Computational Robotics, Fall 2019

Lab 2 - RRT Planner

Andrew Choi, Viacheslav Inderiakin, Yuki Shirai, Yusuke Tanaka

November 2019

1 Introduction

The goal of this lab was to build autonomy into a simple two-wheeled differential drive non-holonomic robot. Using software constructed environments, the robot was shown to be able to find a valid path from an initial starting point to a desired goal state while avoiding all obstacles. Two sampling-based algorithms were used for path planning, RRT and RRT*. Although both methods were capable of finding valid paths in relatively complex environments, a trade-off between finding an optimal trajectory and computational expense was observed between the two methods.

2 Mathematical Formulation

Before delving into the details of the experiment, the kinematics of the robot as well as the path planning algorithms themselves are discussed mathematically.

2.1 Differential Drive Kinematics

A two-wheeled robot that employs differential drive simply describes a robot whose wheels are never steered. Instead, to invoke rotation, the angular wheel velocities are varied. Following this, by using the center point between the wheels as the reference point, the following transition equations can be constructed

$$\dot{x} = \frac{r}{2}(u_l + u_r)cos\theta \tag{1}$$

$$\dot{y} = \frac{r}{2}(u_l + u_r)sin\theta \tag{2}$$

$$\dot{\theta} = \frac{r}{L}(u_r - u_l) \tag{3}$$

where r is the wheel radius, u_r is the right wheel angular velocity (rad/s), u_l is the left wheel angular velocity (rad/s), L is the distance between the two wheels, and θ is the angular position of the robot. From this, it can be inferred that when the angular wheel velocities are equal, pure translation occurs while when the angular wheel velocities are at opposite velocities, pure rotation occurs as shown in figure 1[1]. These equations form the basis for the drive policy of the robot which is discussed later in section 3.4.

Then, from these equations, the dimensionality of the input is 2, dimensionality of the configuration space is 3, and dimensionality of the operational space is 2.

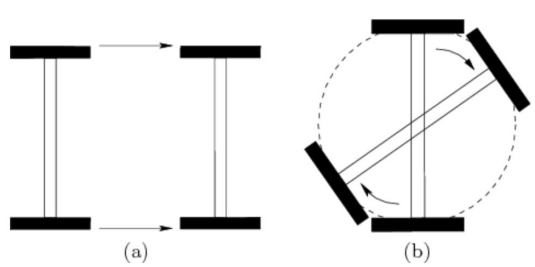

Figure 1: Pure translation and rotation in differential drive kinematics [1]

2.2 Rapidly-Exploring Random Trees: RRT

As the robot resides in a continuous state and action system, sampling-based methods were used for path planning with obstacles. More specifically, the Rapidly Exploring Random Trees (abbreviated RRT from hereon) algorithm was employed. A widely used algorithm for robot motion planning, this algorithm efficiently builds a space-filling tree in high-dimensional spaces where each node is a reachable state and all edges are a sequence of actions/states from node to node.

This tree is constructed by randomly sampling from the state space. From these sampled points, so long as they are a valid state for the robot, the nearest explored node is then used as the start state. From this start state, the robot then attempts to drive towards the sampled point for a certain amount of time. Whichever state that the robot ends up in after the time is up is then added as the new node while the trajectory is added as the edge so long as the whole trajectory was collision-free.

When used in conjunction with a computed object configuration space C_{obs}, RRT easily handles environments with obstacles as well as any differential constraints such as the differential drive kinematics of the robot making it an efficient choice for path planning in our setup. In addition to this, due to the Voronoi regions created by the constructed tree, the algorithm is inherently biased to grow towards unexplored areas of the state space, hence the name *rapidly-exploring*. Following this, as time approaches ∞ with more of the state space being explored, RRT is guaranteed to find a viable path if one exists.

$$\lim_{t \to \infty} Pr(solution) \to 1$$

Although RRT is guaranteed to eventually converge towards a path if such a path exists, it is unable to determine whether or not a path *doesn't* exist. Therefore, RRT is considered a *probabilistically complete* algorithm. Furthermore, there is no promise in terms of optimally of the path generated, something that RRT* addresses. The psuedocode for the RRT algorithm can be seen below.

Algorithm 1: Rapidly Exploring Random Trees (RRT)

```
Result: Construct a space-filling tree which is guaranteed to find a path if such a path exists
Vertices are V, Edges are E
Construct configuration space with obstacles C_obs
Set start state s_s, set k = 0, set number of vertices K, set drive time dt
while k is not K do
    sample x_rand
    if x_rand ∈ C_obs then
        continue
    end
    find x_nearest ∈ V
    from x_nearest drive towards x_rand for dt
    arrive at x_new with trajectory of states
    if trajectory not ∈C_obs then
        add node x_new to V
        add edge (x_nearest, x_new) to E
    end
    increment k
end
```

2.3 Constructing the Configuration Space using Minkowski Sum

As mentioned in the previous section, there must be a way to check whether or not a given state is collision-free or not when constructing the RRT tree. To do so, the configuration space C is computed. The configuration space includes all possible states of the robot that does not collide with obstacles. Essentially, the robot state must obey the following

$$\text{Robot state} \in C/C_{obs}$$

To compute the configuration space C/C_{obs}, the robot is treated as a point object with all obstacles boundaries being enlarged accordingly. By doing so, collision checking reduces to simply checking whether or not the robot point is within an obstacle polygon or not.

To find the appropriate enlarged obstacle boundaries, the Minkowski sum is used. Through this method, the corners of the robot are aligned with the corners of the obstacle after which the center points of the robot are used to construct the new obstacle polygon. Figure 2[2] shows an example of this construction using a rectangular robot angled at 45° around a rectangular obstacle.

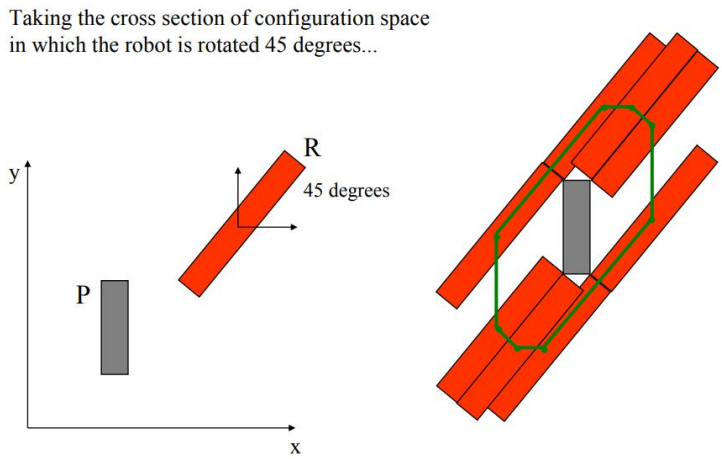

Figure 2: Construction of Minkowski Sum Polygon for $\theta = 45°$ [2]

For robots that are able to rotate with non-symmetrical geometries around the center point, such obstacle polygons must be calculated for all possible angular positions. When dealing with a continuous state system, it is customary to discretize the angular positions in order to obtain a computationally feasible configuration space. At the same time, this discretization must be fine enough to not suffer from misprediction of collisions. To account for any such possible collisions caused by differences between configuration space angle and real robot angle, a small padding can be added to all obstacles as a safety factor.

2.4 Optimized RRT: RRT*

To account for the lack of optimality when using RRT, an optimized variant of RRT, RRT*, was also used for path planning. As a variant of RRT, RRT* has the same algorithmic characteristics of RRT such as being probablistically complete. Two key differences are applied to the algorithm.

1. Once x_{new} is found, instead of connecting to the original parent node right away, search for a more optimal parent node within the tree and connect to that "better" node instead.

2. Rewire nodes within the tree for more optimal connections in terms of a cost parameter.

With these additions, RRT* is able to generate more optimal paths as shown in figure 3[3] where plots a-d show RRT and plots e-h are RRT*.

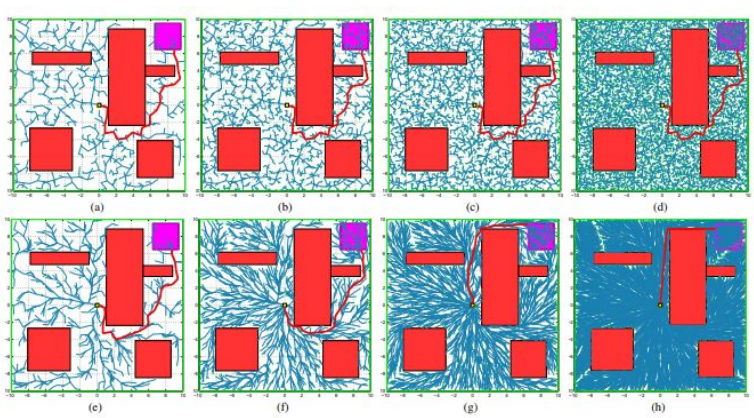

Figure 3: RRT vs. RRT* Tree Generation[3]

As shown, RRT* trees end up with a structure with much less "needless" branching due to the rewiring of the tree and adaptive parent node selection. Due to the optimization of the tree, RRT* is guaranteed converge towards the most optimal path as t approaches ∞ so long as a path exists.

$$\lim_{t \to \infty} Pr(\text{optimal path}) \to 1$$

With this optimization guarantee comes some cons of the RRT* algorithm as it is inherently more expensive computationally due to the overhead of having to search for a better parent node as well as attempting to rewire nodes every iteration. The psuedocode for RRT* can be seen below.

Algorithm 2: Optimized Rapidly Exploring Random Trees (RRT*)

```
Result: Find optimal path
Vertices are V, Edges are E
Construct configuration space with obstacles C_obs
Set start state s_s, set k = 0, set number of vertices K, set drive time dt
while k is not K do
    sample x_rand
    if x_rand ∈ C_obs then
        continue
    end
    find x_nearest ∈ V
    from x_nearest drive towards x_rand for dt
    arrive at x_new with trajectory of states
    if x_better exists then
        redrive from x_better to x_new
    end
    if trajectory not ∈C_obs then
        add x_new to V
        add edge (x_nearest, x_new) to E
    end
    increment k
    rewire tree
end
```

3 Experimental Setup

This section provides the necessary information about the models of the robot and the parking lot environment. The robot's drive policy as well as what constitutes the "nearest" node are also discussed. Furthermore, several experimental assumptions made in the model are also discussed as well as their effects on the solution.

3.1 Map Model

To show the path planning capabilities of RRT, a map simulating a parking lot was constructed as seen in figure 4. On this map, vehicles, walls and road markers were represented as red, yellow, and white polygons, respectively. Only obstacles with rectangular shape were considered. The task of the robot was to navigate around the obstacles and successfully park in space 9 which can be seen marked with green.

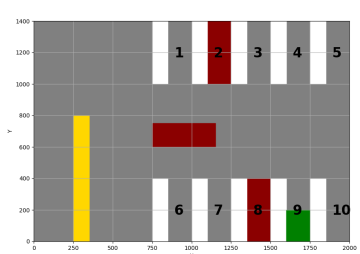

Figure 4: Plane Map for Parking lot environment

3.2 Robot Model and Assumptions

In this work, a two wheeled rectangular robot was considered. Dimensions and shape of the robot are shown in figure 5. The angular wheel velocities of the robot were controlled individually using PWM, allowing for angular rotations ranging from -60 to 60 RPM. Since this experiment focuses primarily on the planning problem, the control problem was omitted for simplicity. Essentially, the following assumptions were used:

1. Only robot kinematics were considered. Forces such as friction were ignored.
2. Instantaneous acceleration was assumed in switching between different wheel velocities.
3. No sliding of the robot occurs.
4. The robot resides in a fully observable system and requires no sensor model.

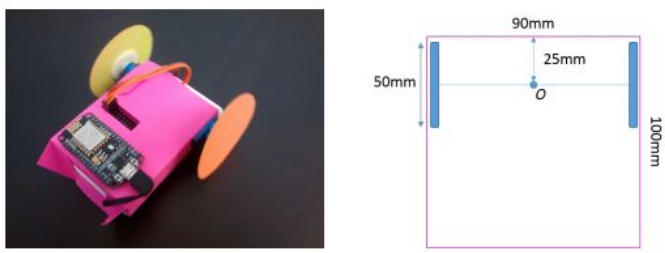

Figure 5: Robot Dimensions

The state space of the robot is continuous as defined by the environment obstacles and boundaries. Furthermore, the action space of the robot was also considered continuous in the range of -60 to 60 RPM. To deal with this continuous system model, discretization of the configuration space is conducted which is discussed in the next section.

3.3 Robot Configuration Space

Before path planning could occur, a 3D configuration space was first created using the Minkowski sum method discussed in section 2.3. Due to the robot's non-symmetrical shape, the configuration space can be seen to have a wavy boundary outline as θ is varied in figure 6 where every tenth degree was colored black to allow for easier visualization of the variation of the obstacle polygons. In computing the configuration space, a precision of 0.1 $^\circ$ was used resulting in a total of 3600 layers of obstacle polygons stored in an array. Using this array, safe and unsafe states could be determined by referencing the obstacle polygons of the current robot θ and then checking whether or not the robot center point lied in an obstacle polygon.

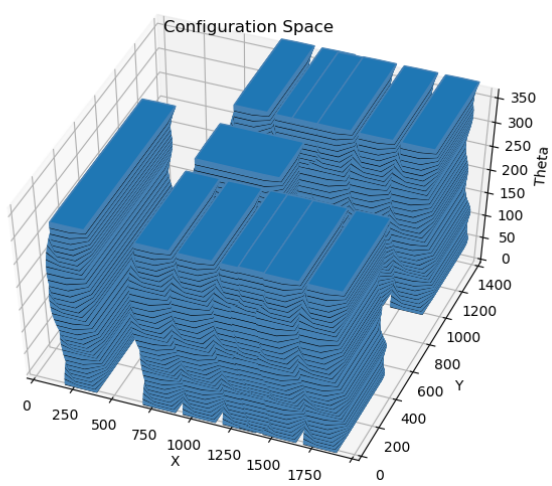

Figure 6: Configuration Space for Parking Lot Environment (without map borders)

While performing RRT and RRT*, new points for the graph were picked randomly from this configuration space. If the point belonged within an obstacle polygon, the sampled point was discarded and a new point was resampled.

3.4 Nearest Qualification and Drive Policy

When determining the nearest point after sampling, the L2-norm was used as the metric as shown below. This metric was used because pure euclidean distance proved to be a poor indicator of the best node to drive from. This was a result of the differential drive in which if the angular positions between the sample point and drive node were significantly different, the robot had a lower chance of driving towards the sample node versus a node that was further away by distance but more aligned in θ.

$$x_{nearest} \leftarrow \text{argmin}_{node}(node_x - sample_x)^2 + (node_y - sample_y)^2 + (node_\theta - sample_\theta)^2$$

To determine which actions were to be used when driving from the nearest node to the sampled point, a system of equations was constructed using the differential drive equations discussed in section 2.1. Using these equations, the left and right wheel angular velocities were computed by minimizing the least squares error of the current position and desired position. These equations are shown below

$$\text{eqn. 1: } \frac{r}{L}y - x = \Delta\theta \tag{4}$$

$$\text{eqn. 2: } \frac{r}{2}(x+y)sin\theta = \Delta x \tag{5}$$

$$\text{eqn. 3: } \frac{r}{2}(x+y)cos\theta = \Delta y \tag{6}$$

$$\text{minimize } (\Delta\theta)^2 + (\Delta x)^2 + (\Delta y)^2 \tag{7}$$

where x and y are the left and right wheel angular velocities (rad/s) u_l and u_r, respectively.

Through minimizing, the computed wheel velocities were then used to drive from node to node with each drive period lasting for 1 second at which the robot travels with the following wheel velocities for 0.1 second periods. In other words, the robot re-optimized its actions 10 times every drive period which allowed for turns of varying arcs. During this drive period, at each time step interval, the current robot state is checked for collision. If a collision is detected, the trajectory is thrown away and new point is sampled. Otherwise, the last robot state and the full trajectory are added as a new node and edge to the tree. It can then be seen that one session of driving will consist of the following.

Algorithm 3: Drive Algorithm

```
Result: Create a path from node towards sample node
Set t = 0, dt = 0.1 seconds
while t ≤ 1 do
    minimize least squares and receive u_l and u_r
    δx ← (r/2)(u_l + u_r)cosθ dt
    δy ← (r/2)(u_l + u_r)sinθ dt
    δθ ← (r/L)(u_r − u_l) dt
    Update x_curr, y_curr, and θ_curr
    if state_curr ∈ C_obs then
        throw away trajectory and resample
    end
    t += dt
end
add state_curr to V
add trajectory to E
```

4 Result and Discussion

In this section, the simulation result of RRT under different environments and RRT* is discussed.

4.1 RRT in a Complex Parking Lot

RRT algorithm was implemented in Python and tested in a complex parking lot, where there are three red cars and one yellow divider as in Fig. 7. The white lot lines were considered as obstacles, and the robot was initially at the left down corner. As the number of sampled nodes increased, the coverage of the RRT tree improved. In Fig. 7, the tree grew to the other end of the parking lot as early as at 250 sample nodes, whereas the robot failed to reach the goal lot, green lot 9, until 3000 samples. The parking was as narrow as the width of the robot itself, and therefore the rover should have approached straight to the destination from its head, which raised the difficulty to reach the goal state.

Comparing 3000 sample case (Fig. 9) to 10000 sample case (Fig. 11), there was no significant improvement in terms of the robot path since RRT did not refine the tree as it grew and larger samples only contributed to enhance the tree cover area and density. Indeed, with 10000 sample in Fig. 10 RRT discovered a path to all open parking space but one space, while with 3000 samples in Fig. 8 RRT failed to screen the half of the open lots.

Using one thread of 3.7GHz Clock speed CPU, the 10000 nodes RRT took about 1000 seconds, while 3000, 1000, and 500 sampled cases required only about 150s, 60s, and 15s, respectively. As the tree nodes expanded, the program had to spend exponentially more time because it would substantially demand more computational expense to analyze the nearest nodes and robot trajectories costs.

RRT had an known issue which was to determine when RRT tree can find a solution. In this complex parking lots, RRT discovered a path to the green goal at 849 samples at average over 10 runs. However, RRT could find reached the goal as soon as 326 samples or even failed at 3000 nodes, at which sample RRT process was halted for the sake of time in this case. The program sampled a random point uniformly from the entire field, yet one point was extracted from the goal area every 25-sample. Nonetheless, RRT frustrated to connecting the sample point from the goal to its existing tree depending on how the tree advanced to or around there due to the tight clearance of the lot space.

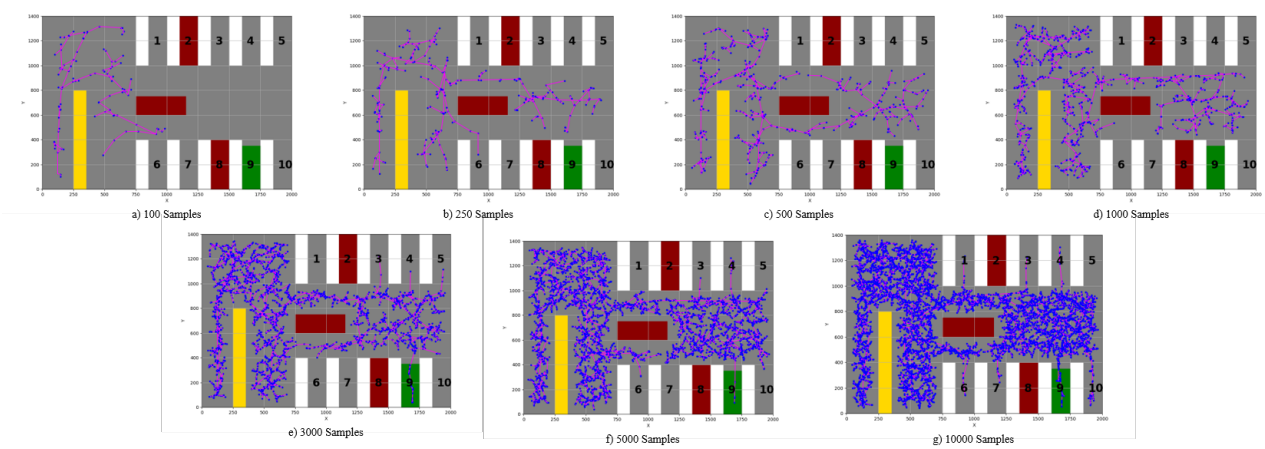

Figure 7: RRT with the different number of sample nodes.

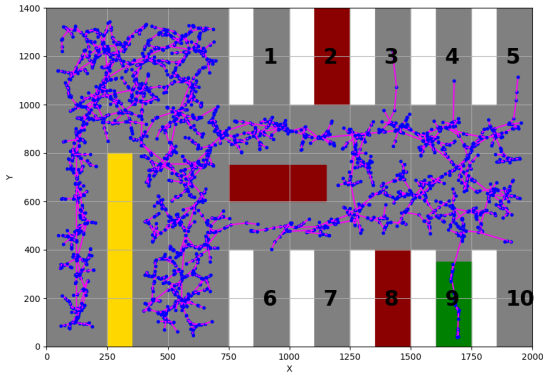

Figure 8: RRT node trees with 3000 node samples

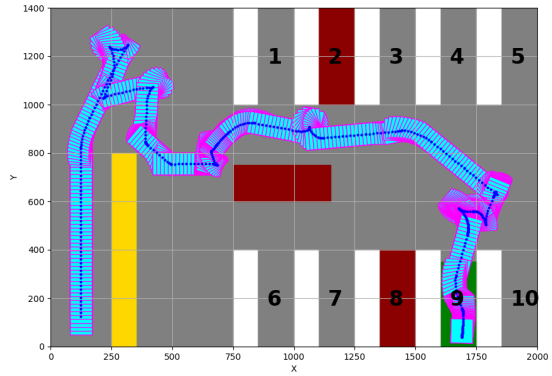

Figure 9: A robot path generated by RRT with 3000 node samples

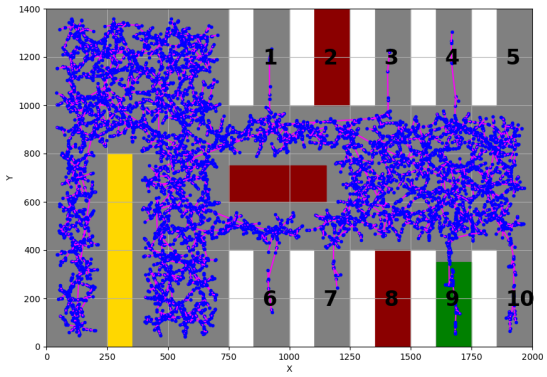

Figure 10: RRT node trees with 10000 node samples

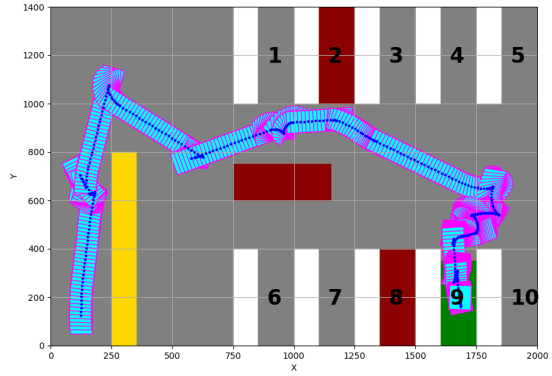

Figure 11: A robot path generated by RRT with 10000 node samples

4.2 RRT Performance in Multi-Environments

Furthermore, RRT was tested in simpler situations where there were less obstacle than the original lots. Fig. 12 and Fig. 13 showed the scenario when the yellow divider was eliminated. With the yellow divider as in Fig. 7, tree expanded vertically around the initial position and other branches toward wall stop growing, which would certainly result in collision. On the other hand, the RRT tree in Fig. 12 spread more evenly from the robot start position. Regarding of the computation time, there was no compelling difference because the region around the yellow obstacle was relatively larger. Thus, this obstacle did not affected much on the time, but RRT struggled from finding path at other narrow space typically.

In Fig. 14 and Fig. 15, red regions which indicated cars were removed from the field, and then RRT was able to allocate a path to the goal distinctively faster than the original parking lots at sample nodes of 300 to 500 range. In Fig. 15 the robot proceeded straight toward the target after avoiding white divider next to the lot 6. In Fig. 13, the robot had to maneuver frequently to pass though the gap between the car in the middle and other obstacles.

When RRT was tested in an open space, a path was found at average of 112 nodes, and in Fig. 16 and Fig. 17 the tree consisted of 25 nodes. In contrast of Fig. 11, the robot went almost straight to the goal and there was less maneuver in those trajectories. Consequently, the complexity of the map would affect not only the computation time, but also path quality in RRT since the tree can grow relatively straight to the goal state.

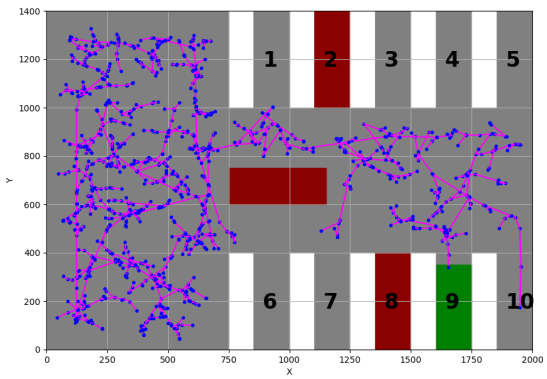

Figure 12: RRT node trees in a parking lot with no yellow divider

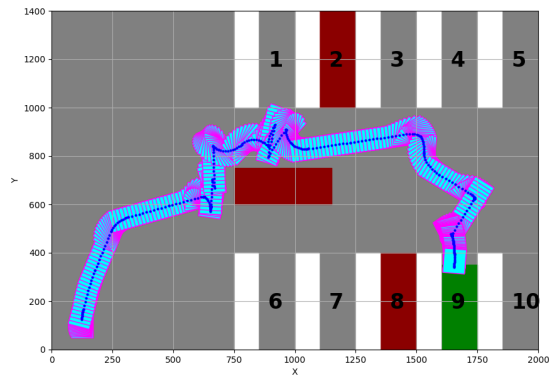

Figure 13: A robot path to the goal generated by RRT in a parking lot with no yellow divider

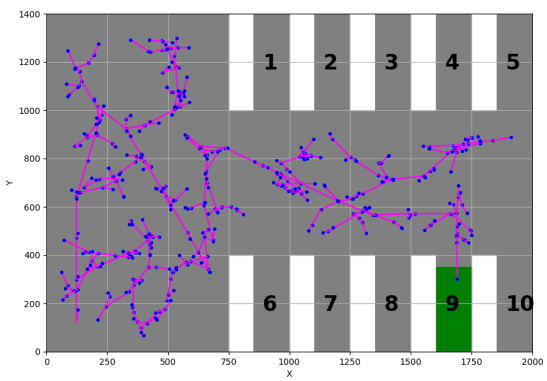

Figure 14: RRT node trees in a parking lot with no yellow divider and no cars

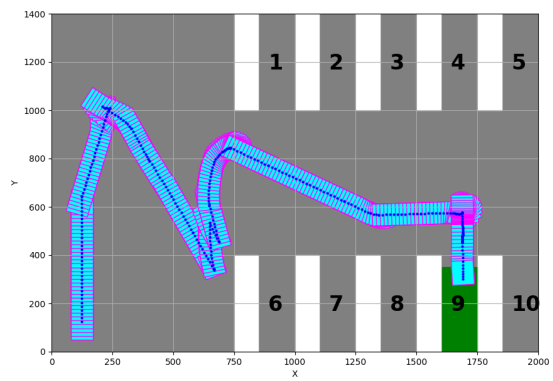

Figure 15: A robot path to the goal generated by RRT in a parking lot with no yellow divider and no cars

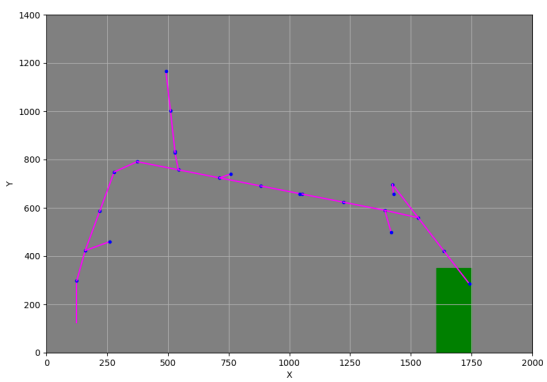

Figure 16: RRT node trees in an open space

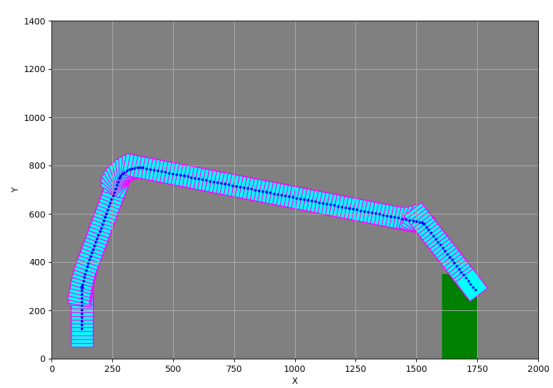

Figure 17: A robot path to the goal generated by RRT in an open space

4.3 RRT*

Here, the trees and trajectories generated by RRT and RRT* are shown in Fig. 18, Fig. 19, Fig. 20, Fig. 21

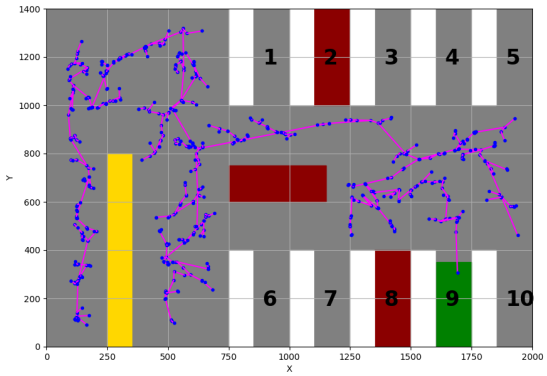

Figure 18: Tree generated by RRT

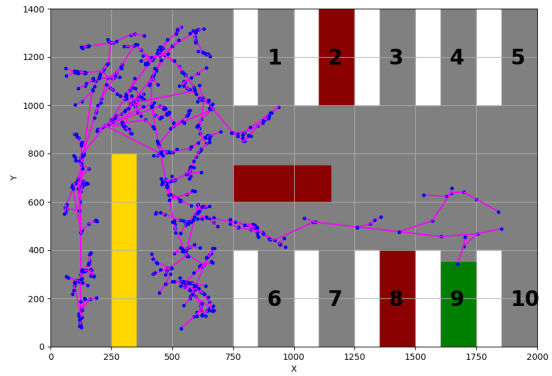

Figure 19: Tree generated by RRT*

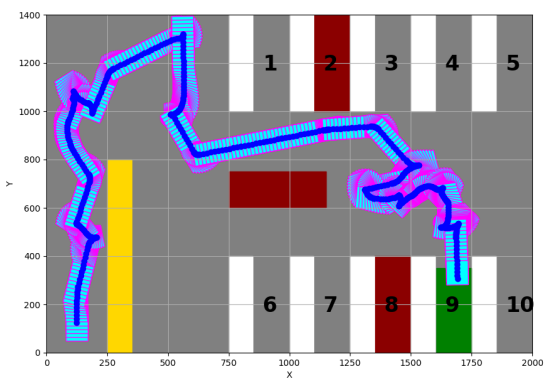

Figure 20: Trajectory generated by RRT

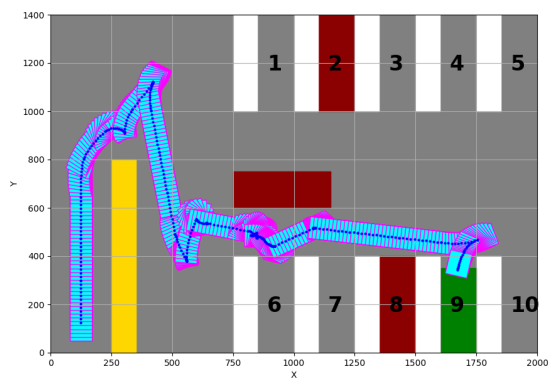

Figure 21: Trajectory generated by RRT*

First, the computational cost is discussed. Under the same condition (the number of maximum iteration is 3000, the number of steps for action is 10 during 1s), RRT takes 172 s, and RRT* takes 219 s. This result is reasonable because RRT* searches for a more optimal parent node and rewire nodes within the tree for more optimal connections.

Compared with Fig. 18, Fig. 19 shows that rewiring operation is confirmed, especially in the region of $250 < x < 500$, $800 < y < 1200$. The reason why the rewiring is conducted in the region is that the robot has basically two ways to arrive at the goal: the first road is over the red obstacle at $750 < x < 1125$, $600 < y < 750$, and the other is under it. Thus, based on the random node, our RRT* planner computes/rewires the optimal parent node, and the nodes in the region of $250 < x < 500$, $800 < y < 1200$ have an effect on these two roads, so the rewiring action is often executed in the region.

As shown in Fig. 20, Fig. 21, the trajectory generated by RRT* is more smooth than that by RRT. In addition, the length of the trajectory is shorter than that by RRT. This is because RRT* eventually converges the optimal trajectory while it is not guaranteed that RRT outputs the optimal trajectory. Therefore, even though RRT* is expensive in terms of computation, the RRT* shows a better trajectory that enables the robot to arrive at the goal with less length of the trajectory while avoiding obstacles.

4.4 RRT Uncertatinty

The proposed RRT and RRT* do not take into account uncertainty, so that if the uncertainty is quite large, the robot may collide to obstacle even though the generated trajectory avoids the obstacle.

There are two ways to solve this problem. The first solution is a practical one. When RRT is implemented, the RRT is executed in the configuration space. So, as long as the planner models the uncertainty, the planner "extends" the obstacle in the configuration space in consideration of the uncertainty. In other words, the safety region is introduced that takes into account the uncertainty by extending the obstacle region in the configuration space.

The other solution is a theoretical one. When RRT* is implemented, by calculating the expectation of the cost, the planner is able to find the optimal trajectory based on the probabilistic description. The dynamics of the robot is not deterministic any more, so that the optimal trajectory is not guaranteed any more. However, still the planner is able to find the optimal path where the robot would arrive at the destination with the highest expected value in terms of cost in probabilistic description.

5 Conclusion

In this work, RRT and RRT* were implemented to obtain a trajectory from the start point to the goal point while avoiding obstacles. In a complex field which mimicked a parking lot with cars, walls, and lane markers, RRT successfully attained a path to the goal state, but with unreasonable trajectories. The number of sample nodes did not contribute to improving the robot path, while it increased completeness of the RRT in the map with exponentially more computation time. In multi-environments, the implemented RRT resulted in practical and straight forward trajectories, while the number of nodes required to solve did not notably decreased unless bottle neck obstacles were eliminated. Furthermore, RRT* was also implemented which led to shorter and smoother trajectories albeit with more expensive computational cost.

6 References

[1] http://planning.cs.uiuc.edu/node659.html
[2] https://www.cs.cmu.edu/motionplanning/lecture/Chap3-Config-Space_howie.pdf