ECE 276B Project 2 – Motion Planning

Yu-Hao Liu
dept. of Electrical and Computer Engineering
University of California, San Diego
San Diego, CA
yul133@eng.ucsd.edu

***Abstract*—In this project, we focus on comparing the performance of search-based and sampling -based planning algorithms in 3-D Euclidean space. We choose Weighted A* algorithm to compare with RRT algorithm.**

Keywords—Motion Planning, Search-Based Planning, Sample-Based Planning, Weighted A*, RRT

I. INTRODUCTION

In this paper, we aims to demonstrate two planning algorithms that are based on different concepts. First algorithm is Weighted A*, which is the advanced version of A* algorithm, and also a common approach of Search-based planning algorithms. Original A* is composed of the Dijkstra's algorithm and a heuristic function. Compared to original A* algorithm, Weighted A* provides a more flexible restriction on the selection of a path. The result would be sub-optimal but the efficiency increases.

The second algorithm is Rapidly Exploring Random Tree, or RRT. RRT is one of the most common Sample-based algorithm. RRT constructs a tree from random samples with root. The tree is grown until it contains a path to the goal. Since it generates a path according to the start position and the goal position, it is well-suited for single-shot planning. However, there are some disadvantages. The paths RRT computes may be sub-optimal and the path smoothing is required as a post-processing step. In addition, finding a feasible path in highly-constrained environment is challenging and expensive.

II. PROBLEM STATEMENT

We formulate this problem as a deterministic shortest path problem and consider following basic elements:

A. Path

A sequence

$$i_{1:q} \coloneqq (i_1, i_2, \dots, i_q)$$

of nodes $i_k \in \mathcal{V}$

B. All path from $s \in \mathcal{V}$ to $\tau \in \mathcal{V}$

$$\mathbb{I}_{s,\tau} \coloneqq \{ i_{1:q} \mid i_k \in \mathcal{V}, i_1 = s, i_q = \tau \}$$

C. Path Length

The sum of the arc lengths over the path

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

D. Objective

Find a path that has the smallest length from node s to node τ

$$i^*_{1:q} = \underset{i_{1:q} \in \mathbb{I}_{s,\tau}}{\operatorname{argmin}} J^{i_{1:q}}$$

E. Assumption

For all $i \in \mathcal{V}$ and for all $i_{1:q} \in \mathbb{I}_{i,i}$, $J^{i_{1:q}} \geq 0$. In other words, there are no negative cycles in the graph.

III. TECHNICAL APPROACH

After well defining all elements we need for a shortest path problem, we implement the several techniques to compute a desired path.

A. Label Correcting Algorithms (LCA)

Instead of visiting all nodes to compute the shortest path, LCA only visit a portion of nodes. It prioritizes the visited nodes i using the cost-to-arrive values $V_t^F(i)$. There are several ideas in this algorithm.

First, we define a label g_i that is an estimate of the lowest cost from node s to each visited node $i \in \mathcal{V}$. Each time g_i is reduced, the labels g_j, which node j is one of the children of node i, can be corrected i.e. $g_j \coloneqq g_i + c_{ij}$. Furthermore, we create an OPEN set that contains set of nodes that can potentially be part of the shortest path to τ. To be more precise, the nodes in OPEN set are the search frontier in every search iteration.

B. A* Algorithm

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

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

where h_j is a positive lower bound on the optimal cost to get from node j to τ and is known as heuristic.

By adding up a more stringent criterion can reduce the number of iterations required by the LCA. The heuristic is constructed depending on "special knowledge" about the problem. The more accurately h_j estimates the optimal cost from j to τ, the more efficient the A* algorithm becomes.

A heuristic function must satisfy following conditions:

- Admissible: $h_i \leq dist(i,\tau)$ for all $i \in \mathcal{V}$
- Consistent: $h_\tau = 0 \; and \; h_i \leq c_{ij} + h_j$ for all $i \neq \tau \; and \; j \in Children(i)$

There are many choices we can choose to set as the heuristic function such as Euclidean distance (l2-norm) or Diagonal distance. In this project, we decide to use the Manhattan distance, which is defined as:

$$h_i := \|x_\tau - x_i\|_1 := \sum_k |x_{\tau,k} - x_{i,k}|$$

C. Weigthed A* Algorithm Psuedocode

Algorithm Weighted A* Algorithm

OPEN ← {s}, CLOSED ← {}, $\epsilon \geq 1$
$g_s = 0, g_\tau = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
while $\tau \notin CLOSED$ **do**
    Remove i with min$(g_i + \epsilon h_i)$ from OPEN
    Insert i into CLOSED
    for $j \in Children(i)$ and j $\notin CLOSED$ **do**
        if $g_i >$ $(g_i + c_{ij})$ **then**
            $g_i \leftarrow (g_i + c_{ij})$
            Parent(j) ← i
            if $j \in OPEN$ **then**
                Update priority of j
            else
                OPEN ← OPEN ∪ $\{j\}$

D. Collision Checking Algorithm

In this project, we are provided with maps that has several obstacles. We have implemented an algorithm to avoid collision between line segments and axis-aligned bounding boxes (AABBs) in continuous 3D space.

In our algorithm, we create an obstacle map whenever the planner is initiated. The algorithm receives the map and turns the map into an obstacle map that is a Boolean grid map. The grid shows True if it is inside an obstacle or has intersected with it and False otherwise.

Therefore, when the planner is exploring the map, each node would be verified whether it is a valid node, which is not out of bound and doesn't exist inside the obstacles. By doing so, we can ensure that the planner only explores the accessible positions and the path it derives is available for the robot.

E. Properties of implementation of Weighted A*

- Optimality

  Using the parameter ϵ gives a suboptimal result compared to the original A*. The path is at most two times longer than the original one.

- Memory and time efficiency

  By taking advantage of heuristic function, it is unnecessary to explore the whole map to find an optimal path. Therefore, we could save time and memory.

F. Procedures for Sampling-based Motion Planning

For sampling-based motion planning, before we start discussing the RRT algorithm. There are several functions that are critical for the implementation.

- SAMPLEFREE

  Return i.i.d. samples from C_{FREE} (Configuration space that doesn't have obstacles)

- NEAREST

  Given a graph G =(V, E) with V ⊂ C and a point $x \in$ C, returns a vertex v ∈ V that is closest to x:

  $$NEAREST\big((V,E),x\big) := \underset{v \in \mathcal{V}}{\operatorname{argmin}} \|x - v\|$$

- STEER

  Given points x,y ∈ C and $\epsilon > 0$, returns a point z ∈ C that minimize the norm of z and y while remaining within ϵ from x:

  $$STEER_\epsilon(x,y) := \underset{z:||z-x|| \leq \epsilon}{\operatorname{argmin}} \|z - y\|$$

- COLLISIONFREE

  Given points x,y ∈ C, return Ture if the line segment between x and y lies C_{free} and False otherwise.

G. Rapidly Exploring Random Tree Algorithm (RRT)

RRT constructs a tree from random samples with root x_s. In detail, RRT samples a new

configuration x_{rand} , find the nearest neighbor $x_{nearest}$ in G. In $x_{nearest}$ 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.

To reach the goal, during the iteration expanding edges and nodes, RRT occasionally add the goal configuration x_τ and see if it gets connected to the tree.

RRT can be implemented in the original workspace as well as in a configuration space. However, if constructing in a C-space, one needs to consider what distance function to use to find the nearest configuration. Also, one needs to ensure the path is collision-free between two configurations.

In this project, we implement the goal-biased sampling to reduce the total number of configurations. With probability (1-p_g), x_{rand} is chosen as a uniform sample in C_{free} and with probability p_g, $x_{rand} = x_\tau$. For results that present in the next section, we set $p_g = 0.5$ as default case.

H. RRT Algorithm Psuedocode

Algorithm RRT Algorithm

V ← $\{x_s\}$; E ← $\emptyset$
for $i = 1 \ldots n$ 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
        V ← V ∪ $\{x_{new}\}$
        E ← E ∪ $\{x_{nearest}, x_{new}\}$
return G = (V, E)

I. RRT Algorithm Implementation

To implement the RRT, we choose the library from Github [1]. The algorithm defines an n-dimensional search space, and n-dimensional obstacles within the space. Also, it utilizes R-Trees to improve the performance by avioding point-wise collision-checking and distance-checking. We assign the start and goal locations as well as parameters such as the number of iterations, the length of edge.

IV. RESULTS

For the last part, I would present results that apply two planning algorithms on seven types of maps. Each map has different layout and therefore could provide various scenarios to test the performance of two planners.

All graphs are presented below, Fig.1 to Fig.7 are the seven maps that is applied with A* algorithm to compute the paths, while the paths of Fig. 8 to Fig. 14 are derived by the RRT algorithm. For the graphs using A*, the control parameters are fixed. The steering distance is 0.1 meter, the epsilon is 2, and the heuristic function is set to l1-norm. However, for the second part, we have tried different parameters to observe the change.

To compare the performance of two planners, we will discuss three perspectives:

A. Quality of computed paths

As you can see, the results of two planners are quite different. The trajectory derived by the A* planner is smoother and more precise compared to the one from the RRT planner. This is predictable since that the principle of two planners are not the same. A* computes the shortest path and combines with the heuristic function. The result would be more like visibility graph, which finds the shortest path within the view. On the other hand, RRT samples a node at each iteration, it is possible that the nodes it sampled is not an optimal choice. As a result, the trajectory seems to be less direct and precise.

B. Number of considered nodes

As for the memory size used by the planner, RRT has better performance in average case. In general, RRT could find a path within 1000 samples while A* needs 2000 iterations to complete it. However, when it comes to a more closed environment or narrow passages, RRT would require significantly large number of samples to find a feasible, but not optimal, path. For example, the maze map cost the RRT planner 20000 samples to find a path while the A* planner takes only 7000 iterations.

C. Effects of different parameters

For this section, we use the results from the RRT planner to illustrate the difference. First parameter is the number of samples, which is discussed in the previous part. As shown in Fig. 8, the RRT planner needs sufficient samples to find a feasible path. Second parameter is the steering distance. The Fig. 10 shows the change while using different length, longer distance results in finding a path that prefers going through sparse space; meanwhile, losing the chance to find a shortcut. Last but not least, we change the probability of sampling the goal. As shown in Fig. 13, for a complicated map

like maze, using larger probability could take fewer cost.

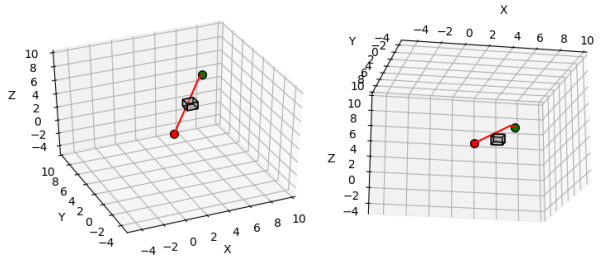

Fig. 1. Single Cube (A*)

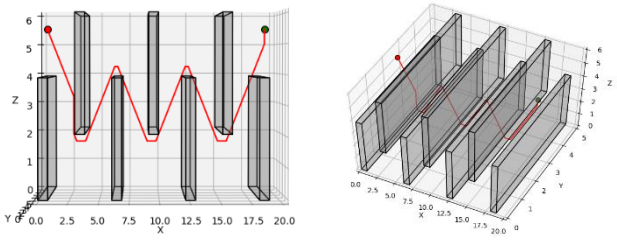

Fig. 2. Flappy Bird (A*)

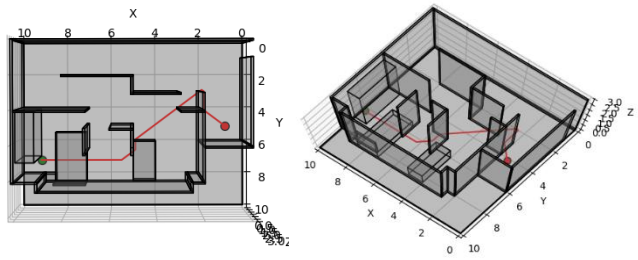

Fig. 3. Room (A*)

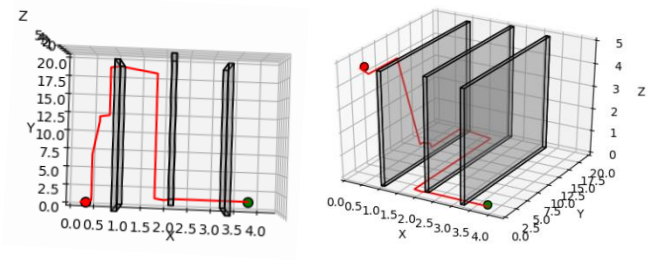

Fig. 4. Monza (A*)

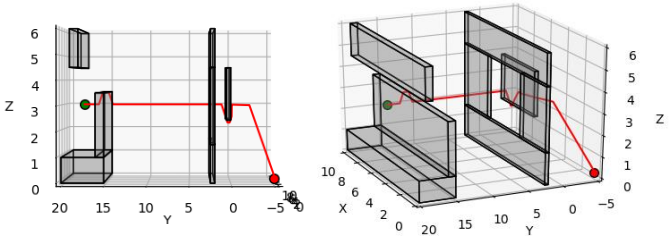

Fig. 5. Window (A*)

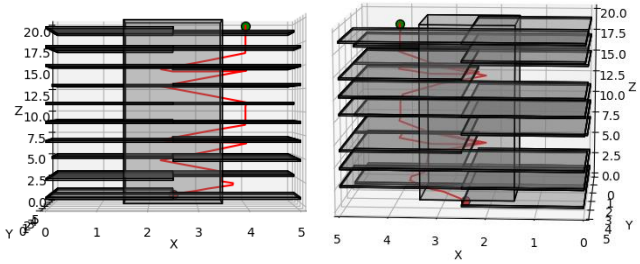

Fig. 6. Tower (A*)

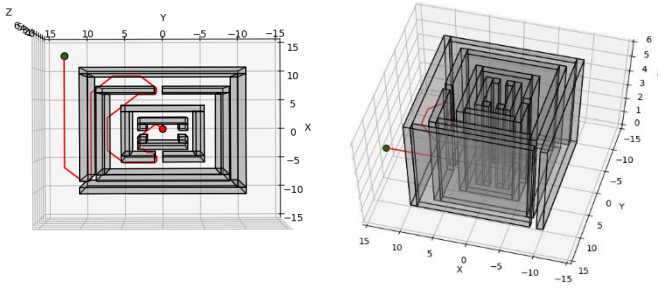

Fig. 7. Maze (A*)

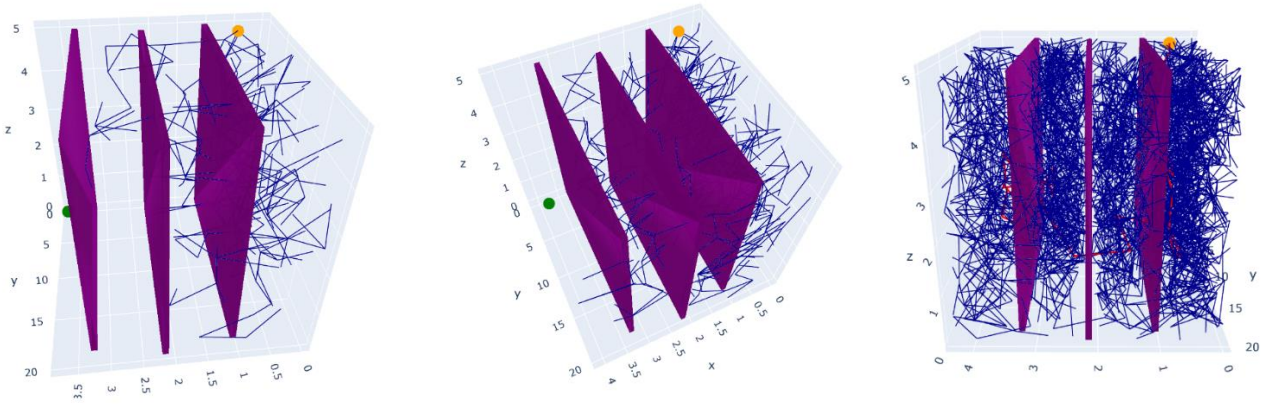

Fig. 8. Monza using RRT with different samples (Left: 1024 / Middle: 2000 / Right: 7802)

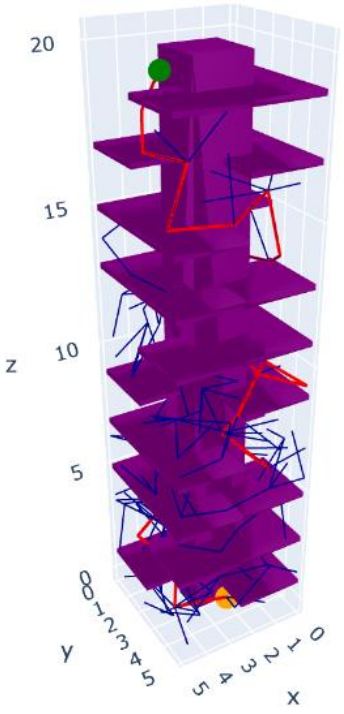

Fig. 9. Tower using RRT

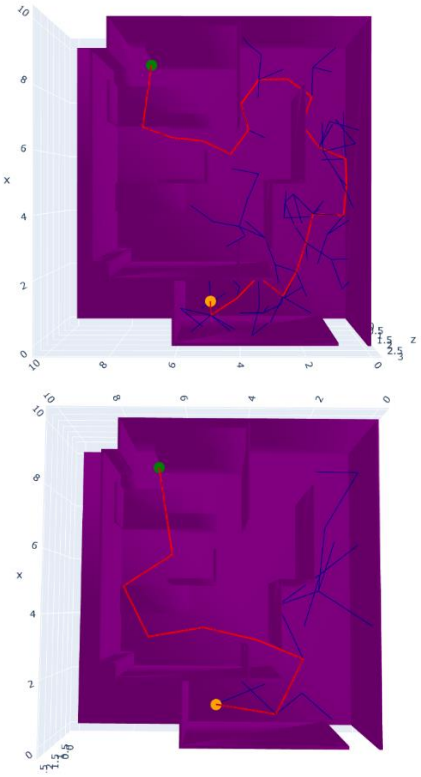

Fig. 10. Room using RRT with different steering distance (Up: sqrt(2) / Down: sqrt(5))

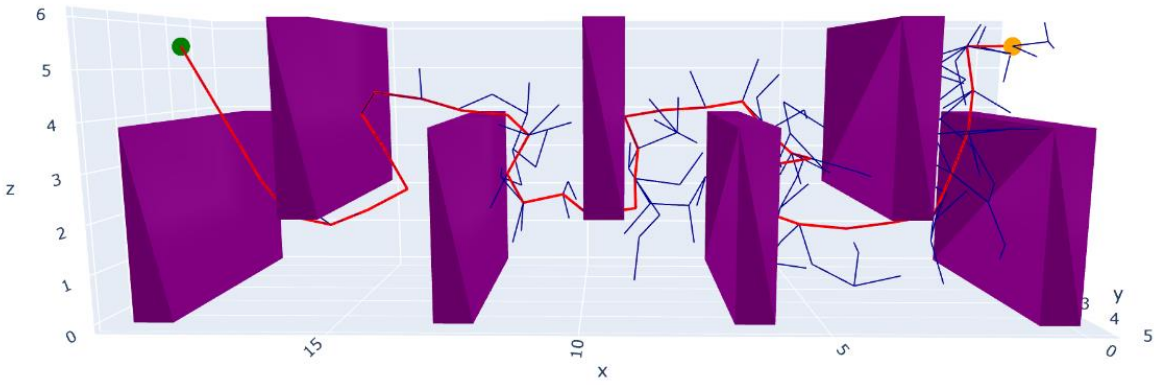

Fig. 11. Flappy Bird using RRT

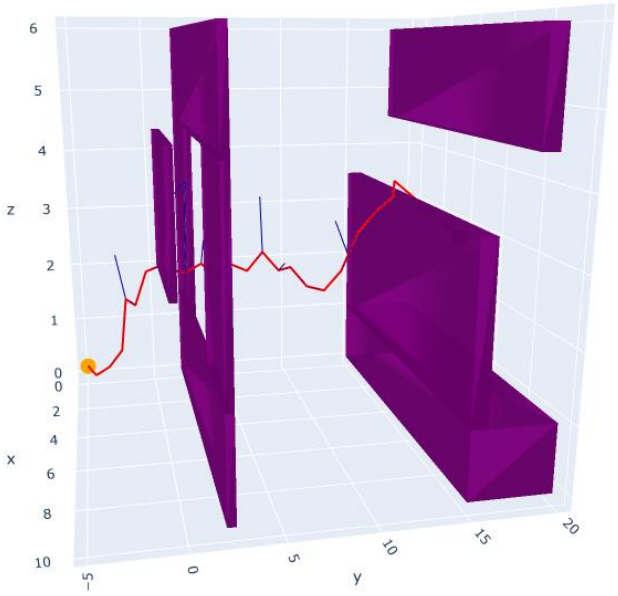

Fig. 12. Window using RRT

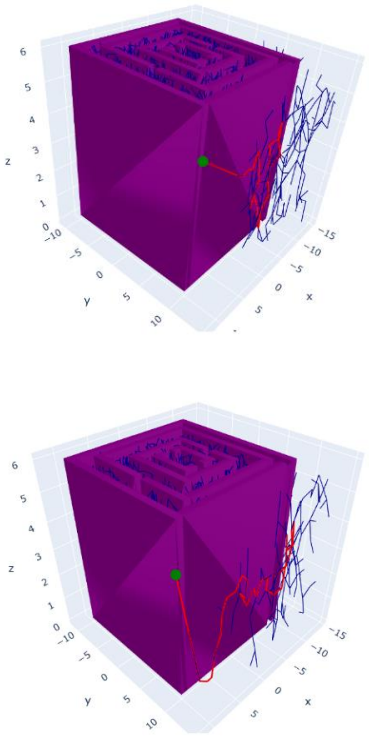

Fig. 13. Maze using RRT with different probability p_g
(Up: 0.1 / Down: 0.5)

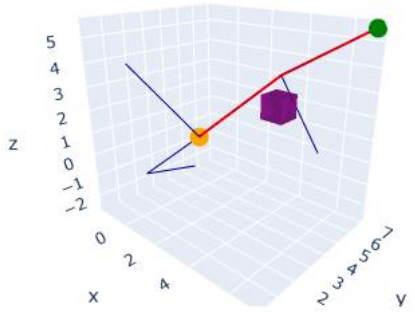

Fig. 14. Single Cube using RRT