Motion Planning

Chung-Pang, Wang
University of California San Diego
Department of Electrical and Computer Engineering

***Abstract*—In this work, we will focus on search-based and sampling-based motion planning algorithms in 3-D Euclidean space. We will test both search-based and sampling-based motion planning algorithms in 7 different environments with obstacles that are axis-aligned bounding boxes (AABBs).**

***Index Terms*—search-based, sampling-based motion planning algorithms**

I. Introduction

Motion planning involves the task of finding a feasible path for a robot or an autonomous agent from a starting point to a destination while avoiding obstacles. This work focuses on two primary categories of motion planning algorithms: search-based and sampling-based methods in a 3-D Euclidean space. Search-based algorithms rely on systematic exploration of the state space to find an optimal path. In contrast, sampling-based algorithms generate paths by randomly sampling the state space and connecting these samples to construct a feasible path. For search-based algorithms, we will implement weighted A* path planning algorithm to find an optimal path. For sampling-based algorithms, we will implement Rapidly-exploring Random Trees (RRT) and it's variations and finally compare the optimally and efficiency between search-based and sampling-based algorithm.

II. Problem Formulation

A. Deterministic Shortest Path Problem

Consider a graph with vertex set $\mathcal{V}$, edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, and edge weights $\mathcal{C} := \{c_{ij} \in \mathbb{R} \cup \{\infty\} \mid (i,j) \in \mathcal{E}\}$ where c_{ij} denotes the cost of transition from vertex i to vertex j. Our objective is to find a shortest path from a start node s to an end node τ. Path can be defined as a sequence $i_{1:q} := (i_1, i_2, \ldots, i_q)$ of nodes $i_k \in \mathcal{V}$ and the length of the path can be written as the sum of edge weights along the path:

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

All paths are from $s \in \mathcal{V}$ to $\tau \in \mathcal{V}$: $\mathcal{P}_{s,\tau} := \{i_{1:q} \mid i_k \in \mathcal{V}, i_1 = s, i_q = \tau\}$. The objective of the path planning algorithm is to find a path that has the minimum length from node s to node τ:

$$\text{dist}(s,\tau) = \min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J_{i_{1:q}} \tag{2}$$

$$i^*_{1:q} \in \arg\min_{i_{1:q} \in \mathcal{P}_{s,\tau}} J_{i_{1:q}} \tag{3}$$

Where $i^*_{1:q}$ is the optimal path from start to goal.

B. A* Path Planning Algorithm

A* path planning algorithm is a variation of label correcting algorithm. Label correcting algorithm assigns the lowest cost discovered so far from s to node $i \in V$ label g_i to the explored nodes and correct the label when g_i is reduced, the labels g_j of the children of i can be corrected with $\bar{g}_j = g_i + c_{ij}$. Furthermore, there is a OPEN list stores the set of nodes that can potentially be part of the shortest path to τ.

Instead of using the condition $g_i + c_{ij} < g_\tau$ to correct label, A* algorithm uses $g_i + c_{ij} + h_j < g_\tau$. where h_j is a non-negative lower bound on the optimal cost-to-go from node j to τ known as a heuristic function:

$$0 \leq h_j \leq \text{dist}(j,\tau) \tag{4}$$

The more stringent criterion for admission to OPEN can reduce the number of iterations required by the label correcting algorithm to find an optimal path. CLOSED stores the set of states that have already been expanded. A* can be significantly slow if the environment is complex. An easy solution is to weight heuristic to increase the tendency of heading directly to the goal, called weighted A* algorithm. We will discuss that in the next section.

Algorithm 1 A* Algorithm

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**
  Remove i with smallest $f_i := g_i + h_i$ 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 $\cup \{j\}$
      end if
    end if
  end for
end while

C. Collision Checking Algorithm

Since there are obstacles in the environment the agent is navigating, we need to check if the path will collide with obstacles or not during the planning process. Luckily, the

obstacles are axis-aligned bounding boxes (AABBs), meaning that we can utilize the geometry properties of AABBs to simplify the general collision checking algorithm to improve the efficiency of the path planning algorithm. The main idea of our collision checking algorithm is that since the environment is discretized due to the need of planning algorithm, we can simplified the 3D collision checking of line segments (paths) and AABBs to the 3D collision checking of sampled points from the line segments and AABBs. Therefore, we only need to check if the points are lie in the AABBs or not. We defined 3D point and AABBs collision detection function:

$$\begin{aligned} f(P,B) = \ & (P_x \geq B_{\text{min}X} \wedge P_x \leq B_{\text{max}X}) \\ & \wedge (P_y \geq B_{\text{min}Y} \wedge P_y \leq B_{\text{max}Y}) \\ & \wedge (P_z \geq B_{\text{min}Z} \wedge P_z \leq B_{\text{max}Z}) \end{aligned} \tag{5}$$

Where P are the sampled points and B are the Vertices of the AABBs. Steer(v_p, r) function leads the start points p_s to the next sample point p_{es}.

Algorithm 2 Collision Checking Algorithm

```
1: p_s ← start; p_e ← end; r ← resolution
2: v_p ← p_e − p_s
3: P_es ← Steer(v_p, r)
4: for p_es ∈ P_es do
5:     if f(P, B) then
6:         return TRUE
7:     else
8:         return FALSE
9:     end if
10: end for
```

D. Rapidly-Exploring Random Trees (RRT)

In contrast to search-based path finding algorithms, sampling-based algorithms generate paths by randomly sampling the free state space C_{free} and connecting these samples to construct a feasible path. There are several 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 a point $\mathbf{x} \in C$, returns a vertex $\mathbf{v} \in V$ that is closest to $\mathbf{x}$:

$$\text{Nearest}((V,E),\mathbf{x}) := \arg\min_{\mathbf{v}\in V} \|\mathbf{x}-\mathbf{v}\|$$

- **Near:** given a graph $G = (V, E)$ with $V \subset C$, a point $\mathbf{x} \in C$, and $r > 0$, returns the vertices in V that are within a distance r from $\mathbf{x}$:

$$\text{Near}((V,E),\mathbf{x},r) := \{\mathbf{v} \in V \mid \|\mathbf{x}-\mathbf{v}\| \leq r\}$$

- **Steer:** given points $\mathbf{x}, \mathbf{y} \in C$ and $\epsilon > 0$, returns a point $\mathbf{z} \in C$ that minimizes $\|\mathbf{z} - \mathbf{y}\|$ while remaining within ϵ from $\mathbf{x}$:

$$\text{Steer}_\epsilon(\mathbf{x},\mathbf{y}) := \arg\min_{\mathbf{z}:\|\mathbf{z}-\mathbf{x}\|\leq\epsilon} \|\mathbf{z}-\mathbf{y}\|$$

- **CollisionFree:** given points $\mathbf{x}, \mathbf{y} \in C$, returns True if the line segment between $\mathbf{x}$ and $\mathbf{y}$ lies in C_{free} and False otherwise.

RRT first sample a new configuration $x_{rand} \in C_{free}$, find the nearest neighbor $x_{nearest}$ in G and connect them if straight line is collision-free, finally stop building the tree when the goal configuration x_τ. Although RRT is a fast way to obtain a feasible path with little memory and computation compare to A* algorithm, RRT is proven to be not optimal [5] since we never modify the tree once we build it. Therefore, we will implement an optimal version of RRT called RRT* which rewires the tree to ensure asymptotic optimality in the next section.

Algorithm 3 RRT

```
1: V ← {x_s}; E ← ∅
2: for i = 1 . . . n do
3:     x_rand ← SampleFree()
4:     x_nearest ← Nearest((V, E), x_rand)
5:     x_new ← Steer(x_nearest, x_rand)
6:     if CollisionFree(x_nearest, x_new) then
7:         V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new)}
8:     end if
9: end for
10: return G = (V, E)
```

III. Methods

A. Weighted A*

An simple improvement of A* path planning algorithm is to weight the heuristic function, forcing the agent to expand toward the goal extensively. The only variation is to change line 4 of the original A* algorithm $f_i := g_i + h_i$ to $f_i := g_i + \epsilon h_i$. Since the importance of the heuristic function had increased, the nodes that has the smallest f_i will more likely be the nodes toward the goal. With weighted heuristic, the agent might find the optimal path in a shorter amount of time. However, if there is a trap (e.g. curved wall) between the start and the goal, the agent might get stuck in that region for a while. Besides, choosing large ϵ might cause the heuristic function to be not admissible and consistent, meaning that it is no longer guarantee to find an optimal path or even a feasible path.

B. RRT*

As mentioned above, RRT is not optimal since we never modify the tree once we build it. Therefore, a variation RRT* introduces rewiring mechanism to retain optimality of RRT. Similarly, RRT* first generates a new potential node x_{new} but instead of finding the closest node in the tree, find all nodes within a neighborhood $\mathcal{N}$ of radius $\min\{r^*, \epsilon\}$ where:

$$r^* > 2\left(1+\frac{1}{d}\right)^{1/d}\left(\frac{\text{Vol}(C_{\text{free}})}{\text{Vol}(\text{Unit } d\text{-ball})}\right)^{1/d}\left(\frac{\log|V|}{|V|}\right)^{1/d} \tag{6}$$

$r*$ is the suggested connection radius of RRT* [6] to build a better tree. Let $x_{\text{nearest}} = \arg\min_{x_{\text{near}} \in \mathcal{N}} g(x_{\text{near}}) + c(x_{\text{near}}, x_{\text{new}})$ be the node in $\mathcal{N}$ on the currently known shortest path from x_s to x_{new}. Set the label of x_{new} to $g(x_{new}) = g(x_{nearest}) + c(x_{nearest}, x_{new})$. Check all nodes $x_{\text{near}} \in \mathcal{N}$ to see if re-routing through x_{new} reduces the path length. If $g(x_{\text{new}}) + c(x_{\text{new}}, x_{\text{near}}) < g(x_{\text{near}})$, then remove the edge between x_{near} and its parent and add a new edge between x_{near} and x_{new}.

Algorithm 4 RRT*

$V \leftarrow \{x_s\}; E \leftarrow \emptyset$
for $i = 1 \dots n$ **do**
  $x_{\text{rand}} \leftarrow \text{SampleFree}()$
  $x_{\text{nearest}} \leftarrow \text{Nearest}((V, E), x_{\text{rand}})$
  $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$
  if $\text{CollisionFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \text{Near}((V, E), x_{\text{new}}, \min\{r^*, \epsilon\})$
    $V \leftarrow V \cup \{x_{\text{new}}\}$
    $c_{\min} \leftarrow \text{Cost}(x_{\text{nearest}}) + \text{Cost}(\text{Line}(x_{\text{nearest}}, x_{\text{new}}))$
    for $x_{\text{near}} \in X_{\text{near}}$ **do**
      if $\text{CollisionFree}(x_{\text{near}}, x_{\text{new}})$ **then**
        if $\text{Cost}(x_{\text{near}}) + \text{Cost}(\text{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
          $x_{\min} \leftarrow x_{\text{near}}$
          $c_{\min} \leftarrow \text{Cost}(x_{\text{near}}) + \text{Cost}(\text{Line}(x_{\text{near}}, x_{\text{new}}))$
        end if
      end if
      end for
      $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$
      for $x_{\text{near}} \in X_{\text{near}}$ **do**
        if $\text{CollisionFree}(x_{\text{new}}, x_{\text{near}})$ **then**
          if $\text{Cost}(x_{\text{new}}) + \text{Cost}(\text{Line}(x_{\text{new}}, x_{\text{near}})) < \text{Cost}(x_{\text{near}})$ **then**
            $x_{\text{parent}} \leftarrow \text{Parent}(x_{\text{near}})$
            $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
          end if
        end if
      end for
    end if
  end for
  return $G = (V, E)$

C. Bi-Directional RRT

While RRT* improves upon the original RRT by providing optimality, there are still potential issues that may arise. If the searching environment is easy to pass through in one direction but very difficult from the other direction, start building the tree from the wrong way might take a huge amount of time to find a feasible path. The problem is called the bug traps problem as shown in Fig.1. A simple and intuitive way to solve it is to build the tree from both start and goal. As shown in Algorithm.5, Bi-Directional RRT build trees from both sides and swap the building priority if the trees from the other side grows faster.

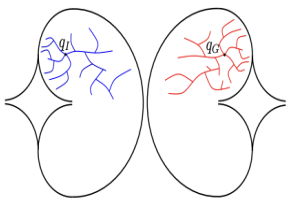

Fig. 1. Bug Trap

Algorithm 5 Bi-Directional RRT

$V_a \leftarrow \{x_s\}; E_a \leftarrow \emptyset; V_b \leftarrow \{x_r\}; E_b \leftarrow \emptyset$
for $i = 1$ **to** n **do**
  $x_{\text{rand}} \leftarrow \text{SampleFree}()$
  $x_{\text{nearest}} \leftarrow \text{Nearest}((V_a, E_a), x_{\text{rand}})$
  $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$
  if $x_{\text{new}} \neq x_{\text{nearest}}$ **then**
    $V_a \leftarrow V_a \cup \{x_{\text{new}}\}; E_a \leftarrow E_a \cup \{(x_{\text{nearest}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{nearest}})\}$
    $x'_{\text{nearest}} \leftarrow \text{Nearest}((V_b, E_b), x_{\text{new}})$
    $x'_{\text{new}} \leftarrow \text{Steer}(x'_{\text{nearest}}, x_{\text{new}})$
    if $x'_{\text{new}} \neq x'_{\text{nearest}}$ **then**
      $V_b \leftarrow V_b \cup \{x'_{\text{new}}\}; E_b \leftarrow E_b \cup \{(x'_{\text{nearest}}, x'_{\text{new}}), (x'_{\text{new}}, x'_{\text{nearest}})\}$
      if $x'_{\text{new}} = x_{\text{new}}$ **then**
        return SOLUTION
      end if
      if $|V_b| < |V_a|$ **then**
        SWAP$((V_a, E_a), (V_b, E_b))$
      end if
    end if
  end if
end for
return FAILURE

IV. Implementation Details

A. Discrete Map and Collision Checking

To run our path planning algorithm in 3D continuous Euclidean space, we need to discretize the map to 3D grids. I had tried different map resolution (from 0.1 to 0.5) and found 0.3 to be the balance of runtime and optimality (smallest path length). For the motion model, I'm using 26 directions at each step with a equal cost 0.5, meaning the agent will try 26 different directions in a 3D sphere with radius 0.5. For collision checking, I use R-tree for spatial indexing AABBs to check if a given point is in the AABBs or not. I can build R-tree before planning started and access efficiently during planning, saving large amounts of time.

B. Search-Based A* and weighted A* Algorithm

For the implementation of A* algorithm, I strictly follow Algorithm.1 and use Euclidean distance $h_i := \|\mathbf{x}_\tau - \mathbf{x}_i\|_2$ as my heuristic function. I utilize priority queue (pqdict in Python) for OPEN list to speed up the process of finding the smallest f value. Furthermore, I test relative large weight $\epsilon = 1.5$ to see how will the OPEN and CLOSE differ from the original A* algorithm.

C. Sampling-Based RRT* Algorithm

For the implementation of sampling-based methods, I utilized Python motion planning library at https://github.com/motion-planning/rrt-algorithms and deploy on our 3D Euclidean space. I test RRT, RRT* and Bi-Directional RRT* and visualize the difference of the trees.

V. Results

A. A* and Weighted A* Planning Algorithm

Table.1 shows the runtime (in seconds), path length/cost (in meters) and the number of OPEN and CLOSED nodes at terminal state of A* path planning algorithm in different environments. We can see from Fig.3, maze environment is the

most complex, as a result, the expanded states are extremely larger the opened node, meaning the algorithm explored many nodes and find the smallest f value throughout many nodes at each iterations, causing long time to terminate but guarantee optimality. Fig.3 are visualization of the non-weighted A* algorithm in 7 different environments, the yellow points are the CLOSED nodes and the grey points are the OPEN nodes in the end. As mention in previous section, weighting heuristic function of A* algorithm improve the efficiency of A* algorithm. Fig.2 shows the comparison of the visualization of weighted A* ($\epsilon = 1.5$) and original A* algorithm. We can see from the bottom left and right in Fig.2 that the numbers OPEN and CLOSED are significantly decreased and aggregated along the path since weighted heuristic increase the tendency of moving toward to the goal. For the Flappy Bird environment weighted A* only takes 25 seconds to find an optimal path and the Window case only takes 1.06 seconds which is 2 4 times faster than A* algorithm.

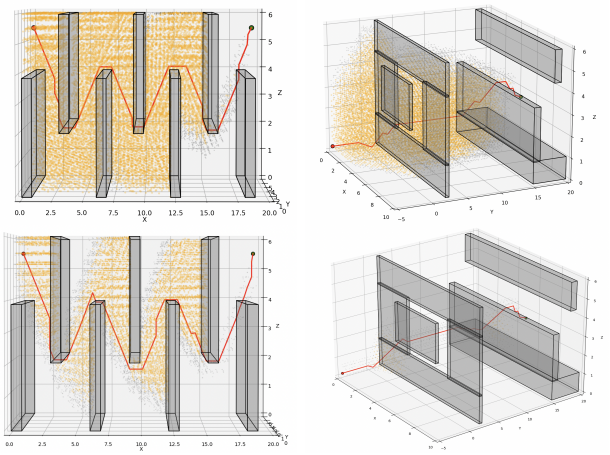

Fig. 2. A* (Top Left and Right) and Weighted A* (Bottom Left and Right) in different test scenario

Test Scenario	Time Taken	Path Length	OPEN	CLOSE
Single Cube	0.5	8.47	353	18
Maze	205.48	75.04	4277	57590
Flappy Bird	47.67	26.25	821	14577
Monza	41.44	76.38	1683	13942
Window	51.04	26.59	5882	14135
Tower	39.29	29.07	1079	11621
Room	4.80	11.55	1162	1288

TABLE I
SUMMARY OF A* PATH PLANNING EXPERIMENTS

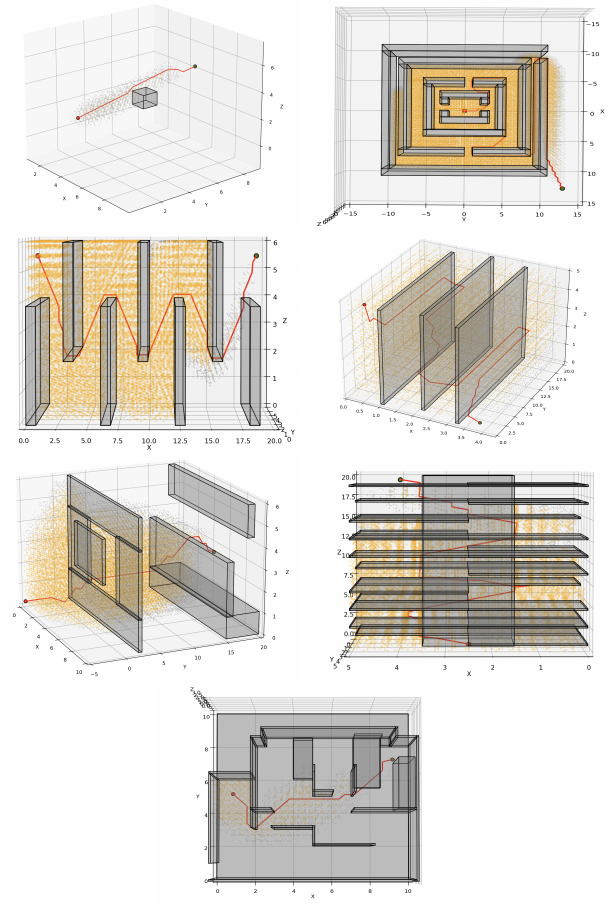

Fig. 3. A* path planning algorithm in different test scenario

B. RRT-Based Planning Algorithm

As mentioned in the previous section, RRT is not optimal since we never modify the tree once we build it. We can see from Fig.4, the paths RRT find are not optimal and take longer path to reach the goal while RRT* rewire a better path when building trees thus the trees appear to be more like straight lines, improves optimality greatly. As the bug trap problem mentioned above, I test Bi-Directional RRT* for 7 different environments. Bi-Directional RRT* not only build trees from both start and goal sides but also rewires trees when building them. We can see from table.2, most of the test scenario are faster than A* search-based algorithm except for the Maze and Tower environment. We can see from Fig.5 that those two environment are more complex and this is when search-based algorithm might out-win the sampling-based algorithm in runtime because the sampling-based algorithm might sample lots of useless tree and get stuck at several free space. But for the other test case, Bi-Directional RRT* is extremely efficient for finding a feasible path with a negligible loss of optimality in path length.

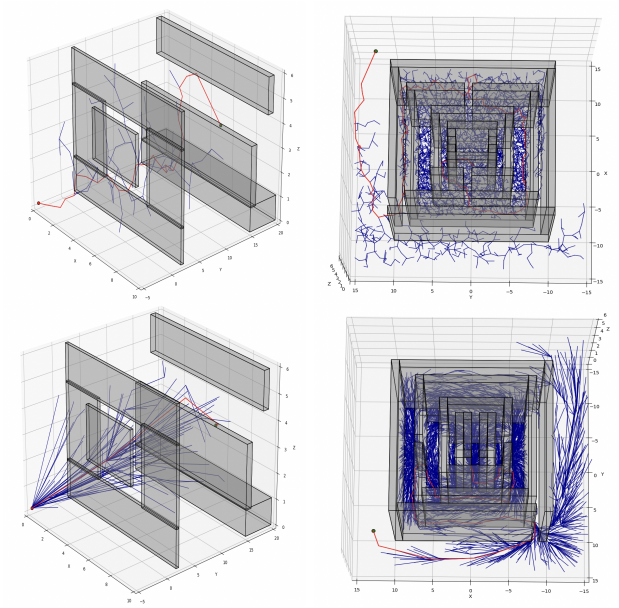

Fig. 4. RRT in Window and Maze Environments(Top Left, Top Right); RRT* in Window and Maze Environments(Bottom Left, Bottom Right)

Test Scenario	Time Taken	Path Length	Samples
Single Cube	0.23	7.92	70
Maze	241.9	93	11047
Flappy Bird	22.16	28.75	614
Monza	35.03	78.99	1148
Window	0.989	24.51	72
Tower	43.77	32.07	1050
Room	0.82	11.55	93

TABLE II
SUMMARY OF BI-DIRECTIONAL RRT* PATH PLANNING EXPERIMENTS

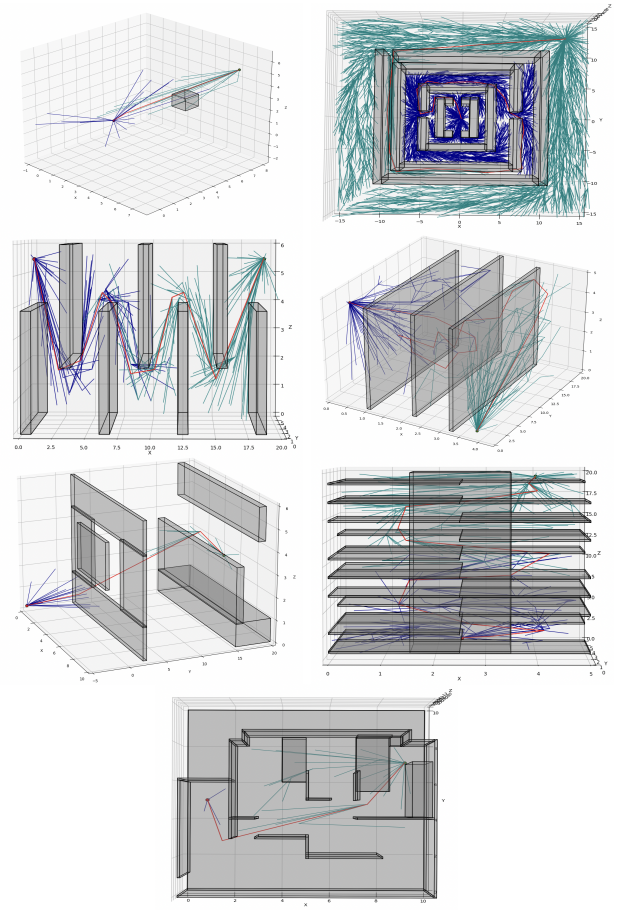

Fig. 5. Bi-Directional RRT* path planning algorithm in different test scenario

REFERENCES

[1] UCSD ECE276B slides on Markov Chains https://natanaso.github.io/ece276b/ref/ECE276B_2_MC.pdf

[2] UCSD ECE276B slides on Markov Decision Processeshttps://natanaso.github.io/ece276b/ref/ECE276B_3_MDP.pdf

[3] UCSD ECE276B slides on Deterministic Shortest Pathhttps://natanaso.github.io/ece276b/ref/ECE276B_5_DSP.pdf

[4] Article of different 3D collision detection methods Pathhttps://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection

[5] Karaman, S., & Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *The International Journal of Robotics Research, 30*(7), 846-894. https://doi.org/10.1177/0278364911406761

[6] Bhattacharya, A., Agarwal, S., & Karaman, S. (2018). Revisiting the Asymptotic Optimality of RRT*. arXiv preprint arXiv:1802.06305.

[7] UCSD ECE276B slieds on Sampling-Based Motion Planning https://natanaso.github.io/ece276b/ref/ECE276B_9_SamplingBasedPlanning.pdf