Motion Planning in 3-D Environments

Jay Paek
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, California
jpaek@ucsd.edu

***Abstract*—This project delves into the realm of motion planning within 3-D Euclidean spaces, focusing on both search-based and sampling-based algorithms. The objective is to navigate through environments defined by rectangular boundaries and obstacles, moving from a start to a goal position. The project is divided into three parts. In the first part, an algorithm for collision detection between line segments and axis-aligned bounding boxes (AABBs) is implemented, ensuring safe navigation. The second part involves the development of a search-based planning algorithm, weighted A*, to enhance efficiency and navigate through the environment. The final part presents sampling-based algorithm, RRT*. This project aims to enhance understanding and application of motion planning techniques in complex 3-D environments, with a strong emphasis on algorithmic efficiency and robustness in collision detection.**

***Index Terms*—Robotics, dynamic programming, optimal control, searching, path-finding, planning**

I. Introduction

Motion planning in robotics involves determining a feasible path for a robot to move from a start to a goal position while avoiding obstacles. This project focuses on implementing two prominent motion planning algorithms: A-star (A*) and Rapidly-exploring Random Tree Star (RRT*), coupled with collision detection using PyBullet. A* is a widely-used search-based algorithm known for its efficiency and optimality in pathfinding, while RRT* is a sampling-based algorithm that ensures asymptotic optimality, making it suitable for complex and high-dimensional spaces. The integration of PyBullet, a physics simulation engine, facilitates accurate collision detection, which is crucial for the safety and reliability of the planned paths.

To implement A*, the environment is discretized into a grid where each cell represents a potential position for the robot. A* searches for the shortest path by evaluating the cost of moving from the start to the goal while avoiding obstacles. The algorithm uses a priority queue to explore nodes with the lowest cost first, based on a heuristic function that estimates the remaining distance to the goal. The heuristic guides the search towards the goal efficiently. During the search, each node is checked for collisions using PyBullet, ensuring that the path remains valid by avoiding any overlaps with obstacles. This collision detection step is performed by querying PyBullet's physics engine to verify if the path between nodes is free of obstacles, thus ensuring the generated path is safe for the robot to traverse.

By combining A* with PyBullet for collision detection and RRT*, this project demonstrates the practical application of search-based and sampling-based algorithms in real-world scenarios. The use of PyBullet not only enhances the accuracy of collision detection but also simplifies the integration of physics-based validation into the motion planning process. This approach ensures that the generated paths are not only optimal and efficient but also physically feasible and safe, making it a robust solution for motion planning in complex 3-D environments.

II. Problem Statement

For this entire problem, any point or vector is in $\mathbb{R}^3$.

For each map, we are given strings with information of the map boundary, starting point s, end point τ, along with obstacles.

The map boundary information is encoded in a vector format known as axis aligned bounding boxes (AABB) in $\mathbb{R}^9$. The entire environment is ensured to be a rectangular prism. The first three entries denote the coordinates of one corner, and the next three entries denote the opposite corner. s and τ are just given as $\mathbb{R}^3$ vectors within these boundaries.

The information for the obstacles are also encoded in a vector in $\mathbb{R}^9$, and conveniently they are all rectangular blocks. Similar to the map, the first six entries capture the positional information of the obstacles.

The agent begins at the coordinate s and aims to reach some ϵ distance away from τ. The agent is free to move in any direction as long as it does not leave the map bounds or collide with an obstacle.

In essence, we have a deterministic shortest path (DSP) problem at hand. Essentially, given a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, we want to find a sequence $\{i_n\}_1^N \subset \mathcal{V}$ such that $||i_N - \tau|| < \epsilon$ and the cost of the path

$$d(\{i_n\}_1^N) = \sum_{n=1}^{N-1} c_{i_n, i_{n+1}}$$

is minimized. $c_{i,j}$ or c_{ij} denotes the cost to move from node i to j. For the sake of the problem, we will assume that all movement has positive cost i.e. $c_{ij} > 0, \forall i, j \in \mathcal{V}$.

Let $\mathcal{P}$ be all possible finite-length paths, or traversable sequence of nodes, such that the first node s and the last

node is ϵ-close to τ. Then the DSP program is formulated as follows:

$$\underset{\{i_n\}_{n=1}^N \in \mathcal{P}}{\text{argmin}} \quad d(\{i_n\}_{n=1}^N)$$

There is no polynomial time algorithm that solves the DSP problem, so a heuristic algorithm is required. Either we cant Some notable searching algorithms include:

- Depth-first search, breadth-first search
- Dijkstra's algorithm
- Bellman-Ford algorithm
- Jump point search
- A* search

In this project, we will be utilizing the A* search algorithm and the RRT* sampling-based planning algorithm, which will be explained in the next section.

III. Technical Approach

In this section, we will discuss the step-by-step approach to each of the subproblems given at hand. All computational aspect of this project are done with Python.

A. Collision Checking

We want to simulate the real-world, and the agent phasing through an obstacle is not very realistic.

We will be using PyBullet, a physics simulation engine library for Python. After launching the PyBullet simulation engine, we need to add all of the environment information. Thankfully, the collision objects can deal with AABB information, which simplifies the preprocessing step.

Algorithm 1 Collision Check Preparation

Require: Environment and obstacle info in AABB: $S \subset \mathbb{R}^9$
  Configure environment bounds
  Set of obstacles $O \leftarrow \{\}$
  for $s \in S$ **do**
    $cs \leftarrow$ CollisionShape(AABB)
    $vs \leftarrow$ VisualShape(AABB)
    $O \leftarrow O \cup \{\text{Multiobject}(cs, vs)\}$
  end for

After setting up the environment in the physics engine, a simple use of the rayTest function will check whether a line with end points $(start, end)$ will collide with any objects.

B. Search-based Motion Planning

Before presenting the proposed solution to this problem, we will present the idea behind the what is proposed.

The label correcting algorithm is a rudimentary approach to a deterministic short path problem. Essentially, we will maintain a list of nodes that seem to have potential to be part of the shortest path towards the goal. Each node possess a running cost g_i, which describes the cost it takes to get to node i from the starting node. In each iteration step, we will go to the node that has potential to reach the goal and is easiest to attain, and we will expand that node i.e. explore further from that specific point. This process in continued until the goal is reached.

Algorithm 2 Label Correcting Algorithm

Require: $s, \tau, c, g, h, \epsilon$
  OPEN $\leftarrow \{s\}, g_s = 0, g_i = \infty \forall i \in \mathcal{V} - \{s\}$
  while OPEN $\neq \emptyset$ **do**
    Remove i from OPEN
    for $j \in$ Children(i) **do**
      if $g_j, g_\tau > g_i + c_{ij}$ **then**
        $g_j \leftarrow g_i + c_{ij}$,
        Parent$(j) \leftarrow i$
        if $j \in$ OPEN **then**
          OPEN $=$ OPEN $\cup \{j\}$
        end if
      end if
    end for
  end while

The weight A* algorithm is a modification of the label correcting algorithm that efficiently navigates towards the goal through a success metric known as the heuristic function denoted as h. We will let h be the Euclidean distance between the input node and the goal. This will help navigate the agent select the best nodes to expand and nodes that progressively move towards the goal in the environment without significant detours i.e. obstacles that encourage the agent to move in the opposite direction of the goal. We have the parameter α, which weighs the trust for the heuristic function. We can select this parameter depending on the map.

Our version of weighted A* is implemented such that when a node is "expanded", it will check whether the children of a node is invalid i.e. if they collide with an obstacle or are out of bounds. Hence, there must be an initial step to assess the validity before evaluating the traversability of a child node.

Algorithm 3 Weighted A* Algorithm

Require: $s, \tau, c, g, h, \epsilon$
  OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$
  while $\tau \notin$ CLOSED **do**
    $i = \underset{i \in \text{OPEN}}{\text{argmin}}\ g_i + \alpha h_i$
    CLOSED $\leftarrow$ CLOSED $\cup \{i\}$
    for $j \in$ Children$(i), j \notin$ 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 $=$ OPEN $\cup \{j\}$
        end if
      end if
    end for
  end while

Notes on A* implementation:

- To efficiently obtain the most promising node in the OPEN set, we will use a priority queue, where the priority is determined by f_i.
- We will assume that the agent can only travel 0.31-units exclusively in the positive and negative x, y, z direction. We can set the travelling distance to other values. It was noticed that setting the travelling distance to an nice fraction caused the agent to move in the infinitesimally small gaps between blocks. Hence the selection of the distance.
- $h(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2$, and the heuristic constant varies by map.
- $\epsilon = 0.5$, which is the acceptable distance from the goal to be considered "at the goal."
- In the worst-case, the A* can be $\mathcal{O}(b^n)$, where b is the average number of edges from each node, and n is the number of nodes on the resulting path.
- The memory complexity of A* is $\mathcal{O}(b^n)$, since every node needs to be tracked.
- We will not be preemptively creating the graph nor discretize the space prior to the search. Starting at s we will expand the node and attach new vertices based on the children of the expanded node.

C. Part 3: Sampling-based Motion Planning

Apart from the other search-based algorithms, RRT* is a probabilistic approach to the DSP problem. In normal RRT, we will grow a tree from the starting node and begin constructing a space filling curve until the goal is reached. However, RRT* will continuous optimize the path by rewiring nodes that are closer together instead of taking an unnecessary longer path.

We will be using a pre-implemented RRT* algorithm at this repository. We must add the map bounds and obstacles in a similar fashion to PyBullet. Then, after confinguring the start and end point, we can call a function that executes the planning algorithm. This search is probabilistically complete, so it will definitely reach the goal almost surely. The time complexity, however is hard to analyze.

Some notes on the RRT* algorithm:

- SampleFree samples points that do not have an obstacle.
- Nearest(.) returns to closest point to the input.
- Steer(.,.) attempts to construct a path from first input to second input node.
- CollisionFree(.,.) checks if the path from the first to second input does not collide with an obstacle.
- Cost is equivalent to c_{ij}

Algorithm 4 RRT*

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

IV. Results

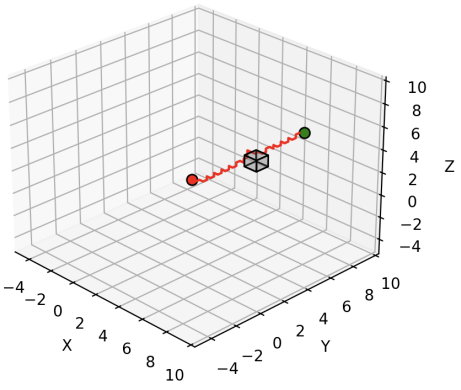
Some observations regarding the results

- A* performed really well in environments where the agent did not need to move in the opposite direction of the goal. Raising the heuristic constant in these settings significantly improved performance.
- A* expanded significantly more nodes in complex environments. In these environments, decreasing the heuristic constant resulted in slightly better performance.
- RRT* did not always necessarily find the most optimal route, but was still relatively efficient.
- RRT* took too long for the maps maze and monza. Most likely due to complex environment (100k+ samples). If paths not provided in the report, their run time information and paths can be found here.
- RRT* parameters needed to be tuned so the path would not phase through walls. Some maps had thinner walls than others.
- Both algorithms tended to trace the edge plane of an obstacles, mainly due to to the heuristics forcing the path to remain as close to the goal as possible.

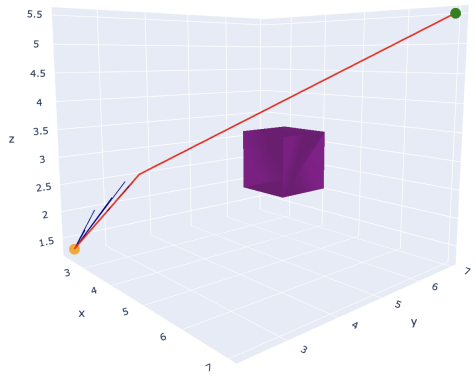
Parameters for each map and algorithm is specified per section.

A. *Single Cube Environment*

heuristic constant: 1, Nodes expanded: 14457, Planning took: 0.8325889110565186 sec, Path length: 13

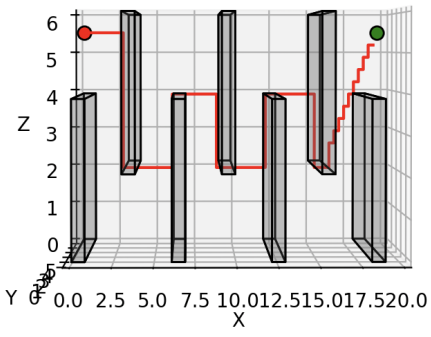


edge length: 0.1, intersection check length: 0.1, path length: 8, runtime: 0.4259, 71 samples.

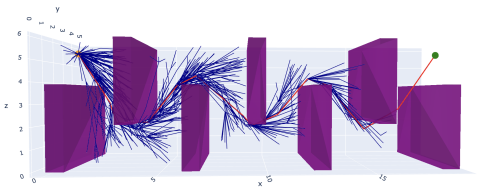


B. *Flappy Bird Environment*

heuristic constant: 1, Nodes expanded: 22478, Planning took: 1.235600233078003 sec, Path length: 32

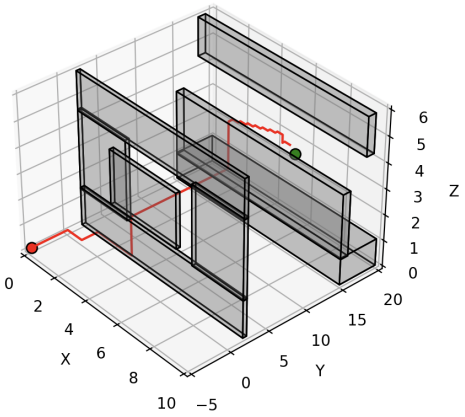


edge length = 0.1, intersection check length = 0.1, path length: 27, runtime: 5.722, 3875 samples

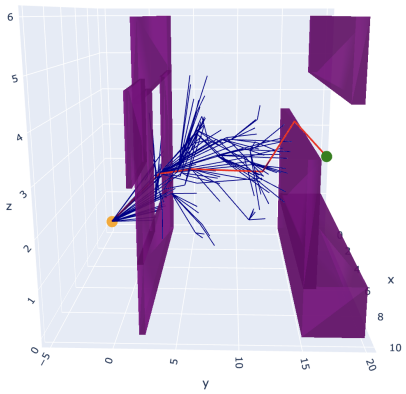


C. *Window Environment*

heuristic constant: 1, Nodes expanded: 31205, Planning took: 1.7627739906311035 sec, Path length: 31



edge length = 0.1, intersection check length = 0.1, path length: 25, runtime: 1.493, 961 samples



D. *Tower Environment*

heuristic constant: 1, Nodes expanded: 31090, Planning took: 1.731515884399414 sec, Path length: 41

edge length = 0.1, intersection check length = 0.1, path length: 29, runtime: 16.174, 9487 samples

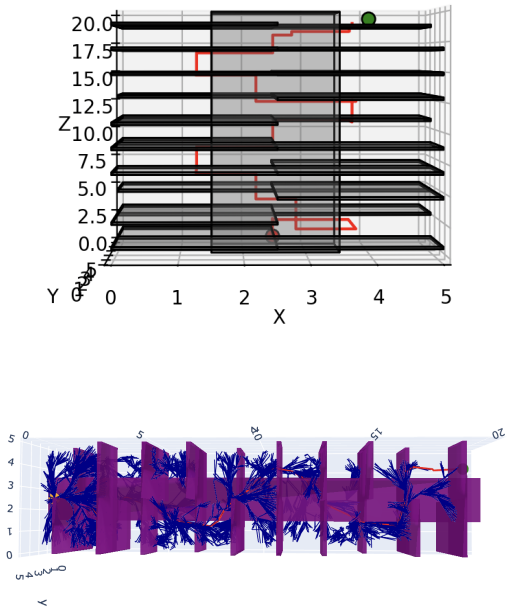


Fig. 1: Tower with A* (above) and RRT* (below)

E. Room Environment

heuristic constant: 1, Nodes expanded: 6102 Planning took: 0.34586501121520996 sec, Path length: 13

edge length = 0.05, intersection check length = 0.01, path length: 16, runtime: 8.136, 4537 samples

F. Maze Environment

heuristic constant: 1, Nodes expanded: 221405, Planning took: 12.49658489227295 sec, Path length: 81

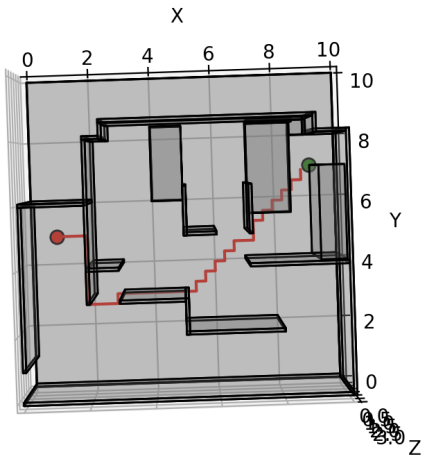


Fig. 2: Room with A*

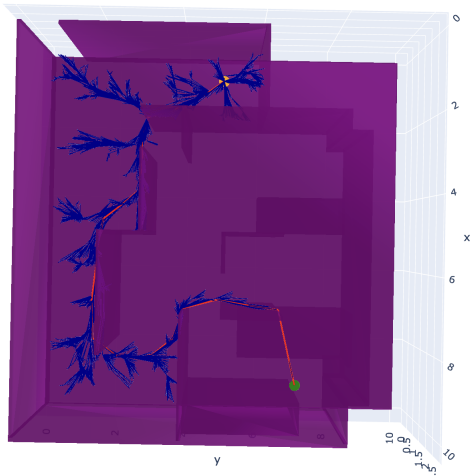


Fig. 3: Room with RRT*

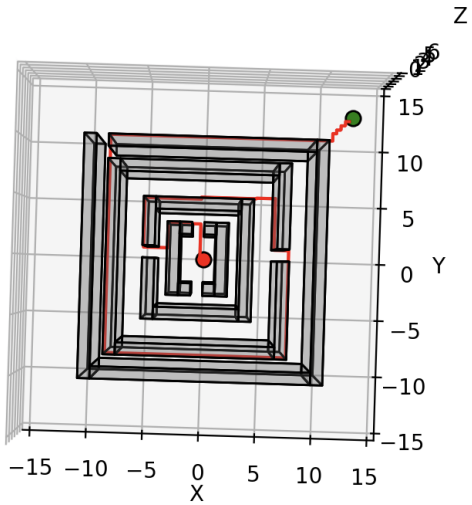


Fig. 4: Maze with A*

G. Monza Environment

heuristic constant: 1, Nodes expanded: 12574, Planning took: 0.6785778999328613 sec, Path length: 81

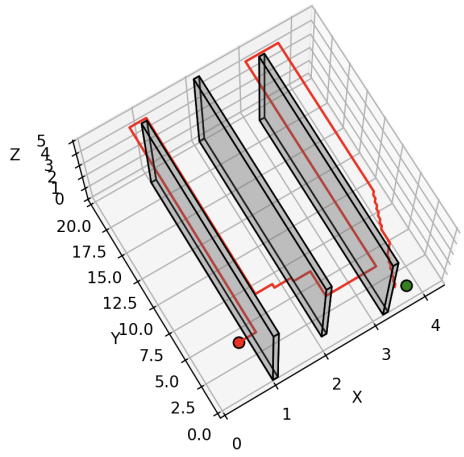


Fig. 5: Monza with A*

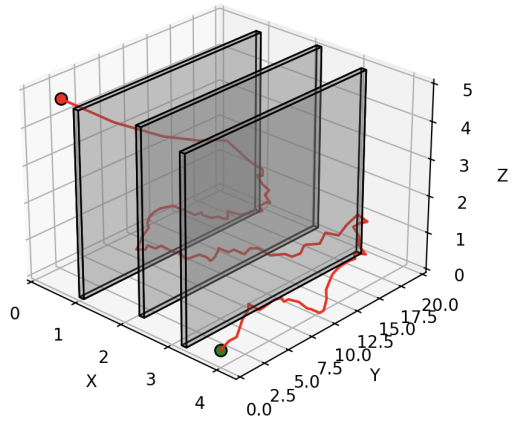


Fig. 6: Monza with RRT*

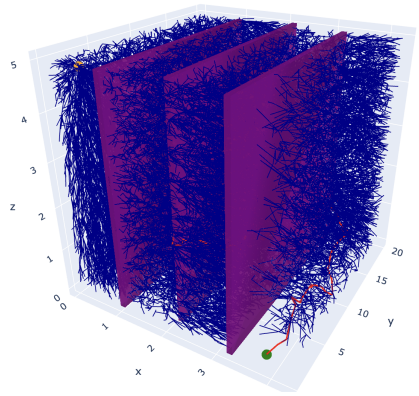


Fig. 7: Sampling trees on Monza with RRT*

```
Checking if can connect to goal at 177375 samples
Can connect to goal
Planning took: 407.7327620983124 sec.

Crash
Success: False
Path length: 75
```

Fig. 8: Computation info on Monza with RRT*