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

Allen Zeng
UCSD
azeng@ucsd.edu

Abstract

This report compares the performance of search-based and sampling-based motion planning algorithms in 3D Euclidean space. Specifically, the author's implementation of a Weighted A* algorithm is analyzed and compared against an open source implementation of RRT*. These algorithms are tested on 7 bounded 3D environments that contain axis-aligned box obstacles.

1 Introduction

Motion planning for robotics is a sub-field of machine learning in which intelligent agents seek an optimal path through their environment. Practically, this involves avoiding obstacles while also minimizing the agent's costs or maximizing rewards. While this is not always the case, in our application the agent assumes full knowledge of its 3D environment including the starting position, ending position, and potential obstacles. There are two primary approaches to this problem: search-based motion planning and sampling-based motion planning.

In search-based motion planning, we can discretize the continuous 3D environments into 26-connected 3D grids and model our problem as the Deterministic Shortest Path Problem [1]. Then, we solve this problem using the Weighted A* algorithm. For sampling-based motion planning, we use an implementation of RRT* [2, 3] to find the shortest path through the continuous environment.

2 Problem Formulation

The motion planning problem is defined as follows, using the notation of [2]. For a space $S \in R^3$, we are given the free space $S_{free} \in S$ and obstacle space $S_{obs} \in S$ where $S_{free} \cap S_{obs} = \emptyset$. The agent starts at state $x_s \in S_{free}$ and has a goal state $x_\tau \in S_{free}$. Let a path be a continuous function $\rho : [0,1] \to S$, and the set of all paths $\mathcal{P}$. Let a feasible path be a continuous function $\rho : [0,1] \to S_{free}$ such that $\rho(0) = x_s$ and $\rho(1) = x_\tau$, and the set of all feasible paths $\mathcal{P}_{s,\tau}$. Given (S_{free}, x_s, x_τ) and a cost function $J : \mathcal{P} \to \mathbb{R}_{\geq 0}$, the path planning problem seeks a feasible path ρ^* such that:

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

and failure is reported if no such optimal path exists. Specifically for our planning problem in R^3, we choose the cost J as the total Euclidean distance traveled along the path. Assuming the agent moves straight along vectors $v \in R^3$ from $\rho(0)$ to $\rho(1)$ over n segments:

$$J = \sum_{i=1}^{n} ||v_i||_2$$

Additionally for the search-based motion planning, we approach the problem using a deterministic shortest path formulation. We discretize S into a 26-connected 3D grid with resolution r. There are no negative cycles in this graph as all edge costs are the Euclidean distances between adjacent grid cells. The overall objective is the same as above, except now the agent's movements $v \in D$ are restricted to

$$D := \left\{ r \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} \middle| \, a, b, c \in \{-1, 0, 1\} \right\} \setminus \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\}$$

in addition to the general requirement that a feasible path does not cross through S_{obs}. Let x'_s, x'_τ be the states on the grid closest to x_s, x_τ respectively. For the set of states on the grid corresponding to free space, $G = \{x_1, x_2, ...\} \in S_{free}$,

$$x'_s = \text{argmin}_{x \in G} ||x - x_s||_2$$
$$x'_\tau = \text{argmin}_{x \in G} ||x - x_\tau||_2$$

These grid-aligned start and goal states serve as the start and goal of the deterministic shortest path formulation.

3 Technical Approach

We approach the motion planning problem through a search-based approach and a sampling-based approach. For the search-based approach, we discretize the continuous space into a 26-connected 3D grid and use weighted A* to solve the resulting deterministic shortest path problem. For the sampling-based approach, we use an open source implementation of *3D Heuristic Bidirectional RRT** [3].

3.1 Search-based Motion Planning

The space S is discretized into a 26-connected 3D grid with resolution $r = 0.2$. This resolution was chosen based on prior knowledge of the obstacle sizes in the given environments. Prior to running weighted A*, we find all of the grid cells that correspond to S_{free}. Practically, these are all cells which fall within the map boundary and are outside of the block obstacles. We know that our boundary and all of our block obstacles are axis-aligned boxes. So, we define a collision-checking function that tests whether a line segment, which connects two adjacent grid cells, intersects with some axis-aligned bounding box. This collision check is necessary for the case that a thin obstacle with width less than resolution r falls between two grid cells. We do not want the agent to plan a path that jumps through walls. We also find the closest x'_s, x'_τ on the grid closest to x_s, x_τ respectively. With this information in place, we run weighted A* on the grid-based graph.

Algorithm 2 Weighted A* Algorithm

```
1:  OPEN ← {s}, CLOSED ← {}, $\epsilon \geq 1$
2:  $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
3:  while $\tau \notin$ CLOSED do                                ▷ $\tau$ not expanded yet
4:      Remove $i$ with smallest $f_i := g_i + \epsilon h_i$ from OPEN       ▷ means $g_i + \epsilon h_i < g_\tau$
5:      Insert $i$ into CLOSED
6:      for $j \in$ Children($i$) and $j \notin$ CLOSED do   ⎫
7:          if $g_j > (g_i + c_{ij})$ then                   ⎪
8:              $g_j \leftarrow (g_i + c_{ij})$              ⎪  expand state $i$:
9:              Parent($j$) ← $i$                            ⎬  ∘ try to decrease $g_j$ using path from $s$ to $i$
10:             if $j \in$ OPEN then                         ⎪
11:                 Update priority of $j$                   ⎪
12:             else                                         ⎪
13:                 OPEN ← OPEN ∪{$j$}                       ⎭
```

Weighted A* Algorithm, reproduced from [1].

The weighted A* algorithm from [1] was implemented as part of the search-based planning, and it is reproduced in Algorithm 2. In our implementation, the function call `Children`(i) is computed on-the-fly. For each node, there are initially 26 possible children, corresponding to directions specified in D. We first filter out these candidate child nodes by requiring `Children`$(i) \in S_{free}$. Then we use the collision-checking function to ensure that there is no obstacle between the current node i and remaining candidate child nodes. Passing candidate nodes are returned as the required list of child nodes. Note that it is important to not pre-compute the value of `Children`(i) for all states $i \in \mathcal{V}$, as this would defeat the purpose of A* not needing to expand every node on the graph. The majority of the computation cost is due to the collision checking function, as the rest of the operations in Algorithm 2 are efficient. Since the function `Children` is only called as needed, this allows our implemented algorithm to scale to larger search spaces as long as a good heuristic function is chosen and not many nodes must be expanded during search.

In our implementation, we choose the Euclidean norm from the queried vertex to the goal as the heuristic function. Edge costs are defined as the Euclidean norm between the current vertex and the (adjacent) child vertex.

For the `OPEN` map, we use a priority queue and therefore have a weighted A* time complexity of $O(|\mathcal{V}|^2)$ [1]. Since our graph is based on a 26-connected grid, the `Children` function is a constant coefficient to the complexity and does not change the $O(|\mathcal{V}|^2)$. Practically however, as shown in the Results (Section 4) less expanded nodes means less calls to the `Children` function and so the algorithm runtime decreases significantly. Also shown in [1], A* and weighted A* have a probably minimum number of expansions $O(|\mathcal{V}|)$ to find the optimal solution. As noted in [1], weighted A* is ϵ-suboptimal ($g_\tau \leq \epsilon \textbf{dist}(s, \tau)$) but trades optimality for speed. The memory requirements of weighted A* are generally better than A*, as typically less nodes are expanded, but not guaranteed. We test weighted A* for ϵ values $\{1, 2, 3\}$. Note that when $\epsilon = 1$, the algorithm is equivalent to the unweighted A* algorithm. The algorithm is complete on a finite graph [1].

After the weighted A* algorithm terminates, we start from the grid goal state x'_τ and lookup `Parent`(x_i) repeatedly until we reach reach the grid starting state x'_s. This sequences of states is the shortest feasible path through the graph. In the case that the grid start and goal location do not match up with the continuous-space start and goal location, we prepend the start x_s and append the goal x_τ to that path. This is so we can make fair comparisons of total path length with the sampling-based planning algorithm.

3.2 Sampling-based Motion Planning

For the sampling-based motion planning algorithm, we chose to use from the *3D Heuristic Bidirectional RRT** the `rrt` library [3]. This implementation contains a method that lazily attempts to shorten the current best path, in addition to the baseline bidirectional RRT* algorithm. Compared to regular RRT, RRT* rewires the exploring random tree to ensure asymptotic optimality. Furthermore this bidirectional implementation grows a tree at both the start node and goal node, which often has better perforamnce in practice [2]. The RRT family of algorithms are probabilistically complete. Meaning that if a feasible path exists, the probability of finding it approaches 1 exponentially as the number of samples approaches infinity.

In using the 3D Heuristic Bidirectional RRT* algorithm, we set the boundary conditions and block obstacles as given by the maps. We alternate between paths of length 1 and 0.1 so that the tree can explore quickly over large empty regions while still being able to maneuver through small gaps. We set parameter $r = 0.05$ to be less than the miniumum dimensions of the block obstacles, so that sampled vertices do not connect through the thin obstacles. We use as many samples as is necessary to converge, the probability of checking for a connection to the goal is 0.01, and the number of nearby branches to rewire is set to 32.

4 Results

The weighted A* algorithm was tested at three different values of $\epsilon \in \{1, 2, 3\}$. Note that when $\epsilon = 1$, the algorithm is equivalent to the unweighted A* algorithm. Weighted A* is ϵ-suboptimal but typically runs faster than unweighted A* [1]. As shown in Table 1, the total path lengths for each environment increases slightly as ϵ increases, but not significantly. Shown in Table 2, higher ϵ values generally lead to faster runtimes. And shown in Table 3, the number of explored nodes is correspondingly lower for higher ϵ. For these negligible increases in path length, the runtimes are greatly reduced for larger values of ϵ in weighted A*.

The weighted A* and bidirectional RRT* algorithms have similar output path lengths. And in this case due to the limited resolution of the grid for weighted A*, RRT* is able to find shorter paths through the continuous environment. The bidirectional RRT* planner finds shorter paths for the simpler Cube, Window, and Monza environments. For the more complex environments, such as Maze, weighted A* gives better results. Shown in Table 2, bidirectional RRT* tends to have faster runtimes than ($\epsilon = 1$) weighted A* in open environments without obstacles that form small gaps. Bidirectional RRT* is slower in Monza and Maze, both of which requires finding narrow openings through the environment. However, bidirectional RRT* is much slower than weighted A* for ϵ values 2 and 3 in all cases. Bidirectional RRT* is able to find shorter paths than weighted A* (performed over a grid) since RRT* works in continuous space, but weighted A* seems to be the much better option if a slightly sub-optimal path is acceptable.

For Table 3, the exact counts for the bidirection RRT* algorithm may not offer much insight, since the algorithm's trees explore randomly and the chance of checking for a feasible path connection is also random. The listed values are from the final run corresponding to the Figures 8-14. However, during debugging and testing, I found that the number of samples varied greatly for the Monza and Maze environments. I believe this is due to the narrow openings between the obstacles in both of these environments. The algorithm may get lucky in its sampling and the number of samples is low on termination, or the algorithm gets unlucky and an order-of-magnitude more samples are needed to find a feasible path. Correspondingly, there was large variation in the algorithm runtimes for those environments.

Under Table 1, bidrectional RRT* has the shortest path for the Monza environment. But looking at Figure 11, it is clear that RRT* could find even shorter paths if there were more samples or if path smoothing [2] were performed. However, bidirectional RRT* already has the slowest runtime on Monza by a wide margin. It is possible that weighted A* may find comparably optimal path lengths if we use a finer grid resolution or if we use a ¿26-connected grid. Path smoothing could also be applied to the weighted A* outputs to find more optimal paths.

As mentioned in the Technical Approach, much of the computational cost in my implementation of weighted A* comes from the collision-checking function. I implemented a naive approach which tests each line segment against all possible axis-aligned bounding box obstacles. A better approach would be to use techniques that subdivide the obstacles, such as a bounding volume hierarchy, which offer logarithmic time complexity in terms of obstacles for checking these intersections. This would allow the algorithm to scale to environments with many obstacles.

Environment	w-A*, $\epsilon = 1$	w-A*, $\epsilon = 2$	w-A*, $\epsilon = 3$	Bi-RRT*
Cube	8.5327	8.5327	8.5327	8.1297
Flappy Bird	25.6307	25.8650	26.5964	27.6605
Window	26.7737	27.0258	26.9087	24.3553
Monza	76.5102	76.6452	76.5681	76.0312
Tower	28.2129	29.2866	30.6331	31.3623
Room	11.6710	12.3053	13.3195	12.2010
Maze	74.4880	74.6229	74.7881	75.2647

Table 1: Path lengths for each environment and algorithm.

Environment	w-A*, $\epsilon = 1$	w-A*, $\epsilon = 2$	w-A*, $\epsilon = 3$	Bi-RRT*
Cube	0.4655s	0.1102s	0.1027s	1.4898s
Flappy Bird	36.4943s	5.0930s	4.9546s	12.1748s
Window	39.6110s	0.2241s	0.1894s	7.7756s
Monza	19.7290s	14.9757s	15.4446s	74.9659s
Tower	48.6257s	5.9098s	6.6708s	22.9510s
Room	5.2219s	0.8717s	1.0212s	2.0890s
Maze	286.0447s	194.3470s	76.5032s	438.4345s

Table 2: Runtimes for each environment and algorithm.

Environment	w-A*, $\epsilon = 1$	w-A*, $\epsilon = 2$	w-A*, $\epsilon = 3$	Bi-RRT*
Cube	921	26	26	420
Flappy Bird	41104	6637	5882	1218
Window	55580	165	116	689
Monza	47600	37283	37178	4355
Tower	32895	3242	3773	2523
Room	2882	526	534	173
Maze	178541	144630	143179	37889

Table 3: For each environment, the number of nodes in the closed set for A* and the number of samples for Bidirectional RRT*.

4.1 Weighted A* Figures

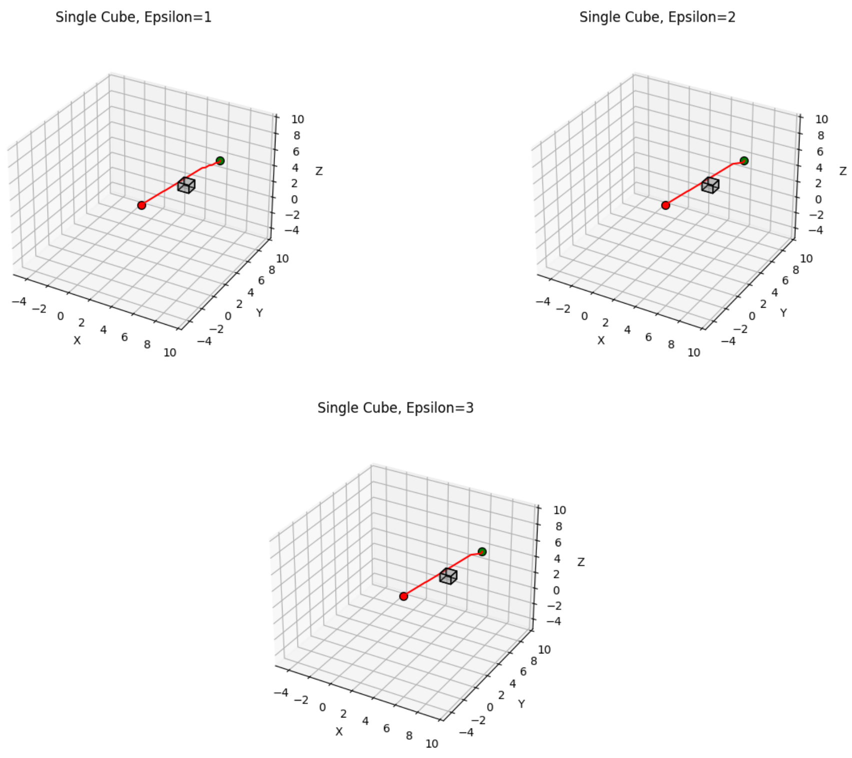


Figure 1: **Cube** environment. Path lengths from low to high epsilon: 8.5327, 8.5327, 8.5327. Runtimes from low to high epsilon: 0.4655s, 0.1102s, 0.1027s.

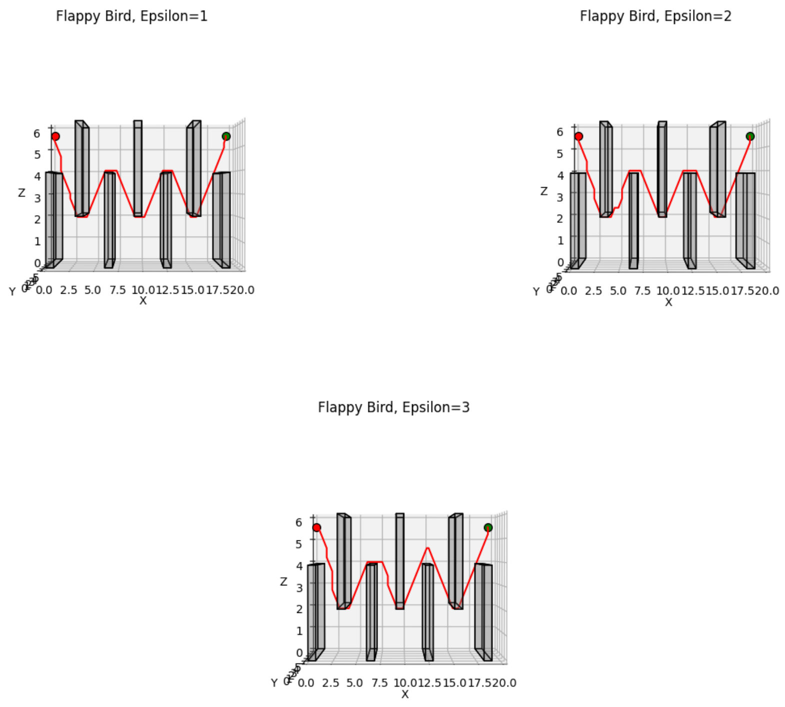


Figure 2: **Flappy Bird** environment. Path lengths from low to high epsilon: 25.6307, 25.8650, 26.5964. Runtimes from low to high epsilon: 36.4943s, 5.0930s, 4.9546s.

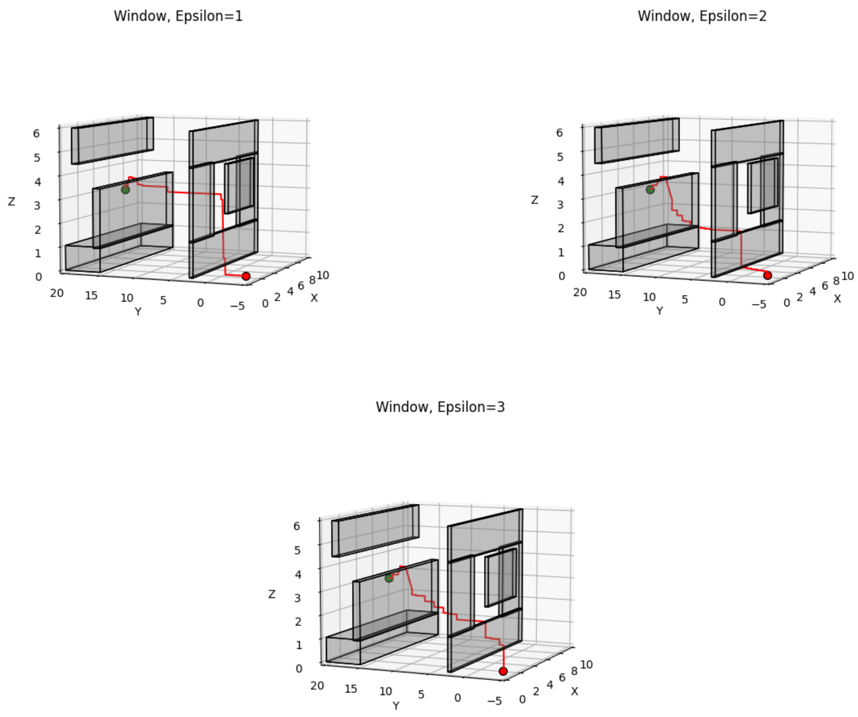


Figure 3: **Window** environment. Path lengths from low to high epsilon: 26.7737, 27.0258, 26.9087. Runtimes from low to high epsilon: 26.7737s, 0.2241s, 0.1894s.

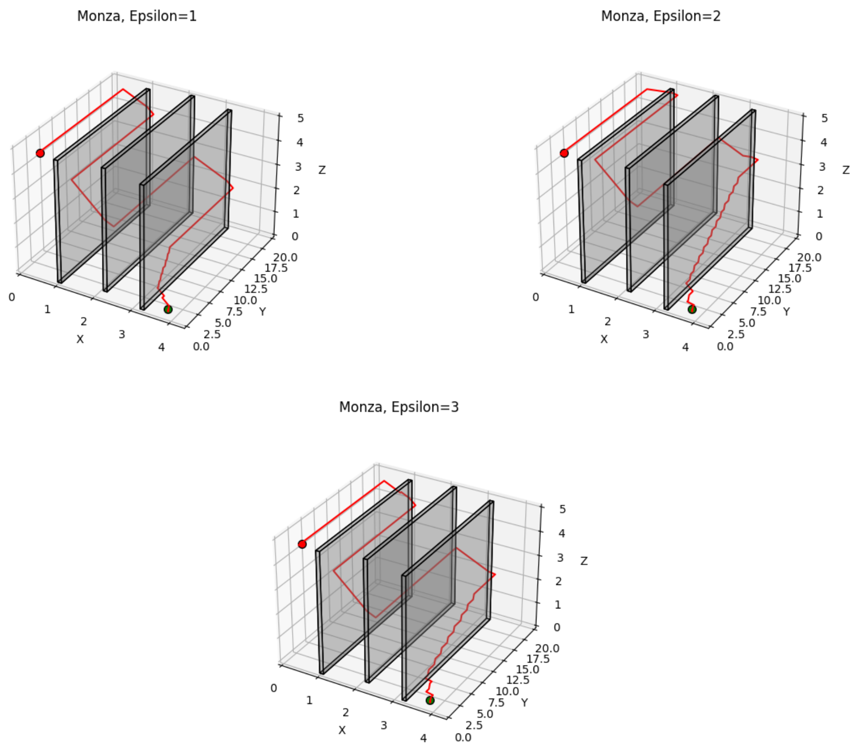
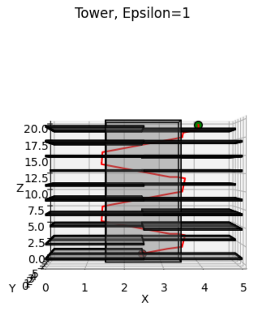
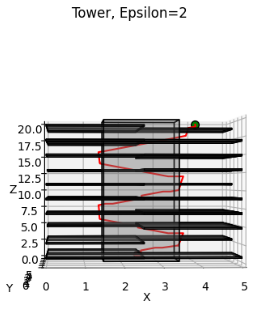


Figure 4: **Monza** environment. Path lengths from low to high epsilon: 76.5102, 76.6452, 76.5681. Runtimes from low to high epsilon: 19.7290s, 14.9757s, 15.4446s.





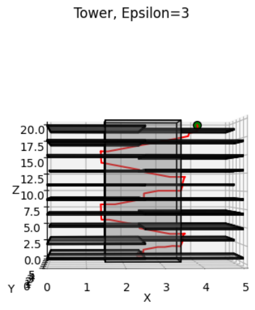


Figure 5: **Tower** environment. Path lengths from low to high epsilon: 28.2129, 29.2866, 30.6331. Runtimes from low to high epsilon: 28.2129s, 5.9098s, 6.6708s.

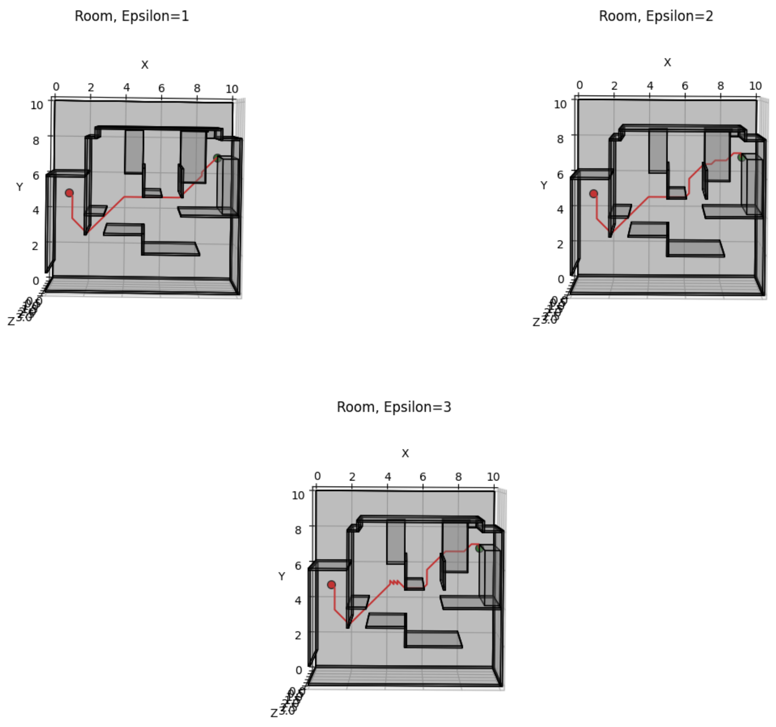


Figure 6: **Room** environment. Path lengths from low to high epsilon: 11.6710, 12.3053, 13.3195. Runtimes from low to high epsilon: 5.2219s, 0.8717s, 1.0212s.

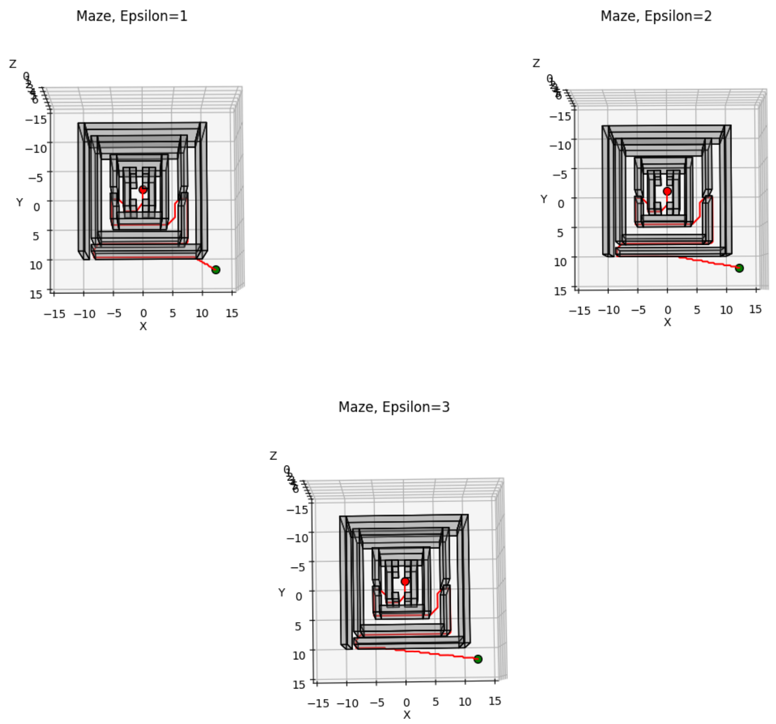


Figure 7: **Maze** environment. Path lengths from low to high epsilon: 74.4880, 74.6229, 74.7881. Runtimes from low to high epsilon: 286.0447s, 194.3470s, 76.5032s.

4.2 RRT* Figures

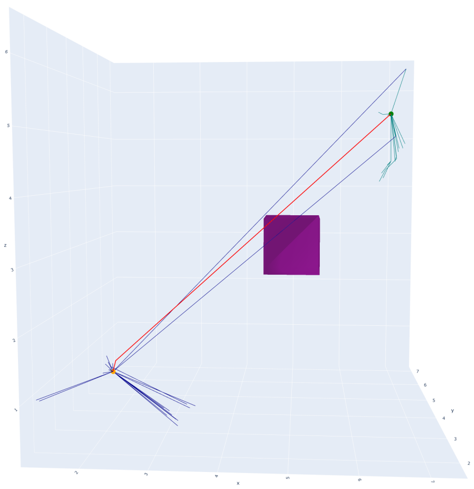


Figure 8: **Cube** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 8.1297. Runtime: 1.4898s. Samples: 420.

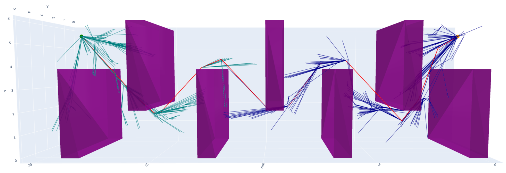


Figure 9: **Flappy Bird** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 27.6605. Runtime: 12.1748s. Samples: 1218.

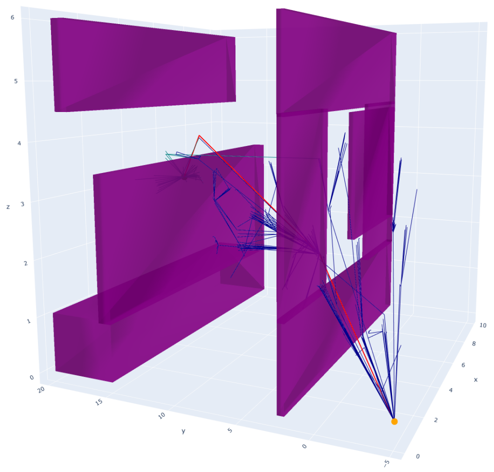


Figure 10: **Window** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 24.3553. Runtime: 7.7756s. Samples: 689.

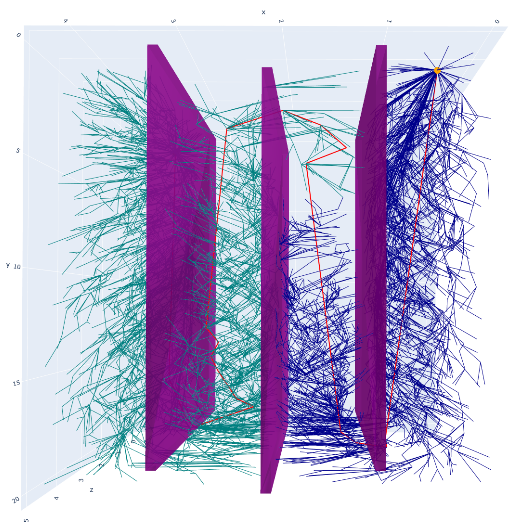


Figure 11: **Monza** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 76.0312. Runtime: 74.9659s. Samples: 4355.

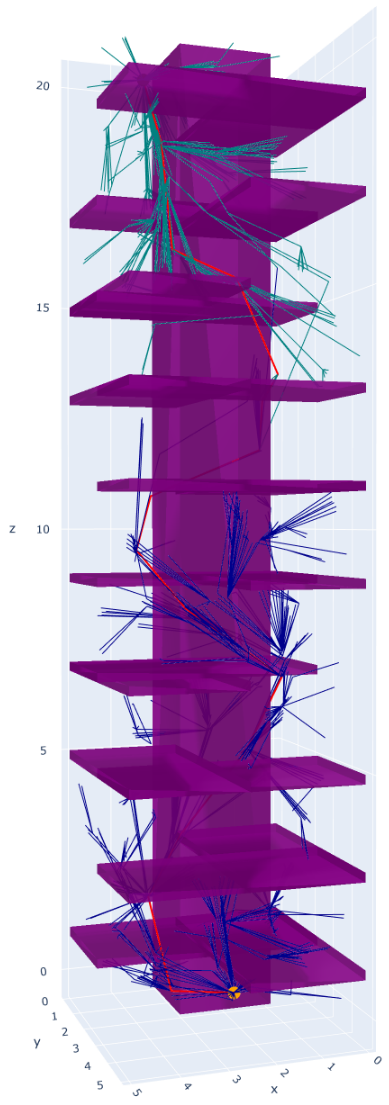


Figure 12: **Tower** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 31.3623. Runtime: 22.9510s. Samples: 2523.

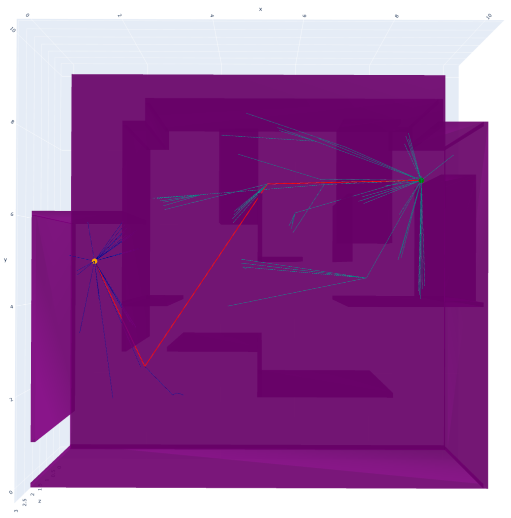


Figure 13: **Room** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 12.2010. Runtime: 2.0890s. Samples: 173.

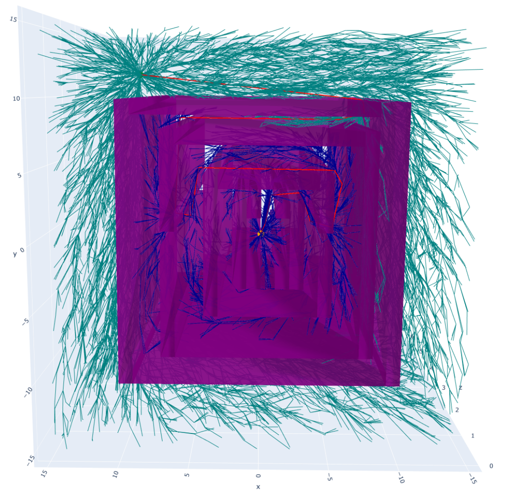


Figure 14: **Maze** environment. Final path in red, start-based tree in blue, goal-based tree in green. Path length: 75.2647. Runtime: 438.4345s. Samples: 37889.

References

[1] Nikolay Atanasov. Lecture 7: Search-based motion planning. In *ECE276B: Planning & Learning in Robotics*, 2021.

[2] Nikolay Atanasov. Lecture 9: Sampling-based motion planning. In *ECE276B: Planning & Learning in Robotics*, 2021.

[3] SZanlongo. rrt. `https://github.com/motion-planning/rrt-algorithms`.