3DPathFinder: Motion Planning in Complex Environments

Yixin Zhang

I. INTRODUCTION

A. Overview of Motion Planning

Motion planning, crucial for autonomous navigation, seeks the shortest path through continuous state and control spaces, navigating around obstacles within a known environment. This problem, also known as the Piano Movers Problem, involves designing trajectories that are not only feasible but also minimize costs related to factors such as distance, time, energy, and risk. The primary goal is to generate global, collision-free trajectories automatically. These paths must connect an initial state to a designated goal region while adhering to environmental constraints (like physical obstacles) and the robot's kinematic and dynamic limitations.

B. Approaches to Motion Planning

Our project explores two main algorithmic approaches for motion planning in three-dimensional spaces across seven distinct environments:

- Our project utilizes Weighted A* as an example of search-based planning algorithms. This method discretizes the state space into a regular grid to construct a graph and tackles the deterministic shortest path (DSP) problem via label correction. While this approach is made efficient with the use of heuristic functions, it faces challenges in high-dimensional spaces due to its regular discretization method. Nonetheless, it is resolution complete and provides finite-time (sub)optimality guarantees, making it a reliable choice under certain conditions.
- In contrast, the Rapidly-exploring Random Tree (RRT) exemplifies our sampling-based planning algorithms. This technique irregularly discretizes the state space by sampling states, which are then used to incrementally construct a graph. It efficiently addresses the DSP problem through label correction and excels in high-dimensional spaces. However, it may struggle with "narrow passages," a common challenge in complex environments. Despite these hurdles, RRT is probabilistically complete and offers asymptotic (sub)optimality guarantees.

Throughout the project, we will conduct a thorough analysis of both algorithms, focusing on how different hyperparameters influence their performance and strengths. This analysis will shed light on each algorithm's suitability across various environmental contexts, aiming to enhance the understanding of their practical applications and limitations in real-world scenarios.

II. PROBLEM FORMULATION

A. Motion Planning Problem Definition

The motion planning problem seeks to determine a feasible path ρ^* from a start state x_s to a goal state x_τ within a defined configuration space C that avoids obstacles. The objective is to minimize a cost function J over all feasible paths $\mathcal{P}_{s,\tau}$, such that $J(\rho^*) = \min_{\rho \in \mathcal{P}_{s,\tau}} J(\rho)$, or to report failure if no such path can be found. This involves navigating through a complex environment delineated by spatial constraints and obstacles, requiring efficient and effective pathfinding algorithms.

- **Configuration Space** C: Defined by a 3-D rectangular boundary with limits $(x_{\min}, y_{\min}, z_{\min}, x_{\max}, y_{\max}, z_{\max})$. This space encompasses all possible states of the system.
- **Obstacle Space** C_{obs}: Comprises several rectangular blocks defined by corners at $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$. These blocks delineate areas to avoid.
- **Free Space** C_{free}: The portion of C not occupied by obstacles, calculated as $C \setminus C_{\text{obs}}$. It represents the areas available for navigation.
- **Start and Goal States**: The start state x_s and goal state x_τ, both located within C_{free}, define the beginning and desired endpoint of the journey.
- **Path**: Defined as a continuous function $\rho : [0,1] \to C$, which maps a normalized parameter to the configuration space.
- **Feasible Path**: A path ρ that remains entirely within C_{free} from x_s to x_τ, constituting the set of all feasible paths $\mathcal{P}_{s,\tau}$.
- **Cost Function** J: Evaluates each path according to criteria such as length, energy, and safety, aiming to identify the most efficient route within the set of feasible paths.

B. DSP Problem Definition

In our project, we employ both Search-based and Sampling-based planning algorithms to incrementally construct a graph, transforming the motion planning problem into a Deterministic Shortest Path (DSP) problem. This problem is tackled using a label correcting algorithm, which is effective in environments without negative cycles.

Consider a graph defined by a vertex set V, an edge set $E \subseteq V \times V$, and edge weights $C := \{c_{ij} \in \mathbb{R} \cup \{\infty\} \mid (i,j) \in E\}$ where c_{ij} denotes the cost of transition from vertex i to vertex j. The assumption of no negative cycles, i.e., $J'_{i_{1:q}} \geq 0$, for all paths $i_{1:q} \in P_{i,j}$ and all vertices $i \in V$, simplifies the

computational complexity. The primary objective is to find the shortest path from a start node s to an end node τ:

$$\text{dist}(s,\tau) = \min_{i_{1:q} \in P_{s,\tau}} J_{i_{1:q}}$$

$$i^*_{1:q} \in \arg\min_{i_{1:q} \in P_{s,\tau}} J_{i_{1:q}}$$

A path is defined as a sequence $i_{1:q} := (i_1, i_2, \ldots, i_q)$ of nodes $i_k \in V$, and the path length is the sum of edge weights along the path: $J_{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k, i_{k+1}}$.

The project details are as below:

- **Vertex Representation:** Each vertex i corresponds to the position of an agent in 3D space (x, y, z).
- **Edge Definition:** The edges are based on the motion model of the agent. In the A* algorithm, edges represent 26 possible movements within a cubic lattice around the agent, where each move is to one of the 26 surrounding lattice points, ensuring no collisions and adherence to boundary conditions. In the RRT algorithm, edges are formulated using the STEER_ε function in conjunction with a COLLISIONFREE condition to guarantee viable paths.
- **Costs c_{ij}:** The cost between two nodes is calculated using the Euclidean distance:

$$c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

- **Start and Goal Nodes:** The start and goal nodes are denoted by s and τ, respectively.

III. TECHNICAL APPROACH

A. Collision Detection

A Line Segment in this context is defined as a straight part of a line bounded by two distinct end points, $p0$ and $p1$. An Axis-Aligned Bounding Box (AABB) is a rectangular box that is aligned with the coordinate axes, making it easy to compute intersections due to its regular shape.

This algorithm checks whether a given line segment intersects with an AABB. It does so by transforming both the segment and the box relative to the origin, simplifying the intersection tests. The algorithm first calculates the center and half-lengths of the AABB, as well as the midpoint and half-length vector of the line segment. By translating these points to the origin, the algorithm then employs the Separating Axis Theorem (SAT) to determine if there is any overlap along the primary axes and cross-product axes, which effectively checks for any separating axis that would indicate no intersection.

The primary advantage of using this algorithm is its efficiency. By leveraging axis alignments and the properties of the SAT, it can quickly conclude non-intersection scenarios without extensive computation. This makes it particularly suitable for applications in computer graphics and collision detection in gaming, where multiple intersection tests are performed frequently and speed is critical.

For evaluating whether a path is collision-free, we need to check each consecutive pair of points along the path against all blocks in the environment. This is done using the 'Check Intersection Between Line Segment and AABB' function. Specifically, we:

- Take each consecutive pair of points in the path.
- Check these points against all blocks in the environment using the 'Check Intersection Between Line Segment and AABB' function.
- If any of these checks return 'True', a collision is detected, indicating that the path is not collision-free.
- If all checks return 'False', then the path is deemed collision-free.

This process provides a valuable tool for verifying whether a path calculated by our algorithms is free of collisions. It not only ensures the overall integrity of the proposed path but also serves as an integral part of our algorithms to validate each individual movement. By systematically check-

Algorithm 1 Check Intersection Between Line Segment and AABB

procedure LINESEGMENTAABB($p0, p1, b$)
  $b_{\min} \leftarrow b[:3]$
  $b_{\max} \leftarrow b[3:]$
  $c \leftarrow (b_{\min} + b_{\max}) * 0.5$ ▷ Calculate center of the box
  $e \leftarrow b_{\max} - c$ ▷ Calculate half-length extents
  $m \leftarrow (p0 + p1) * 0.5$ ▷ Calculate segment midpoint
  $d \leftarrow p1 - m$ ▷ Calculate segment half-length vector
  $m \leftarrow m - c$ ▷ Translate box and segment to origin
  $adx, ady, adz \leftarrow \text{abs}(d)$ ▷ Absolute values of the half-length vector
  EPSILON $\leftarrow 1e-12$
  $adx \leftarrow adx + \text{EPSILON}$
  $ady \leftarrow ady + \text{EPSILON}$
  $adz \leftarrow adz + \text{EPSILON}$
  if $\text{abs}(m[0]) > e[0] + adx$ **then**
    return False
  end if
  if $\text{abs}(m[1]) > e[1] + ady$ **then**
    return False
  end if
  if $\text{abs}(m[2]) > e[2] + adz$ **then**
    return False
  end if
  if $\text{abs}(m[1]*d[2] - m[2]*d[1]) > e[1]*adz + e[2]*ady$ **then**
    return False
  end if
  if $\text{abs}(m[2]*d[0] - m[0]*d[2]) > e[0]*adz + e[2]*adx$ **then**
    return False
  end if
  if $\text{abs}(m[0]*d[1] - m[1]*d[0]) > e[0]*ady + e[1]*adx$ **then**
    return False
  end if
  return True ▷ No separating axis found
end procedure

ing each segment of the path against potential obstacles, we can guarantee that the path is both feasible and safe for navigation. This collision detection method is essential for maintaining the reliability and accuracy of our motion planning algorithms, making it a critical component in the development of robust autonomous navigation systems.

Algorithm 2 Collision Detection

```
function COLLISIONDETECTION(path, blocks)
    for i = 0 to len(path) − 2 do
        pt1 ← path[i], pt2 ← path[i + 1]
        for blk in blocks do
            if LINE_SEGMENT_AABB(pt1, pt2, blk) then
                return True
            end if
        end for
    end for
    return False
end function
```

B. Weighted A* Algorithm

First, we will include the A* algorithm, outlining how we construct the graph, choose the heuristic function, and provide a of the algorithm, along with an analysis of its properties—optimality, completeness, memory, and time efficiency.

1) Motion Model & Graph Construction: Originally, we have two options to construct the graph: one is to discretize the entire configuration space and then build the edges; to save memory and enhance computing efficiency, we opt to build the motion model leveraging an on-the-fly motion model. Since our problem is in 3D space, we aim to build a 26-connected model, visualizations of which are shown in Fig. 1. For each dimension, there are three possible movements: -1, 0, and 1. Combining three dimensions (x, y, and z), we have 27 possible directions, noting that we should also delete the (0,0,0) vector, which indicates no movement. Thus, there are 26 directions. Each direction can be represented as $d_i = \{x_i, y_i, z_i\}$, where x_i, y_i, z_i can be -1, 0, or 1, excluding (0,0,0).

Therefore, the motion model can be written as

$$v_{i+1} = f(v_i, d_i) = v_i + r \cdot d_i,$$

where r is the resolution or step size, and it is a tunable parameter which we will discuss later.

Given the motion model, we can construct the graph on-the-fly using the `GetNeighbours` function. For all possible 26 directions, if a neighbour is within the boundary and the path from the current node to this neighbour has no collisions, it is added to the valid neighbours. The movement step size is defined by r. Consequently, the cost from node v_i to v_{i+1} is calculated as

$$c_{i,i+1} = r \cdot \|d_i\|_2,$$

where $\|d_i\|$ represents the Euclidean norm of the direction vector d_i, which can be 1, $\sqrt{2}$, or $\sqrt{3}$, corresponding to movements along one axis, movements diagonally across a face of the cube, and movements diagonally through the cube, respectively.

Algorithm 3 Get Neighbors

```
function GETNEIGHBOURS(current_node, resolution)
    neighbours ← []
    for k ∈ [0, ..., 25] do  ▷ Iterating over 26 directions
        next ← current_node + d_i × resolution
        if Out of Boundary then
            pass                    ▷ Skip if out of boundary
        end if
        path ← (current_node, next)
        collision ← collisionDetection(path, blocks)
        if collision then
            pass                    ▷ Skip if path is blocked
        end if
        Add next to neighbours ▷ Add valid next node
    end for
    return neighbours
end function
```

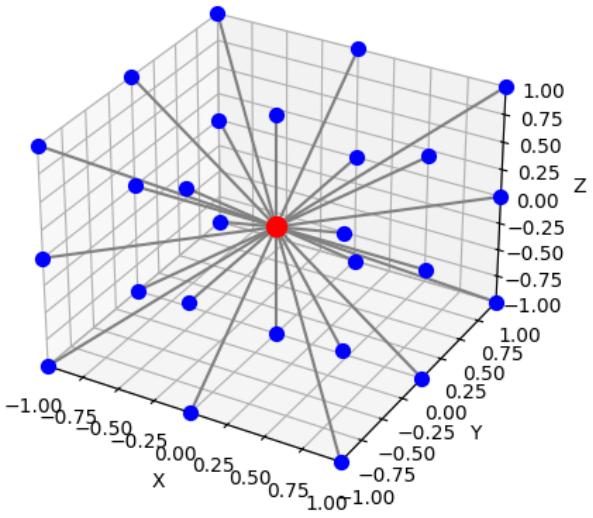

Fig. 1: 26-connected graph: the red center node represents the current node, the blue nodes represent its neighbors, and all connecting edges are depicted as gray lines.

2) Heuristic Function Selection: To ensure our algorithm retrieves an optimal path, the heuristic function must be admissible. If A* uses an admissible but inconsistent heuristic, it is still guaranteed to return an optimal path as long as closed states are reopened. A consistent heuristic is preferable as it eliminates the need to reopen the *CLOSE* list, thus saving significant computational effort.

In this project, we choose the Euclidean distance as our heuristic function:

$$h(v_i) = \|v_i - \tau\|_2 = \sqrt{(x_i - x_\tau)^2 + (y_i - y_\tau)^2 + (z_i - z_\tau)^2}$$

To prove that the Euclidean distance is consistent, we must demonstrate the triangle inequality: Suppose v_i is a node, and v_j is any child of v_i,

$$\begin{aligned} h(v_i) &= \|v_i - \tau\|_2 \\ &= \|v_i - v_j + v_j - \tau\|_2 \\ &\leq \|v_i - v_j\|_2 + \|v_j - \tau\|_2 \\ &= c_{ij} + h(v_j) \end{aligned}$$

Thus, the Euclidean distance is consistent. Since consistency implies admissibility, this heuristic guarantees the identification of an optimal path.

3) Adaptive Step Size r: Because the starting and ending coordinates are precise to one decimal place, and only if the Euclidean distance from the current node to the end node is regarded as having reached the goal, if we choose a step size that is too large, the program will run indefinitely because this type of step size is too large and will never reach the goal. However, if the step size is too small, the program will run much slower because the graph in this case will be much larger, and therefore, more time is needed to search for a path.

Thus, we propose an Adaptive Step Size r, defined by:

$$r = \begin{cases} start\_resolution & \text{if } h(v) > threshold, \\ end\_resolution & \text{if } h(v) \leq threshold \end{cases}$$

Ideally, $end\_resolution \leq start\_resolution$. This strategy is a good trade-off since, if we are far from the goal, we use a larger step size to reach the goal as soon as possible, and as we get closer, we use a smaller step size to ensure accuracy and success in searching for a finer path.

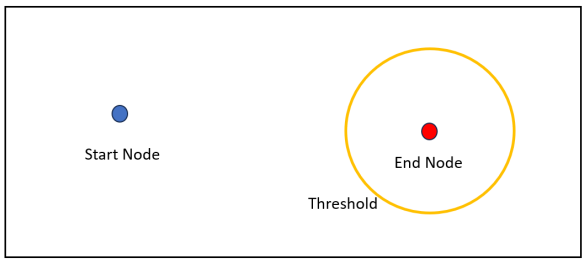

Fig. 2: Adaptive Step Size Visualization, the yellow circle is the threshold

4) Detailed Procedure: If A* uses an admissible but inconsistent heuristic, it is guaranteed to return an optimal path as long as closed states are reopened. However, if we do not reopen the CLOSED states, the algorithm is only guaranteed to return an ε-suboptimal path. In our project, aiming for optimal pathfinding, we choose to reopen the CLOSED states.

The Weighted A* Algorithm as described begins by initializing the OPEN set with the start node s and the CLOSED set as empty, with a flexibility factor $\varepsilon \geq 1$ indicating the weighting of the heuristic. Each node i in the graph, excluding s, starts with an infinite cost g_i, while g_s is set to zero, indicating the start of the pathfinding process. The algorithm proceeds by exploring nodes until the target node τ is in the CLOSED set. Nodes are selected from the OPEN set based on the smallest value of $f_i = g_i + \varepsilon h_i$, where h_i is the heuristic estimate from node i to the goal.

Upon expanding a node i, it is moved to the CLOSED set. An adaptive step size is determined based on whether the heuristic value $h(i)$ exceeds a predefined threshold, selecting between a start or end resolution to balance exploration speed and accuracy. For each neighbor j of i, retrieved through the `GetNeighbours` function with the current resolution, the algorithm checks if a cheaper path to j is found via i. If so, g_j is updated and i is set as j's parent. If j is already in the OPEN set, its priority is updated. If j was previously in the CLOSED set but now has a better path, it is moved back to OPEN, ensuring that the best path will be considered. Nodes not yet explored are simply added to the OPEN set. This method ensures that all possible paths are considered for achieving the optimal route to the target under the constraints of the heuristic's accuracy and computational efficiency.

Algorithm 4 Weighted A* Algorithm

OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$, $\varepsilon \geq 1$
$g_s = 0$, $g_i = \infty$ for all $i \in V \setminus \{s\}$
while $\tau \notin$ CLOSED **do**
    Remove i with smallest $f_i = g_i + \varepsilon h_i$ from OPEN
    Insert i into CLOSED
    if i reaches the goal **then**

    end if
    if $h(i) \geq threshold$ **then** ▷ Adaptive step size logic
        $resolution \leftarrow start\_resolution$
    else
        $resolution \leftarrow end\_resolution$
    end if
    for each j in GETNEIGHBOURS(i, $resolution$) **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 if $j \in$ CLOSED **then**
                Remove j from CLOSED
                OPEN $\leftarrow$ OPEN $\cup \{j\}$
            else ▷ Node is unexplored
                OPEN $\leftarrow$ OPEN $\cup \{j\}$
            end if
        end if
    end for
end while

5) Analysis of properties: In this section, we will discuss our A* algorithms based on optimality, completeness, memory, and time efficiency.

- **Optimality:** Since we reopen the CLOSED set, we

can guarantee that we will find an optimal path. This approach allows us to reconsider nodes if a shorter path to them is discovered, ensuring that the shortest path is always selected.

- **Completeness:** A* algorithm is complete, meaning it will always find a solution if one exists, as long as there are finite nodes and the branching factor is finite. The reopening of the CLOSED set ensures that all possible paths are explored, and none are prematurely discarded.
- **Memory Efficiency:** The memory efficiency of the A* algorithm is primarily influenced by the size of the OPEN and CLOSED sets. Reopening nodes in the CLOSED set can increase memory usage since more nodes might need to be stored at one time. However, this can be mitigated by efficient data structures such as priority queues for the OPEN set and hash sets for the CLOSED set.
- **Time Efficiency:** The time efficiency of A* depends significantly on the heuristic's accuracy. An admissible and consistent heuristic can dramatically reduce the search space, thus improving efficiency. However, reopening the CLOSED set, although ensuring optimality, might lead to increased computation time as more nodes are processed multiple times.

C. RRT Algorithm

In this part of the project, we will not implement our own RRT algorithm. Instead, we will directly utilize the RRT algorithm package available at `https://github.com/motion-planning/rrt-algorithms`. This approach allows us to focus on application-specific aspects of our project while leveraging well-tested and optimized code for RRT implementation.

1) RRT Procedure: The Rapidly-exploring Random Tree (RRT) algorithm is a path planning algorithm designed to efficiently navigate complex spaces by randomly building a tree rooted at the starting point. The algorithm iteratively processes steps until a maximum number of samples, defined by *Max_samples*, are reached. During each iteration, it performs the following actions:

- Generates a random point x_{rand} in the search space.
- Identifies the nearest existing node in the tree (x_{nearest}) to the random point.
- Attempts to create a new node x_{new} by moving from x_{nearest} towards x_{rand}, controlled by a steering function that may consider constraints like maximum edge lengths stored in q.
- Validates that the path from x_{nearest} to x_{new} is free of obstacles using a specified resolution r.
- If the path is valid, x_{new} is added to the vertex set V and the edge from x_{nearest} to x_{new} is added to the edge set E.
- Periodically checks if a valid path to the goal has been established based on a predefined probability *prc*. If a valid path is found, it returns this path immediately.

If no path is found by the time all samples are processed, the function returns *None*, indicating that the path planning was unsuccessful within the given constraints and number of iterations.

Algorithm 5 Rapidly-exploring Random Tree (RRT)

$V \leftarrow \{x_s\}$, $E \leftarrow \emptyset$
for $i = 1 \ldots$ Max_samples **do**
  $x_{\text{rand}} \leftarrow$ SAMPLE_FREE()
  $x_{\text{nearest}} \leftarrow$ NEAREST$((V,E), x_{\text{rand}})$
  $x_{\text{new}} \leftarrow$ STEER$(x_{\text{nearest}}, x_{\text{rand}}, q)$
  if COLLISION_FREE$(x_{\text{nearest}}, x_{\text{new}}, r)$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$
    $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$
  end if
  $success, path \leftarrow$ CHECK_SOLUTION(prc)
  if *success* **then return** *path*
  end if
end for
return None ▷ Return None if no path is found

2) Interface with Our Environment: In this section, we detail how our algorithms interface with the environment:

- **Search Space**: The search space is defined precisely by the environment's boundaries. These boundaries are assigned as follows: $[(x_{\text{low}}, x_{\text{up}}), (y_{\text{low}}, y_{\text{up}}), (z_{\text{low}}, z_{\text{up}})]$.
- **Start and Goal Points**: Points are represented by tuples in the form (x, y, z).
- **Obstacles**: These are axis-aligned rectangles represented by tuples in the form $(x_{\text{low}}, y_{\text{low}}, z_{\text{low}}, x_{\text{up}}, y_{\text{up}}, z_{\text{up}})$.

3) Analysis of properties: In this section, we discuss our RRT algorithms based on optimality, completeness, memory, and time efficiency.

- **Optimality**: RRT is not inherently optimal. Under reasonable assumptions, the probability that RRT converges to an optimal solution as the number of samples approaches infinity is zero.
- **Completeness**: RRT is probabilistically complete; the probability that a feasible path will be found, if one exists, approaches 1 exponentially as the number of samples increases.
- **Memory**: RRT constructs a sparse graph, which requires less memory compared to denser graph representations.
- **Time Efficiency**: Due to its sparse graph nature, RRT generally requires fewer computations, enhancing its time efficiency.

4) RRT Parameter Analysis: The key parameters influencing the RRT algorithm's performance are:

- **Max_samples**: Specifies the maximum number of sampling attempts. To ensure the algorithm is probabilistically complete, **Max_samples** is set to 5,000,000.
- **prc**: The probability of checking for a solution, set at 0.01, to promptly identify a feasible solution.
- **r**: Defines the resolution for collision checking along edges, balancing precision and computational efficiency.
- **q**: Records the lengths of edges added to the tree, crucial

for assessing the spatial distribution and connectivity within the generated tree.

These parameters are set to optimize the balance between the algorithm's completeness, efficiency, and practical applicability in diverse scenarios.

IV. Results

We are implementing the A* and RRT algorithms across seven distinct environments: Single Cube, Maze, Flappy Bird, Monza, Window, Tower, and Room. We will compare their correctness and effectiveness in each environment to evaluate how well these algorithms perform under different conditions.

A. Qualitative Results Analysis

The plots display the configuration of each environment, highlighting obstacles in gray, the start position in red, and the goal position in green.

From these visualizations, it is evident that the paths generated by the Rapidly-exploring Random Tree (RRT) algorithm tend to be more erratic and lengthier compared to those produced by the A* algorithm. This discrepancy is particularly noticeable in environments rich with obstacles, such as Monza and Maze. In these scenarios, RRT exhibits numerous unnecessary movements and appears to become ensnared in local minima, struggling to find efficient pathways through complex obstacle layouts. Conversely, the paths determined by the A* algorithm are noticeably smoother and more systematic. This method consistently leads to more direct and coherent movements, efficiently navigating through the environments by effectively utilizing its heuristic to gauge and minimize the total path cost from the start to the goal. This strategic approach allows A* to avoid obstacles more adeptly and achieve a shorter path to the target.

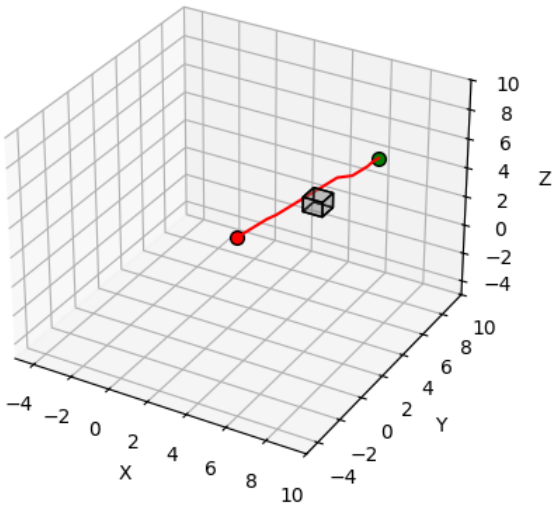

Fig. 3: Single Cube: A*

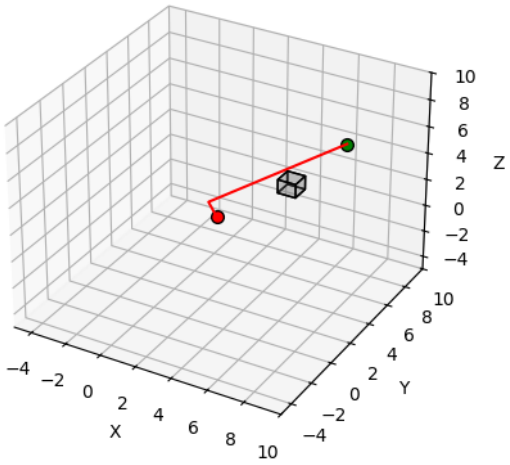

Fig. 4: Single Cube: RRT

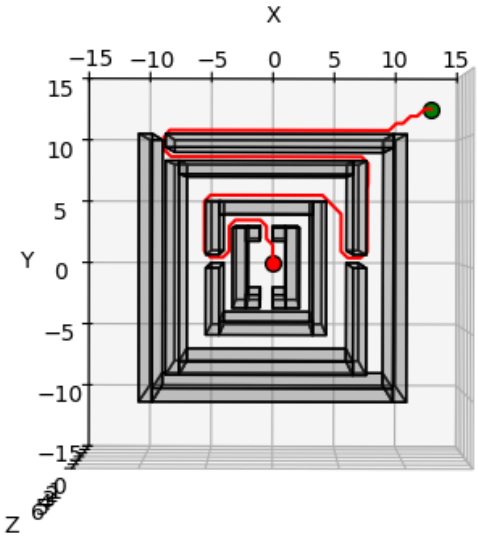

Fig. 5: Maze: A*

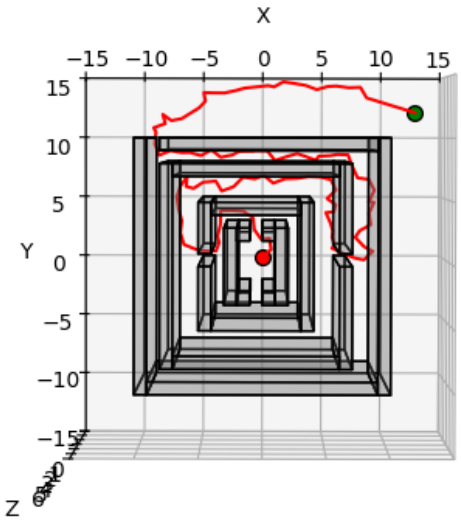

Fig. 6: Maze: RRT

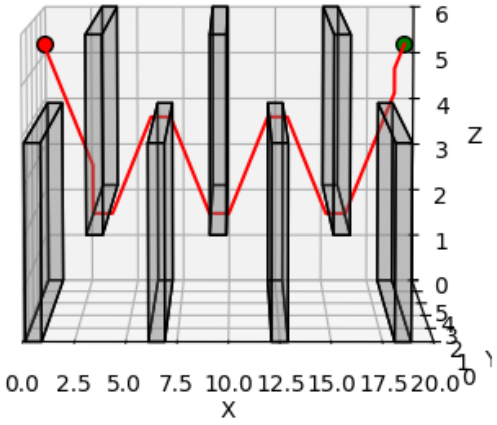

Fig. 7: Flappy Bird: A*

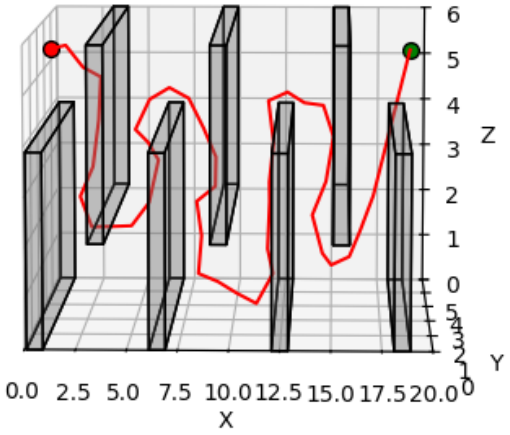

Fig. 8: Flappy Bird: RRT

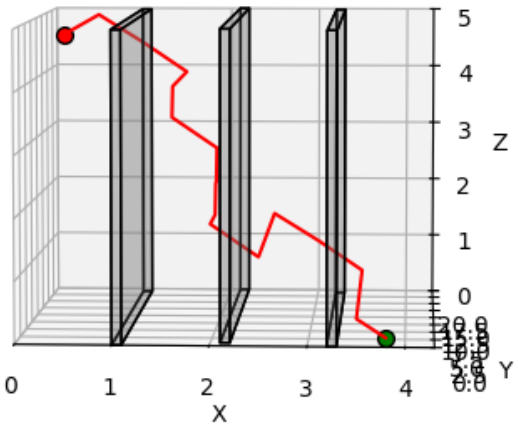

Fig. 9: Monza: A*

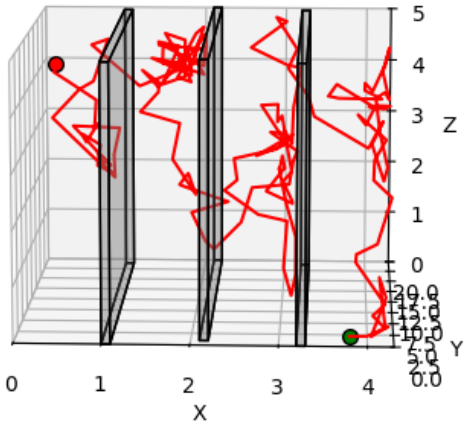

Fig. 10: Monza: RRT

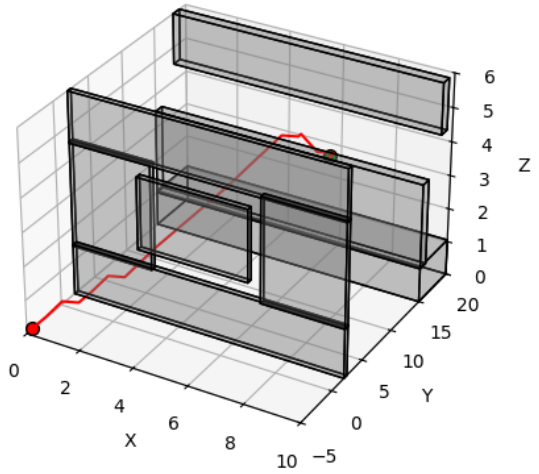

Fig. 11: Window: A*

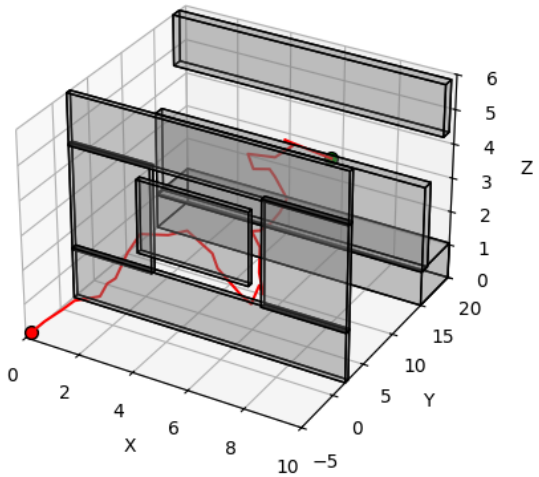

Fig. 12: Window: RRT

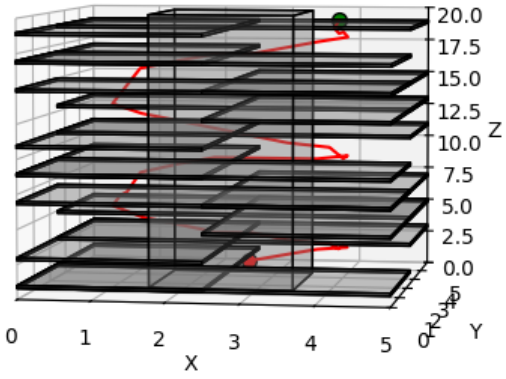

Fig. 13: Tower: A*

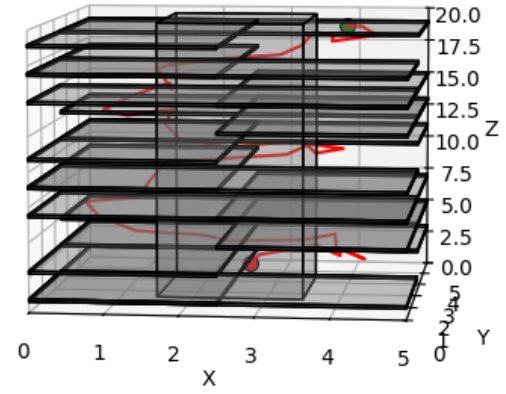

Fig. 14: Tower: RRT

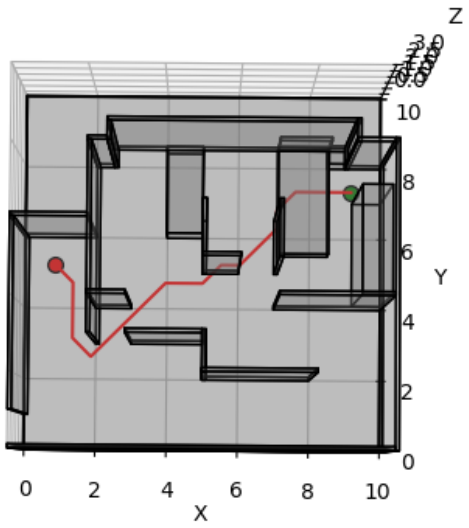

Fig. 15: Room: A*

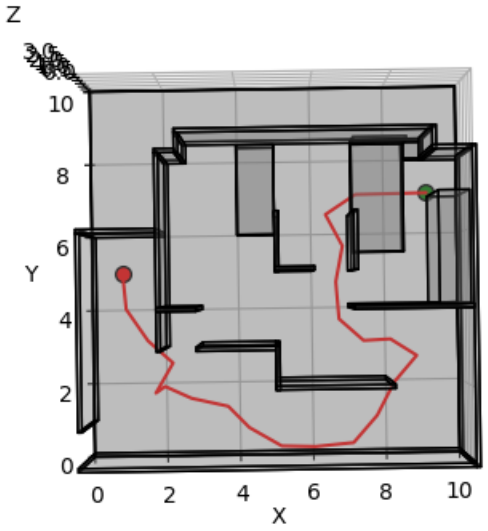

Fig. 16: Room: RRT

B. *Quantitative Results Analysis*

We are implementing the A* and RRT algorithms across seven distinct environments: Single Cube, Maze, Flappy Bird, Monza, Window, Tower, and Room. We will compare their correctness and effectiveness in each environment to evaluate how well these algorithms perform under different conditions.

In our A* algorithm, we set start resolution to be 0.5 and the end resolution to be 0.1. Based on the table displaying A* algorithm performance metrics across various environments with different epsilon values, it's evident that the parameter adjustments significantly impact the algorithm's efficiency and effectiveness. As epsilon increases from 1 to 10, there is a notable decrease in the number of expanded nodes across nearly all environments, suggesting an increase in search efficiency. For instance, in the maze environment, the expanded nodes decrease from 9018 at epsilon = 1 to 6568 at epsilon = 10, indicating less computational overhead. However, this efficiency sometimes comes at the cost of slight increases in path length, as seen in the same maze environment where path length increases marginally from 79 to 81. The planning time consistently decreases with higher epsilon values, reflecting faster decision-making despite the less optimal paths. For example, in the flappy bird environment, time reduces dramatically from 3.965 seconds to 0.550 seconds as epsilon changes from 1 to 10. This trend suggests that while higher epsilon values can reduce computational load and speed up the algorithm, they may also lead to less optimal pathfinding results, particularly in complex or cluttered environments.

In our Rapidly-exploring Random Tree (RRT) algorithm, we set the following parameters:

- MAX_sample: The maximum number of samples, set to 5×10^6.
- prt: The probability of checking whether a solution exists, set to 0.01. This ensures that the solution is

TABLE I: Detailed A* Performance Metrics for Various Environments

ε	Env	Exp. Nodes	Path Len.	Time (s)
1	single cube	125	8	0.065
	maze	9018	79	38.10
	flappy bird	3456	25	3.965
	monza	3146	78	3.485
	window	6376	26	12.529
	tower	2441	32	6.800
	room	378	12	1.198
2	single cube	12	8	0.009
	maze	6938	80	30.417
	flappy bird	739	26	1.024
	monza	2476	78	1.449
	window	65	27	0.109
	tower	673	35	2.311
	room	127	12	0.499
10	single cube	12	8	0.007
	maze	6568	81	16.369
	flappy bird	501	31	0.550
	monza	2476	78	1.515
	window	49	26	0.0921
	tower	227	39	0.784
	room	88	12	0.320

TABLE II: Detailed RRT Performance Metrics for Various Environments

q	Env	Path Len.	Time (s)
1	single cube	10	0.080
	maze	121	15.79
	flappy bird	47	0.973
	monza	107	28.122
	window	30	0.445
	tower	48	1.219
	room	25	0.779
0.1	single cube	9	0.172
	maze	116	228.206
	flappy bird	38	1.596
	monza	103	324.212
	window	28	1.189
	tower	40	5.294
	room	19	0.6283
0.01	single cube	8	0.048
	maze	113	303.283
	flappy bird	34	1.934
	monza	106	844.996
	window	29	1.980
	tower	41	8.167
	room	22	1.198

checked in a timely manner.

- r: The resolution of points to sample along an edge when checking for collisions, set to 0.01. This ensures that the path is collision-free.
- q: A list of lengths of edges added to the tree, which we will analyze in terms of performance.

In various environments, a smaller q value typically leads to shorter paths. However, this is often accompanied by increased computation times, especially in complex environments. This suggests that when choosing q values, there needs to be a balance between path optimization and computational efficiency. In simple environments, smaller q values may be appropriate as they achieve shorter paths at a smaller time cost. However, in complex environments, excessively small q values may lead to significant increases in computational overhead.

C. Comparison Between Weighted A Algorithm and RRT Algorithm*

Based on the detailed analysis of the performance metrics, it is evident that the Weighted A* algorithm generally outperforms the RRT algorithm in terms of path length and execution time across various environments. The Weighted A* consistently finds shorter paths and exhibits a predictable behavior with respect to the parameter ε. As ε increases, the execution time decreases significantly while maintaining relatively stable path lengths. This makes Weighted A* particularly efficient in structured environments like mazes, where it also expands fewer nodes at higher ε values. On the other hand, the RRT algorithm produces longer paths and demonstrates high variability in performance based on the parameter q. Lower q values lead to longer execution times, especially noticeable in complex environments like "monza". Despite this, RRT can still be effective in less structured environments, but its performance is less predictable compared to Weighted A*. Overall, for scenarios requiring consistency and efficiency in path planning, the Weighted A* algorithm is more suitable, while RRT might be preferable in dynamically changing or unstructured environments where exploration is more critical.

V. CONCLUSIONS

Throughout our project, we explored two key algorithmic approaches for motion planning in three-dimensional spaces: Weighted A* and Rapidly-exploring Random Tree (RRT). Our analysis across seven environments revealed that Weighted A* offers robustness through resolution completeness and finite-time (sub)optimality guarantees, making it effective in grid-discretized state spaces despite challenges in high-dimensional settings. In contrast, RRT excels in high-dimensional spaces with its sampling-based approach but struggles with narrow passages, highlighting the need for careful consideration of environmental complexity.

Our comprehensive evaluation of hyperparameters for both algorithms provided valuable insights into their performance dynamics. This understanding is crucial for optimizing their real-world applications and informs the development of more adaptive motion planning strategies. By leveraging the strengths and addressing the weaknesses of both Weighted A* and RRT, we can enhance autonomous navigation systems' ability to generate collision-free, optimal trajectories. Future research may focus on hybrid approaches that combine the deterministic accuracy of search-based methods with the flexibility of sampling-based techniques for further advancements in motion planning technology.

ACKNOWLEDGMENT

The author would also like to thank for the constructive advice from Professor Nikolay Atanasov, all teaching assistants, and everyone who shared their ideas on Piazza and during office hours.