying rewire RRT*'s edges are almost straight line which implies optimal path.

Algorithm 6 RRT*

```
V ← {x_s}; E ← ∅
for i = 1 ... n do
    x_rand ← SampleFree()
    x_nearest ← Nearest((V, E), x_rand)
    x_new ← Steer(x_nearest, x_rand)
    if CollisionFree(x_nearest, x_new) then
        X_near ← Near((V, E), x_new, min{r*, ε})
        V ← V ∪ {x_new}
        c_min ← Cost(x_nearest) + Cost(Line(x_nearest, x_new))
        for x_near ∈ X_near do                    ▷ Extend along a minimum-cost path
            if CollisionFree(x_near, x_new) then
                if Cost(x_near) + Cost(Line(x_near, x_new)) < c_min then
                    x_min ← x_near
                    c_min ← Cost(x_near) + Cost(Line(x_near, x_new))
        E ← E ∪ {(x_min, x_new)}
        for x_near ∈ X_near do                    ▷ Rewire the tree
            if CollisionFree(x_new, x_near) then
                if Cost(x_new) + Cost(Line(x_new, x_near)) < Cost(x_near) then
                    x_parent ← Parent(x_near)
                    E ← (E \ {(x_parent, x_near)}) ∪ {(x_new, x_near)}
```

Fig. 5. *RRT* Algorithm*

L. OMPL Implementation of RRT*

In the experiment of RRT*, we used the state-of-the-art sampling-based motion planning library OMPL [1]. We set the objective to be path length which means our goal is to find the shortest and optimal path.

Since RRT* use random sampling, we test each environment using RRT* with 6 different seed of random generator to find a suitable planning time which ensure we will have a desired result path from x_s to x_τ. Two types of validation checker were used to check if the state or action to that state is valid which will be discussed in detail in subsection M.

M. Collision Detection

As mentions before, we have two options to start the motion planning

- Pre-compute the C-Space (e.g., inflate the obstacles with the robot radius)
- Perform collision checking on the fly

1) Collision Detection in A:* In our implementation of path planning with weighted A* Algorithm, we discretized the environment in to a 3D Gird. Meanwhile, we also encode the information of boundary and blocks into the 3D Grid.

Therefore, we pre-compute the C, C_{obs}, and C_{free} into 3D Grid where value of each entry can be either 0 or 1 representing free space or obstacles as described in equation(8). This pretty much prevent collision happen when planning with A*.

In advance, we utilize Flexible Collision Library (FCL) [2] to perform collision checking. We consider boundary or obstacle as a hyper-rectangle or a box, and our agent as a sphere with tiny volume. The radius of agent can be relatively small compared to the specified environment. In the planning time, we perform one-to-many Discrete Collision Checking with contact allowed instead of one-to-one Continues collision checking.

This is reasonable approach since we convert the configuration space into Discrete 3D Gird, and obstacle's shape is simple. This approach will speed up the collision checking process dramatically.

2) Collision Detection in RRT:* In our implementation of path planning with RRT* Algorithm, we choose to perform collision checking on the fly. There are two validation checking we need to perform:

- State Validation: check if the state is valid
- Motion Validation: check if the action to the state is valid

The state validation is exactly the same as above. Considering agent as sphere and boundary or block as box to perform one-to-many Discrete Collision Checking.

For Motion Validation, we perform a combination of line-segment collision checking and continuous motion collision checking.

1) Given two state s_1 and s_2 and consider those two state as point first, where s_2 is the child node of s_1. Draw a line between s1 and s2 and form a line segment.
2) Consider obstacles as axis-aligned bounding boxes (AABBs) in continuous 3D space, check if this line intersect with any AABB in the environment.

3) If intersect exists, return FALSE (Motion is not Valid).
4) Otherwise, consider agent at state s_1 as a tiny sphere and perform continuous motion collision checking.
5) if collision occurs, return FALSE (not Valid). Otherwise, return True (Motion is Valid)

There are 2 reason to preform double collision checking. For time concern, one-to-one continuous collision checking is typically slow when there are lots of obstacles in the environment. The time complexity is $O(n^2)$. Line segment collision checking has advantage over speed and efficiency. However, for generalization concern, the assumption that agent is a point can not be true in real world. We can think of using line segment collision checking as a filter which filter out absolutely invalid action and leave potential feasible and valid actions for continuous collision checking to do the job. The combination of motion validation outperform single motion validation checking either in line segment collision detection or continuous collision detection. Also, it is more generalize to much complex environments.

TABLE I
WEIGHT A* ALGORITHM WITH ϵ 2

Test Env	Weight A* Algorithm		
	Path Length	*Considered Nodes*	*heuristic function*
single cube	11.85	2667	Manhattan
single cube	11.09	1678	Euclidean
single cube	9.971	6195	Diagonal
single cube	11.4	3084	Octile
monza	83.08	13470	Manhattan
monza	79.97	47072	Euclidean
monza	85.84	28689	Diagonal
monza	83.23	35818	Octile
flappy bird	32.31	21589	Manhattan
flappy bird	33.07	22151	Euclidean
flappy bird	32.14	22859	Diagonal
flappy bird	31.52	16008	Octile
window	34.21	11954	Manhattan
window	29.78	7950	Euclidean
window	30.68	8414	Diagonal
window	31.04	9582	Octile
room	18.43	9050	Manhattan
room	15.7	10628	Euclidean
room	13.64	6993	Diagonal
room	13.11	5649	Octile
maze	84.47	30417	Manhattan
maze	86.25	121180	Euclidean
maze	110.27	268500	Diagonal
maze	84.19	92652	Octile
tower	36.1	18919	Manhattan
tower	34.76	24199	Euclidean
tower	35.13	23653	Diagonal
tower	36.54	21799	Octile

IV. RESULT

The visualization of path of each environments are include in Appendix.

A. Weighted A* vs A*

In Search-based Motion Planning Algorithm, we implement our own version of weighted A* with ϵ-consistence. We test the algorithm under 7 different 3D Euclidean environments.

The performance results of Weighted A* with ϵ 2 is shown in Table 1, and the performance result of classic A* Algorithm(which has ϵ 1) is shown in Table 2. The red color marks the shortest path length and its heuristic function, and the blue color marks the least number of node explored with its heuristic function. If the candidate achieve both it is marked with orange color.

1) Path Length: The rest of environments are more complex and difficult than "single cube". Motion Planning in some environments like "Maze" can be very challenging. From opitmality perspective, A* definitely find a shorter path than Weight A*. In most environment A* almost outperform weighted A* by at least 15%.

2) Number of Considered Nodes: The results consist with the theory that A* performs the minimal number of state expansions to guarantee optimality. Any path with node expansions less than A*'s is not optimal. Weighted A* trade the optimality with speed. Although A* outperforms weighted A*, some path of weighted A* is very close. For instance, in window, the best result in A* is 26.92 with 193359 nodes expanded with Octile Distance as heuristic. In weighted A*, Euclidean Distance as heuristic has a path length of 29.78 with only 7950 nodes expanded. In this case, we trade off 2.86 meters for speed which is 24 times faster. Even with the same heuristic function in weighted A*, the path length is 31.04 with only 9050 node expanded which is 21 times faster than A*.

3) Effect of Heuristic: There is a simple toy environments, "single cube", with only one obstacles. Agent can almost go straight to from start position to goal. We can see in this case Euclidean performs well in both weight A* and A*.

In complex environments, Manhattan or Octile distance performs better, since the path from start to goal will not be a straight line. There are several challenging environments such as maze which requires agent to explore and maneuver far away from the goal in order to get out of the maze. We can see Manhattan distance expand least number of node in most complex environments either in Weight A* or A*. Weighted A* guaranteed to expand no more states than A* as long as the heuristic is consistent.

4) Memory consumption: A* does provably minimum number of expansions, $O(|V|)$, to find the optimal solution but this might require an infeasible amount of memory. Since Weighted A* is able to expand less nodes, the memory requirements of weighted A* are often better, but this is not always true.

B. RRT*vs A*

RRT is probabilistically complete but not optimal.On the other hand, RRT* combines RRT plus rewiring of the tree to ensure asymptotic optimality. The performance results are shown in Table 3. The path length is the mean value across 4 different random seed. As we provide RRT* enough time for planing, it almost ensure a feasible path if it exists. As we increase the time of planning time, the performance of RRT* increase as well. However, this is not always the case as we

TABLE II
A* ALGORITHM WITH ϵ 1

Test Env	Weight A* Algorithm		
	Path Length	*Considered Nodes*	*heuristic function*
single cube	11.68	3110	Manhattan
single cube	8.76	12500	Euclidean
single cube	13.61	1105397	Diagonal
single cube	9.16	910123	Octile
monza	76.75	17253	Manhattan
monza	82.76	104315	Euclidean
monza	86.77	98341	Diagonal
monza	76.80	56443	Octile
flappy bird	28.74	33993	Manhattan
flappy bird	34.5	111025	Euclidean
flappy bird	30.05	73738	Diagonal
flappy bird	26.92	56777	Octile
window	29.96	99678	Manhattan
window	30.48	132182	Euclidean
window	29.75	185047	Diagonal
window	26.82	193359	Octile
room	14.92	9098	Manhattan
room	12.82	51332	Euclidean
room	14.5	53747	Diagonal
room	12.19	20109	Octile
maze	76.4	70929	Manhattan
maze	91.33	1222020	Euclidean
maze	99.29	586764	Diagonal
maze	75.9	166613	Octile
tower	33.24	25478	Manhattan
tower	32.97	74360	Euclidean
tower	33.4	62271	Diagonal
tower	31.28	41638	Octile

see in room environment. One reasonable explanation is that since we run RRT* with different random seed, it is possible in one or several experiments the path is much longer. As long as we provide more than sufficient time, RRT* will provide a asymptotic optimal path as time increase.

TABLE III
RRT* ALGORITHM

Test Env	RRT* Algorithm Limited Time	
	Mean Path Length	*Plan Time(sec)*
single cube	7.96	5
single cube	7.89	10
monza	74.12	300
monza	73.71	1000
flappy bird	27.27	30
flappy bird	26.78	60
window	24.26	30
window	24.22	60
room	11.36	30
room	11.51	800
maze	82.50	300
maze	77.59	300
tower	29.04	60
tower	28.70	300

1) Quality of Computed Paths and efficiency: Since OMPL is a state-of-art sampling-base library implemented in C++, it may be more memory and computation efficient than our own implementation of A*. We can see the difference is not huge as well. If the dimensionality of state space is low, A* is able to find a path faster and ensure its optimality. The reason why RRT* might be better than our implementation of A* is that, we convert the environment into a 3D Grid with 26-connectivity. RRT* can find children in any directions. Also, we use resolution 1 in A*, RRT* might take action much smaller since the action space is continuous. We can see from the plots of A* the edge connect the path is not always a straight line. In some case it is more like a not smoothed twisted curve. RRT* applied rewire to connect node with smaller cost without collision. Path Smoothing may also applied automatically by OMPL connect and smooth the path.

The advantage of RRT* is that it is quite easy to implement as shown in section K in Technical Approach. It creates a sparse graph so that it requires little memory and computation, and find feasible paths quickly in practice. It can also add heuristics on top to make the searching algorithm more efficient.

The disadvantage of RRT* is thaat the computed path may be sub-optiaml and require path smoothing as a post processing step. Finding a feasible path in highly constrained environments for instance "maze" environment is challenging and requires a long time to compute. Beside, RRT* doesn't works well with narrow path. It is likely to be trapped and never get out of the environments(Bug Traps) with narrow exits.

C. Potential Improvement of A*

From the above discussion, there can be some improvement on our A* implementation to catch up to the performance of RRT*.

1) Breaking Ties: In some environments, there might exists many path with the equal length. A* might explore all the path with the same f value. Since A* sorts by f value, making the f value diff will ensure only one of the f value will be explored. One way to break ties is to add a deterministic random number to the heuristic. Or, computes the sum of vector cross-product between the start to goal vector and the current point to goal vector to let a* prefer paths that are along the straight line from the starting point to the goal. vector cross-product approach works better only when the environment is simple and direct path can be obtain.

2) Reduce the number of nodes or visit : Alternative Map Representation can solve the problem by reducing the number of nodes in the graph. Some approaches leave the number of nodes alone but reduce the number of nodes visited for instance jump Point search. The main idea is that accessing the contents of many points on a grid in a few iterations of A* is more efficient than maintaining a priority queue over many iterations of A*

3) Improve Heuristic: A heuristic that is exactly equal to the exact distance is ideal for making A* correct and efficient, but this is usually impractical. ALT A* [3] uses "landmarks" and triangle inequality to preprocess the path finding graph in order to make path finding much faster.

REFERENCES

[1] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, https://ompl.kavrakilab.org.

[2] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 3859–3866.

[3] A. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," Tech. Rep. MSR-TR-2004-24, July 2004. [Online]. Available: https://www.microsoft.com/en-us/research/publication/computing-the-shortest-path-a-search-meets-graph-theory/

V. APPENDIX

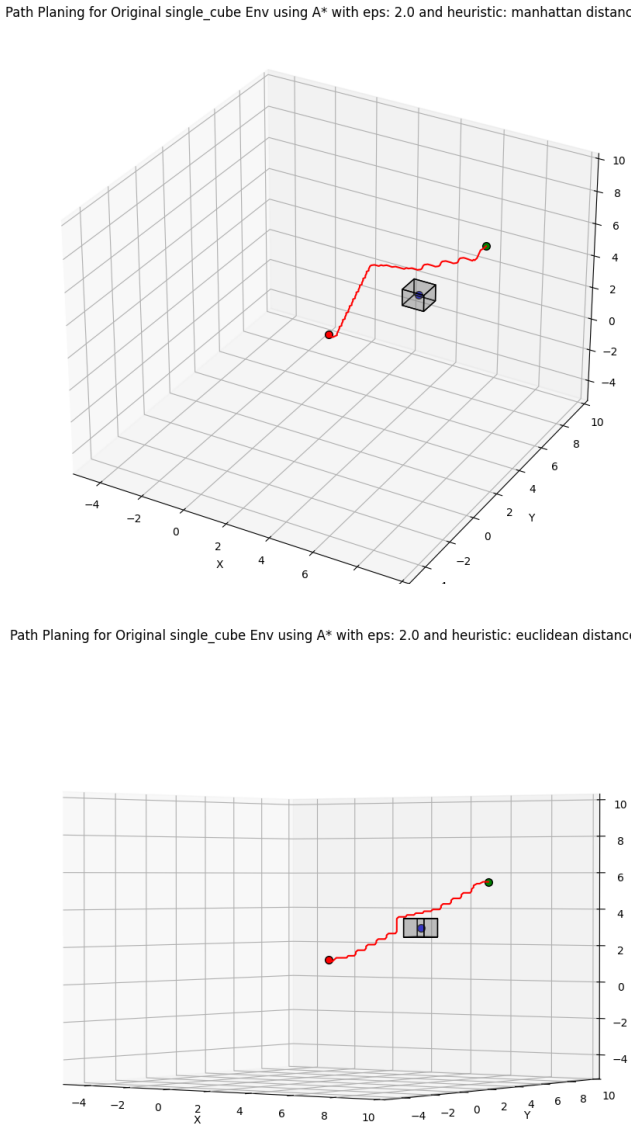


Fig. 6. *Path Planing Weighted A* ϵ 2 in Single Cube Env*

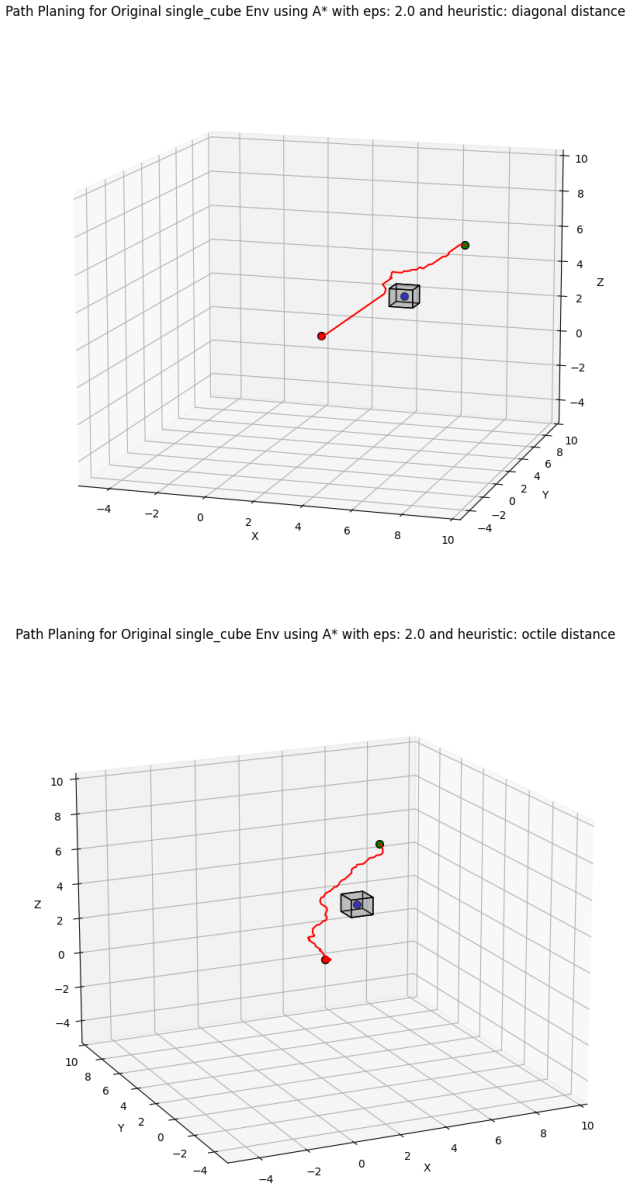
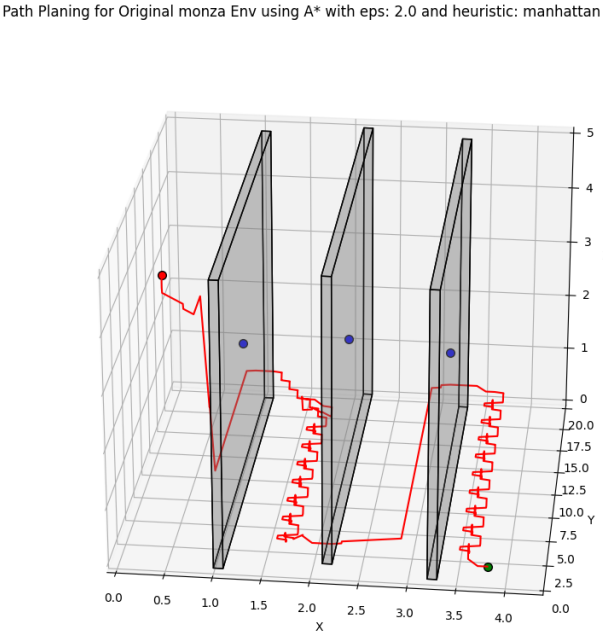


Fig. 7. *Path Planing Weighted A* ϵ 2 in Single Cube Env*



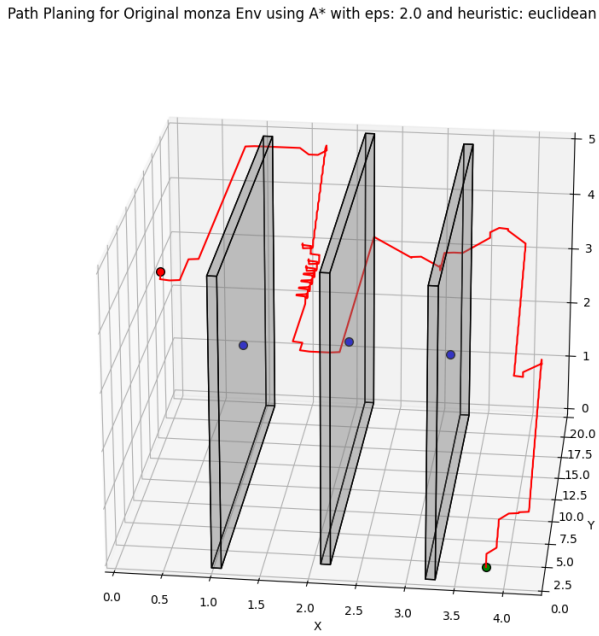
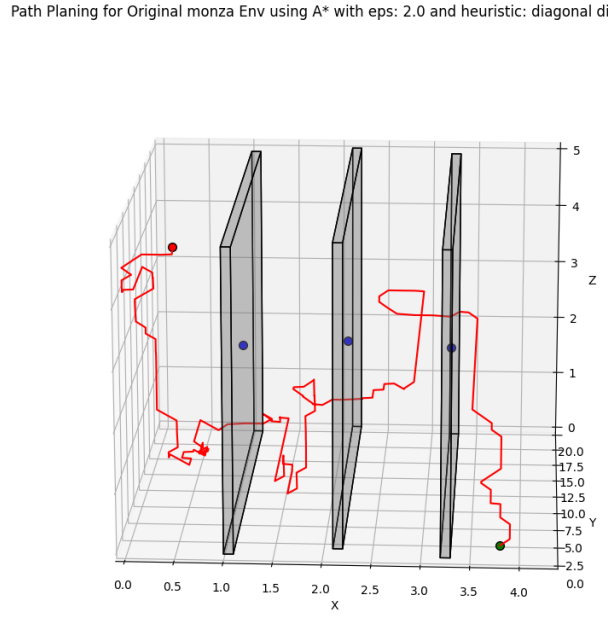


Fig. 8. *Path Planing Weighted A* ϵ 2 in Monza Env*



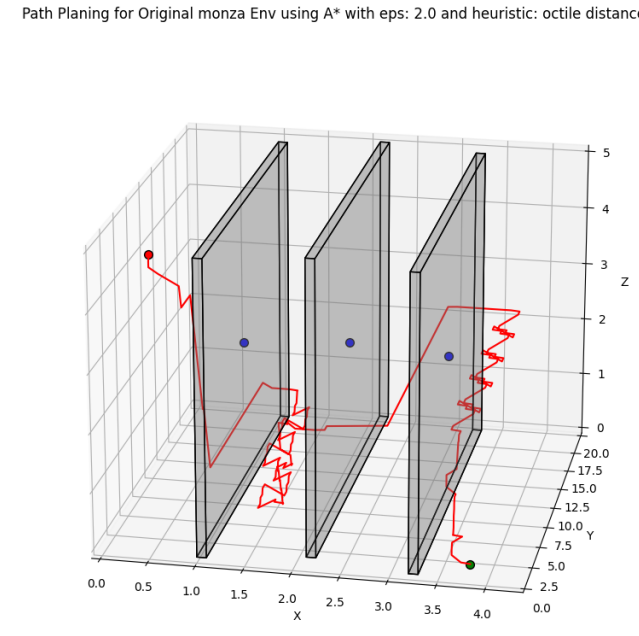


Fig. 9. *Path Planing Weighted A* ϵ 2 in Monza Env*

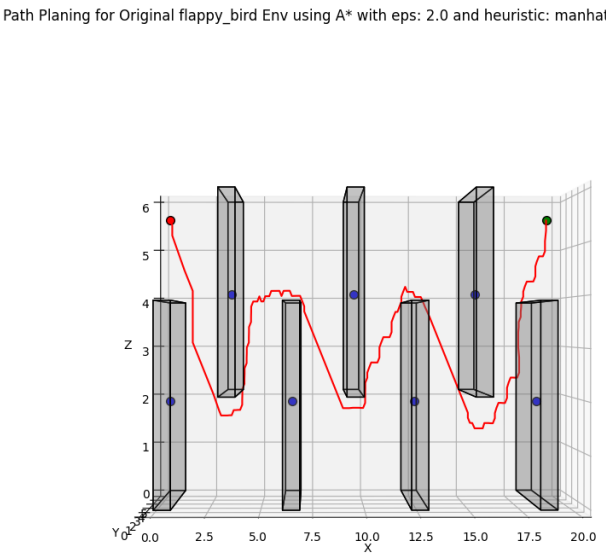
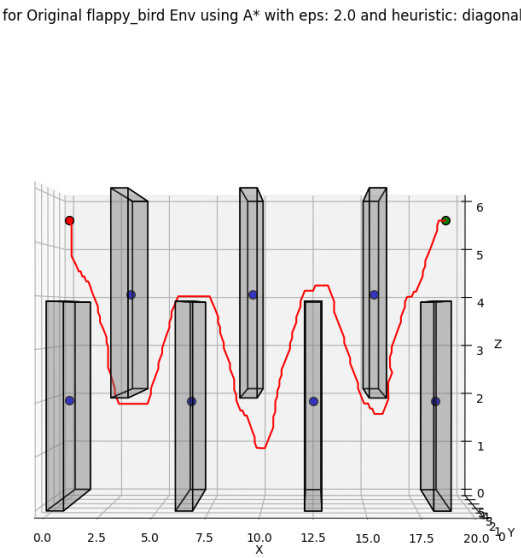


Fig. 10. *Path Planing Weighted A* ϵ 2 in Flappy Bird Env*



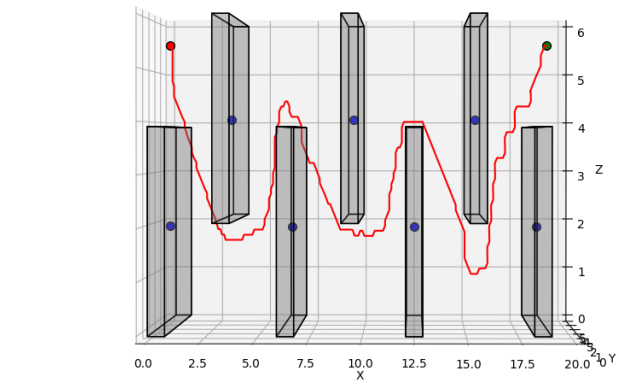
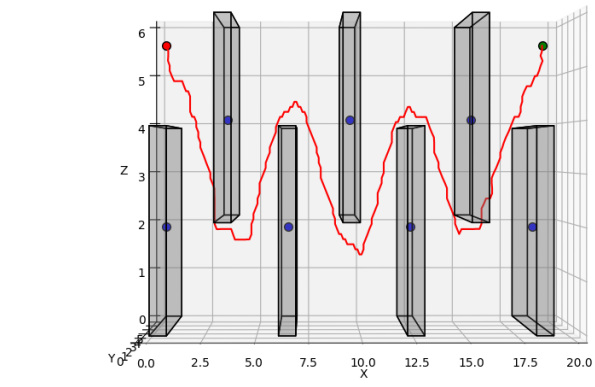
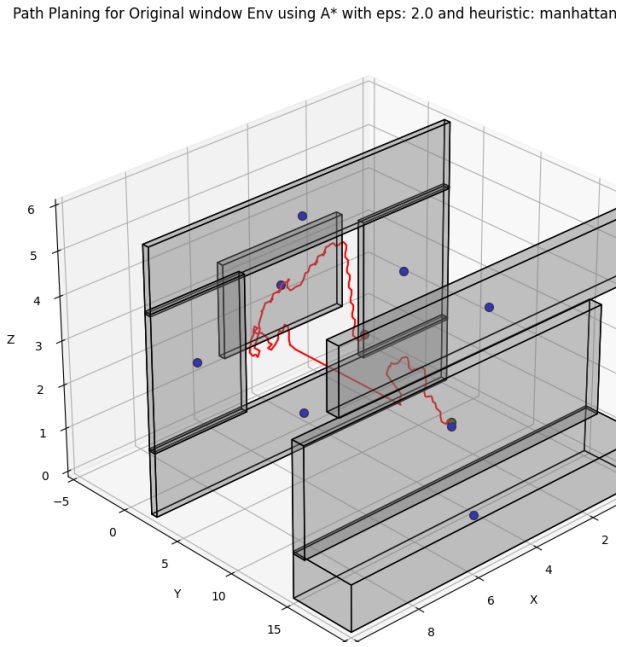
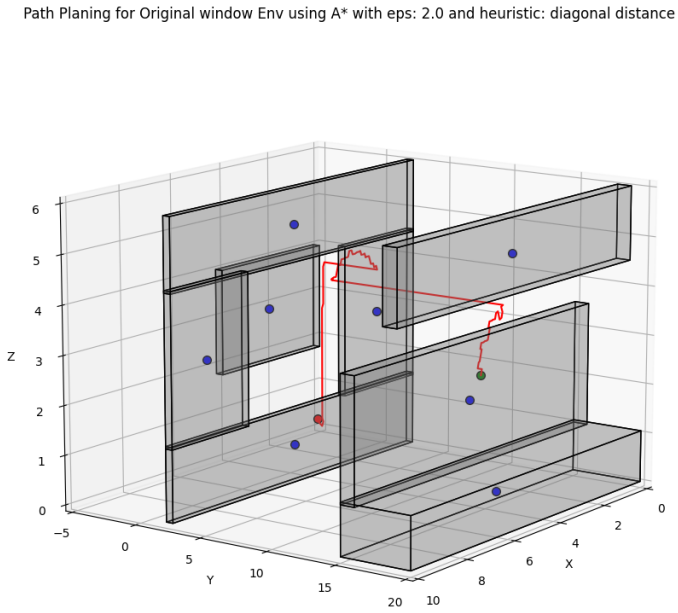
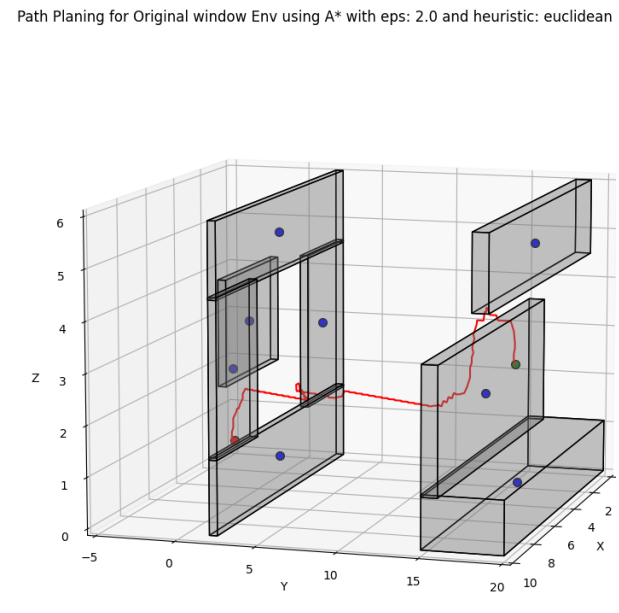


Fig. 11. *Path Planing Weighted A* ϵ 2 in Flappy Bird Env*









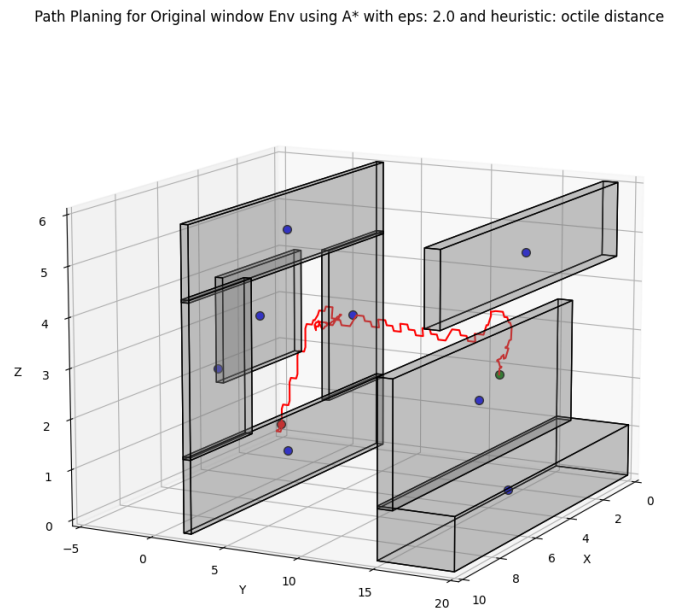
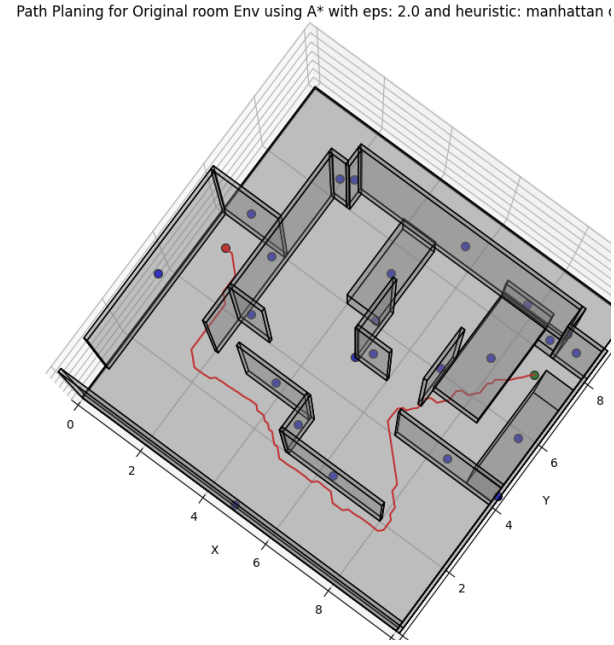
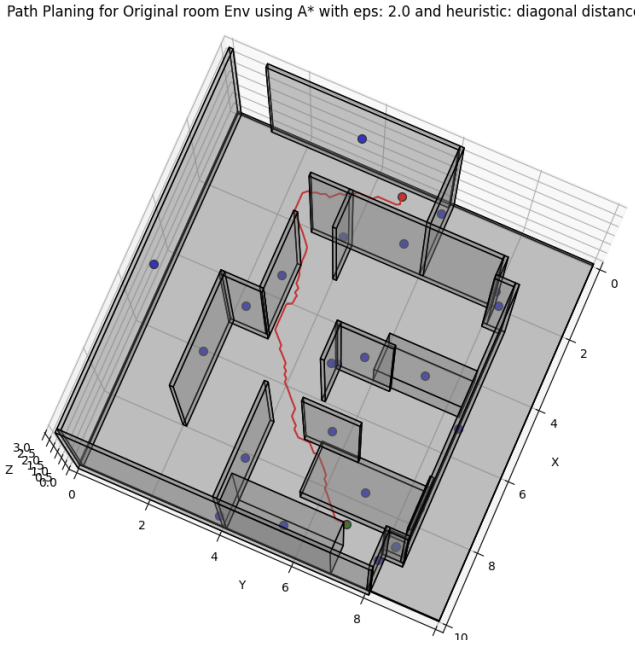
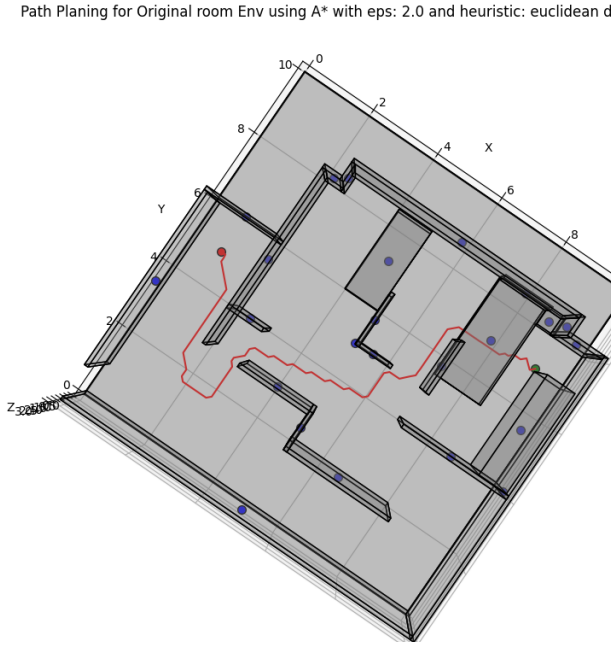


Fig. 12. *Path Planing Weighted A* ϵ 2 in Window Env*

Fig. 13. *Path Planing Weighted A* ϵ 2 in Window Env*







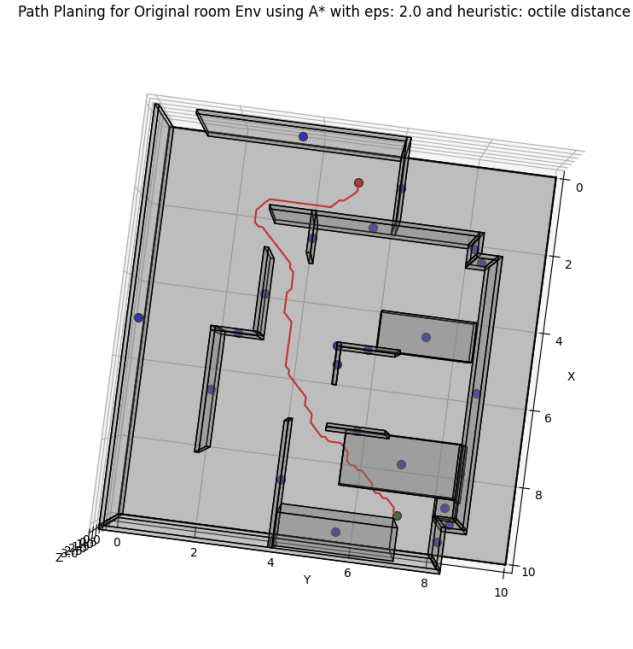
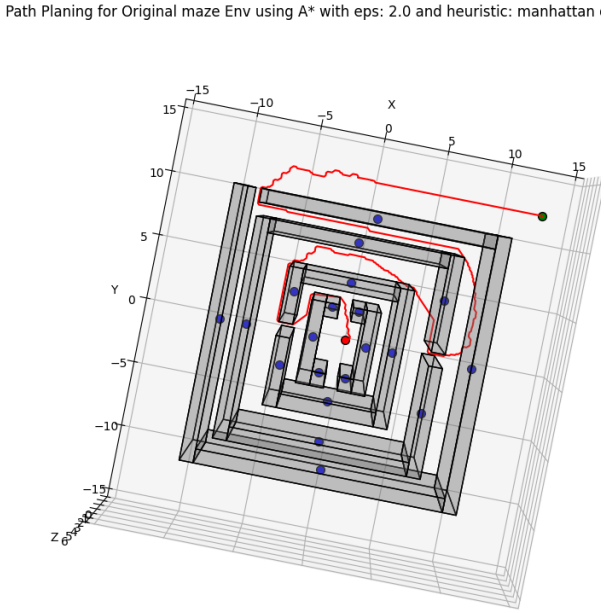


Fig. 14. *Path Planing Weighted A* ϵ 2 in Room Env*

Fig. 15. *Path Planing Weighted A* ϵ 2 in Room Env*



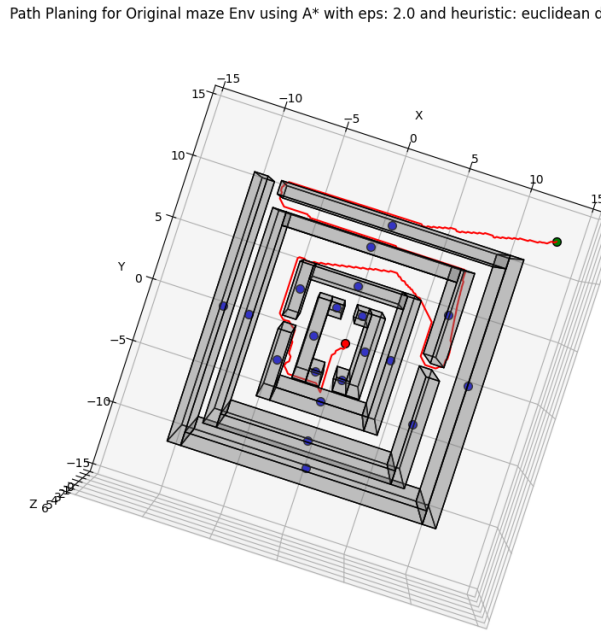
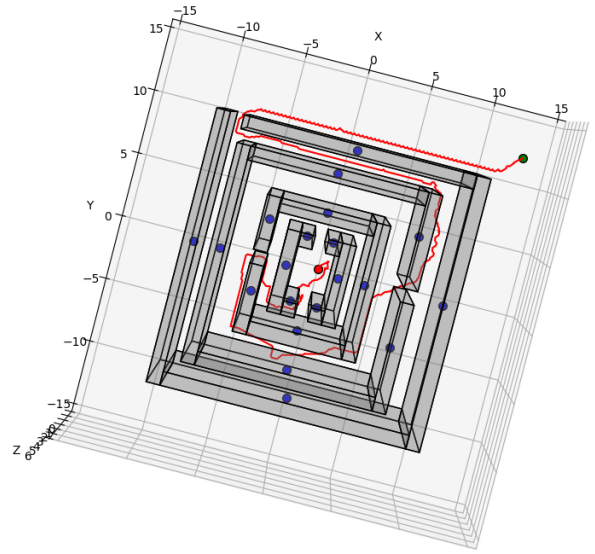


Fig. 16. *Path Planing Weighted A* ϵ 2 in Maze Env*



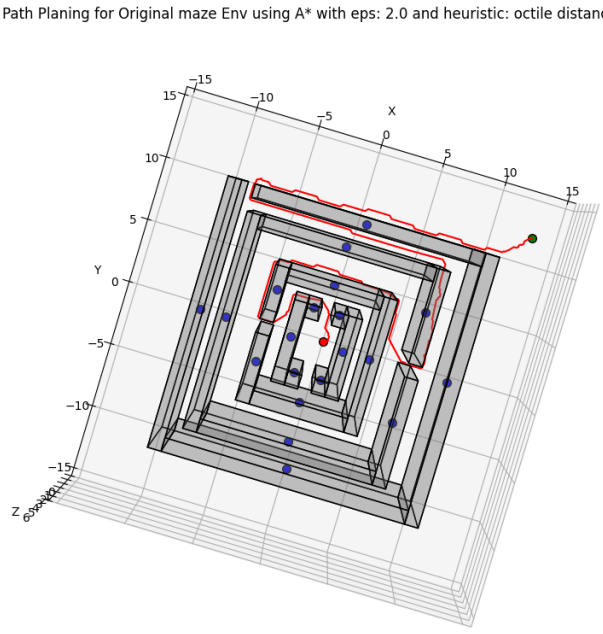


Fig. 17. *Path Planing Weighted A* ϵ 2 in Maze Env*

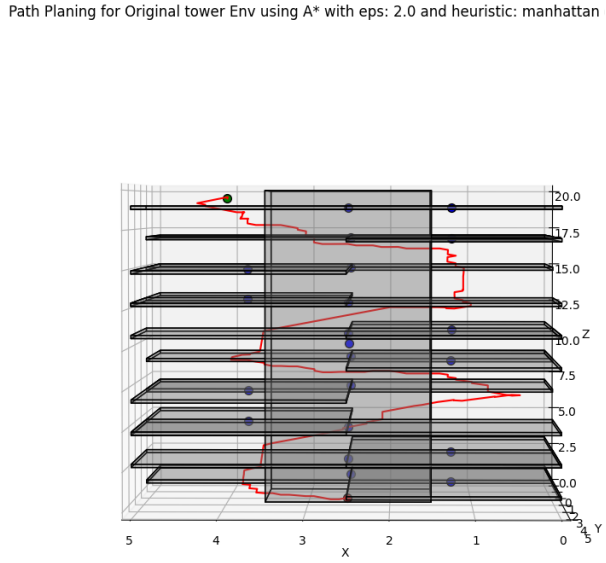


Fig. 18. *Path Planing Weighted A* ϵ 2 in Tower Env*

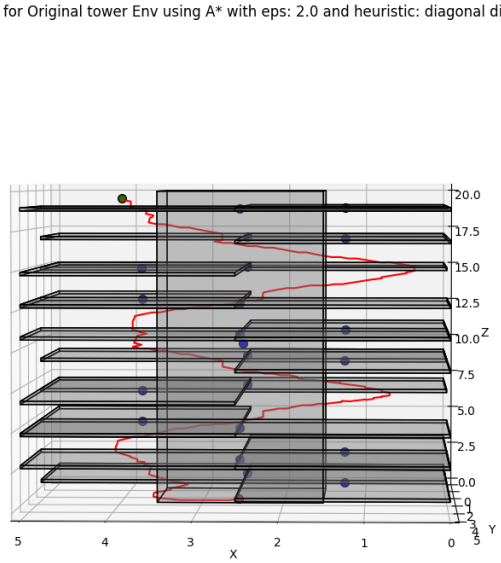
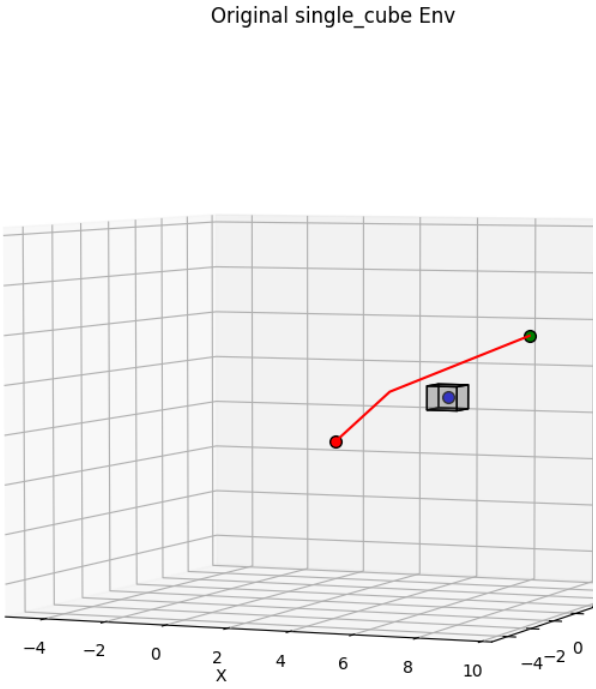


Fig. 19. *Path Planing Weighted A* ϵ 2 in Tower Env*



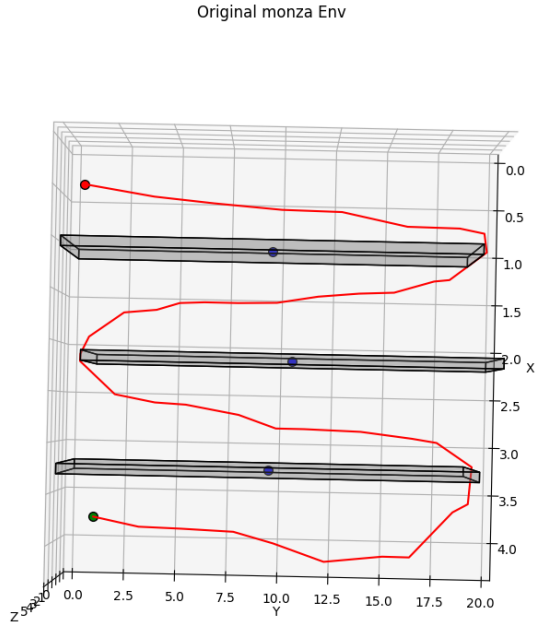
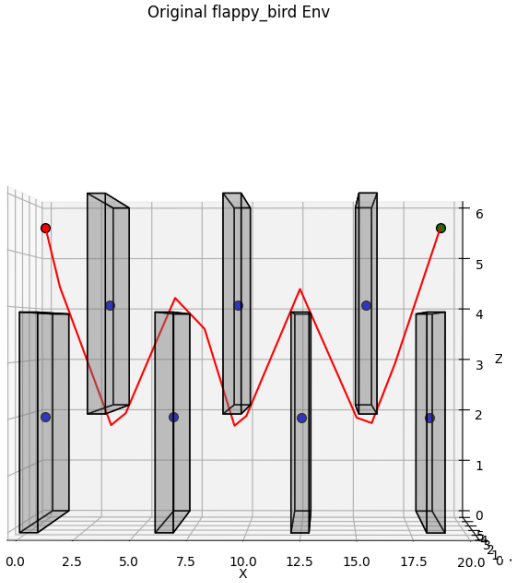


Fig. 20. *Path Planing RRT* in Single Cube Env*



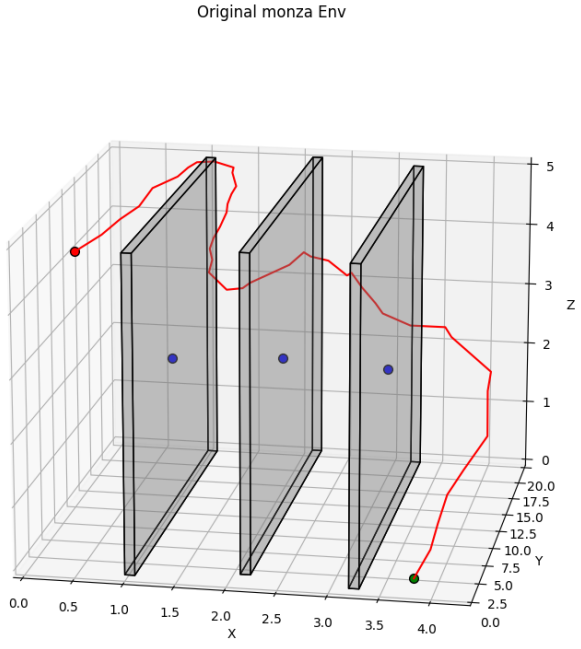


Fig. 21. *Path Planing RRT* in Flappy Bird Env*

Fig. 22. *Path Planing RRT* in Monza Env*

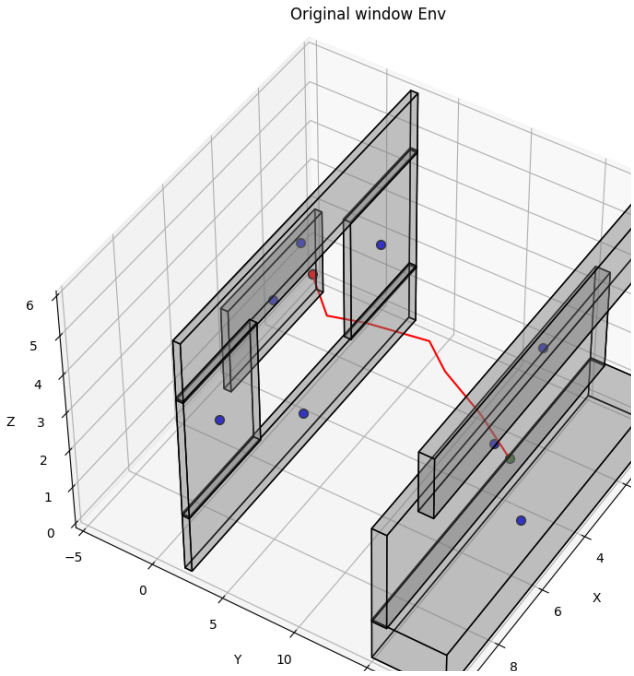
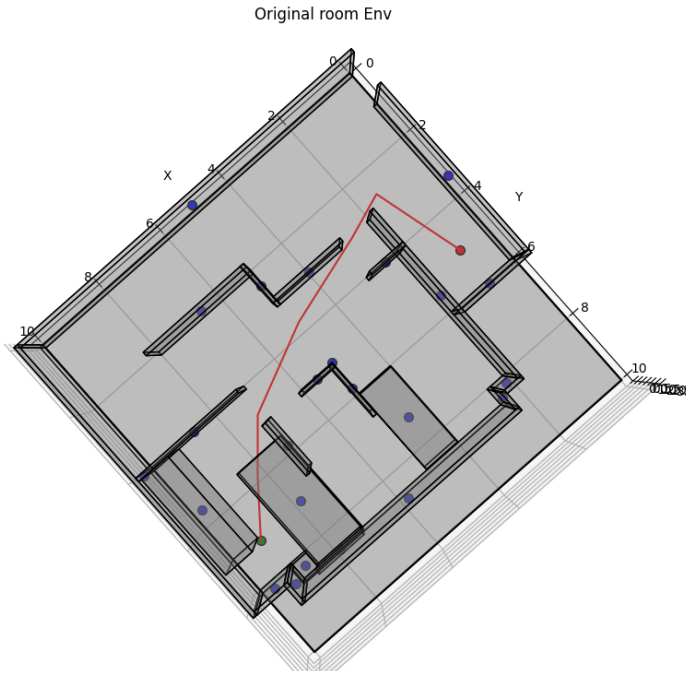


Fig. 23. *Path Planing RRT* in Window Env*



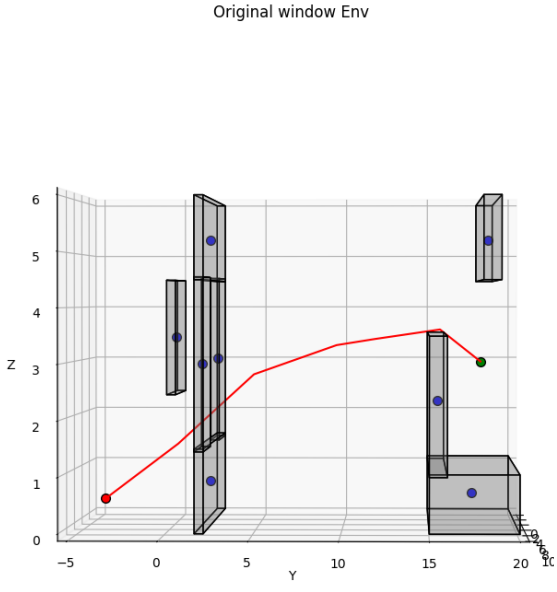
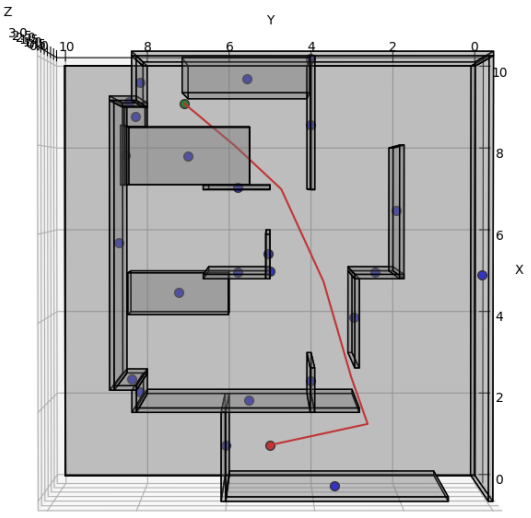
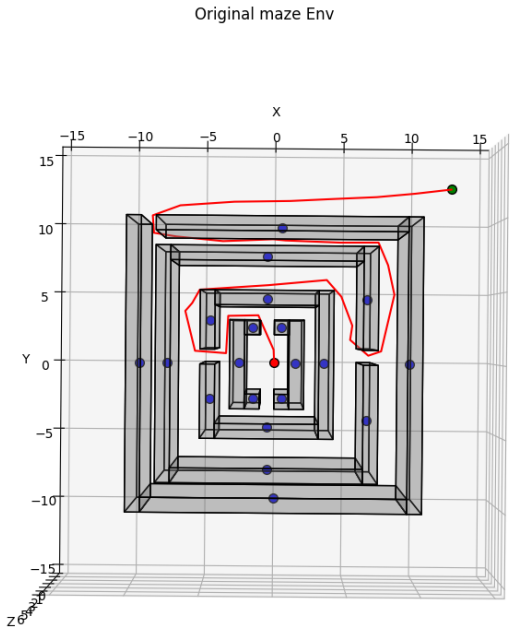
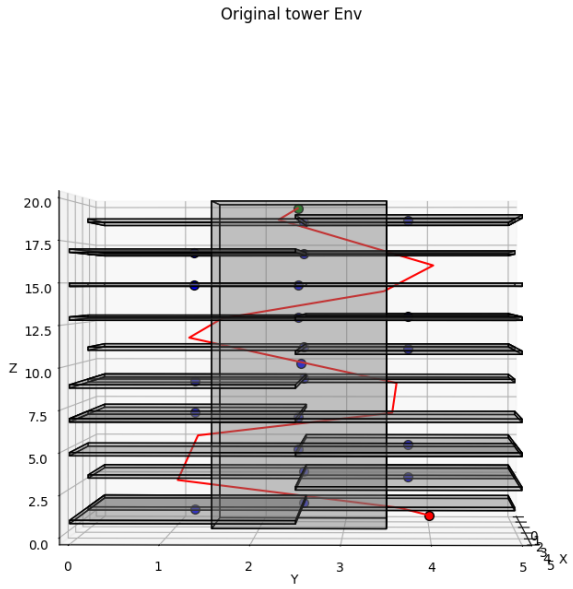
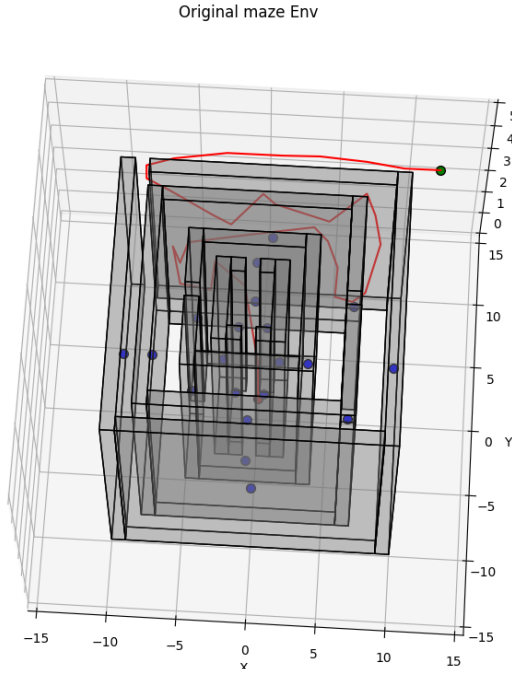


Fig. 24. *Path Planing RRT* in Room Env*









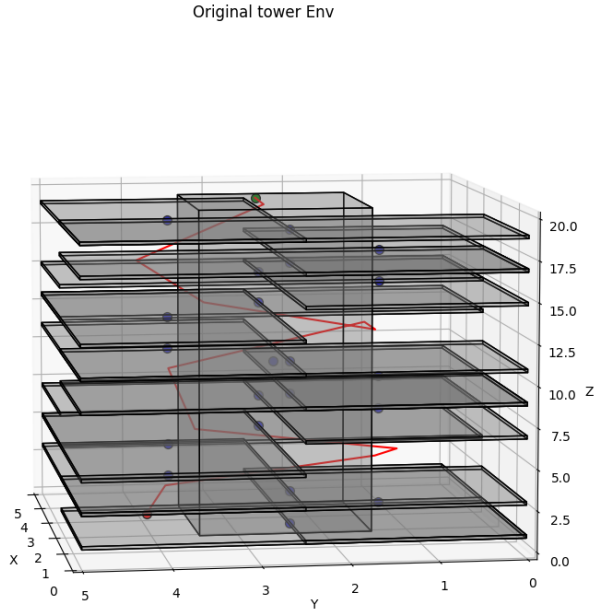


Fig. 25. *Path Planing RRT* in Maze Env*

Fig. 26. *Path Planing RRT* in Tower Env*