ECE276B PR2 Report

1st Weixiao Zhan
weixiao-zhan[at]ucsd[dot]edu

I. Introduction

Motion planning in robotics involves determining a policy or a path from a certain start state to a certain goal state while minimizing the cost and avoiding obstacles. This project focuses on solving a deterministic motion planning problem in 3-D Euclidean space with axis-aligned bounding boxes (AABB) obstacles using search-based and sample-based algorithms.

Search-based algorithms systematically explore the state space to find a path, while sample-based algorithms randomly sample the space and connect samples to construct a feasible path. Specifically, this project implements collision checking for AABB, A* algorithm, probabilistic road map (PRM), rapidly exploring random tree (RRT), and RRT*. Lastly, evaluations are made between search-based and sampling-based methods.

II. Problem Formulation

The deterministic motion planning problem can be described as the collection of state space $\mathcal{X}$, control space $\mathcal{U}$, motion model f, time horizon T stage cost l, and terminal cost q. Denote the start state as s and the goal state as τ.

A. State Space

In this project, the environment is a collection of one boundary AABB (denote as B) and m obstacle AABBs (denote as O_i). Each AABB represents a set of points within a box-shaped range. An AABB can be compactly parameterized by two $\mathbb{R}^3$ points, i.e., the lower left l and the top right r:

$$\text{env} = \{B, O_1, O_2, \cdots, O_m \in \text{AABB}\}$$
$$\text{AABB}_{l,r} = \left\{ p \middle| \begin{matrix} l, p, r \in \mathbb{R}^3 \\ l_i \le p_i \le r_i, i \in \{x, y, z\} \end{matrix} \right\}$$

The state space is all free space in the environment:

$$\mathcal{X} = \left\{ p \middle| p \in \left(B \setminus \bigcup_i O_i \right) \right\}$$

Due to the limited numerical precision on computers, the search-based planner must explicitly discretize the state space. Denote the discretization scale as ϕ, that is, discretize 1 unit of each axis into ϕ cells. All points within each cell are considered the same and located at the center of the cell. The sample-based planner only consider a finite sampled subset of state space.

$$\mathcal{X}_{search} = \{\text{int}(\phi p), \forall p \in \mathcal{X}\}$$
$$\mathcal{X}_{sample} \subset \mathcal{X}$$

B. Control Space

In search-based planning, the point can move in all diagonal directions, allowing for 26 possible moves from each state. Whereas in sample-based planning, the point can move any length in any direction.

$$\mathcal{U}_{\text{search}} = \left\{ -\frac{1}{\phi}, 0, \frac{1}{\phi} \right\}^3$$
$$\mathcal{U}_{\text{sample}} = \mathbb{R}^3$$

C. Motion Model

The motion model is deterministic.

$$f(x, u) = x + u \quad \text{iff} \quad x + u \in \mathcal{X}$$

D. Stage Cost and Terminal Cost

$$l(x, u) = \|u\|_2$$
$$q(x) = \begin{cases} 0 & x = \text{goal} \\ \infty & \text{otherwise} \end{cases}$$

E. Equivalent Shortest Path Problem

The deterministic motion planning problem is equivalent to the shortest path problem. Thus, finding an optimal control policy is equivalent to finding the shortest path in a certain graph G:

$$G_\Lambda = \left\{ \begin{matrix} V = \mathcal{X}_\Lambda \\ E = \left\{ (x \to f(x,u)) \middle| \begin{matrix} \forall x \in \mathcal{X}_\Lambda \\ \forall u \in \mathcal{U}_\Lambda \end{matrix} \right\} \end{matrix} \right\}$$
$$\Lambda = search, sample$$

Given a shortest path $(s \to p_1 \to \cdots p_i \to \cdots \tau)$, the corresponding optimal control policy is

$$\pi_{T-i}(p_i) = p_{i+1} - p_i$$

III. Technical Approach

A. Collision Checking Environment Class

The `CollisionCheckingEnvironment` class is the framework for representing the environment and the state space. It implemented following primitives:

1) `load`*:* load environment from text file and apply discretization if desired.

2) `point_collide`*:* checks whether a point locate inside any obstacles or outside the boundary. The method is cached using LRU-cache decorator to improve performance by avoiding redundant checks, especially in search-based planner.

3) `line_collide`: checks whether a line segment collides with any obstacles. This is achieved using the `trimesh` library: each AABB obstacle is converted to a 3-D mesh composed of 12 triangles; the line segment is converted to a ray; and ray tracing is used to find intersections. Then, the intersections are checked whether lying on the line segment.

Note if a line segment compactly lies within one obstacle, this method will not report collision. Thus, `point_collide` need to be used in conjunction to ensure entire line segment lies in free space.

4) `sample_free`: sample random points in the environment's free space.

5) visualization: Additionally, the class includes visualization functions to render the environment with obstacles, start, and goal points, aiding in visual debugging and analysis.

B. Search-Based Planner

The `SearchBasedPlanner` class provides an abstract framework for implementing label correcting algorithm in a 3-D environment. It maintains:

- `start` and `goal`: The start and goal positions in the environment.
- `g`: A dictionary to hold the cost from the start to each node, initialized to infinity.
- `parent`: A dictionary to hold the shortest path tree, mapping each node to its parent.
- `open_list`: A set containing the frontier nodes during the search.
- `close_set`: A set to keep track of expanded nodes.

and support:

- `iter_children`: generates valid (no collision) child nodes (neighbors) and their respective stage costs.
- `build_path`: reconstructs the path from the goal to the start using the parent pointers, computing the total path cost.

All search-based implementation only need to overload the `search` method.

1) A:* inherits from `SearchBasedPlanner` and implements the A* algorithm. The heuristic function used is the L2 norm (Euclidean distance), which is also cached for efficiency. The `open_list` is revised to a priority queue sorted by $g + heuristic$.

Its `search` method initializes the cost to the start node and inserts it into the open list with its heuristic value. The main loop iteratively expands the node with the lowest estimated total cost until the goal is reached. During expansion, it updates the cost and parent for each valid child node and adds them to the open list if a shorter path is found.

C. Sample-Based Planner

The `SampleBasedPlanner` class provides an abstract framework for implementing sample-based motion planning algorithms in a 3-D environment. It maintains:

- `start` and `goal`: The start and goal positions in the environment.
- `radius`: The critical radius

$$r^* = 2\left(1+\frac{1}{d}\right)^{\frac{1}{d}}\left(\frac{Vol(C_{free})}{Vol(Unit-ball)}\right)^{\frac{1}{d}}\left(\frac{\log(n)}{n}\right)^{\frac{1}{d}}$$

- `graph`: A `networkx` graph to represent the sampled nodes and edges.

and supports:

- `build_path`: Search shortest path from start to goal using `networkX` build in dijkstra algorithm.
- `near` and `nearest`: Finds near nodes of a given node using `cKDTree` from `sklean` library to accelerate computation.
- `steer`: Generates a point that steers from the start towards the goal by a specified distance.

All sample-based implementation only need to overload the `sample` method.

1) PRM: inherits from `SampleBasedPlanner`. Its `sample` method generates a roadmap by randomly sampling nodes in the free space and connecting new nodes to their nearby nodes if the path between them is collision-free.

2) RRT: inherits from `SampleBasedPlanner` and revise the `graph` to a directed graph. Its `sample` method generate a tree by randomly sampling nodes in the free space and connecting new nodes to their nearest node if the path between them is collision-free. Every certain period, RRT will try to connect the goal with nearest node on the tree.

3) RRT:* extends `RRT` with optimal capabilities. It maintains a cost dictionary to track the cost to each node from the start. Its `sample` method, in addition to the routine in RRT's `sample`, also rewire the tree in the local region around the new node if there is a better path to nearby nodes via the new node. After rewiring, the cost reduction is cascade to all descendant of the rewired near node.

IV. Results

Five planner are compared: A* ($\phi = 5$), A* ($\phi = 8$), PRM, RRT, RRT* (first path and final path). For A*, two discretization scale are used and compared. For sample-based planner, the radius is set to critical radius r^*; the number of iteration is set to terminate when the graph had $N = 2000$ valid nodes; the steer step is set to 0.2. For RRT*, both the first path and final path are reported.

A. quality of path

Figure 1 illustrates the shortest path lengths found by various planner across different environments. The RRT and RRT* algorithms failed to find any valid path in the "monza" environment within a reasonable (50,000) iterations due to the special construction of the environment, which will be discussed in a later section.

A* planner with larger discretization scale and finer path granularity can produce slightly shorter paths.

A* and other Search-based planners are limited to 45-degree movements, thus, despite explored all nodes systematically,

their results are not the true shortest path. In face, sample-based planners can sometimes produce slightly shorter path than A*.

Among sample-based planners, the standard RRT produces the longest paths. Whereas, PRM and RRT* can find equally good paths compared to search-based planners. In addition, RRT* can keep improving the path quality with more iterations and rewiring.

B. Runtime and Solution Size

The time complexity and memory complexity is measured by the total runtime (shown in Figure 2) and the size of solution (shown in Figure 3) respectively.

The size of solution is defined to as:

1) A*: the number of nodes in open and close set;
2) PRM: the number of edges in graph. Note: the PRM always iterates N times and produce a graph with N nodes.
3) RRT and RRT*: the number of sample iteration to build the N node tree. Note: not every iteration in RRT and RRT* can add new nodes to the graph. Thus, the solution size is measured in number of sample iterations.

Based on Figure 2 and 3, when the l2 heuristic works well, such as "single cube" and "room" environment, A* planner is the fastest and uses the fewest memory. However, when the heuristic doesn't work well, such as "maze" environment, A* planner is the slowest and uses significantly more memory. Moreover, any slightly increment in discretization scale would case the solution size cubically increase in worse case.

For PRM planner, the number edge stays largely the same across different environments.

For RRT and RRT* planner, the more complex the environment, the more iteration it needs to build N node graph, meaning each iteration has smaller probability of extending the tree.

C. profiling

`cProfile` library is used to further investigate program profile of A* and PRM planner. Figure 4 shows that A* spend most of its time on iterating child nodes and compute hashes for indexing. Where as PRM spend most of its time on collision checking.

D. PRM vs RRT

Based on Figure 2 and 3, RRT runs faster and consume fewer memory than PRM. However, this comes with the cost of less robust and more sensitive to the environment layout.

Figure 5 shows the PRM and RRT on monza environment after $\frac{N}{2} = 1000$ iterations. The PRM has already expand though the entire free space and connect to the goal. Where as the RRT is stuck behind the first wall.

PRM always add the new node into graph in every iterations. The new node may not connect to the graph imminently, but they are place holders to support future new nodes and improvements.

On the other hand, RRT only try to connect the nearest point. Since the wall is skinny, most of the nearest nodes will lay on two side of the wall. Their connection attempt would fail due to collision checking. Thus RRT is struggle to make progress in such environments.

E. RRT vs RRT*: rewire effect

Figure 6 shows RRT and RRT* in single cube environment after $\frac{N}{2} = 1000$ iterations. The rewiring in RRT* is able to revise the randomly generated tree into radial shape, representing the optimal path from start to any sampled nodes.

F. future work

To investigate the effect of different step size in RRT and RRT*.

G. result summary

A* rely on good heuristic function to work well. PRM is probabilistically optimal and most robust among sample-based planners. but it is slow to compute and consume much memory. RRT reduces the runtime and memory footprint, but is less robust and not probabilistically optimal. RRT* introduced rewiring step to achieved probabilistically optimal but the runtime is slow down to PRM level.

APPENDIX

Shortest path of all methods in all environment is shown in Figure 7. For interactive result, run `plot.py`

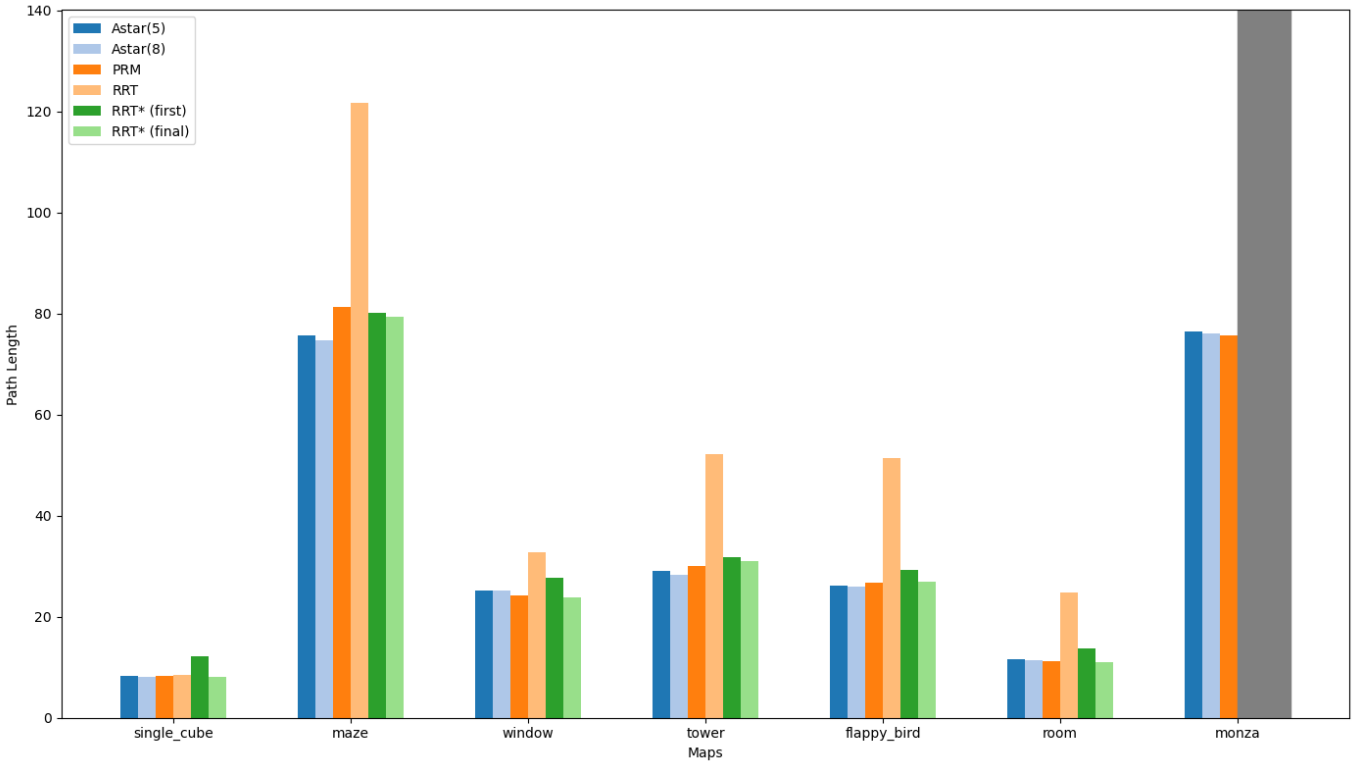


Fig. 1: Path Length Comparison

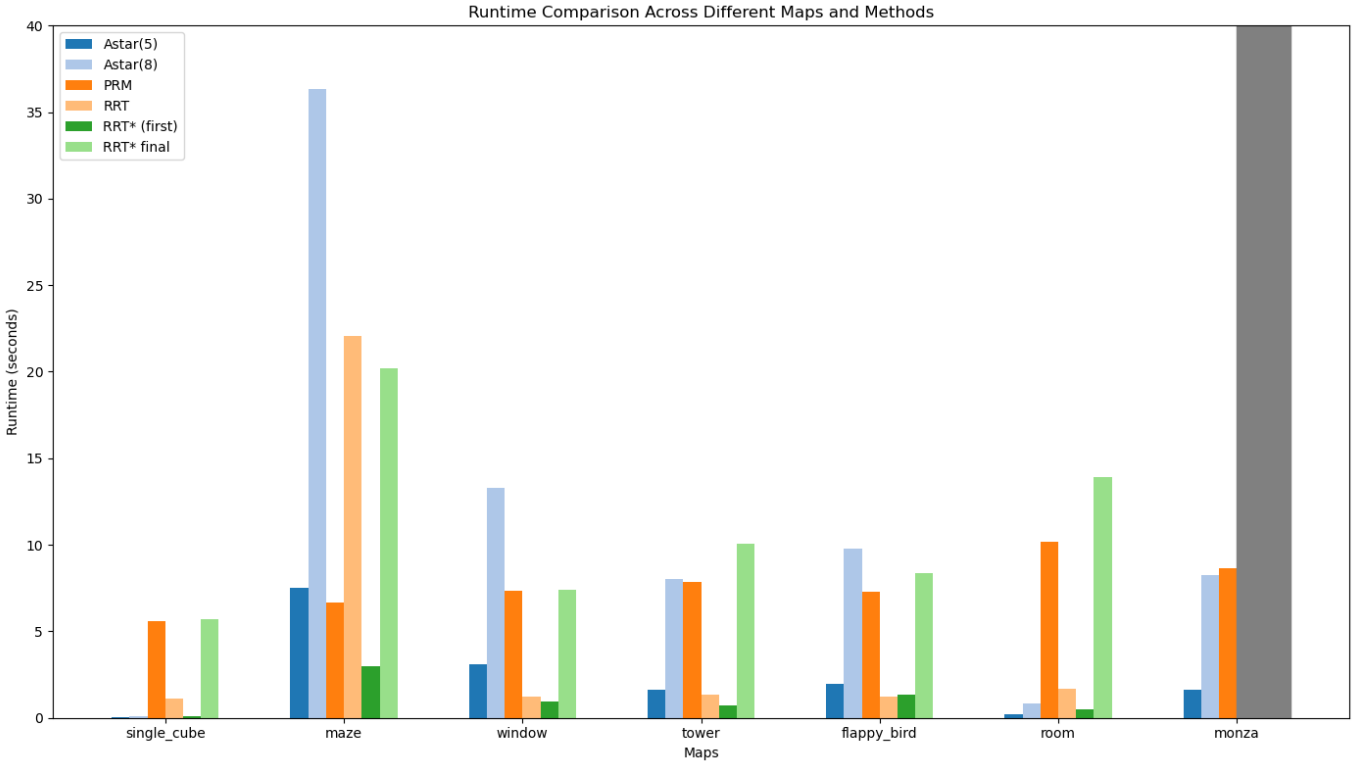


Fig. 2: Runtime

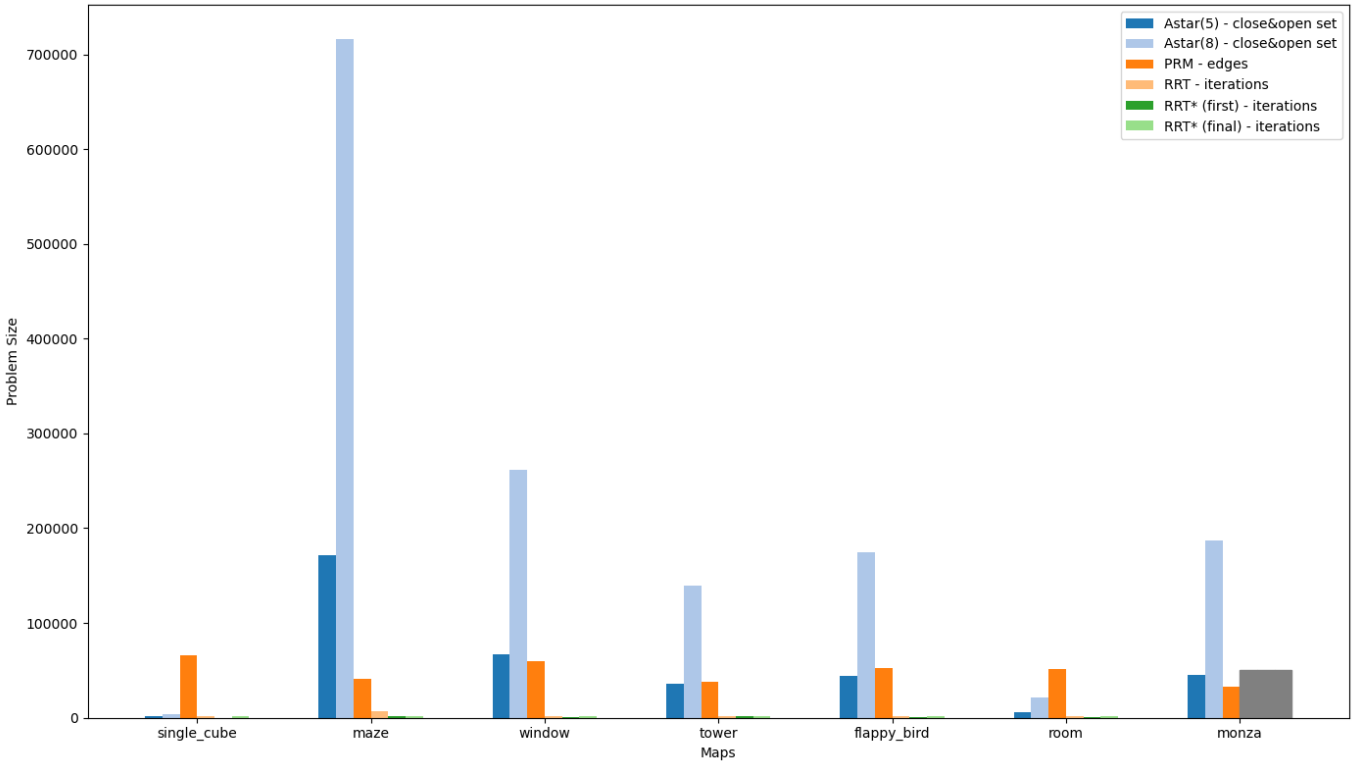


Fig. 3: Solution Size

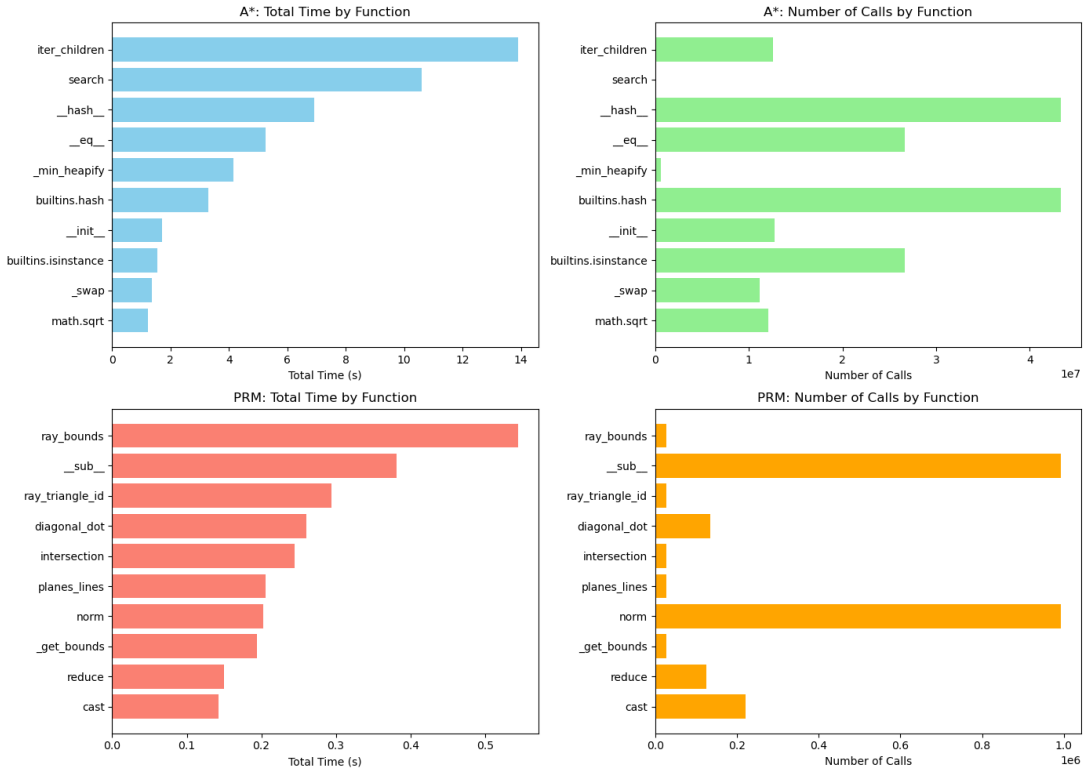
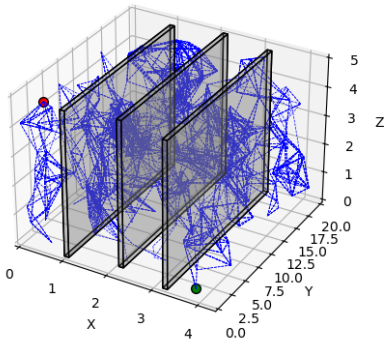
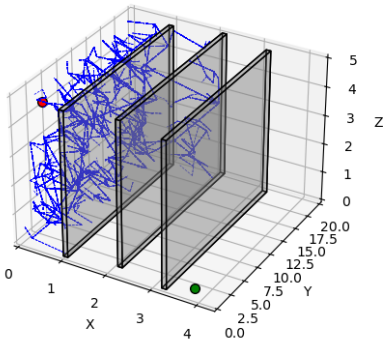


Fig. 4: Profile

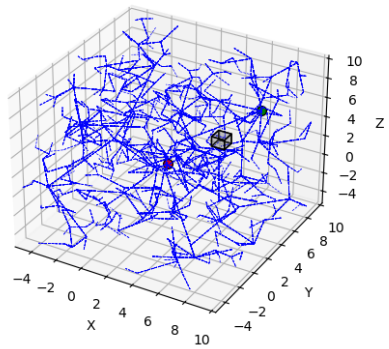


(a) PRM after $\frac{N}{2}$ iterations

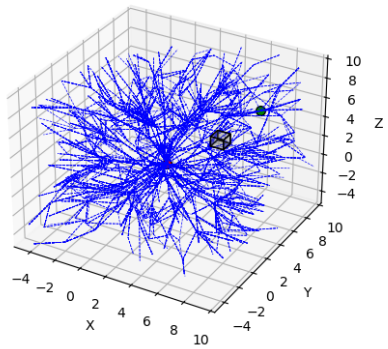


(b) RRT after $\frac{N}{2}$ iterations

Fig. 5: Monza environment

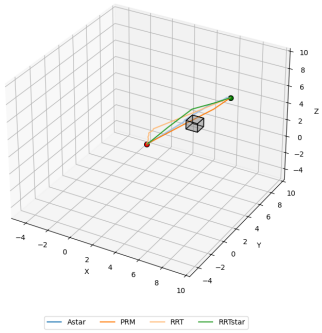


(a) RRT after $\frac{N}{2}$ iterations

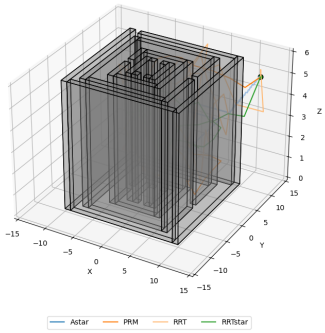


(b) RRT* after $\frac{N}{2}$ iterations

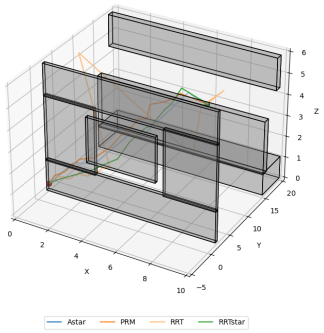
Fig. 6: Single cube environment



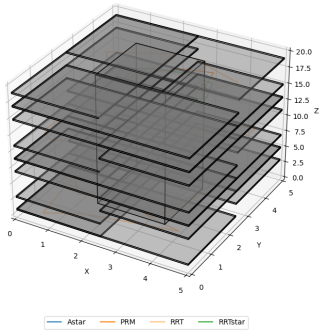
(a) Single Cube



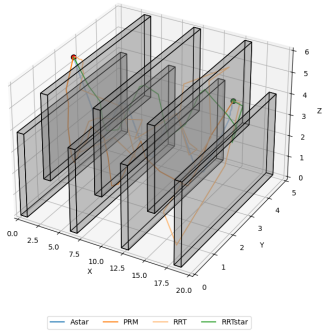
(b) Maze



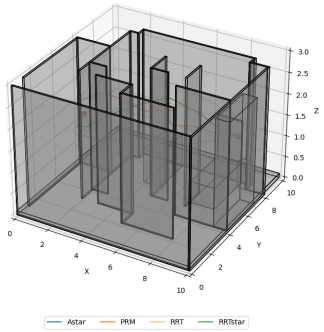
(c) Window



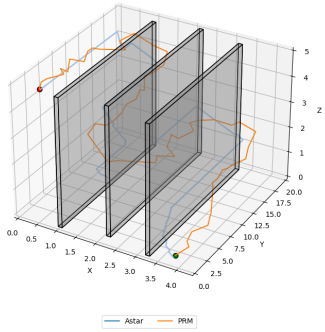
(d) Tower



(e) Flappy Bird



(f) Room



(g) Monza

Fig. 7: hortest path of all methods in all environment