Rapidly-Exploring Random Trees (RRT) Algorithm

Ansari Abrar

February 4, 2026

1 Preface

Almost no good article on this topic was found. So, I decided to write on my own and summarize a lot in a short form of a sheet. I also added extra resources that might help at the end of the sheet.

Contents

2 Introduction

Path planning involves finding a collision-free path from a starting position to a goal position in an environment cluttered with obstacles. You might've already hear about the Dijkstra algorithm, in your algorithm course (CSE-221), right? It is a path planning algorithm, but a specific type. There are three general classifications of path planning algorithms:

1. **Grid-Based Methods**: Discretize the environment into a grid (Dijkstra, A*, etc.).
2. **Sampling-Based Methods**: Sample configurations from continuous space (RRT, RRT*, PRM).
3. **Potential Field Methods**: Use artificial potential functions to guide motion

This document focuses on **Rapidly-Exploring Random Trees (RRT)**, a powerful sampling-based algorithm that efficiently searches high-dimensional configuration spaces. Unlike grid-based methods such as Dijkstra's algorithm, RRT does not require discretization of the environment and can handle continuous, high-dimensional problems effectively.

3 Overview of Sampling-Based Path Planning

Sampling-based methods follow a general framework:

1. **Sample**: Generate random configurations q_{rand} from the robot's free configuration space $\mathcal{C}_{free}$
2. **Nearest Neighbor**: Find the nearest existing node $q_{nearest}$ in the current tree using some distance metric
3. **Local Planner**: Connect $q_{nearest}$ toward q_{rand} with a simple path (usually straight line)
4. **Collision Check**: Verify the new path segment is collision-free
5. **Add Node**: If collision-free, add the new node q_{new} to the tree
6. **Check Goal**: If q_{new} is close to the goal, we may have found a solution
7. **Repeat**: Continue until a path is found or max iterations reached

4 The RRT Algorithm

4.1 Core Concept

The Rapidly-Exploring Random Tree (RRT) is a randomized data structure that grows a tree from the starting configuration by iteratively adding random points toward unexplored regions of the configuration space. The algorithm is **probabilistically complete**, meaning it will find a solution if one exists, given enough time and samples. In other words, the premise of RRT is actually quite straightforward. The points are generated randomly and connected to the closest available node. Each time a vertex is created, the vertex must be checked to be outside of an obstacle. Furthermore, chaining the vertex to its closest neighbor must also avoid obstacles. The algorithm ends when a node is generated within the target region or a limit is hit.

4.2 Key Advantages

- **High-Dimensional Spaces**: Works efficiently in 6D, 7D, or higher dimensions where grid-based methods fail
- **Non-Convex Obstacles**: Handles arbitrary obstacle geometries without special preprocessing
- **Fast Exploration**: Random sampling explores space faster than systematic methods in high dimensions
- **Simple Implementation**: Relatively straightforward to code and understand
- **Probabilistically Complete**: Guaranteed to find a path eventually (if one exists)

4.3 Key Limitations

- **Suboptimal Paths**: Solutions are often longer and more jagged than optimal paths
- **No Optimality Guarantee**: Cannot guarantee the shortest path
- **Random Behavior**: Results vary between runs (non-deterministic)
- **Inefficient for Simple Scenes**: Overkill for 2D grid-based navigation where Dijkstra is better

5 Mathematical Foundation

5.1 Configuration Space

The robot's **configuration space** $\mathcal{C}$ is the space of all possible positions and orientations of the robot. For a 2D point robot in a 2D environment, $\mathcal{C} = \mathbb{R}^2$. For a 6-DOF robot arm, $\mathcal{C} = \mathbb{R}^6$ or a non-Euclidean manifold.

- $\mathcal{C}_{free}$: Configurations where the robot does not collide with obstacles
- $\mathcal{C}_{obst}$: Configurations where the robot collides with obstacles
- $\mathcal{C}_{free} \cup \mathcal{C}_{obst} = \mathcal{C}$

5.2 Distance Metric

RRT uses a distance function to measure how far a sampled configuration is from existing tree nodes:

$$d(q_a, q_b) = \|q_a - q_b\|_2 = \sqrt{\sum_{i=1}^{n} (q_a^i - q_b^i)^2}$$

;where n is the dimension of the configuration space.

5.3 Steering Function

The steering function $\text{Steer}(q_{from}, q_{toward}, \Delta s)$ attempts to move from q_{from} toward q_{toward} by a maximum distance of Δs:

$$q_{new} = q_{from} + \Delta s \cdot \frac{q_{toward} - q_{from}}{\|q_{toward} - q_{from}\|}$$

For a point robot, this is straightforward. For more complex systems (vehicles with non-holonomic constraints), the steering function is more complex.

6 RRT Algorithm Pseudocode

6.1 Basic RRT

Listing 1: RRT Algorithm Pseudocode

```
Algorithm RRT(q_start, q_goal, max_iterations, step_size):
    Tree.add_node(q_start)

    for i = 1 to max_iterations do
        # Step 1: Sample a random configuration
        q_rand = Sample_Configuration()

        # Step 2: Find the nearest node in the tree
        q_nearest = Nearest_Node(Tree, q_rand)

        # Step 3: Steer from nearest toward random
        q_new = Steer(q_nearest, q_rand, step_size)

        # Step 4: Check for collisions
        if not Collision_Free(q_nearest, q_new) then
            continue

        # Step 5: Add the new node to the tree
        Tree.add_node(q_new)
        Tree.add_edge(q_nearest, q_new)

        # Step 6: Check if goal is reached
        if Distance(q_new, q_goal) < step_size then
            q_goal.parent = q_new
            return Path_Found(Tree, q_goal)

    return No_Path_Found
End Algorithm
```

6.2 RRT with Goal Biasing

A common extension is to bias the random sampling toward the goal with probability p:

Listing 2: RRT with Goal Biasing

```
Algorithm RRT_GoalBias(q_start, q_goal, max_iterations, step_size, goal_bias=0.1):
    Tree.add_node(q_start)
```

```
    for i = 1 to max_iterations do
        # Sample randomly with probability (1-goal_bias),
        # or sample goal with probability goal_bias
        if random() < goal_bias then
            q_rand = q_goal
        else
            q_rand = Sample_Configuration()

        q_nearest = Nearest_Node(Tree, q_rand)
        q_new = Steer(q_nearest, q_rand, step_size)

        if not Collision_Free(q_nearest, q_new) then
            continue

        Tree.add_node(q_new)
        Tree.add_edge(q_nearest, q_new)

        if Distance(q_new, q_goal) < step_size then
            q_goal.parent = q_new
            return Path_Found(Tree, q_goal)

    return No_Path_Found
End Algorithm
```

7 Visual Representation of RRT Growth

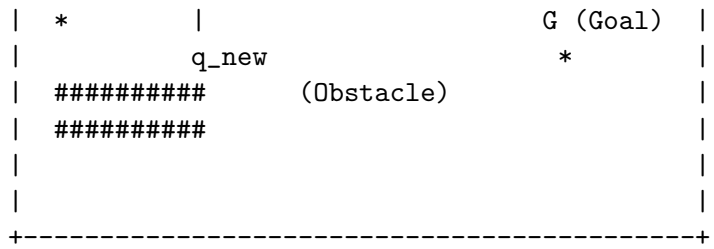
The following ASCII diagrams illustrate how RRT grows over iterations:

7.1 Initial State

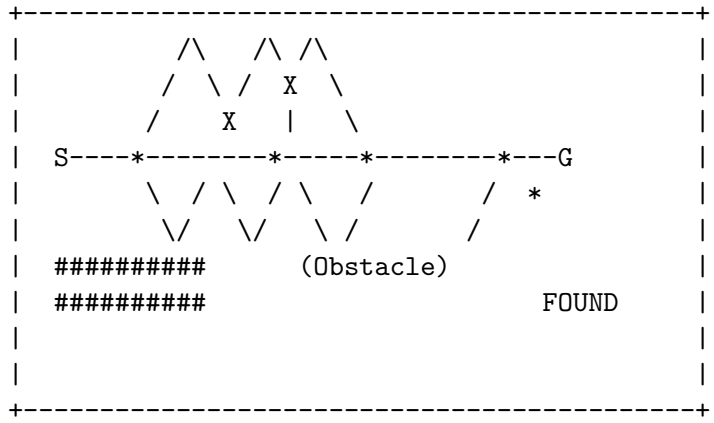
```
+-------------------------------------------+
|                                           |
|                                           |
|  S (Start)                    G (Goal)    |
|  *                            *           |
|                                           |
|  ##########      (Obstacle)               |
|  ##########                              |
|                                           |
|                                           |
+-------------------------------------------+
```

7.2 After First Sample

```
+-------------------------------------------+
|                                           |
|          q_rand                           |
|            x                              |
|  S ------ nearest, add q_new              |
```



7.3 After Multiple Iterations



The tree grows rapidly into empty space while avoiding obstacles. Eventually, it reaches the goal region.

8 RRT* (RRT Star) - The Optimized Version

8.1 Motivation

Basic RRT finds *any* solution quickly, but the path is often suboptimal (long and jagged). **RRT*** improves solution quality by:

1. Continuing to search even after finding a path
2. **Rewiring** the tree to improve path cost
3. Using a larger search radius for connections

8.2 Key Differences from RRT

Aspect	RRT	RRT*
Goal	Find any path quickly	Find optimal path
Termination	When path found	After max iterations
Connection	Nearest neighbor only	k-nearest neighbors
Rewiring	None	Yes, improves cost
Path Quality	Suboptimal (jagged)	Approaches optimal
Computation	Faster	Slower

Table 1: Comparison of RRT and RRT*

8.3 RRT* Pseudocode

Listing 3: RRT* Algorithm

```
Algorithm RRT_Star(q_start, q_goal, max_iterations, step_size):
    Tree.add_node(q_start)
    best_path_cost = infinity
    best_path = None

    for i = 1 to max_iterations do
        # Sample
        if random() < goal_bias then
            q_rand = q_goal
        else

            q_rand = Sample_Configuration()

        # Nearest neighbor
        q_nearest = Nearest_Node(Tree, q_rand)
        q_new = Steer(q_nearest, q_rand, step_size)

        if Collision_Free(q_nearest, q_new) then
            # Instead of just connecting to nearest,
            # find all neighbors within radius r
            k = ceil(log(i) / i) # Number of neighbors
```

```
            r = min(r_max, eta * (log(i) / i)^(1/d))

            neighbors = Near_Nodes(Tree, q_new, r)

            # Connect to neighbor with minimum cost
            q_min = q_nearest
            cost_min = Cost(q_nearest) + Distance(q_nearest, q_new)

            for each q_neighbor in neighbors do
                if Collision_Free(q_neighbor, q_new) then
                    cost = Cost(q_neighbor) + Distance(q_neighbor, q_new)
                    if cost < cost_min then
                        cost_min = cost
                        q_min = q_neighbor

            Tree.add_node(q_new)
            Tree.add_edge(q_min, q_new)
            Set_Cost(q_new, cost_min)

            # Rewiring: check if q_new can improve neighbors
            for each q_neighbor in neighbors do
                cost = Cost(q_new) + Distance(q_new, q_neighbor)
                if cost < Cost(q_neighbor) then
                    if Collision_Free(q_new, q_neighbor) then
                        Old_parent = Parent(q_neighbor)
                        Rewire(q_neighbor, q_new)
                        Set_Cost(q_neighbor, cost)

            # Update best path
            if Distance(q_new, q_goal) < step_size then
                path_cost = Cost(q_new) + Distance(q_new, q_goal)
                if path_cost < best_path_cost then
                    best_path_cost = path_cost
                    best_path = Path(Tree, q_new)

    return best_path
End Algorithm
```

9 Implementation Details

9.1 Nearest Neighbor Search

Finding the nearest neighbor is critical for RRT performance. Common approaches:

- **Brute Force**: Check distance to all nodes - $O(n)$ per query
- **KD-Tree**: Spatial partitioning - $O(\log n)$ average case
- **Ball Tree**: Hierarchical partitioning - $O(\log n)$ average case

For small trees ($n < 1000$), brute force is acceptable. For larger trees, use KD-tree.

9.2 Collision Detection

For a line segment from q_1 to q_2:

1. Discretize the segment: $q(t) = q_1 + t(q_2 - q_1)$ for $t \in [0, 1]$
2. Check collision at regular intervals: $t = 0, \Delta t, 2\Delta t, ..., 1$
3. If any sample collides, return collision
4. Use finer intervals for higher accuracy

9.3 Distance Metrics

For different configuration spaces:

- **Euclidean**: $d(q_a, q_b) = \|q_a - q_b\|_2$ (points, positions)
- **SO(2) (rotations)**: $d(\theta_a, \theta_b) = |\theta_a - \theta_b|$ (with wraparound)
- **SE(2) (position + orientation)**: Combined Euclidean + rotation
- **Weighted**: $d(q_a, q_b) = \sqrt{\sum w_i (q_a^i - q_b^i)^2}$

10 Performance Analysis

10.1 Asymptotic Properties

- **Probabilistic Completeness**: $P(\text{find path}) \to 1$ as iterations $\to \infty$ (if path exists)
- **Not Resolution Complete**: Unlike grid-based methods, cannot guarantee finding paths in narrow passages unless step size is appropriate
- **Computational Complexity**: $O(n \log n)$ per iteration with KD-tree, $O(n)$ with brute force

10.2 Convergence Rate

For RRT, the expected time to find a path decreases exponentially with the "visibility" of the goal from the start region. In cluttered environments, convergence can be slow.

RRT* converges to the optimal solution:

$$\lim_{n \to \infty} C(T_n) = C^* \text{ (optimal cost)}$$

However, convergence can be slow in high dimensions.

11 Parameters and Tuning

11.1 Tuning Guidelines

- **Larger step size**: Faster exploration but larger jumps, may miss narrow passages
- **Higher goal bias**: Converges faster but explores less thoroughly
- **Smaller collision resolution**: More accurate but slower
- **More iterations**: Better solution quality but slower computation

Parameter	Description	Typical Values
Δs (step size)	Max distance per extension	1-5% of workspace
max_iterations	Maximum iterations before timeout	1000-10000
goal_bias	Probability of sampling goal	0.05-0.2
collision_resolution	Distance between collision checks	0.1-0.5 units
search_radius (RRT*)	Max distance to consider neighbors	$r_{max} = 30\sqrt{\log(n)/n}$

Table 2: Key RRT Parameters

12 Comparison with Other Methods

12.1 RRT vs. Dijkstra's Algorithm

Aspect	Dijkstra	RRT
Space Type	Discrete (grid)	Continuous
Dimensions	Limited (2D-3D practical)	High dimensions (6D+)
Path Optimality	Optimal (guaranteed)	Suboptimal
Memory Usage	High (full grid)	Low (tree only)
Exploration	Systematic	Random
Narrow Passages	Requires fine grid	Depends on sampling
Non-Convex Obstacles	Handles well	Handles well

Table 3: Detailed Comparison: Dijkstra vs. RRT

12.2 RRT vs. RRT*

Aspect	RRT	RRT*
First Solution	Fast (early termination)	Slower
Final Solution	Suboptimal	Near-optimal
Complexity	Higher per iteration	Much higher
Use Cases	Any-time planning	Offline planning
Best For	Real-time systems	Optimization

Table 4: RRT vs. RRT*

13 Practical Applications

13.1 Robotics

- **Robotic Arms**: 6-7 DOF manipulators navigating in constrained environments
- **Humanoid Robots**: High-dimensional whole-body motion planning
- **Autonomous Vehicles**: Path planning in cluttered environments
- **Drones**: 3D aerial path planning with collision avoidance

13.2 Game Development and Animation

- NPC path planning in complex game worlds
- Character animation with collision avoidance
- Multi-agent crowd simulation

13.3 Other Domains

- Protein folding in computational biology
- Motion synthesis for virtual characters
- Automated inspection path planning

14 Variants and Extensions

14.1 Bidirectional RRT (BiRRT)

Grow trees from both start and goal toward each other, reducing exploration time:

Listing 4: Bidirectional RRT Concept

```
T1 grows from q_start
T2 grows from q_goal

Each iteration:
    Expand T1 with random sample
    If T1 node is close to T2:
        Try to connect to T2
        If connected: Path found!
    Swap T1 and T2
```

14.2 Informed RRT*

Uses heuristic information (e.g., straight-line distance) to guide rewiring decisions, improving convergence to optimal solution.

14.3 Batch Informed Trees (BIT*)

Combines benefits of sampling-based and graph-based methods with anytime optimization.

15 Pseudocode for Our Ship Navigation Game

Here's how RRT is implemented in our ship navigation example:

Listing 5: RRT Implementation (Java)

```
void buildRRT() {
    for (int i = 0; i < MAX_ITER; i++) {
        // Sample random point
        Node randNode = randomNode();
```

```
5
6           // Find nearest node in tree
7           Node nearest = nearestNode(randNode);
8
9           // Steer from nearest toward random point
10          Node newNode = steer(nearest, randNode);
11
12          // Check collision
13          if (!collision(nearest, newNode)) {
14              tree.add(newNode);
15
16              // Check if goal reached
17              if (dist(newNode, goal) < STEP) {
18                  goal.parent = newNode;
19                  extractPath();
20                  break;
21              }
22          }
23      }
24  }
25
26  Node steer(Node a, Node b) {
27      double theta = Math.atan2(b.y - a.y, b.x - a.x);
28      return new Node(
29          a.x + STEP * Math.cos(theta),
30          a.y + STEP * Math.sin(theta),
31          a
32      );
33  }
```

16 Challenges and Solutions

16.1 Narrow Passages

Problem: RRT struggles to find paths through narrow corridors.
Solutions:

- Use smaller step size
- Increase goal bias
- Use BiRRT or specialized planning methods

16.2 High-Dimensional Spaces

Problem: Curse of dimensionality makes random sampling inefficient.
Solutions:

- Use informed sampling (e.g., Informed RRT*)
- Exploit problem structure (subset of dimensions)
- Use better nearest neighbor structures

16.3 Real-Time Constraints

Problem: Need path in limited time.
Solutions:

- Use anytime algorithms (return best path found so far)
- Parallel RRT trees
- Precompute paths when possible

17 Conclusion

The Rapidly-Exploring Random Tree algorithm is a powerful tool for path planning in high-dimensional configuration spaces. Its key advantages are:

1. **Efficiency in High Dimensions**: Outperforms grid-based methods in 4D+ spaces
2. **Simplicity**: Straightforward to implement and understand
3. **Probabilistic Completeness**: Guaranteed to find a path (eventually)
4. **Flexibility**: Works with arbitrary obstacle geometries

While basic RRT produces suboptimal paths, variants like RRT* and Informed RRT* improve solution quality. The choice between RRT, RRT*, or other methods depends on the specific application requirements:

- **Real-time systems**: Use basic RRT with early termination
- **Offline planning**: Use RRT* for optimal solutions
- **Complex dynamics**: Use specialized steering functions
- **2D grid navigation**: Use Dijkstra or A* for guaranteed optimality

18 Resources

Some additional resources I recommend:

- A Really Good Medium Article
- A Project
- A YouTube Playlist