Search-based and Sampling-based Motion Planning in 3D Euclidean Space

Zhuoqun Chen
Dept. Electrical and Computer Engineering
UC San Diego
La Jolla, USA
zhc057@ucsd.edu

***Abstract*—This report focus on comparing the performance of search-based and sampling-based motion planning algorithms in 3D Euclidean Space with obstacles. By formulating the Motion Planning Problem and its sub-problems that is suitable for our environment settings, we solved it with search-based Weighted A* and sampling-based RRT Algorithm. To show the effectiveness of the algorithms, we tested it 7 environments with different layouts. We presented the experiment results and conclude the report in the end.**

***Index Terms*—Motion Planning, Search-based Path Planning, Sampling-based Path Planning, Weighted A*, Rapidly-exploring Random Trees(RRT)**

I. Introduction

Motion planning is a fundamental problem in robotics and autonomous systems, which involves determining a feasible path for a robot or agent to navigate from a start to a goal configuration while avoiding obstacles in its environment. This problem is crucial in various scenarios, such as autonomous vehicles, industrial automation, and mobile robotics. To tackle this challenge, search-based path planning algorithms and sampling-based path planning algorithms have emerged as powerful techniques. Search-based algorithms, such as A* and Dijkstra's algorithm, explore a discrete search space to find an optimal or near-optimal path. On the other hand, sampling-based algorithms, like Rapidly-exploring Random Trees (RRT) and Probabilistic Roadmaps (PRM), construct a roadmap or a tree of feasible configurations by randomly sampling the configuration space. These algorithms enable efficient and robust path planning in different scenarios, offering versatile solutions to the motion planning problem.

In this report, we mainly focus on comparing the performance of search-based and sampling-based motion planning algorithms in 3D Euclidean Space with obstacles. The organization of this report is as following: We formulate the motion planning problem in section II, then we give our performance analysis of different algorithms in section III, then we did some experiments to show the effectiveness our implemented algorithms in section IV. At the end of the report, we also appended some technical solutions and experiment debugging logs that helped us to improve the robustness of our self-implemented path planning algorithms.

II. Problem Formulation

Generally speaking, given a robot with its dynamics, a configuration space C built on top of an environment with obstacles, an initial state of the robot $\mathbf{x}_s$, a goal state $\mathbf{x}_\tau$, the **Motion Planning Problem** in $(C_{free}, \mathbf{x}_s, \mathbf{x}_\tau)$ is to find a sequence of valid configurations that robot can move from $\mathbf{x}_s$ to $\mathbf{x}_\tau$. In our settings, the robot is treated as a point in $\mathbb{R}^3$ without Kinodynamic constrains that moves inside a regular Axis-Aligned Bounding Box(AABB) with also AABB obstacles, thus our problem can be natually converted to a **Path Planning Problem** in $\mathbb{R}^3$. In many cases, the dimension of the problem is very high so it's not always possible to develop a *complete* algorithm that terminates in finite time[1]. Thus for a Path Planning Problem, for different type

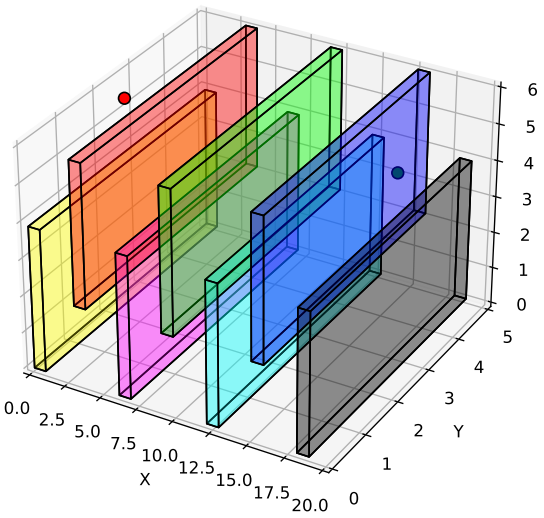

Fig. 1. Flappy Bird Environment Example: The goal is to plan a 3D collision-free path from the *start*(red) to the *goal*(green). The boundary box is the overall limited configuration space in $\mathbb{R}^3$. The colorful blocks in the middle are the obstacles and any line segment along the planned path cannot have intersection within those space. The complementary free space is the area that is not covered by obstacles within the boundaries.

of algorithms, the original problem can be divided into two sub-problems:

A. Feasible Path Planning Problem

We formulate a **Feasible Path Planning Problem** as the following: given a path planning problem $(C_{free}, \mathbf{x}_s, \mathbf{x}_\tau)$, find a feasible path $\rho : [0,1] \rightarrow C_{\text{free}}$ such that $\rho(0) = \mathbf{x}_s$ and $\rho(1) = \mathbf{x}_\tau$ if one exists. If there is no such path, report failure.

Many Sampling-based planning algorithms can be used to solve this sub-problem and provide some theoretical guarantees and we will talk about more details in section III-F.

B. Optimal Path Planning Problem

If we associate a cost function $J : \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$ with a feasible path ρ^*, then we can formulate a **Optimal Path Planning Problem**: given a path planning problem $(C_{free}, \mathbf{x}_s, \mathbf{x}_\tau)$ and a cost function $J : \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$, find a feasible path ρ^* such that:

$$J(\rho^*) = \min_{\rho \in \mathcal{P}_{s,\tau}} J(\rho) \tag{1}$$

Sampling-based algorithms are not suitable for directly finding such path in eq. (1) because the randomness of the steer distance when sampling new nodes $\mathbf{x}_{rand}$. However, many Search-based planning algorithms can be used to solve such problem because the planning usually operates on *undirected uniform-cost grid*. We will talk about more details from the algorithm side including how we design the control space $\mathcal{U}$ for 3D scenario in section III-E.

In later sections, we focus more on the details and algorithms developed to solve the above feasible path planning problem and optimal path planning problem.

III. Technical Approach

A. Framework Design

We designed our software pipeline as in Figure 2. Search-based planners and Sampling-based planners are at the same level of abstraction and implment the *plan* or *search path* API.

B. Collision Detection Design

Implementing a 3D *collision checker* is critical for finding a shortest path in 3D Euclidean space with obstacles. Without collision detection during the planning in each step, we won't be able to tell which children nodes shouldn't be expanded and may eventually end up with an unusable planned path in real-world applications. In our problem, the obstacles are a set of rectangular blocks specified by lower-left corner $(x_{\min}, y_{\min}, z_{\min})$ and upper-right corner $(x_{\max}, y_{\max}, z_{\max})$.

1) 3D Line-AABB Collision Detection: There are many possible configurations for a directed line segment $\overrightarrow{v_0 v_1}$ specified by start point v_0 and end point v_1 and an *Axis-Aligned Bounding Box(AABB)* denoted as B in $\mathbb{R}^3$ can be placed, an example can be found in Figure 3.

And from Figure 3, we can deduct some common rules to check whether a line $\overrightarrow{v_0 v_1}$ and an AABB B will collide:

- $\overrightarrow{v_0 v_1}$ collides with B if v_0 or v_1 is within B

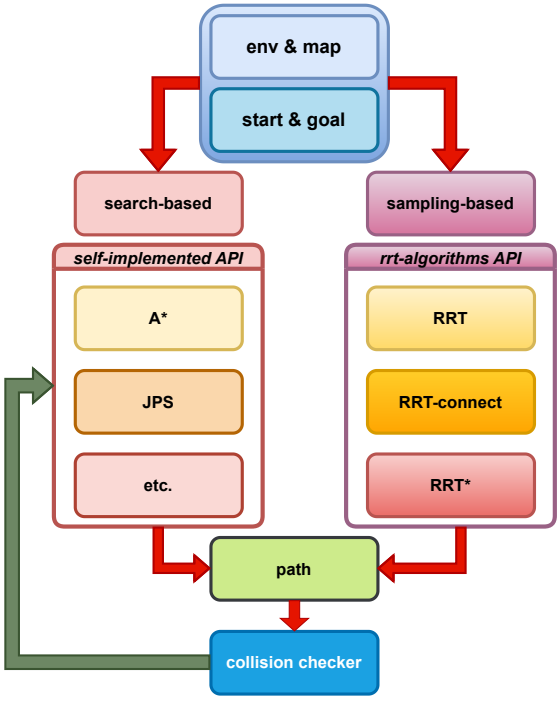

Fig. 2. An illustration of our pipeline: given the environment read from the map and the positions of the start and the goal point, we can specify to use search-based algorithms or sampling-based algorithms to plan a 3D collision-free path, for sampling-based algorithms, we use the APIs provided by [2]. *Collision Checker* will be used in search-based algorithms to ensure only feasible children nodes and paths given by any algorithm will be double-checked whether it's collsision-free. More details can be found in Section III-B.

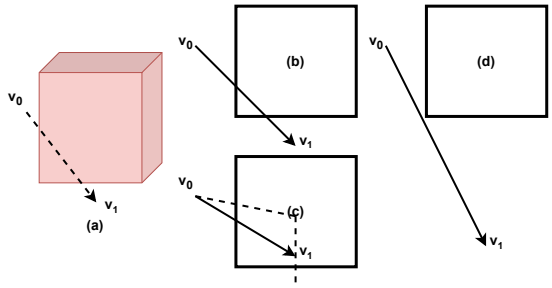

Fig. 3. Different scenarios of a directed line segment and an AABB in Euclidean Space: (a) the viewpoint(x-axis pointing right, y-axis pointing inside, z-axis up) of seeing a layout of a directed line segment and an AABB, and when projected onto the x-y 2D plane, there are 3 possible cases: (b) line $\overrightarrow{v_0v_1}$ intersects the bottom plane on a point outside the box (c) line $\overrightarrow{v_0v_1}$ intersects the left plane on a point outside the box

- $\overrightarrow{v_0v_1}$ collides with B if the intersection between $\overrightarrow{v_0v_1}$ and any plane $\mathbf{E}^2$ of B is within B

The above rules assume that any plane extends to infinity and is represented as *Hessian Normal Form*[3]:

$$\mathbf{n} \cdot \mathbf{x} = -p \tag{2}$$

where $\mathbf{n}$ is the face normal vector of the plane and p is the distance of the origin to the plane(p is positive if origin is located in the same side of $\mathbf{n}$ and the opposite side otherwise). Thus the point-plane distance can be easily calculated as:

$$D(\mathbf{x}_0, \mathbf{E}^2) = \mathbf{n} \cdot \mathbf{x}_0 + p \tag{3}$$

Positive distance D indicates that the queried point $\mathbf{x}_0$ lives in the same side of $\mathbf{n}$ and the opposite side otherwise.

Equation (3) can be used to determine whether a line will collide with a plane (Algorithm 1):

Algorithm 1 3D Line-Plane Intersection Detection

Input: Line $\overrightarrow{v_0v_1}$ (vector $\mathbf{v}$)
Input: Plane $\mathbf{E}^2$ with $\mathbf{n}$ and p
Output: intersection indicator and intersection point if there is

denominator = $\mathbf{n} \cdot \mathbf{v}$
if denominator $= 0$ **then**
  if $D(\mathbf{x}_0, \mathbf{E}^2) = 0$ **then**
    return True, v_0
  end if
  return False, None
end if
$t = -(\mathbf{n} \cdot v_0 + p)/\text{denominator}$
if $t < 0$ or $t > 1$ **then**
  return False, None
end if
intersection point $= v_0 + t \cdot \mathbf{v}$
return True, intersection point

We present our 3D Line-Plane Collision Detection algorithm as Algorithm 2:

Notice that we also need to check whether a line segment will collide with an additional AABB consists of the rectangular boundaries.

Algorithm 2 3D Line-AABB Collision Detection

Input: Line $\overrightarrow{v_0v_1}$ (vector $\mathbf{v}$)
Input: AABB B with 6 planes: left, front, bottom, right, back, top
Output: collision indicator and collision point if there is

if $v_0 \in B$ **then**
  return True, v_0
else if $v_1 \in B$ **then**
  return True, v_1
end if
for each plane $\mathbf{E}^2 \in B$ **do**
  intersects, point = $Algorithm\ 1\ (\overrightarrow{v_0v_1}, \mathbf{E}^2)$
  if intersects = True **then**
    if point $\in$ B **then**
      return True, point
    end if
  end if
end for
return False, None

Algorithm 3 3D Path Collision Free Detection

Input: a path p in $\mathbb{R}^3$
Input: boundary box V and obstacle AABBs $\{B_1, B_2, \cdots, B_m\}$
Output: collision-free indicator

for each line $\overrightarrow{v_0v_1}$ in p **do**
  for each $B_i \in \{B_1, B_2, \cdots, B_m\}$ **do**
    if $v_0 \notin V$ or $v_1 \notin V$ **then**
      return False
    end if
    intersects, point = $Algorithm\ 2\ (\overrightarrow{v_0v_1}, B_i)$
    if intersects = True **then**
      return False
    end if
  end for
end for
return True

2) 3D Path Collision Detection: With Algorithm 2, we can then give our 3D Path Collision Detection algorithm (see Algorithm 3), this algorithm will be executed everytime the planner has given a path to double check(especially in sampling-based algorithms because the validness of newly added edges is checked inside the library itself) that every line segment along the path is collision free.

The Path Collision Detection Algorithm will check the paths returned by both search-based and sampling-based planners.

C. Pruning Successors

D. Additional Techniques to Improve Robustness of Path Finding Algorithms

Notice that this section is better be placed at Appendix section as an experiment log of debugging.

1) Early Termination When Open Set Is Empty:

2) Line and AABB Box Collision Checker Filter out Unreachable Children:

E. Search-based Planning Algorithms

Search-based planning algorithms are suitable for solving Optimal Path Planning Problem(eq. (1)) when we discretize the continuous 3D Euclidean Space as a collection of 3D grids with the same grid size specified by *resolution*. Then we construct a graph representation of the 3D grids and calculate the costs between two nodes i and j and convert the problem into a *Deterministic Shortest Path Problem(DSP)* and use label correction algorithms to solve it(no negative cycles). In our settings, we select resolution $= 0.5$ to discretize the configuration space C and use L2-norm to represent the true cost between two nodes. In the following discussion, we mainly talk about the implementation and theoretical analysis of *Weighted* A^* algorithm.

1) Control Input for 3D Grids: We can see from Figure 4 that for each descretized state $\mathbf{x}$ during planning, there are 26 control inputs in total can be selected to go to the next potential children nodes and each one can be encoded into a discretized steering direction dR:

$$dR = \{i, j, k\} \tag{4}$$

where $i, j, k \in \{-1, 0, 1\}$ and $|i| + |j| + |k| \neq 0$ because we don't allow the next state to be the the same as current state. Thus the next state $\mathbf{x}'$ determined by the *transition function* $f(\mathbf{x}, \mathbf{u})$ can

be written as:

$$\mathbf{x}' = f(\mathbf{x}, \mathbf{u}) = \mathbf{x} + dR \cdot resolution \tag{5}$$

If we interpolate the notation and view current state as current note i on graph G, next state $\mathbf{x}'$ as its child node j, then the stage cost(edge cost) is:

$$c_{ij} = l(\mathbf{x}, f(\mathbf{x}, \mathbf{u})) = \|dR\|_2 \cdot \text{resolution} \tag{6}$$

And $\|dR\|_2$ can be one of the value in $\{D_1, D_2, D_3\}$ where $D_1 = 1$, $D_2 = \sqrt{2}$ and $D_3 = \sqrt{3}$. Notice that the cost defined above indicates *undirected uniform-cost grid*, where the cost $l(\mathbf{x}, \mathbf{u})$ is only determined by the control input no matter what the state $\mathbf{x}$ is. This assumption will also allow us to develop *Jump Point Search* Algorithm to solve the problem.

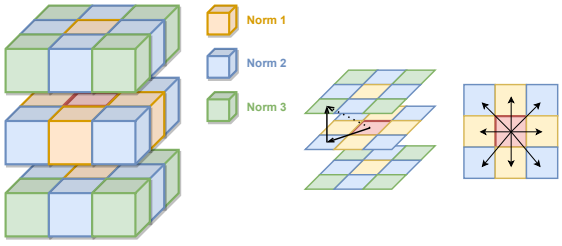

Fig. 4. An overview of 26 possible actions for a descretized state $\mathbf{x}$ during planning: state $\mathbf{x}$ correspondes to the red grid in the center of the middle layer, orange grid indicates Norm-1 action with cost D_1(6 in total), blue grid indicates Norm-2 action with cost D_2(12 in total), green grid indicates Norm-3 action with cost D_3(8 in total)(for simplicity, we can assume resolution $= 1$)

2) Weighted A^* Algorithm: The Weighted A^* Algorithm can be seen in Algorithm 4. And in practice, we implement the OPEN list as an *priority queue* where the key is stored as the discretized 3D grid index converted from the coordinate of the node and the value is $f_i = g_i + \epsilon h_i$. During each iteration, the node i with the minimal f value will be poped from OPEN list and marked as CLOSED. Then we try to correct the labels of valid children nodes of i. Notice that we call *valid children nodes* if they have been filtered by algorithm 1 where the input line is $\overrightarrow{v_i v_j}$ and we iterate over all the obstacle AABBs $\{B_1, B_2, \cdots, B_m\}$ specified by the environment.

Algorithm 4 Weighted A^* Planning

OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}, \epsilon \geq 1$
$g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
while $\tau \notin$ CLOSED **do**
  Pop i with min $f_i := g_i + \epsilon h_i$ from OPEN
  Insert i into CLOSED
  for $j \in$ Children(i) and $j \notin$ CLOSED **do**
    if $g_j > (g_i + c_{ij})$ **then**
      $g_j \leftarrow (g_i + c_{ij})$
      Parent$(j) \leftarrow i$
      if $j \in$ OPEN **then**
        Update priority of j
      else
        OPEN $\leftarrow$ OPEN $\cup\{\cup\}$
      end if
    end if
  end for
end while
return path if there is or None if time out

3) The choice of Heuristic Functions: A heuristic function is *consistent* if it satisfies the triangle inequality:

$$h_i \leq c_{ij} + h_j \ for\ all\ i \neq \tau\ and\ j \in Children(i) \tag{7}$$

and also $h_\tau = 0$.

For our search-based planning on a 3D grid environment settings, both *3D Euclidean distance* and *3D Diagonal distance* are *consistent* heuristic functions when $\epsilon = 1$. And we tried both in our experiments, more details are discussed in section IV.

$$\mathbf{3DEuclideandistance}: h_i = \|\mathbf{x}_\tau - \mathbf{x}_i\|_2 \tag{8}$$

4) Optimality and Completeness Analysis of Weighted A^*: Compared to A^* Algorithm, because *Weighted* A^* uses an $\epsilon - consistent$ heuristic, thus only an $\epsilon - optimal$ path with cost:

$$\mathbf{dist}(s, \tau) \leq g_\tau \leq \epsilon \cdot \mathbf{dist}(s, \tau) \tag{9}$$

will be returned in the end. However, the *completeness* of the algorithm is guaranteed(finite time convergence assuming a finite graph). Thus ϵ is a trade-off factor between optimality and searching speed.

5) Time and Memory Complexity Analysis: We analyze the complexity of the A^* Algorithm assuming that we use *Binary heap*, e.g., *boost::heap::d_ary_heap* in *C++*. Given a graph G with number of nodes $|\mathcal{V}|$, number of edges $|\epsilon|$, the time complexity is:

$$O((|\mathcal{E}| + |\mathcal{V}|) \log |\mathcal{V}|) \tag{10}$$

As for the memory complexity, A^* does minimum number of expansions $O(|\mathcal{V}|)$, however, this require an infeasible amount of memory, and Weighted A^* are often but not always better[4].

6) Scale Up When Extending to Larger Map: The efficiency of Weighted A^* Algorithm is largely affected by the selection of ϵ in different layouts of the environments because this value determines how much the planner is biased towards states closor to the goal by calculating

$$f_i = g_i + \epsilon h_i \text{ with } \epsilon \geq 1 \tag{11}$$

For our current python implementation, the grid-size is fixed, if there are large and dense obstacles between the start point and the goal point, in such environments like *monza* or *flappy_bird*, the planning time is a little bit longer. One possible solution is to use grid with different levels of sparsity. For wide and collision-free area, we use larger grid-size and when the obstacles are dense in the area, we shrink the grid-size. Another solution is to implement *C++* version of the algorithm and use better underlying data structure to implement the *priority queue*.

When $\epsilon = 1$, the Weighted A^* algorithm will be reduced to regular A^* algorithm.

In our experiment, when $\epsilon = 1.0$, the total expanded nodes are 111 for *single cube* environment, however, when we increase ϵ to 1.1, the total expanded nodes are reduced to 11 for the same environment using the same heuristic function.

F. Sampling-based Planning Algorithms

Basically in our settings, for sampling-based planning in 3D, the cost of the path can also be calculated by summing over the length of line segments along the path. But unlike maintaining an *open list*, the tree is expanded randomly in C_{free}. Thus we have no way to evaluate the optimality of a found feasible path at termination, this is why we develop sampling-based algorithms mainly to solve the Feasible Path Planning Problem described in section II-A.

1) Rapidly Exploring Tree(RRT) Algorithm: Rapidly Exploring Tree(RRT) is one of the most popular planning techniques and has many versions of variants when adding kinodynamic constrains. We mainly focus on exploring the original algorithm to get a better understanding of the basic concepts of sampling-based path planning. We present the original RRT algorithm in Algorithm 5.

Algorithm 5 3D RRT

for $i = 1 \ldots n$ **do**
  $\mathbf{x}_{\text{rand}} \leftarrow$ SAMPLEFREE()
  $\mathbf{x}_{\text{nearest}} \leftarrow \text{NEAREST}\left((V, E), \mathbf{x}_{\text{rand}}\right)$
  $\mathbf{x}_{\text{new}} \leftarrow \text{STEER}_{\epsilon}\left(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{rand}}\right)$
  if CollisionFREE $\left(\mathbf{x}_{\text{nearest}}, \mathbf{x}_{\text{new}}\right)$ **then**
    $V \leftarrow V \cup \{\mathrm{x}_{\text{new}}\}$
    $E \leftarrow E \cup \{(\mathrm{x}_{\text{nearest}}, \mathrm{x}_{\text{new}})\}$
  end if
end for
return G=(V, E)

2) Tunable Parameters of RRT: There are 4 tunable parameters when using *RRT* algorithm in *rrt-algorithms* library: length of tree edges Q, maximum number of samples before termination *max_samples*, length of smallest edge to check for intersection with obstacles r and probability of checking for a connection to goal *prc*. In our later on experiment results, we found that the Q value is very important and a key factor when there are many obstacles between the start and the goal point. For the algorithm can always return a feasible path, we maully fix the max samples to a very large number to ensure probabilistically completeness. Then for the parameter r, it will steer a small distance towards the x_{rand} from $x_{nearest}$ if there is an obstacle within them, if the collision is still detected, nothing will be connected to the tree and we will continue to generate a new sample in the configuration space.

3) Addition Effort to Use the Third-Party API for Our Settings: We used a third-party of python implementation of RRT-based Algorithms called *rrt-examples*[2]. And we used the original RRTBase Algorithm in the package and tuned its parameters specified above to find feasible path in our 3D environment settings. We converted the start point and goal point to tuple, converted the boundaries to a 2-D numpy array, only use the first 6 columns of the blocks(the other 3 dimensions are used to specify block colors). Later on when plotting, we also add a second 3D axis to plot the tree itself and add some additional labels to help get a better visualization of the metrics that can show the effectiveness of the algorithm. We also colorized the blocks for better effect.

G. Comparison between Search-based planners and Sampling-based planners

From previous discussion, search-based algorithms will provides finite-time sub-optimality bounds on the solution and guarantes to find a feasible path if it exists in the 3D grid environment settings thus they are resolution complete. However, the problem is that when the problem has very high dimensions, the memory and time consumption can be computationally expensive. On the other hand, however, searching-based algorithms try to construct a graph or a tree representation of the environment by randomly sampling in C_{free}. Therefore they are faster and usually requires less memory than search-based planning in high dimensions. Even though the rigorous *completeness* can't be guaranteed, when the number of iterations go to ∞, the probability of finding a feasible path if it exists will approaches 1, which means that sampling-based algorithms are *probabilistically complete*. And accordingly, the *asymptotic sub-optimality* bounds on the solution is guaranteed. In section IV, we give some performance comparison of both classes of algorithms by showing our experiment results.

IV. Experiment Results

In section III, we discussed about Weighted A^* algorithm and RRT algorithm. In this section, we visualize the path returned by above algorithms in 7 different environments and we give some analysis on the results and conclude in the end of this section.

A. Experiment Settings

We have 7 environments of different layouts: *single cube*, *maze*, *flappy bird*, *monza*, *window*, *tower* and *room*. We first evaluate the effectiveness of our self-implemented Weighted A^* algorithm and then evaluate and visualize the performance of RRT algorithm implemented in *rrt-examples* library.

Experiment results are as following:

B. Weighted A^*

The optimal path length is rounded to *.1f*.

1) single cube: Figure 5: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expande 111 nodes.

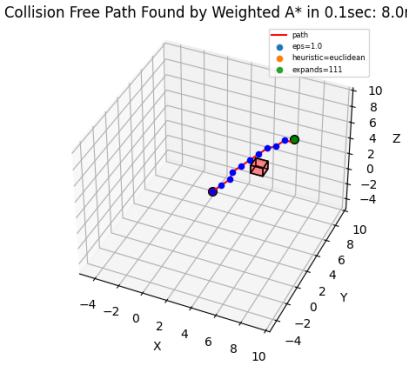

Fig. 5. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

Figure 6: Weighted A*: $\epsilon = 2.0$ with 3D Euclidean Heuristic will expand only 11 nodes.

From above comparison, we can show that ϵ is a trade of between optimality and speed(because there is no obstacles around goal so the path returned by real weighted version with $\epsilon = 2.0$ is also optimal).

2) maze: Figure 7: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expand 8938 nodes.

Figure 8: Weighted A*: $\epsilon = 2.0$ with 3D Euclidean Heuristic will expand 6934 nodes and path found.

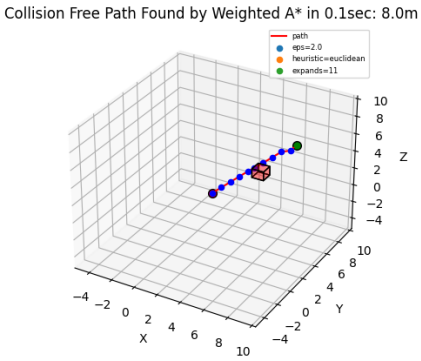

Fig. 6. Weighted A*: $\epsilon = 2.0$ with 3D Euclidean Heuristic

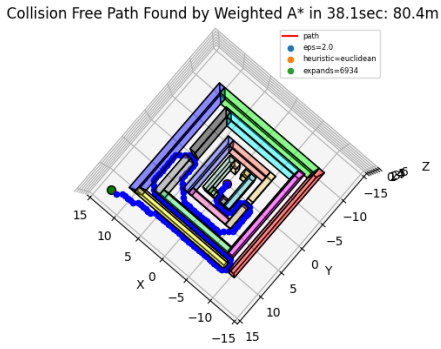

Fig. 8. Weighted A*: $\epsilon = 2.0$ with 3D Euclidean Heuristic

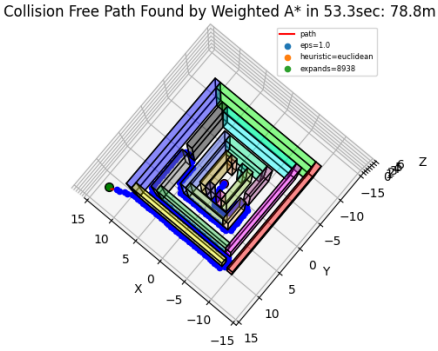

Fig. 7. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

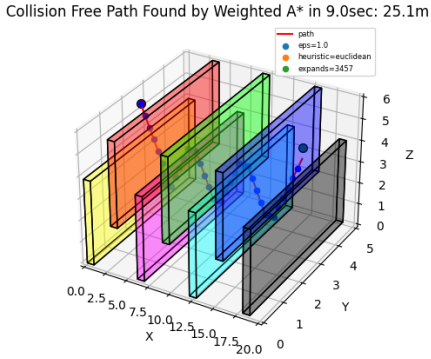

Fig. 9. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

We can see that the weighted A* compared to A* is faster and expands less nodes.

3) flappy bird: Figure 9: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expand 3457 nodes.

Figure 10: Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic will expand 634 nodes and path found.

when $\epsilon = 3.0$, weighted A* only expands about $\frac{1}{6}$ of the nodes compared to A*.

4) monza: Figure 11: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expand 3140 nodes.

Figure 12: Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic will expand 2473 nodes and path found.

5) window: Figure 13: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expand 3978 nodes.

Figure 14: Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic will expand 48 nodes and path found.

6) tower: Figure 15: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expand 2447 nodes.

Figure 16: Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic will expand 311 nodes and path found.

7) room: Figure 17: Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic will expand 287 nodes.

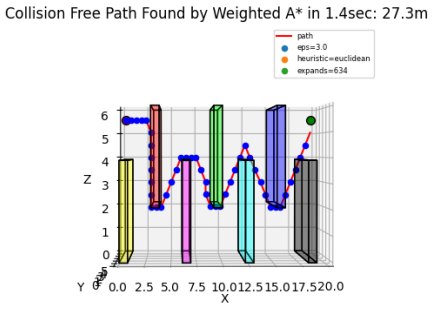

Fig. 10. Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic

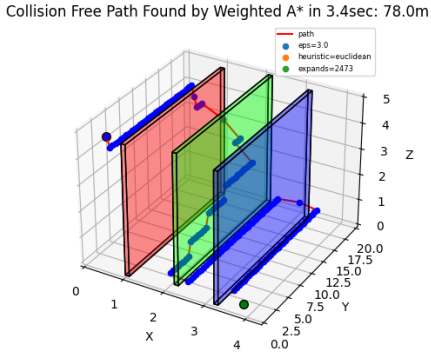

Fig. 12. Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic

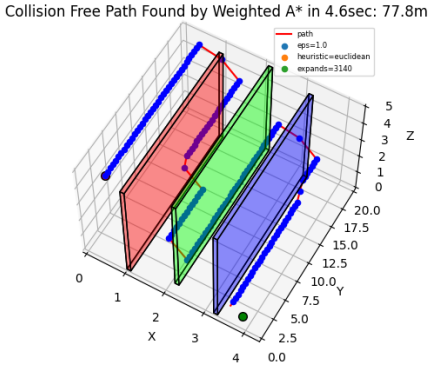

Fig. 11. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

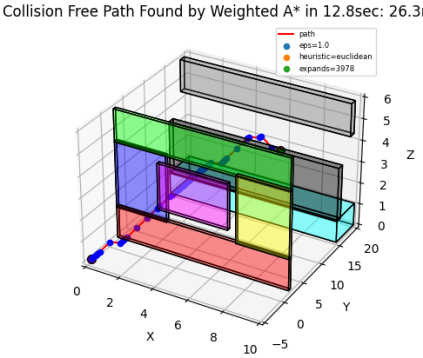

Fig. 13. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

Figure 18: Weighted A*: $\epsilon = 5.0$ with 3D Euclidean Heuristic will expand 77 nodes and path found.

C. RRT

The optimal path length is rounded to *.1f*. There are overall 4 tunable parameters in RRT algorithm because we always want to find a feasible path so we fixed the maximum of iteration number to a very large number $400,000$ to ensure the whole configuration space will nearly be probabilistically convered. Then we mainly tuned Q, r and prc, their meaning is explained in section III-F2

1) single cube: Figure 19: Q=1 r=0.01 prc=0.1 samples 47

Figure 20: Q=1 r=0.01 prc=0.99 samples 7

When there are few obstacles between the start point and the goal point, we can simpliy just increse the probability to check whether the goal point can be directly connected to the tree so that we can find a path using fewer samples.

2) maze: Figure 21: Q=1 r=0.01 prc=0.1 samples 11269

Compared to Weighted A*, which report optimal cost of the path is less than 80m, the RRT find a feasible path of 116m. But the good thing about RRT is that it returns the path in less than 20s,

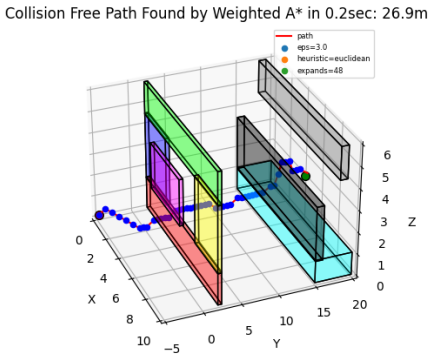

Fig. 14. Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic

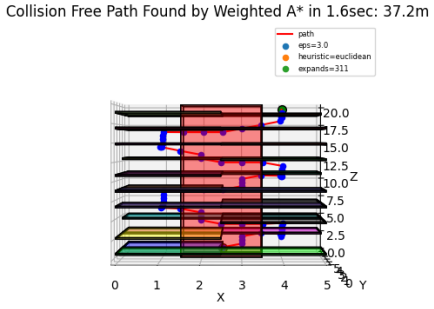

Fig. 16. Weighted A*: $\epsilon = 3.0$ with 3D Euclidean Heuristic

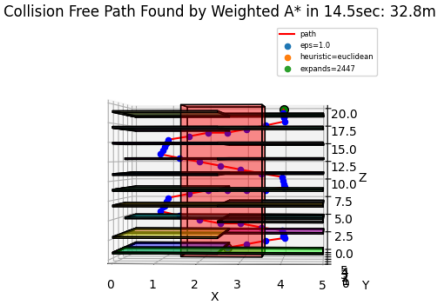

Fig. 15. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

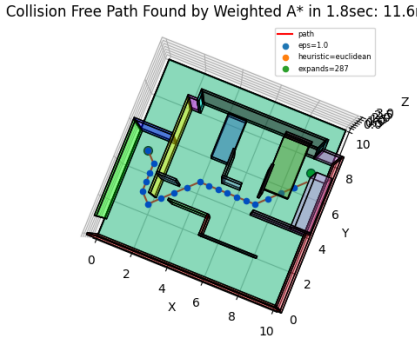

Fig. 17. Weighted A*: $\epsilon = 1.0$ with 3D Euclidean Heuristic

which is so much faster than Weighted A*.

3) bird: Figure 22: Q=1 r=0.01 prc=0.1 samples 536

4) monza: Figure 23: Q=1 r=0.01 prc=0.1 samples 46670

Figure 24: Q=8 r=0.01 prc=0.1 samples 12699

The performance of RRT is worse than Weighted A* in all metrics. And larger Q will sample less even though the found path is longer. This may be because larger Q helps the tree to expand to wider area that is not blocked by local obstacles.

5) window: Figure 25: Q=1 r=0.01 prc=0.1 samples 223

Figure 26: Q=1 r=0.01 prc=0.5 samples 132

Figure 27: Q=1 r=0.01 prc=0.99 samples 329

Figure 28: Q=8 r=0.01 prc=0.1 samples 226

Figure 29: Q=8 r=0.01 prc=0.5 samples 40

Figure 30: Q=8 r=0.01 prc=0.99 samples 17

When there are many obstacles between start point and goal point, increase the probability of directly connect to the goal is not that helpful to find a feasible path.

6) tower: Figure 31: Q=1 r=0.01 prc=0.1 samples 1377

Figure 32: Q=1 r=0.01 prc=0.5 samples 1639

Figure 33: Q=8 r=0.01 prc=0.1 samples 2002

Figure 34: Q=8 r=0.01 prc=0.5 samples 3470

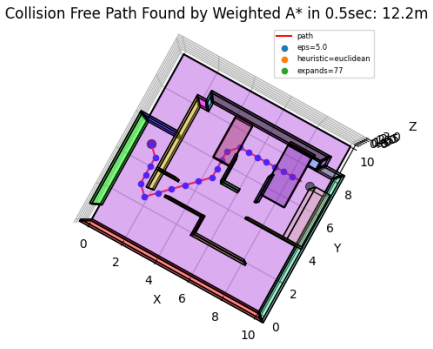

Fig. 18. Weighted A*: $\epsilon = 5.0$ with 3D Euclidean Heuristic

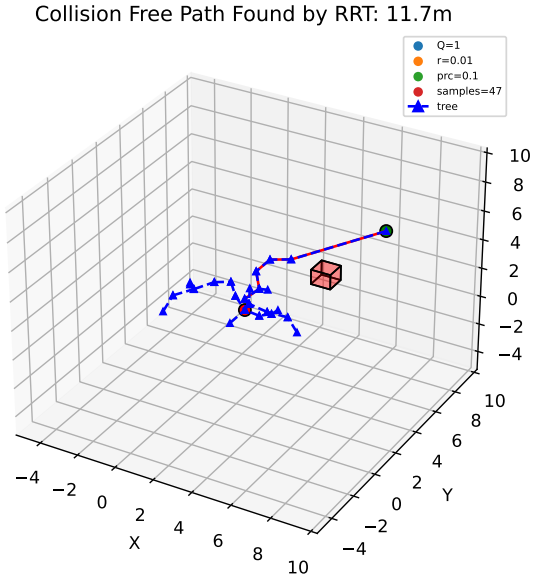

Fig. 19. RRT: Q=1 r=0.01 prc=0.1 samples 47

When the obstacles are very dense, we should use smaller Q and smaller prc and this will help us find the goal more effiently.

7) room: Figure 35: Q=1 r=0.01 prc=0.1 samples 194

Figure 36:

Figure 37: Q=8 r=0.01 prc=0.1 samples 1444

Figure 38: Q=8 r=0.01 prc=0.5 samples 2339

The same conclusion as in tower environment.

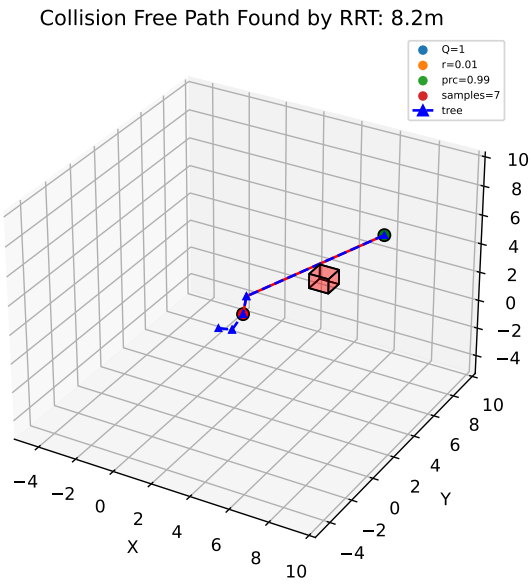

Fig. 20. RRT: Q=1 r=0.01 prc=0.99 samples 7

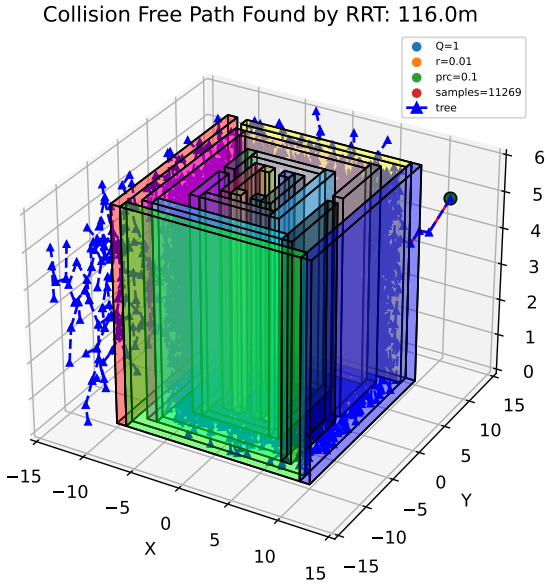

Fig. 21. RRT: Q=1 r=0.01 prc=0.1 samples 11269

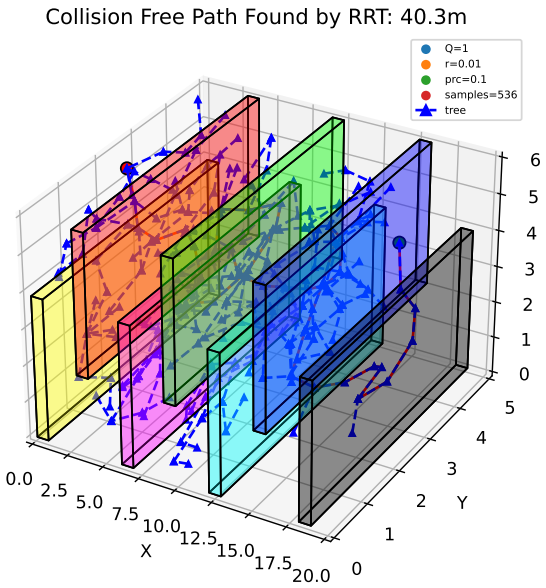

Fig. 22. RRT: Q=1 r=0.01 prc=0.1 samples 536

D. Part B

In Part B's setting, the environment is randomly generated from all 36 possible environments, so we randomly initiated the slover for several times and collected the optimal control sequencies along their according trajectories. And the other observation is that all the grid size is 8x8 and there are only a few wall cells in the fixed positions, so the overall search time is way much longer than in Part A.

E. Results Analysis

From the results, we can see that our experiment basically align with our description in section III, where search-based algorithms will provides finite-time sub-optimality bounds on the solution and guarantes to find a feasible path if it exists in the 3D grid environment settings thus they are resolution complete. And the path given by RRT algorithm is not much faster than Weighted A* algorithm. This may be because our configuration space is of low dimensionality and the advantages of sampling-based algorithms can not be fully observaed. If we use wider and scale up our map, maybe at that time we can see more speed advantage.

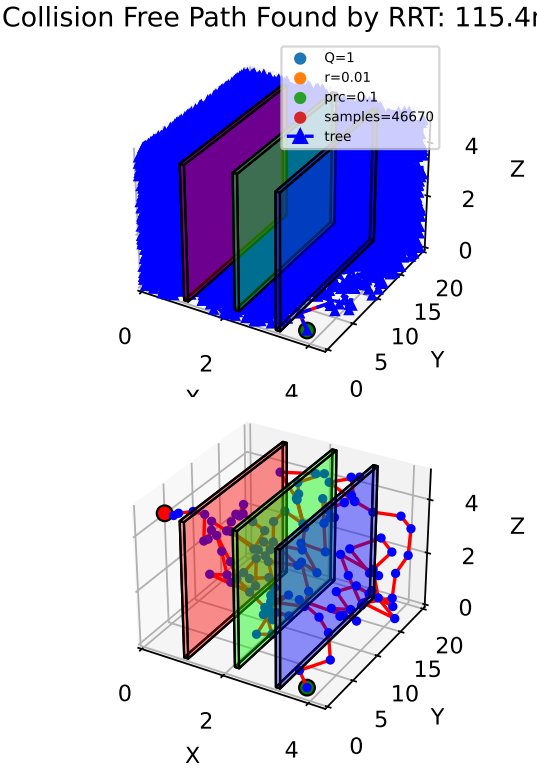

Fig. 23. RRT: Q=1 r=0.01 prc=0.1 samples 46670

F. Conclusions

We performed extensive experiments to compare the different performance between search-based and sampling-based algorithms among 7 different environments and the results basically align with the conclusion in technical approach part.

Acknowledgment

Thanks Prof. Nicolay for such meaningful lectures and the commitment of TA Zhirui Dai.

References

[1] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *CoRR*, vol. abs/1105.1186, 2011. arXiv: 1105.1186. [Online]. Available: http://arxiv.org/abs/1105.1186.

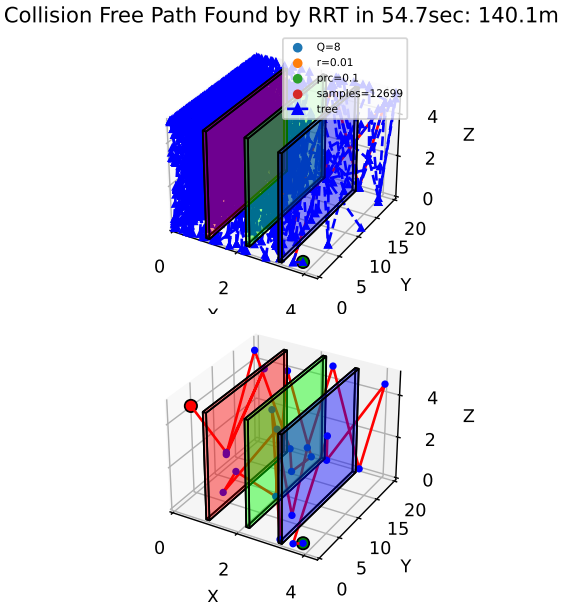

Fig. 24. RRT: Q=8 r=0.01 prc=0.1 samples 12699

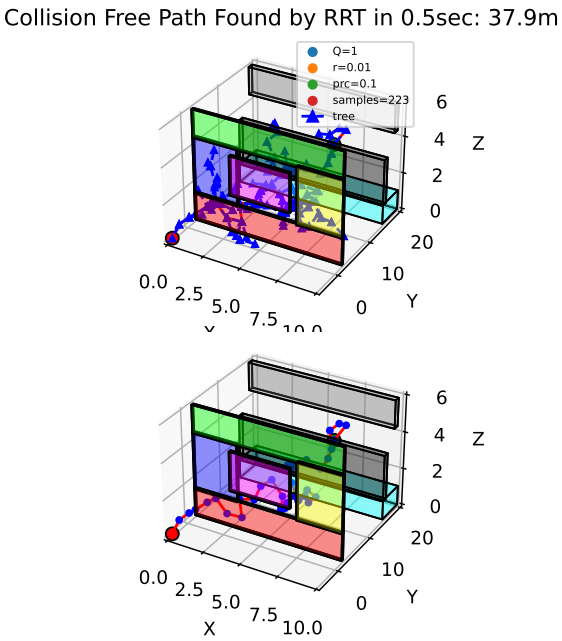

Fig. 25. RRT: Q=1 r=0.01 prc=0.1 samples 223

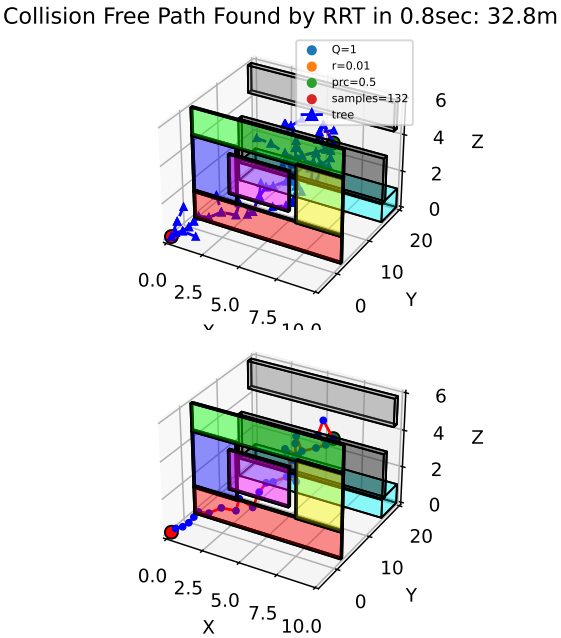

Fig. 26. RRT: Q=1 r=0.01 prc=0.5 samples 132

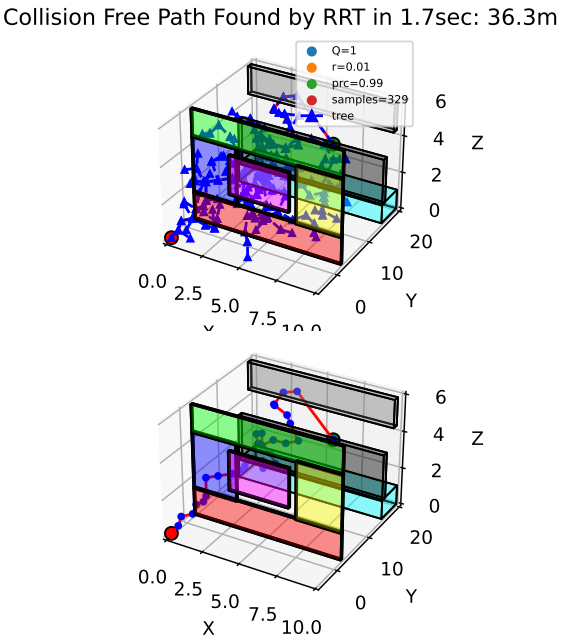

Fig. 27. RRT: Q=1 r=0.01 prc=0.99 samples 329

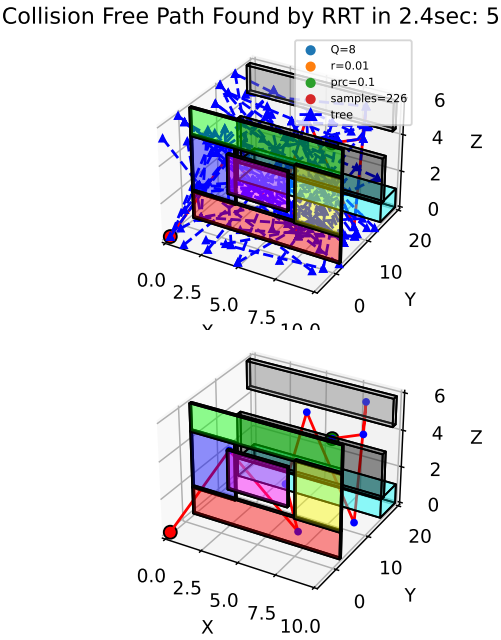

Fig. 28. RRT: Q=8 r=0.01 prc=0.1 samples 226

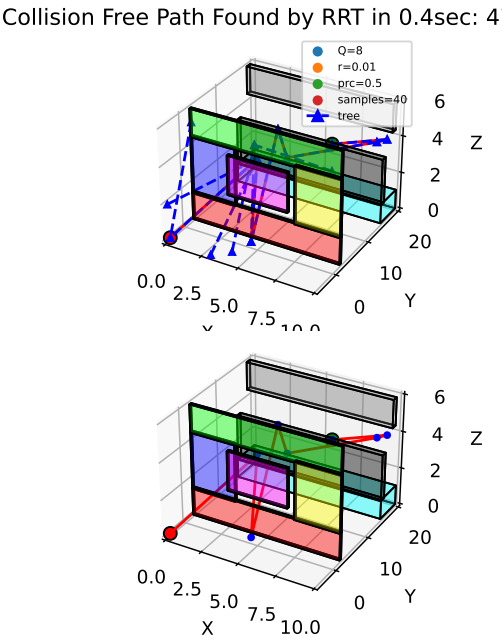

Fig. 29. RRT: Q=8 r=0.01 prc=0.5 samples 40

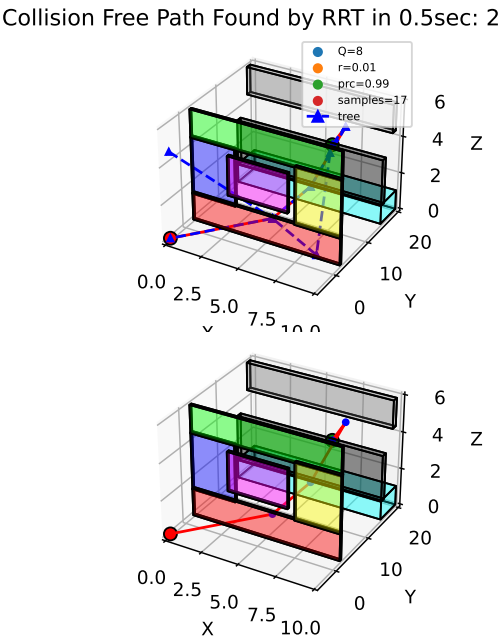

Fig. 30. RRT: Q=8 r=0.01 prc=0.99 samples 17

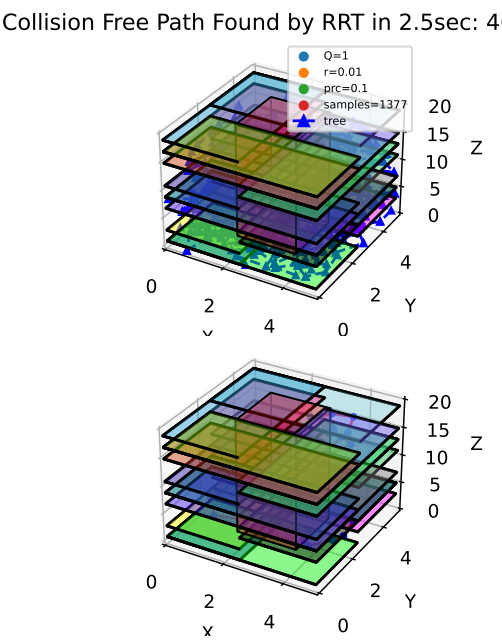

Fig. 31. RRT: Q=1 r=0.01 prc=0.1 samples 1377

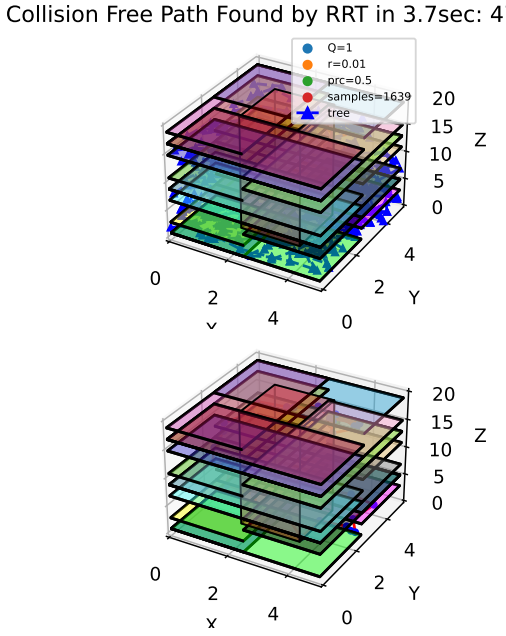

Fig. 32. RRT: Q=1 r=0.01 prc=0.5 samples 1639

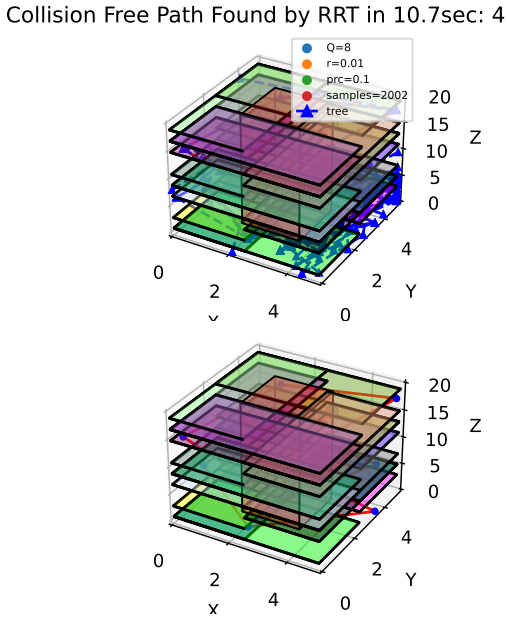

Fig. 33. RRT: Q=8 r=0.01 prc=0.1 samples 2002

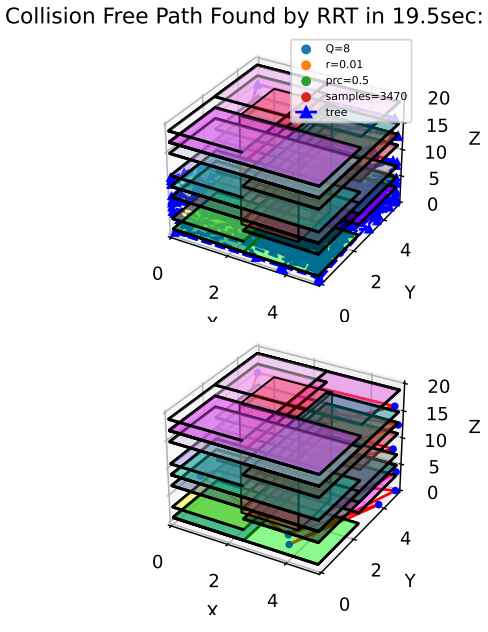

Fig. 34. RRT: Q=8 r=0.01 prc=0.5 samples 3470

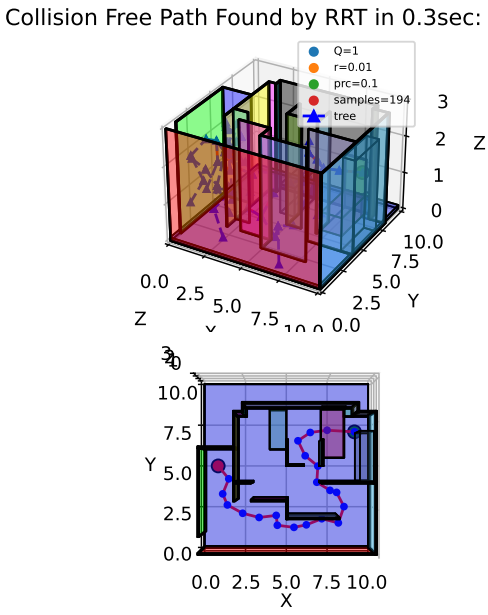

Fig. 35. RRT: Q=1 r=0.01 prc=0.1 samples 194

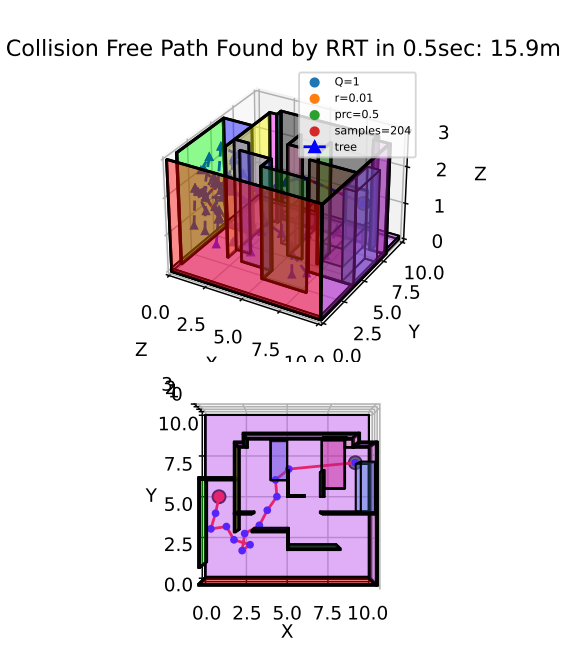

Fig. 36. RRT: Q=1 r=0.01 prc=0.5 samples 204

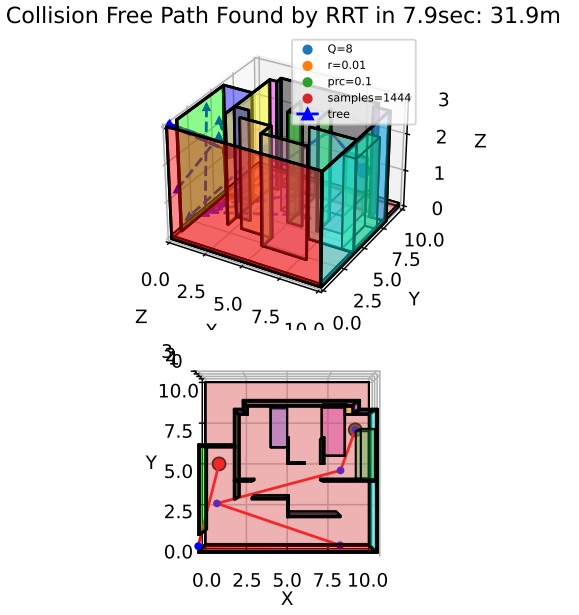

Fig. 37. RRT: Q=8 r=0.01 prc=0.1 samples 1444

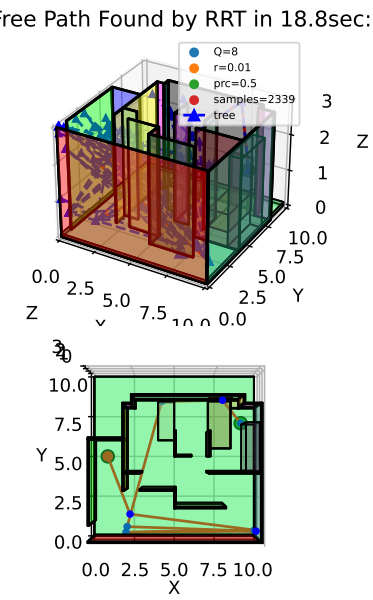

Fig. 38. RRT: Q=8 r=0.01 prc=0.5 samples 2339

[2] *GitHub - motion-planning/rrt-algorithms: N-dimensional RRT, RRT* (RRT-Star) — github.com*, https : / / github . com / motion - planning / rrt - algorithms, [Accessed 22-May-2023].

[3] E. W. Weisstein, *Hessian Normal Form*, en, Text. [Online]. Available: https://mathworld.wolfram.com/ (visited on 05/22/2023).

[4] N. Atanasov, *Lecture pdfs and course website resources*, 2023.

APPENDIX

Before finding the bug of our collision checker, A^* will generate paths that are not collision free in the following 3 environments:

- flappy_bird
- monza
- room

And later we found what is wrong with our checker by examining the *flappy_bird* environment and found a good test case: testing whether line(start point: [3.0, 2.5, 2.5], end point: [3.5, 2.5, 2.0]) will collide with the left plane of the first red obstacle block box(we added some colors to the blocks

for debugging and better visualization). This line indeed collides with the plane but our checker won't detect this. We found this is because the distance sign of the plane to the origin is incorrect. When the left plane is perpendicular to the bottom plane of the boundary and the x-axis offset is greater than 0, the distance sign of the plane should be negative instead of positive because the origin is located in the half-plane specified by the opposite direction of the left plane's normal vertor $\mathbf{n}$. After such modification, we found a collision free path in *flappy_bird*, but a new problem occured: A^* can't find a collision free path for *window* now. And we later found that this is because we set the resolution too small, after changing it to 0.5, we were able to generate optimal feasible path for all the environments.