16-350: Planning Techniques for Robotics
Homework 2: High-DoF Arm Planning
Due: Mar 12 (Wed), 11:59pm

David Seong

Spring 2025

Professor: Maxim Likhachev

March 13, 2025

Contents

1 Introduction

This report details the implementation of planning algorithms of a multi-DOF robotic arm. The goal is to navigate the arm from a start configuration to a goal configuration while avoiding obstacles in the environment. The implemented algorithms are:

- Rapidly-exploring Random Tree (RRT)
- RRT-Connect
- RRT* (RRT-Star)
- Probabilistic Roadmap (PRM)

The performance of the algorithms is evaluated on the basis of the planning time, the cost of the route, the number of vertices to reach the goal, and the success rate of finding solutions within 5 seconds.

2 Planner Algorithms

2.1 Helper Functions

This section provides an explanation of the helper functions shared among the planner algorithms.

2.1.1 Node

A node structure was defined to represent configurations in the planning space:

```
struct Node {
    double* joint_comb;   // Joint angles
    int parent;           // Index of parent node
    double cost;          // Path cost from start
};
```

2.1.2 ExtendStatus

Classifies node-to-node extend status as `TRAPPED, ADVANCED, REACHED`

```
enum ExtendStatus {
    TRAPPED = 0,
    ADVANCED = 1,
    REACHED = 2
};
```

2.1.3 Distance Metrics

Euclidean distance between joint configurations:

```
double euclidean_distance(double* angles1, double* angles2, int
   numofDOFs) {
    double dist = 0.0;
    for (int i = 0; i < numofDOFs; i++) {
        double diff = angles1[i] - angles2[i];
        dist += diff * diff;
    }
    return sqrt(dist);
}
```

2.1.4 Path Validation

The function `isValidStraightLinePath`, `isConfigWithinBounds` check if a straight line path between two configurations is collision-free by interpolating and checking intermediate configurations separated by step size and within bounds, respectively.

2.1.5 Path Shortcutting

The function `shortcutPath` creates a "shortcut" path by skipping over intermediate nodes, given that the following node is directly reachable from the current point.

2.2 Hyper Parameters

- `K = 50000` : maximum number of nodes generated (increasing leads to a higher solution discovery rate and reduced cost for RRT*, PRM at the cost of planning time, reducing the sample number too much may lead to failure at generating path or a less optimized path.)

- `bias_check = 20` : how often to generate configuration equal to the goal (increasing leads to fewer attempts at trying to connect with the goal possibly increasing planning time and path cost, reducing may improve chances of reaching the goal faster, but being excessively biased may interfere with the planner generating enough random points to create valid intermediate points.)

- `epsilon = PI/5` : maximum distance arm configurations can be separated to be able to connect (increasing this value may cause the arm to generate more faulty points by jumping over obstacles. Although this is accounted for by `isValidStraightLinePath` function, a higher epsilon will lead to reduced valid points, which may lead to a higher chance of planning failure or a suboptimal path.

- `stepsize = PI/10` : how far robotic arm moves in visualization

- `radius`: The radius for finding nearby nodes in RRT* scales with $\gamma \cdot \log(V)/V^{1/d}$

- `max_neighbor = 10`: In PRM, each node is limited to 10 connections for maintainable graph density.
- (For both the raidus and max_neighbor, the values' increase may lead to a more accurate, optimized path due to nodes finding more optimized node options to connect and rewire to, it requires more of extend and connect functions which are expensive commands that result in high planning time.)

2.3 RRT (Rapidly-exploring Random Tree) — Planner 0

The implemented RRT algorithm works as follows:

1. Initialize a tree with the start configuration
2. Randomly sample a configuration in the joint space, biasing towards goal node every bias_check nodes
3. Find the nearest node (`nearest_neighbor`) in the tree to the sampled configuration
4. Extend the tree from the nearest node toward the sampled configuration (`extend`)
5. If the goal is reached or maximum iterations are reached, backtrack to original node and return path
6. Otherwise, go to step 2

Following helper functions have been developed and used for the RRT algorithm:

2.3.1 Nearest Neighbor Search

Finds the index of the nearest node from the constructed configuration graph.

```
std::vector<int> near_neighbors(const std::vector<Node>& nodes, int
   node_idx, double radius, int numofDOFs) {
    std::vector<int> neighbors;
    neighbors.reserve(20);

    for (int i = 0; i < nodes.size(); i++) {
        if (i == node_idx) continue;

        double dist = euclidean_distance(nodes[i].joint_comb, nodes[
           node_idx].joint_comb, numofDOFs);
        if (dist <= radius) {
            neighbors.push_back(i);
        }
    }

    return neighbors;
}
```

2.3.2 Node Extension

The `extend` function grows the tree to a randomly sampled configuration:

```cpp
ExtendStatus extend(std::vector<Node>& nodes, Node q_rand, double stepsize, int numofDOFs, double* map, int x_size, int y_size) {
        int neighbor_node_idx = nearest_neighbor(q_rand, nodes, numofDOFs);

        // q_new : step size from q_near to q_rand
        Node q_new;
        q_new.joint_comb = new double[numofDOFs];
        q_new.parent = neighbor_node_idx;

        // calculate vector to q_rand
        double magnitude = 0;
        std::vector<double> direction_vector(numofDOFs);

        for (int i = 0; i < numofDOFs; i++) {
                double diff = q_rand.joint_comb[i] - nodes[neighbor_node_idx].joint_comb[i];
                magnitude += diff * diff;
                direction_vector[i] = diff;
        }

        magnitude = sqrt(magnitude);
        bool reached = magnitude <= stepsize;

    for (int i = 0; i < numofDOFs; i++) {
        if (reached) {
            q_new.joint_comb[i] = q_rand.joint_comb[i];
        } else {
            q_new.joint_comb[i] = nodes[neighbor_node_idx].joint_comb[i] +
                                 (direction_vector[i] / magnitude) * stepsize;
        }
    }
                q_new.cost = nodes[neighbor_node_idx].cost + (reached ? magnitude : stepsize);

        // if outside of map
    if (!isConfigWithinBounds(q_new.joint_comb, numofDOFs, x_size, y_size)) {
        delete[] q_new.joint_comb;
        return TRAPPED;
    }
    //check both the configuration, transition path
    if (IsValidArmConfiguration(q_new.joint_comb, numofDOFs, map,
       x_size, y_size) &&
        isValidStraightLinePath(nodes[neighbor_node_idx].joint_comb,
            q_new.joint_comb, numofDOFs, map, x_size, y_size)) {
        nodes.push_back(q_new);
        if (reached) {
            return REACHED;
        }
        return ADVANCED;
    } else {
        delete[] q_new.joint_comb;
        return TRAPPED;
    }
}
```

2.4 RRT-Connect — Planner 1

RRT-Connect grows two trees simultaneously, one from the start configuration and one from the goal. The key addition in this algorithm is the `connect` function, which attempts to connect the two trees:

1. Initialize tree A with start configuration and tree B with goal configuration
2. Randomly sample a joint configuration
3. Find the nearest node (`nearest_neighbor`) in tree A to the sampled configuration
4. `extend` the tree from the nearest node toward the sampled configuration, given tree is not trapped.
5. `connect` to last added node in tree A from tree B
6. If the trees are connected, construct a path from start to connection node, connection node to goal node, and return the path
7. Otherwise, swap trees A, B and go to step 2

2.4.1 Tree Connection

The `extend` function attempts to connect a node from one tree to another tree effectively using the `extend` function.

```
ExtendStatus connect(std::vector<Node>& nodes, const Node& q_target,
     double stepsize, int numofDOFs, double* map, int x_size, int
    y_size) {
    ExtendStatus status = ADVANCED;

    while (status == ADVANCED) {
        status = extend(nodes, q_target, stepsize, numofDOFs, map,
            x_size, y_size);
    }

    return status;
}
```

2.5 RRT* (RRT-Star) — planner 2

RRT* is an asymptotically optimal variant of RRT that has a rewiring step at the end to minimize the path cost:

1. Extend the tree toward a random sample as in RRT
2. Find all nodes within a neighborhood of the new node
3. Rewire to parent node that minimizes the path cost
4. Continue until K nodes have been generated
5. Return final path

Key helper functions of the RRT* implementation include:

2.5.1 Near Neighbor Search

Returns indices of neighbors nodes within a set radius.

```
std::vector<int> near_neighbors(const std::vector<Node>& nodes, int
    node_idx, double radius, int numofDOFs) {
    std::vector<int> neighbors;
    neighbors.reserve(20);

    for (int i = 0; i < nodes.size(); i++) {
        if (i == node_idx) continue;

        double dist = euclidean_distance(nodes[i].joint_comb, nodes[
            node_idx].joint_comb, numofDOFs);
        if (dist <= radius) {
            neighbors.push_back(i);
        }
    }

    return neighbors;
}
```

2.5.2 Choose Parent

Chooses a single node with the lowest cost.

```
int choose_parent(std::vector<Node>& nodes, int new_node_idx, const std::vector<int>& near_indices,
                 double* map, int x_size, int y_size, int numofDOFs) {
    int min_cost_idx = nodes[new_node_idx].parent;
    double min_cost = nodes[new_node_idx].cost;

    for (int near_idx : near_indices) {
        double edge_cost = euclidean_distance(nodes[near_idx].joint_comb, nodes[new_node_idx].joint_comb, numofDOFs);
        double potential_cost = nodes[near_idx].cost + edge_cost;

        if (potential_cost < min_cost &&
            isValidStraightLinePath(nodes[near_idx].joint_comb, nodes[new_node_idx].joint_comb, numofDOFs, map, x_size, y_size)) {
            min_cost = potential_cost;
            min_cost_idx = near_idx;
        }
    }

    // update parent and cost
    nodes[new_node_idx].parent = min_cost_idx;
    nodes[new_node_idx].cost = min_cost;

    return min_cost_idx;
}
```

2.5.3 Rewire

Rewires the path according to the cost of neighboring nodes.

```
    void rewire(std::vector<Node>& nodes, int new_node_idx, const std::vector<int>& near_indices,
           double* map, int x_size, int y_size, int numofDOFs) {
    for (int near_idx : near_indices) {
        // skip parent
        if (near_idx == nodes[new_node_idx].parent) continue;

        if (isValidStraightLinePath(nodes[new_node_idx].joint_comb, nodes[near_idx].joint_comb, numofDOFs, map, x_size, y_size)) {
            double edge_cost = euclidean_distance(nodes[new_node_idx].joint_comb, nodes[near_idx].joint_comb, numofDOFs);
            double potential_cost = nodes[new_node_idx].cost +
               edge_cost;

            // rewire
            if (potential_cost < nodes[near_idx].cost) {
                // add new edge
                nodes[near_idx].parent = new_node_idx;
                nodes[near_idx].cost = potential_cost;
            }
        }
    }
}
```

2.6 PRM (Probabilistic Roadmap) — planner 3

PRM works by constructing a graph in the configuration space first and then querying for a path:

1. Sample random configurations in joint space
2. For each valid sample, connect it to nearby `max_neighbor` samples
3. Use a graph search algorithm A* and priority queue to find a path
4. Once a path between the start and goal is found, return the path

3 Experimental Results

4 planners were evaluated in 5 randomly generated 3-7 DoF start-goal arm configuration pairs for map2.txt, running each pair 4 times per planner.

Config 1 (3 DoF) Start: 0.392699,2.356194,3.141592 End: 1.570796,0.785398,1.570796						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	2	2	2	2	2	0.00
Cost	4.32	4.32	4.32	4.32	4.32	0.00
TimeSpent	0.0058	0.0064	0.0064	0.00544	0.00601	0.00
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	2	2	2	2	2	0.00
Cost	4.32	4.32	4.32	4.32	4.32	0.00
TimeSpent	0.0035	0.0046	0.00353	0.0048	0.0041075	0.00
Success Rate	1	1	1	1	1	0.00
RRT*						
Num Steps	2	2	2	2	2	0.00
Cost	4.32	4.32	4.32	4.32	4.32	0.00
TimeSpent	2.24	2.08	2.13	2.01	2.115	0.10
Success Rate	1	1	1	1	1	0.00
PRM						
Num Steps	2	2	2	2	2	0.00
Cost	4.32	4.32	4.32	4.32	4.32	0.00
TimeSpent	0.0184	0.018	0.0188	0.018	0.0183	0.00
Success Rate	1	1	1	1	1	0.00

Table 1: Performance comparison for Config 1 (3 DoF)

Config 2 (4 DoF) Start: 0.87,5.62,3.27,0.76 End: 1.14,2.19,3.81,0.16						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	3	6	3	3	3.75	1.50
Cost	6.7	13.19	6.93	6.84	8.42	3.18
TimeSpent	0.047	0.0069	0.0335	0.032	0.02985	0.02
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	4	5	4	4	4.25	0.5
Cost	5.82	7.12	5.84	5.75	6.13	0.66
TimeSpent	0.449	0.466	0.428	0.448	0.44775	0.02
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	3	3	3	3	3	0
Cost	7.012	6.73	7.16	7.22	7.03	0.22
TimeSpent	0.0488	0.035	0.0354	0.03	0.0373	0.01
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	4.26	4.26	4.26	4.263	4.26	0.00
TimeSpent	4.01	4.22	4.16	4.222	4.153	0.10
Success Rate	1	1	1	1	1	0

Table 2: Performance comparison for Config 2 (4 DoF)

Config 3 (5 DoF) Start: 1.32,3.12,6.15,1.65,3.27 End: 1.00,2.51,6.10,5.24,2.93						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	3	2	3	2	2.5	0.58
Cost	4.84	4.01	4.8577	4.013	4.43	0.48
TimeSpent	1.66	1.67	1.663	1.64	1.65825	0.01
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	6	7	3	3	4.75	2.06
Cost	7.26	9.77	7.15	13.57	9.44	3.01
TimeSpent	0.511	0.486	0.0354	0.00488	0.25932	0.28
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	4	2	2	2	2	0
Cost	4.21	4.013	4.013	4.013	4.06	0.10
TimeSpent	2.22	2.2226	3.065	2.215	2.43	0.42
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	4.013	4.013	4.013	4.013	4.01	0.00
TimeSpent	2.11	2.094	2.094	2.069	2.09175	0.02
Success Rate	1	1	1	1	1	0

Table 3: Performance comparison for Config 3 (5 DoF)

Config 4 (6 DoF) Start: 1.02,5.70,3.44,0.84,2.01,4.61 End:1.18,1.37,4.83,2.94,1.52,4.64						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	13	3	3	3	5.5	5.00
Cost	41.91	8.514	8.39	9.83	17.16	16.51
TimeSpent	2.04	4	0.0386	0.0387	1.529325	1.90
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	4	4	2	6	4	1.63
Cost	8.38	8.095	6.12	25.109	11.93	8.85
TimeSpent	0.6432	0.612	0.569	0.04	0.46605	0.29
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	3	3	4	3	3.25	0.50
Cost	8.32	8.066	10.164	10.17	9.18	1.14
TimeSpent	0.0712	0.078	0.05263	0.04896	0.0627	0.01
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	6.12	6.12	6.12	6.123	6.12	0.00
TimeSpent	1.22	1.125	1.079	1.1145	1.13	0.06
Success Rate	1	1	1	1	1	1

Table 4: Performance comparison for Config 4 (6 DoF)

Config 5 (7 DoF) Start: 1.44,2.10,0.06,2.92,6.03,1.76,6.10 End: 1.18,1.37,4.83,2.94,1.52,4.64						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	3	3	3	3	3	0.00
Cost	13.5	13.86	13.7	14.24	13.83	0.31
TimeSpent	2.22	2.37	2.2026	2.398	2.29765	0.10
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	9	12	11	13	11.25	1.71
Cost	43.88	47.37	50.97	49	47.81	3.00
TimeSpent	0.24	0.297	0.31	0.116	0.24075	0.09
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	3	3	3	4	3.25	0.50
Cost	13.74	13.66	13.7	14.27	14.84	0.29
TimeSpent	3.15	3.36	3.27	3.07	3.2125	0.13
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	12.38	12.38	12.383	12.38	12.38	0.00
TimeSpent	0.59	0.6	0.587	0.61	0.59675	0.01
Success Rate	1	1	1	1	1	0

Table 5: Performance comparison for Config 5 (7 DoF)

3.1 Discussion of Experimental Results

Based on the data from the experiment with all configurations (3-7 DoF):

3.1.1 Planning Time

- RRT-Connect consistently had a fast planning time due to its two-end extension nature.
- RRT* and PRM generally have a slower planning time as they don't have an early break and are forced through generate K number of samples before returning a path
- RRT's performance is randomized due to its random nature

3.1.2 Path Cost

- PRM tends to find the most optimal paths and with a low standard deviation. This is due to its two step planning where it first generates a map and uses A* to find the most optimal path
- RRT* also consistently finds a relatively optimal path due to its continuously optimizing nature
- RRT, RRT-Connect's performance are heavily randomized due to their randomly expanding process

3.1.3 Number of Steps

- PRM consistently uses the fewest steps across all configurations
- RRT-Connect performs better than RRT-connect and RRT but it requires more steps as DoF increases
- RRT and RRT-connect have a high step count due to their randomly expanding nature.

3.1.4 Success

All planners were able to reach their goal and successfully plan a path in 5 seconds. Looking at their standard deviations, RRT has the highest, followed by RRT-Connect also due to their random sampling process. Whle RRT* and PRM also start with random sampling, their rewiring and optimized graph search enable them to converge to a optimized path that mostly converges to a similar cost path.

3.2 Extra Credit

3.2.1 RRT* vs RRT* first solution (Configurations 4, 5, 6)

To compare RRT*'s first solution path with RRT * and other planners, its performance was measured the same as other in Tables 1-5.

Config 3 (5 DoF) Start: 1.32,3.12,6.15,1.65,3.27 End: 1.00,2.51,6.10,5.24,2.93		
	Mean	Standard Dev
RRT*		
Num Steps	2	0.00
Cost	4.06	0.10
TimeSpent	2.43065	0.42
Success Rate	1	0.00
RRT* (first)		
Num Steps	2.25	0.50
Cost	4.68	1.34
TimeSpent	1.70175	0.06
Success Rate	1	0.00

Table 6: Comparison between standard RRT* and RRT* (first path) for Config 3 (5 DoF)

Config 4 (6 DoF) Start: 1.02,5.70,3.44,0.84,2.01,4.61 End: 1.18,1.37,4.83,2.94,1.52,4.64		
	Mean	Standard Dev
RRT*		
Num Steps	3.25	0.50
Cost	9.18	1.14
TimeSpent	0.0626975	0.01
Success Rate	1	0.00
RRT* (first)		
Num Steps	3.75	1.50
Cost	14.14	9.52
TimeSpent	0.0537175	0.03
Success Rate	1	0.00

Table 7: Comparison between standard RRT* and RRT* (first path) for Config 4 (6 DoF)

Config 5 (7 DoF) Start: 1.44,2.10,0.06,2.92,6.03,1.76,6.10 End: 0.65,0.36,1.83,2.45,3.91,5.56,3.0		
	Mean	Standard Dev
RRT*		
Num Steps	3.25	0.50
Cost	13.84	0.29
TimeSpent	3.2125	0.13
Success Rate	1	0.00
RRT* (first)		
Num Steps	5	4.00
Cost	22.00	16.13
TimeSpent	2.37	0.03
Success Rate	1	0.00

Table 8: Comparison between standard RRT* and RRT* (first path) for Config 5 (7 DoF)

From Tables 6, 7, 8, it can be seen that while the RRT* first step reaches a solution in a shorter period of time, it does so at the cost of the path cost and number of steps. Furthermore, its cost standard deviation is significantly increased due to its randomness, and its values are close to RRT, which again validates how RRT*'s rewiring process optimizes the path at the cost of planning time.

3.2.2 Similar Configurations for Consistency (5,6,7)

To validate the robustness and consistency of the planners, two similar configuration pairs were put to the test with configuration 5 - start: 1.32,3.12,6.15,1.65,3.27, end: 1.00,2.51,6.10,5.24,2.93 and start: 1.2,3.12,6.15,1.65,3.27, end: 1.00,2.51,6.10,5.24,2.93.

Config 3 (5 DoF) Start: 1.32,3.12,6.15,1.65,3.27 End: 1.00,2.51,6.10,5.24,2.93						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	3	2	3	3	2.75	0.50
Cost	5	4.01	4.83	4.83	4.67	0.44
TimeSpent	2.13	2.25	2.09	2.23	2.17475	0.08
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	3	3	4	5	3.75	0.96
Cost	14.24	15.45	4.80	6.14	10.16	5.46
TimeSpent	0.005	0.0067	0.65	0.0052	0.166725	0.32
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	2	2	2	2	2	0.00
Cost	4.01	4.01	4.01	4.01	4.01	0.00
TimeSpent	2.26	2.18	2.21	2.21	2.215	0.03
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	4.01	4.01	4.01	4.01	4.01	0.00
TimeSpent	2.78	2.79	2.59	2.79	2.74	0.10
Success Rate	1	1	1	1	1	0

Table 9: Similar Config Test - First Dataset

Config 3 (5 DoF) Start: 1.2,3.12,6.15,1.65,3.27 End: 1.00,2.51,6.10,5.24,2.93						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	3	3	3	3	3	0.00
Cost	5.36	5.232	4.92	6.6	5.53	0.74
TimeSpent	2.10	2.22	2.18	2.17	2.17	0.05
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	8	5	3	3	4.75	2.36
Cost	11.35	6.42	14.24	14.6	11.65	3.78
TimeSpent	0.64	0.65	0.01	0.0067	0.33	0.37
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	2	2	2	2	2	0.00
Cost	3.89	3.89	3.89	3.89	3.89	0.00
TimeSpent	2.26	2.22	2.14	2.2	2.20	0.05
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	3.89	3.89	3.89	3.89	3.89	0.00
TimeSpent	2.72	2.66	2.98	2.8	2.79	0.14
Success Rate	1	1	1	1	1	0

Table 10: Similar Config Test - Second Dataset

The results again show the already proven results that RRT is the fastest, yet most widely distributed, and RRT* and PRM are consistent and optimized in terms of cost and number of steps at the cost of planning time.

3.2.3 Reverse Configurations for Consistency (5,6,7)

A reversed configuration of the original configuration 3, 5 DoF, was graded to validate the planners' robustness and consistency.

Config 3 (5 DoF) - Reversed Start: 1.00,2.51,6.10,5.24,2.93 End: 1.32,3.12,6.15,1.65,3.27						
	Trial 1	Trial 2	Trial 3	Trial 4	Mean	Standard Dev
RRT						
Num Steps	3	3	3	3	3	0.00
Cost	8.4	11.293	8.162	8.14	9.88	1.39
TimeSpent	0.09	0.00	0.09	0.08	0.09	0.04
Success Rate	1	1	1	1	1	0.00
RRT-Connect						
Num Steps	6	3	5	4	3	1.30
Cost	10.15	14.53	14.41	13.31	11.32	1.94
TimeSpent	0.68	0.01	0.01	0.008	0.01	0.30
Success Rate	1	1	1	1	1	0
RRT*						
Num Steps	3	3	3	3	3	0.00
Cost	10.03	5.75	7.97	6.14	6.40	1.76
TimeSpent	2.16	2.33	2.63	2.01	2.20	0.23
Success Rate	1	1	1	1	1	0
PRM						
Num Steps	2	2	2	2	2	0
Cost	4.01	4.01	4.01	4.01	4.01	0.00
TimeSpent	2.90	2.8	2.67	2.65	3.04	0.16
Success Rate	1	1	1	1	1	0

Table 11: Performance comparison for Config 3 (5 DoF) with reversed start and goal positions

From this test, it was again determined that PRM and RRT* are the more optimized planners, and RRT-Connect and RRT are the faster, yet more random planners. Another observation was that although RRT may be more optimized, it is often less optimized than PRM because it uses some nodes for biased node generation and possible "rewiring" instead of creating a graph of the whole map and then finding the most optimized path like PRM.

3.3 Conclusions

- For time-critical applications: RRT-Connect offers the best balanced performance (speed and cost)

- For applications where path quality is paramount: RRT* is the best choice if planning time constraints are relaxed
- For some test trials it can be seen that some planners that are expected to perform better/ worse than others actually does the opposite. This is due to the `shortcutPath` function that optimizes the returned path, improving the cost of paths that normally were suboptimal.

3.4 Potential Improvements

Future work could focus on the following improvements:

1. **Optimized Nearest Neighbor Search**: Implementing a KD-tree or other spatial data structure could significantly improve performance for all algorithms.
2. **Optimized Hyper Parameters**: Hyper parameters such as step size, number of samples affect the quality of solutions significantly. These values can be further tested and optimized to find a better combination of hyper parameters.
3. **RRT*-Connect**: Combining the speed of RRT-Connect with the optimality properties of RRT* could yield better overall performance.

4 Compilation Instructions

The code can be compiled with the following command:

```
g++ -std=c++17 planner.cpp -o planner.out
```

To run the planner with a specific configuration:

```
./planner.out map1.txt 5 1.57,0.78,1.57,0.78,1.57
    0.392,2.35,3.14,2.82,4.71 2 myOutput.txt
```

To visualize the output:

```
python visualizer.py myOutput.txt --gifFilepath=myGif.gif
```