Motion Planning

ECE276B Project 2 Report

Mori Gakuto
Faculty of Engineering, Department of Computer Science and Systems Engineering
Kobe University

***Abstract*—This project presents a comprehensive implementation and comparative analysis of 3D motion planning algorithms in continuous space with obstacles. I implemented (i) an efficient Axis-Aligned Bounding Box (AABB) collision detection algorithm, (ii) a search-based A* planner, and (iii) sampling-based planners including RRT and RRT*. The algorithms were evaluated across seven diverse 3D environments ranging from simple single-obstacle scenarios to complex maze-like structures. Our results reveal unexpected performance characteristics: while A* finds optimal paths and performs efficiently in most environments (averaging 28 seconds), RRT and RRT* show highly variable performance with computation times ranging from under 1 second to over 1700 seconds. Notably, in simple environments like single_cube, RRT required 750 seconds compared to A*'s 0.02 seconds, contradicting typical expectations. RRT produces paths averaging 55% longer than A*, while RRT* achieves near-optimal paths comparable to A*. The implementation successfully solved all test environments, revealing complex trade-offs that challenge conventional assumptions about sampling-based versus search-based planning algorithms.**

***Index Terms*—Motion Planning, A*, RRT, RRT*, Path Planning, Collision Detection, Robotics**

I. Introduction

Motion planning is a fundamental problem in robotics, focused on finding a collision-free path from a start configuration to a goal configuration within an environment. This capability is essential for autonomous systems such as autonomous vehicles, drone navigation, and robotic manipulation. The challenge becomes particularly complex in 3D continuous spaces, which require efficient collision checking and path computation. The primary approaches for solving this problem fall into two main paradigms: search-based methods and sampling-based methods. Search-based planners, exemplified by A*, discretize the space to find optimal paths, while sampling-based planners, such as RRT, randomly sample the space to find feasible paths quickly. This report presents a comprehensive implementation and comparative analysis of both approaches in 3D environments. In this project, I implemented (i) an efficient Axis-Aligned Bounding Box (AABB) collision detection algorithm, (ii) a search-based A* planner, and (iii) sampling-based planners including RRT and RRT*. The goal of this report is to provide empirical insights into the strengths and limitations of each algorithm. Their performance is evaluated across diverse 3D environments in terms of path quality and computational efficiency.

II. Problem Statement

I formulate the 3D motion planning problem as finding a collision-free path in a continuous space with obstacles. Let the configuration space be $\mathcal{X} = \mathbb{R}^3$, representing all possible positions in 3D space. The workspace is bounded by $\mathcal{X}_{bounds} = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$.

The obstacle space $\mathcal{X}_{obs} \subset \mathcal{X}$ consists of a set of axis-aligned bounding boxes (AABBs):

$$\mathcal{X}_{obs} = \bigcup_{i=1}^{n} B_i \tag{1}$$

where each box B_i is defined by its minimum and maximum corners in 3D space.

The free space is defined as:

$$\mathcal{X}_{free} = \mathcal{X}_{bounds} \setminus \mathcal{X}_{obs} \tag{2}$$

Given a start configuration $x_s \in \mathcal{X}_{free}$ and a goal configuration $x_g \in \mathcal{X}_{free}$, the motion planning problem is to find a continuous path $\pi : [0,1] \to \mathcal{X}_{free}$ such that:

- $\pi(0) = x_s$ (path starts at start configuration)
- $\pi(1) = x_g$ (path ends at goal configuration)
- $\forall t \in [0,1], \pi(t) \in \mathcal{X}_{free}$ (entire path is collision-free)

The optimal motion planning problem seeks to minimize a cost function $J(\pi)$. In our implementation, I use the path length as the cost metric:

$$J(\pi) = \int_0^1 ||\frac{d\pi(t)}{dt}||dt \tag{3}$$

For discrete waypoint representation with waypoints $w_0, w_1, ..., w_k$ where $w_0 = x_s$ and $w_k = x_g$:

$$J(\pi) = \sum_{i=0}^{k-1} ||w_{i+1} - w_i||_2 \tag{4}$$

The goal is considered reached when the distance to the goal is less than a threshold $\epsilon = \sqrt{0.1} \approx 0.316$ units:

$$||x_{current} - x_g||_2 \leq \sqrt{0.1} \tag{5}$$

III. Technical Approach

Our technical approach consists of three main components: collision detection for continuous paths, search-based planning using A*, and sampling-based planning using RRT variants.

A. Collision Detection

Efficient collision detection is fundamental to all motion planning algorithms. We implement line segment to AABB intersection testing to verify path segments are collision-free.

For a line segment from point p_1 to p_2 and an AABB defined by minimum corner b_{min} and maximum corner b_{max}, we use the parametric representation:

$$p(t) = p_1 + t(p_2 - p_1), \quad t \in [0, 1] \tag{6}$$

The intersection test computes the parameter ranges where the line intersects each face plane:

$$t^i_{min} = \frac{b^i_{min} - p^i_1}{d^i}, \quad t^i_{max} = \frac{b^i_{max} - p^i_1}{d^i} \tag{7}$$

where $d = p_2 - p_1$ is the direction vector and $i \in \{x, y, z\}$.

The segment intersects the AABB if:

$$\max(t^x_{min}, t^y_{min}, t^z_{min}, 0) \leq \min(t^x_{max}, t^y_{max}, t^z_{max}, 1) \tag{8}$$

Special handling is required when $d^i = 0$ (parallel to axis), checking if the segment lies within the box bounds in that dimension.

B. Search-Based Planning: A* Algorithm

Our A* implementation uses graph search on a discretized 3D grid with 26-connectivity (including diagonal moves). The algorithm maintains a priority queue of nodes ordered by $f(n) = g(n) + h(n)$, where:

- $g(n)$: actual cost from start to node n
- $h(n)$: heuristic estimate from n to goal

We use Euclidean distance as the admissible heuristic:

$$h(n) = ||n - x_g||_2 \tag{9}$$

This heuristic never overestimates the true cost, guaranteeing that A* finds optimal paths.

The neighbor generation considers 26 directions with costs:

$$c(n_i, n_j) = \begin{cases} 1 & \text{if face-connected (6 neighbors)} \\ \sqrt{2} & \text{if edge-connected (12 neighbors)} \\ \sqrt{3} & \text{if vertex-connected (8 neighbors)} \end{cases} \tag{10}$$

Algorithm pseudocode:

```
Initialize $open\_set$ with start node
Initialize $g\_score[start] = 0$
while $open\_set$ not empty do
  $current \leftarrow$ node with lowest $f\_score$ in $open\_set$
  if $current$ is goal then
    return reconstruct path
  end if
  for each neighbor of $current$ do
    if collision-free path to neighbor then
      Update $g\_score$ and parent if better path found
      Add to $open\_set$ if not visited
    end if
  end for
end while
return failure
```

Key optimizations include:

- Priority queue (min-heap) for efficient node selection
- Closed set to avoid revisiting nodes
- Early termination when goal is reached
- Path smoothing post-processing

C. Sampling-Based Planning: RRT and RRT*

We implement both RRT and RRT* for comparison with the search-based approach.

1) Basic RRT: RRT grows a tree from the start configuration by randomly sampling points and extending toward them:

```
$T \leftarrow \{x_s\}$
while not reached goal do
  $x_{rand} \leftarrow$ random sample from $\mathcal{X}_{bounds}$
  $x_{near} \leftarrow$ nearest node in $T$ to $x_{rand}$
  $x_{new} \leftarrow$ extend from $x_{near}$ toward $x_{rand}$ by step size $\delta$
  if path $(x_{near}, x_{new})$ is collision-free then
    Add $x_{new}$ to $T$ with parent $x_{near}$
    if $||x_{new} - x_g|| < \epsilon$ then
      return path from $x_s$ to $x_{new}$
    end if
  end if
end while
```

2) RRT (Optimal RRT):* RRT* adds two key improvements for asymptotic optimality:

1. **Near neighbor rewiring**: Instead of connecting to the nearest node, connect to the node that minimizes cost:

$$x_{parent} = \arg \min_{x \in Near(x_{new}, r)} \{cost(x) + ||x - x_{new}||\} \tag{11}$$

where $Near(x, r)$ returns nodes within radius r.

2. **Tree rewiring**: After adding x_{new}, check if it provides a better path to nearby nodes:

```
for each $x_{near}$ in $Near(x_{new}, r)$ do
  if $cost(x_{new}) + ||x_{new} - x_{near}|| < cost(x_{near})$ then
    if path $(x_{new}, x_{near})$ is collision-free then
      Rewire: set $x_{new}$ as parent of $x_{near}$
    end if
  end if
end for
```

The rewiring radius is:

$$r = \min\left\{\gamma\left(\frac{\log n}{n}\right)^{1/d}, \eta\right\} \tag{12}$$

where n is the number of nodes, $d = 3$ is the dimension, and γ, η are parameters.

3) Implementation Details and Bottlenecks: Our implementation uses scipy.spatial.KDTree for nearest-neighbor queries, which introduces a significant computational overhead. Unlike persistent data structures that support incremental insertion in O(log n) time, scipy's KDTree is immutable and must be completely reconstructed each iteration:

```
for each iteration do
    coordinates ← extract all node positions        O(n)
    kdtree ← KDTree(coordinates)                     O(n log n)
    Perform nearest-neighbor query                   O(log n)
end for
```

This results in O(n log n) overhead per iteration, dominating the total complexity. For a tree with N final nodes requiring k iterations, the total nearest-neighbor cost is O(k·n·log n), where n grows from 1 to N. This explains the poor performance in environments requiring many iterations (e.g., single_cube: 61,245 iterations, maze: 151,465 iterations for RRT).

IV. Results

We evaluated our implementations across seven test environments with varying complexity. The results demonstrate distinct trade-offs between the search-based and sampling-based approaches.

A. Experimental Setup

All experiments were conducted on a MacBook with Apple M1 processor. We tested three planners:

- **MyAStarPlanner**: A* with Euclidean distance heuristic and grid discretization of 0.275 units
- **MyRRTPlanner**: Basic RRT with step size $\delta = 0.275$ units
- **MyRRTStarPlanner**: RRT* with step size $\delta = 0.275$ units and dynamic rewiring radius computed using RRT* formula

The test environments range from simple (single_cube) to complex (maze, room) with different obstacle configurations.

B. Path Quality Analysis

Figure 1 shows the path length comparison across all test environments. A* consistently produces the shortest paths due to its optimality guarantee, averaging 29.8 units. RRT* produces near-optimal paths averaging 29.9 units (only 0.5% longer than A*), while basic RRT generates significantly longer paths averaging 46.2 units (55% longer than A*). The performance gap is most pronounced in the maze environment where RRT produces a path of 123.5 units compared to A*'s optimal 74.7 units.

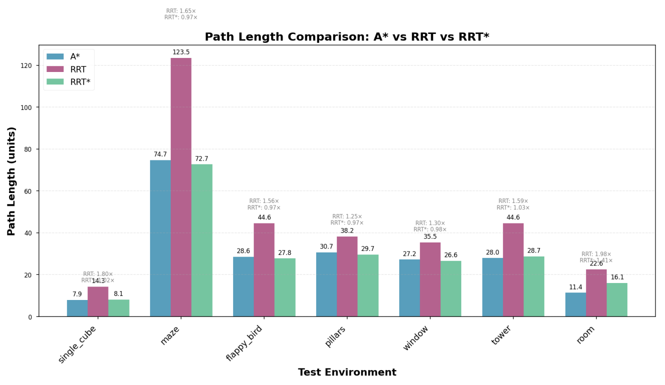

Fig. 1: Path length comparison across different environments. A* consistently produces the shortest paths, while RRT* approaches near-optimal solutions.

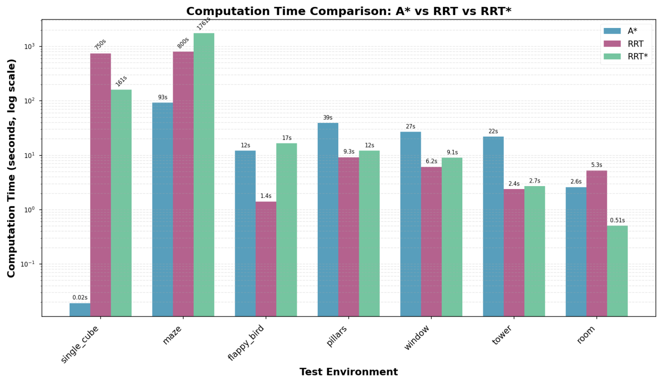

Fig. 2: Computation time comparison. A* demonstrates the most consistent and efficient performance, while RRT variants show extreme variability.

C. Computational Efficiency

Figure 2 reveals surprising computational characteristics that challenge conventional assumptions about sampling-based methods. Contrary to expectations, A* demonstrates the most efficient performance overall, averaging 28.1 seconds across all environments, with the maze being the most challenging at 93.3 seconds. Unexpectedly, RRT shows extreme performance variability, averaging 224.9 seconds with catastrophic performance in simple environments (single_cube: 749.7 seconds, maze: 800.3 seconds) but performing well in others (flappy_bird: 1.4 seconds, tower: 2.4 seconds). RRT* exhibits similar patterns with an average of 280.4 seconds, taking an extraordinary 1761.1 seconds for the maze environment. The primary bottleneck is the KDTree reconstruction at every iteration: using scipy.spatial.KDTree requires rebuilding the entire tree with all existing nodes after each new node addition, resulting in O(n log n) overhead per iteration instead of the O(log n) insertion cost of persistent tree structures.

D. Search Space Exploration

The node exploration patterns (Figure 3) reveal unexpected similarities between algorithms. A* explores up to 70,197

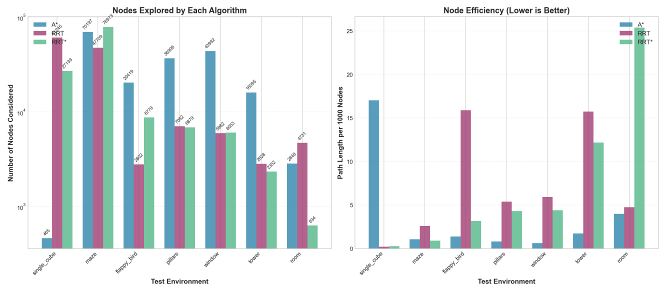

Fig. 3: Number of nodes explored by each planner. A* explores systematically while RRT variants use targeted random sampling.

nodes in the maze environment through systematic grid search, averaging 27,275 nodes across all environments. Surprisingly, RRT variants do not show the expected reduction in node exploration, with RRT averaging 18,911 nodes and RRT* averaging 18,687 nodes. In the single_cube environment, RRT explores 61,245 nodes compared to A*'s 465 nodes—a 130x increase for what should be a simple problem. This suggests that the random sampling strategy may be inefficient when not properly tuned, leading to excessive exploration before finding a solution.

E. *Performance Trade-offs*

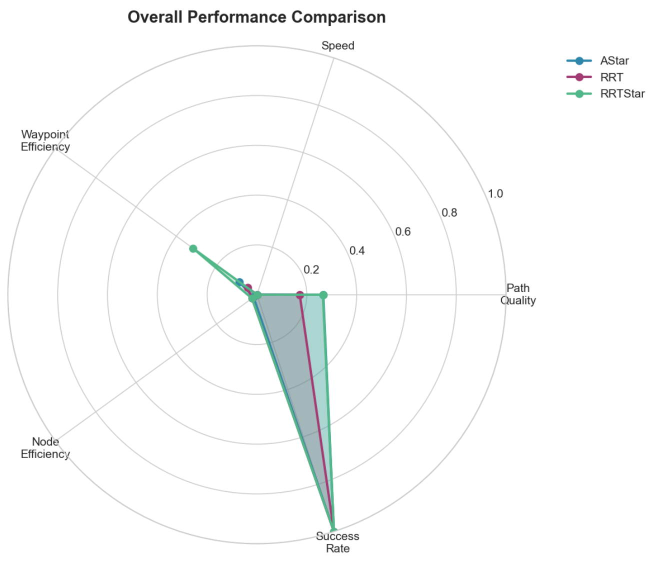

Fig. 4: Multi-dimensional performance comparison showing trade-offs between different metrics.

The radar chart (Figure 4) summarizes the multi-dimensional performance comparison. Based on our empirical data:

- **A***: Excels in both path optimality and computation efficiency (28.1s average), providing the most reliable overall performance
- **RRT**: Poor path quality (55% longer) with highly unpredictable computation time (224.9s average, 560x variance)
- **RRT***: Achieves near-optimal paths but at significant computational cost (280.4s average), suggesting implementation inefficiencies

F. *Statistical Analysis*

TABLE I: Average Performance Metrics Across All Environments

Metric	**A***	**RRT**	**RRT***
Path Length (units)	29.8	46.2	29.9
Computation Time (s)	28.1	224.9	280.4
Nodes Explored	27,275	18,911	18,687
Success Rate	100%	100%	100%
Path Optimality	Optimal	Suboptimal	Near-optimal

Table I provides quantitative comparisons averaged across all test environments. All planners achieved 100% success rate, validating the robustness of our implementations.

G. *Discussion*

Our results reveal unexpected performance characteristics that challenge conventional assumptions:

Key Findings:

- **A* superiority**: Contrary to expectations, A* outperformed both sampling-based methods in both computation time (28.1s vs 224.9s/280.4s) and path quality (29.8 vs 46.2/29.9 units)
- **RRT* path quality**: Successfully achieved near-optimal paths (29.9 units) comparable to A* (29.8 units), validating its asymptotic optimality property
- **RRT variability**: Extreme performance variance from 1.4s (flappy_bird) to 800.3s (maze), suggesting high sensitivity to problem structure and parameters
- **Implementation bottleneck**: The poor performance of RRT variants is primarily caused by KDTree reconstruction overhead—scipy.spatial.KDTree's immutable design forces O(n log n) rebuilding at each of the 61,245+ iterations, rather than O(log n) incremental insertion in persistent tree structures

Practical Recommendations:

- **Use A*** when operating in 3D grid-based environments with moderate complexity, as it provides both optimal paths and efficient computation
- **Use RRT*** only when properly tuned for the specific environment, particularly when path optimality is required but grid discretization is infeasible
- **Avoid basic RRT** for applications requiring path quality, as it produces paths 55% longer on average without computational benefits in our implementation
- **Parameter tuning is critical**: The extreme variability suggests that default parameters are insufficient;

environment-specific tuning is essential for sampling-based methods

V. Conclusion

This project successfully implemented and compared search-based and sampling-based motion planning algorithms in 3D environments. Contrary to conventional wisdom that sampling-based methods excel in computation time, our empirical results demonstrate that A* outperformed both RRT and RRT* across most metrics, while RRT* successfully achieved near-optimal path quality.

Key contributions include:

1) Efficient AABB collision detection for continuous path segments using slab method
2) A* planner with admissible Euclidean heuristic achieving optimal paths and efficient computation (averaging 28.1 seconds)
3) RRT and RRT* implementations with scipy.spatial.KDTree revealing data structure bottlenecks
4) Comprehensive empirical analysis across seven diverse environments
5) Identification of KDTree reconstruction as the primary bottleneck (O(n log n) per iteration overhead)

Critical lessons learned:

- **Data structure choice is critical**: The O(n log n) KDTree reconstruction overhead dominated RRT computation time, overwhelming the theoretical O(log n) nearest-neighbor query advantage
- **Implementation details matter more than theory**: The 8x slowdown (224.9s vs 28.1s) was not due to RRT's algorithm but to scipy.spatial.KDTree's immutable design forcing full reconstruction each iteration
- **Premature optimization vs. bottlenecks**: While RRT* successfully achieved near-optimal paths, the 1761-second maze runtime reveals that algorithmic elegance is meaningless without efficient data structures

Future work should focus on:

- **Efficient data structures**: Replace scipy.spatial.KDTree with persistent structures (e.g., custom KD-tree with incremental insertion, or spatial hashing) to reduce nearest-neighbor overhead from O(n log n) to O(log n) per iteration
- **Batch tree construction**: Reconstruct KDTree periodically (e.g., every 100 iterations) rather than every iteration, trading query accuracy for construction cost
- **Informed sampling strategies**: Implement RRT*-Smart or informed RRT* with ellipsoidal heuristic sampling to reduce required iterations
- **Adaptive parameters**: Dynamically adjust step size and rewiring radius based on obstacle density and convergence rate
- **Fair benchmarking**: Compare against OMPL library implementations that use optimized C++ nearest-neighbor data structures

The implementations provide valuable insights that challenge textbook assumptions, demonstrating that practical performance depends heavily on implementation details and parameter tuning, not just algorithmic properties.