Trajectory Planning of a 2D Omni-Directional Robot using RRT Connect and RRT*

David Bick[1], Serris Lew[2], and Shashank Ojha[3]

Abstract— **In this paper we describe a specific trajectory planning scenario of an omni-directional circular robot with radius r. Given a map filled with convex obstacles specified with their vertices, we generate a feasible trajectory from the initial position to the goal position. We implement and explore the performance in terms of compute time and path distance of both the RRT-Connect and RRT* algorithms. Our results show that RRT* generates much more optimal paths but comes at a cost of computational time. On the other hand, RRT-Connect plans rapidly and has the advantage of being able to discover narrow corridors. It may be worth exploring using the path of RRT-Connect as a heuristic for RRT* for future work.**

I. INTRODUCTION

In this paper, we explore the performance of RRT Connect and RRT* in Trajectory Planning. Formally, we define the problem as follows. We are given the following:

- Initial position of the robot (x_{start}, y_{start})
- Goal position of the robot (x_{goal}, y_{goal})
- Radius r of the robot
- The height and width of our space denoted as H and W respectively
- A list of obstacles given as convex polygons, where each polygon is specified using an ordered list of (x, y) vertices representing the convex hull

Now given these inputs, we wish to find a feasible trajectory for an omni-directional robot. Ideally, we want to find the trajectory with shortest path distance, but we focus on feasibility first.

Our approach to solving this is, as mentioned earlier, RRT Connect and and RRT*. We are interested in how they perform in practice and what the advantages and disadvantages of each are. In addition, we were interested in learning about the obstacle collision problems mentioned in a future section.

II. BACKGROUND

A. Probabilistic Roadmaps

Often times in planning the most important choice we can make is how we're going to represent our search space. Some possible choices are grids, visibility graphs, lattice graphs, or voronoi diagrams. Unfortunately, the issue with all these approaches is that they don't scale to higher dimensions. For example, planning for a 6-DOF robot arm is too expensive to do with any of the above. All of the potential choices above have drawbacks in either computational time or space that make planning in higher dimensions infeasible.

Instead people lean to probabilistic roadmaps for such huge search spaces. Probabilisitic roadmaps make use of random samples to construct a graph of the underlying configuration space. These graphs by nature are generally very sparse with respect to the true configuration space. This allows us to easily represent them in memory as desired. While there are many different algorithms that each have their own advantages to construct these graphs, we focus on RRT Connect and RRT*. We describe each in the next two sections. Once the graph has been constructed and we know there exists a path from the initial position to the goal, we simply perform an A* search on our graph to find the actual path. These paths are generally pretty jerky, but that's not of interest in this paper.

B. RRT Connect

A rapidly exploring random tree (RRT) is a search algorithm that constructs a tree from random samples in the search space. Often utilized for robotic path planning, it handles obstacles well and is designed to be biased and grow towards unsearched areas. At each sample, the tree attempts to extend to it with its nearest node; it does so by moving an incremental distance in the direction of the random sample.

RRT-Connect involves two RRTs rooted at the start and goal configurations, taking turns attempting to extend and connect to one another. A key difference between RRT-Connect and the basic RRT algorithm is that the former includes a Connect heuristic that continues to extend itself until it reaches an obstacle or reaches the goal. Figure 1 shows the RRT-Connect algorithm [1] as it initially starts with two RRTs and at each iteration, a random configuration will allow one tree to explore the space and the other tree to extend in that direction as much as possible. The two trees are swapped and the other tree can now sample a new configuration to extend. This repeats until the two RRTs are connected, which ultimately result in a path between the start and goal configurations.

C. RRT*

RRT* is very similar in many ways, but there are a few differences that we note. First of all, RRT* is an extension of normal RRT that doesn't do the connection part mentioned in the RRT Connect section. Secondly, when extending the

*This work was not supported by any organization

[1]David Bick is a 4th year undergraduate studying Statistics and Machine Learning at Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA `dbick@andrew.cmu.edu`

[2]Serris Lew is a 4th year undergraduate studying Electrical and Computer Engineering at Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA `snlew@andrew.cmu.edu`

[3]Shashank Ojha is a 4th year undergraduate studying Artificial Intelligence at Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA `shashanko@cmu.edu`

```
EXTEND($\mathcal{T}, q$)
1  $q_{near} \leftarrow$ NEAREST_NEIGHBOR($q, \mathcal{T}$);
2  if NEW_CONFIG($q, q_{near}, q_{new}$) then
3      $\mathcal{T}$.add_vertex($q_{new}$);
4      $\mathcal{T}$.add_edge($q_{near}, q_{new}$);
5      if $q_{new} = q$ then
6          Return Reached;
7      else
8          Return Advanced;
9  Return Trapped;
```

```
CONNECT($\mathcal{T}, q$)
1  repeat
2      $S \leftarrow$ EXTEND($\mathcal{T}, q$);
3  until not ($S =$ Advanced)
4  Return $S$;
```

```
RRT_CONNECT_PLANNER($q_{init}, q_{goal}$)
1  $\mathcal{T}_a$.init($q_{init}$); $\mathcal{T}_b$.init($q_{goal}$);
2  for $k = 1$ to $K$ do
3      $q_{rand} \leftarrow$ RANDOM_CONFIG();
4      if not (EXTEND($\mathcal{T}_a, q_{rand}$) = Trapped) then
5          if (CONNECT($\mathcal{T}_b, q_{new}$) = Reached) then
6              Return PATH($\mathcal{T}_a, \mathcal{T}_b$);
7      SWAP($\mathcal{T}_a, \mathcal{T}_b$);
8  Return Failure
```

Fig. 1. RRT-Connect Algorithm

```
Main
$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$; $i \leftarrow 0$;
while $i < N$ do
    $G \leftarrow (V, E)$;
    $x_{\text{rand}} \leftarrow$ Sample($i$); $i \leftarrow i + 1$;
    $(V, E) \leftarrow$ Extend($G, x_{\text{rand}}$);
```

```
$Connect_{RRT^*}(G, x)$
$V' \leftarrow V$; $E' \leftarrow E$;
$x_{\text{nearest}} \leftarrow$ Nearest($G, x$);
$x_{\text{new}} \leftarrow$ Steer($x_{\text{nearest}}, x$);
if ObstacleFree($x_{\text{nearest}}, x_{\text{new}}$) then
    $V' \leftarrow V' \cup \{x_{\text{new}}\}$;
    $x_{\text{min}} \leftarrow x_{\text{nearest}}$;
    $X_{\text{near}} \leftarrow$ Near($G, x_{\text{new}}, |V|$);
    for all $x_{\text{near}} \in X_{\text{near}}$ do
        if ObstacleFree($x_{\text{near}}, x_{\text{new}}$) then
            $c' \leftarrow$ Cost($x_{\text{near}}$) $+ c$(Line($x_{\text{near}}, x_{\text{new}}$));
            if $c' <$ Cost($x_{\text{new}}$) then
                $x_{\text{min}} \leftarrow x_{\text{near}}$;
    $E' \leftarrow E' \cup \{(x_{\text{min}}, x_{\text{new}})\}$;
    for all $x_{near} \in X_{\text{near}} \setminus \{x_{\text{min}}\}$ do
        if ObstacleFree($x_{\text{new}}, x_{\text{near}}$) and
        Cost($x_{\text{near}}$) > Cost($x_{\text{new}}$) $+ c$(Line($x_{\text{new}}, x_{\text{near}}$))
        then
            $x_{\text{parent}} \leftarrow$ Parent($x_{\text{near}}$);
            $E' \leftarrow E' \setminus \{(x_{\text{parent}}, x_{\text{near}})\}$;
            $E' \leftarrow E' \cup \{(x_{\text{new}}, x_{\text{near}})\}$;
return $G' = (V', E')$
```

Fig. 2. RRT-Connect Algorithm

tree towards a new random configuration, we don't simply extend from the nearest neighbor. We do create our target configuration based on the nearest neighbor, but then we consider all configurations within some specific radius r of that target configuration. For all such configurations c_i, we compute the path from the initial point to the target point passing through c_i. If we find a more optimal path than the nearest neighbor, we instead extend towards the new point through the c_{min}. Now given this new branch, we re-evaluate the configurations c_i to check if there is a shorter path from the initial configuration to c_i passing through the new configuration. If there is, we rewire the path c_i by changing its original parent to the new configuration. RRT* is generally more expensive per time step because of the additional rewiring, but it is known for generating much more optimal paths. The RRT* Algorithm [2] is shown in Figure 2.

III. HANDLING OBSTACLES

As mentioned in the introduction, we've specified our obstacles using the convex hull. This is because this sparse representation is used by many state of the art systems today. It is very memory efficient and is sufficient in describing everything we need to know about the obstacle.

A. Convex Assumption

We assume our obstacles are convex in order to take advantage of some computational geometry algorithms. It's easier to check whether a point lies within a convex polygon than any other shape. It's also simple to check user input to confirm whether a given obstacle is indeed convex.

B. Computational Geometry

There are a couple of computational geometry algorithms we used. We briefly go over each here.

- **Line Side Test:**
  The line side test allows us to determine whether a given point P is LEFT or RIGHT of a line passing through two points A and B. If neither is true, then the point is ON the line.
  The test works as follows. We first construct vectors AP and AB. Then we compute $AB \times AP$. If the cross product is positive, we return LEFT. If the cross product is negative, we return RIGHT. Otherwise, we return ON.
- **Segment Intersection Test:**
  The segment interaction test is used to determine if two line segments AB and CD intersect. It is very simple given the line side test. We simply need to check if points A and B are on opposite sides of the line passing through C and D and then also check if points C and

D are on opposite sides of the line passing through A and B.

- **Convex Hull Test:**
  The convex hull test is used to determine if an ordered set of points - the convex hull - is indeed convex a convex hull. The intuition behind this test is that if we were to walk around the perimeter of a convex hull in a counter-clockwise fashion, then it should never be the case that we turn right.
  Thus, to check if an ordered list of n points forms a convex hull, we simply check that the point P_{i+2} is to the left of the line passing though P_i and P_{i+1} for all $0 \leq i \leq n$. We will have to wrap around our indices for the last two iterations.
- **Interior Point Test:**
  The interior point test is used to determine whether a point P is interior to a given convex hull. The intuition behind this test is that if I were to again walk counter-clockwise along the perimeter of my polygon, then the point P will always be to the left. This can easily be checked by using the line side test for each edge of the polygon.

Given these functions, we can quickly determine if a sampled point collides with an obstacle using the interior point test and whether a path crosses one of the obstacles edges using the segment intersection test.

C. Minkowski Sum

The Minkowski sum is used to handle the fact that robot is not a point, but a physical object that indeed takes up space. In our case, we've chosen our robot to be a circle with radius r. This complicates our planning because we don't want to check for every point on our path whether there is an obstacle at range r. Instead, it's easier to preprocess the obstacles by expanding it by r and then plan as if our robot was a point once again. To do this, we invoke the Minkowski Sum. The mathematical definition of this is as follows. Let A and B be two sets. Then our Minkowski addition given by the set $A + B$ is

$$A + B = \{\mathbf{a} + \mathbf{b} | \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}\}$$

In addition to this, we also make use of another important property as our entire space is represented by convex hulls. Specifically,

$$Conv(A + B) = Conv(A) + Conv(B)$$

where $Conv(X)$ is the convex hull of the set X. The above statement is saying that for all non-empty subsets A and B of a real vector space, the convex hull of their Minkowski sum is the Minkowski sum of their convex hulls. This is important since our obstacles are denoted by the convex hull.

In our case, although we have the convex hull of an obstacle A, we don't exactly have a convex hull of the robot since it is defined by a circle which would have an infinite amount of points in its convex hull. Thus, we bound our robot with a square with side lengths $2r$. Note that now we can represent our second convex hull set B with just 4 points. This allows us to perform the Minkowski sum easily.

IV. NEAREST NEIGHBOR

The main distinction between RRT and the original random trees, which failed to explore the space very well, is the use of nearest neighbors. Regular random trees would sample from the space and connect it to the tree. RRT's instead extend from the nearest neighbor of the tree to create a fixed length branch in the direction of the sample.

Starting from the nearest neighbor causes to the tree to spider out in a way that covers a lot more territory because the branches extend rather than the center. A dense exploration cannot cover volume as quickly as a more sparse exploration.

To find the nearest neighbor of the RRT to the sampled point, we implemented a k-d tree data structure. The k-d tree is essentially a binary tree in k dimensions. The tree stores points with smaller values at the current dimension to the left, and larger to the right, as in a standard binary tree. To generalize to k-dimensions, you compare the test node with the current tree node at the depth % k dimension, assuming the depth of the first node is 0, and each subsequent level is one level deeper. Essentially you start by comparing with the first dimension, then the second dimension, and continue across, and cycle back to the first dimension after reaching the final dimension.

Every point in the RRT was stored in the k-d tree, so after finding the nearest neighbor of the sample point, we inserted the new point into the tree. The k-d tree generally allows us to ignore half of the space along the axis we are looking at, which is crucial to increase the efficiency of the nearest neighbor operation. We run this algorithm for every point, so it must be fast for our algorithm to be fast.

One other important detail is that is possible that the distance between the test point and the current point in the k-d tree is greater than the distance of the test point to any part of the hyper-plane that the current point forms in the space. For example, the test point could be quite far from the point in the tree, but much closer to the boundary hyper-plane, which could mean that there is a point just on the other side of the hyper-plane that is closer to the test point. Therefore, if the distance between the test point and the current tree point is greater than the distance between the test point and the hyper-plane, we also search the opposite side of the hyper-plane. This will ensure correctness, at the cost of some efficiency.

V. RESULTS

We've written all our code in C++. Most of it is written from scratch besides some of the built-in STL data structures and GLUT toolkit, which helped us visualize our results. The entire codebase can be found here at `https://github.com/Shashank-Ojha/RRT-Trajectory-Planner`. The repository includes a README.md that documents how to compile and run the code. We've provided 2 maps to test the code, but more can easily be added by following $template.txt$ in the $maps/$ folder.

In order to measure performance we used three criteria: the path length, planning time, and # nodes in final graph. This allowed us to tell which algorithm gave the better path and which algorithm was faster. The last criteria was also an indicator of how many nodes were needed to generate a result. We've summarized these statistics in Table I. It shows the averages taken over 100 trials on the same Map 2.

TABLE I

SUMMARY OF RRT CONNECT AND RRT* PERFORMANCE ON MAP 2

Average Value	RRT Connect	RRT*
Path Length	115.37 ± 11.652	19.04 ± 6.501
Planning Time (ms)	2.237 ± 0.463	76.81 ± 59.263
# Nodes in Final Tree	220.06 ± 34.31	1057.21 ± 571.792

Note that RRT* had a goal bias of 0.1.

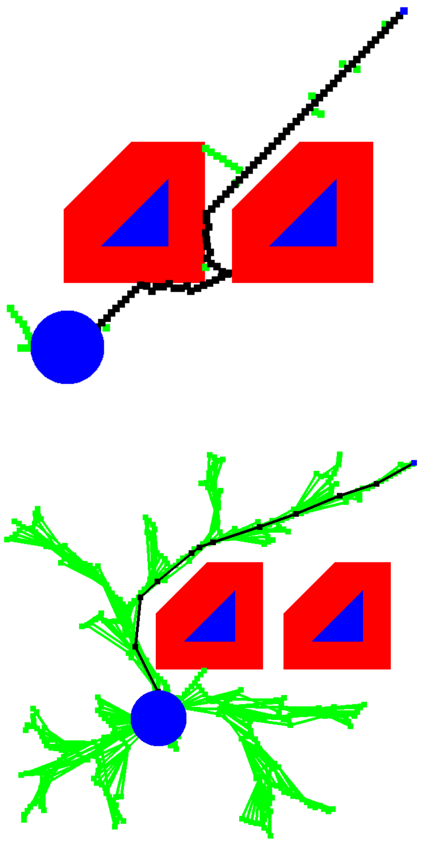

Fig. 3. Results on Map 1. The first picture denotes the output of RRT-Connect whereas the second picture denotes the output of RRT-Star. The blue circle represents the robot at its starting position. The green branches represent the generated tree for the respective algorithms. The other blue shapes represent the obstacles on the map while the red boundary around it shows the expansion using the Minkowski Sum. The final path is shown in black.

From Table I, we can see that on average RRT connect runs significantly faster - about $40x$. However, the path length is also much worse. RRT* tends to generate more optimal paths that was almost 6 times shorter. This was the expected result as RRT* has the additional rewiring step that leads to more straighter and optimal paths but comes at a huge computational cost. Thus, we expected RRT* to be slower but more optimal.

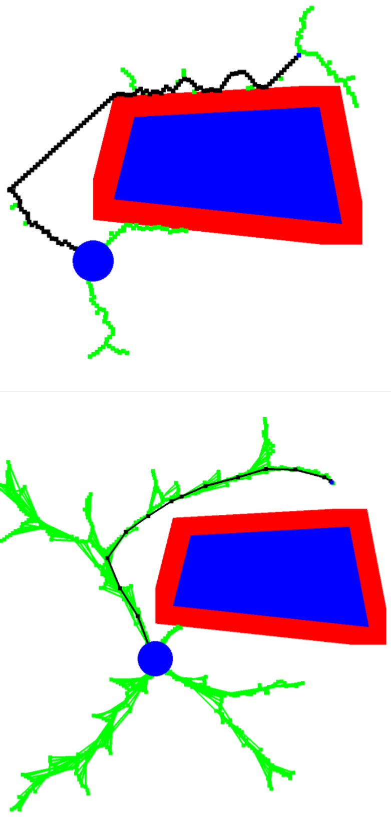

Fig. 4. Results on Map 2. The first picture denotes the output of RRT-Connect whereas the second picture denotes the output of RRT-Star.

In addition to the statistics, it's also helpful to visualize some of the results. In Figure 3, we can see the results of RRT-Connect and RRT* on our first map. We can see that the underlying tree generated is much larger for RRT*. In addition, the path given by RRT* is much smoother and straighter. In contrast the final path given by RRT-connect is very unstable and jerky. Nevertheless, the visualization on the first map shows us a key insight into the RRT-connect algorithm. There is a very narrow corridor of legal configuration space between the two obstacles. As we can see that in RRT*, this corridor goes unexplored. With RRT-connect, however, the final path generated is through that corridor. This is the power of RRT connect. If there is a light of sight between the start tree and goal trees, RRT does a pretty good job of finding it.

In Figure 4, we see another example where RRT-connect gives a very noisy path whereas RRT* provides a much smoother and reasonble one.

VI. DISCUSSION AND FURTHER WORK

From our results, we conclude that indeed RRT* can give us much smoother and straighter paths to the goal with the expense of more compute time. Nevertheless, RRT* doesn't do well in exploring the entire search space as was seen in the results from Map 1. When it comes to just finding a feasible path, it may be better to use RRT-connect. One possible extension to this work could be to see whether first using RRT-Connect to find a feasible path and then using RRT* with a bias along the RRT-connect's path could be used to generate more optimal paths. This seems promising given the results from Map 1. In addition, it might be worthwhile to see how these results hold up in higher dimensional planning. It is questionable if RRT* is even feasible in higher dimensions given the significant additional computational cost required.

ACKNOWLEDGMENT

We'd like to thank our professor Chris Atkeson for guiding us through this project. He's the boy. Thanks for providing a unique voice and embodying authenticity.

REFERENCES

[1] J. J. Kuffner, Jr and S. M. LaValle. RRT-Connect: An Efficient Approach to Single-Query Path Planning. In Proc. 2000 IEEE Int'l Conf. on Robotics and Automation (ICRA 2000)

[2] S Karaman and E. Frazzoli. Incremental Sampling-based Algorithms for Optimal Motion Planning. In International Journal of Robotics Research, 2000.