TP4: Path planning using RRT

Planification et contrôle

ROB316

Yu WANG

January 16, 2022

Remarque : All the results (code, images, etc.) presented in this report can be found on this gitHub : https://github.com/Rescon/ROB316

1 Introduction

1.1 Summary of the courses

There are two levels of planning: local and global. Global planning has already been studied in TP3, which stores environmental information in a map and uses this map to find a feasible path. But this is not suitable in unknown environments. In this TP, we will study local path planning, which only takes into account the instantaneous environmental information of the robot, which helps us to reduce the computation time.

Path planning consists of four parts: obstacle avoidance, reactive planning, stochastic path search and exploration. Vector Field Histogram, Dynamic Window and Potential Fields can be used to perform obstacle avoidance. The BUG algorithm is used in the unknown map to finish the reactive planning. In stochastic path search, the RRT algorithm and its variants are used. And the exploration of an unknown environment can be done by the A* algorithm.

1.2 Description of TP

In this practical work, we will work on the Rapidly Exploring Random Trees (RRT) algorithm [2]. For this, we will use the python code that implements RRT and one of its variant RRT* [1], on different environments. This code is modified from the code of the repository of Huiming Zhou [1] that implements and

[1] https://github.com/zhm-real/PathPlanning

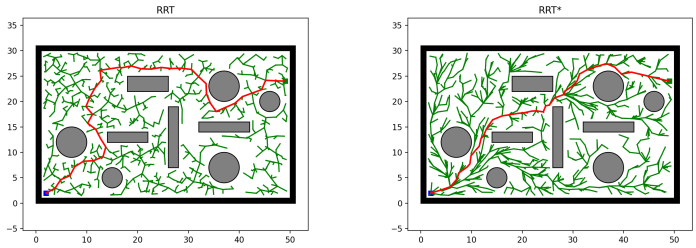
illustrates many path planing algorithms. Then we will apply the obstacle-based rapidly-exploring random tree (OBRRT) [3] for some special environment.

2 RRT vs RRT*

In this section, we keep the map, the default start and end positions and do experiments to compare the RRT algorithm with RRT*.

2.1 Question 1

Both RRT and RRT* are tested by changing the maximum number of iterations. The results obtained according to the default parameters are shown in Figure 1.



(a) (b)

Figure 1: Result of RTT and RTT*

In theory, RRT stops when it finds a trajectory, but RRT* will keep going until the maximum number of iteration to find an optimal trajectory. The result of different values is shown in Table 2.1 and 2.2. An iteration cycle of 10 on RRT and RRT* is used to avoid stochastic errors.

According to Table 1 and Table 2, we find that when the maximum number of iterations is not large enough, the RRT and RRT* algorithms cannot succeed in finding the trajectory and there will be failures.

For RRT, we see that there is no direct relationship between the maximum number of iterations and the other two values. The path length cannot be improved by increasing the number of iterations. But for RRT*, the case is different. The larger the number of iterations, the smaller the path length, which means that more iterations allow us to obtain a more optimal path. The time to find the first trajectory is changed but the total computation time is still related to the maximum number of iterations.

Maximum number of iterations	Computation time	Path length
100	0.042838	Nan
200	0.174101	Nan
500	0.314507	71.56
1000	0.710250	72.21
2000	2.116602	71.29
5000	6.382090	73.82
10000	22.846642	71.90

Table 1: Result of different maximum number of iterations for RRT

Maximum number of iterations	Computation time	Path length
100	Nan	Nan
200	Nan	Nan
500	2.880791	71.49
1000	8.335840	68.53
2000	12.582366	62.31
5000	275.396494	58.85
10000	1975.234109	56.29

Table 2: Result of different maximum number of iterations for RRT*

2.2 Question 2

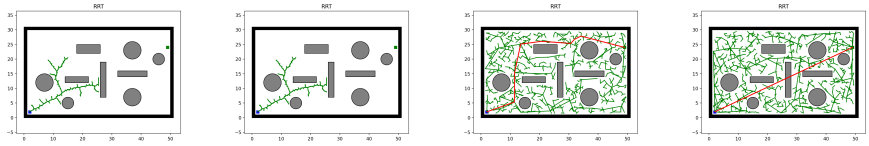
We now study the influence of *step_len* on both algorithms by changing its value. The maximum number of steps is set by default (1500).

Step Length	Computation time	Path length
0.1	Nan	Nan
1	31.405949	74.81
5	72.126588	67.54
10	132.259305	62.34
100	103.596341	52.79

Table 3: Result of different *step_len* for RRT

According to the results presented in Table 3 and 4, it can be found that a *step_len* that is too short will slow down the search speed of the algorithm, and too long will lead to growth in overcoming obstacles.

Specifically, when *step_len* is small, it will take a long time to find a path, as shown in Figure 2 and Figure 3. The larger *step_len* is, the less time it needs to find a trajectory. The associated length is also smaller. When the value of *step_len* is large, the algorithm can find a path quickly. However, when the value is too large, the trajectory found is no longer correct because obstacle avoidance is not done, as shown in Figure 2 and Figure 3.



(a) *step_len*=0.1 (b) *step_len*=1 (c) *step_len*=10 (d) *step_len*=100

Figure 2: Result of different *step_len* for RRT

Step Length	Computation time	Path length
0.1	Nan	Nan
1	9.973501	65.93
5	1.201872	67.54
10	1.259305	66.91
100	1.596341	51.89

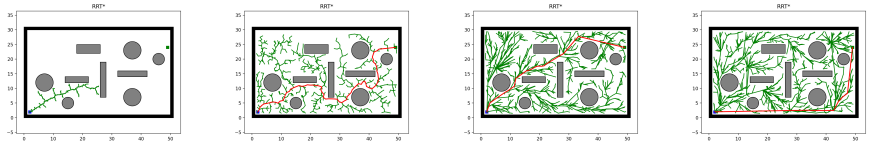
Table 4: Result of different *step_len* for RRT*

3 Planification in narrow corridors

In this section, the *Env2* environment (*environment = env.Env2 ()*) is used and all other parameters are set to default. A narrow corridor is located in the middle of the map, which makes it difficult to find a path.

3.1 Question 3

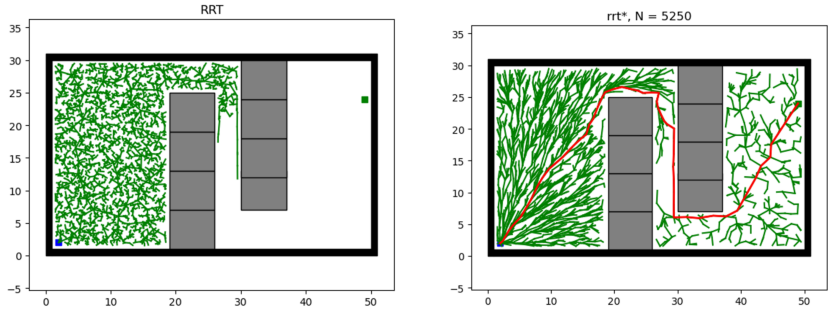
According to Figure 4, it is found that it is much more difficult to grow the tree quickly in this environment. This can be explained by the RRT principle. When the tree tries to grow to a random point from the nearest node, it will judge whether the line between the point and the nearest node passes through any obstacles. If so, this random point will be eliminated. In the map used in this section, the width of the corridor is not large enough compared to the *step_len*, which prevents us from expanding the tree as new random points are eliminated. For RRT*, a larger number of iterations is required then the final



(a) *step_len*=0.1 (b) *step_len*=1 (c) *step_len*=10 (d) *step_len*=100

Figure 3: Result of different *step_len* for RRT*

result may be obtained.



(a) (b)

Figure 4: Result for RRT and RRT*

3.2 Question 4

To solve the problem in question 3, we apply the OBRRT algorithm, which chooses those nodes who aren't in the obstacle area as the code shows. The idea of this algorithm is to sample points taking into account obstacles in order to increase the chances of the tree crossing difficult areas. The associated code is shown below. The result is showed in the Figure 5.

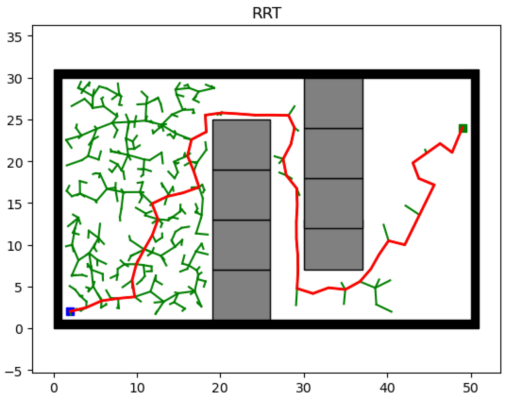


Figure 5: OBRRT Algorithm

```
flag = True
List = self.env.obs_rectangle
while flag:
    x = np.random.uniform(self.x_range[0] + Δ,
```

```
5           self.x_range[1] - Δ)
6       y = np.random.uniform(self.y_range[0] + Δ,
7           self.y_range[1] - Δ)
8       if not self.utils.is_inside_obs(Node((x,y))):
9           flag = False
10  return Node((x,y))
```

References

[1] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[2] S. M. LaValle, J. J. Kuffner, B. Donald, et al. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and computational robotics: new directions*, 5:293–308, 2001.

[3] S. Rodriguez, X. Tang, J.-M. Lien, and N. M. Amato. An obstacle-based rapidly-exploring random tree. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 895–900. IEEE, 2006.