Lab 3 - Rapidly-Exploring Random Trees Algorithm

Songyou Peng
psy920710@gmail.com

I. INTRODUCTION

The rapidly-exploring random trees (RRT) is one of the most well-known sampled-based algorithms for robotics path planning. This report will mainly discuss the implementation details of the algorithm. Besides, we play with different parameters of the algorithm and will talk about our insights in the last part.

II. IMPLEMENTATION

A. RRT

The main idea of the RRT algorithm is to grow a tree rooted at the given starting point by using random samples from the search space [1]. When each random sample is acquired, we obtain a vector from the nearest point in the tree to the sample and grow the tree in this direction with a certain distance. Finally, when the goal point is included in the tree, the algorithm stops and the path is reconstructed by checking the connected points in the edge list.

Firstly we initialize the *vertices* list with the starting point q_{start}. RRT iteratively samples points q_{rand} in the given map. In each iteration, there is 30% of possibility p that q_{rand} is directly set to the goal point. Here in order to realize this function, we create a random value within the range of $[0,1]$ and if the value is smaller than p, q_{rand} is set to q_{goal}, otherwise a random point within the map.

After obtaining q_{rand}, we calculate the Euclidean distances from all the points in the *vertices* list to the q_{rand}, and choose the nearest vertex as the q_{near}. Then, the tree grows from q_{near} along the direction to the q_{rand}. If the distance between q_{near} and q_{rand} is larger than a certain threshold $delta\_q$, the tree still grows only $delta\_q$ and acquire a new vertex q_{new}, or q_{new} is just equal to q_{rand}.

However, q_{new} is not always valid. As long as one of the following three conditions is satisfied, the q_{new} will be discarded.

- q_{new} lies out of the map
- q_{new} is already in the *vertices*
- there exist obstacles between q_{near} and q_{new}

The first two conditions can be tested easily, but a few more steps are required in order to check the third one. We get a unit vector from q_{near} to q_{new} and then use incremental strategy to check if the 10 middle points lying between these two vertices are within obstacle or not.

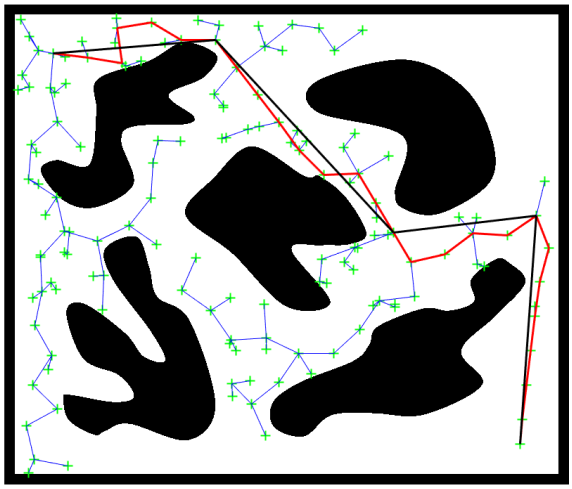
If the current q_{new} is valid, then we can put it into the *vertices* list and insert its index and the index of the corresponding q_{near} in the *vertices* to the *edges* list.

Finally, we reconstruct the path from the *edges* list. Starting from the goal point, we check which vertex is the q_{near} of goal point, put it into the *path*, and then check the q_{near} of the goal point's q_{near} point, and so on. When the q_{near} appears to be the starting point, the path is successfully reconstructed. The result is the red lines shown in the Fig. 1.

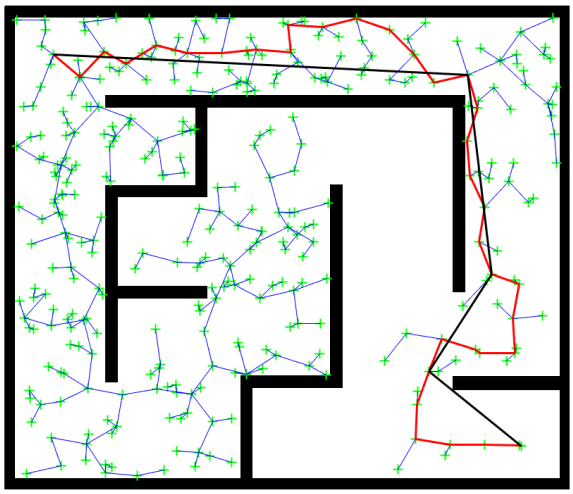
B. Smoothing

As we know, the RRT is a randomly sampling process. The acquired paths are different and not optimized for each time. Although the RRT cannot bring us the real optimized path, we can still try to get a shorter or less noisy path. The idea is as follows:

1) In each iteration, at first q_{start} is always set to the start point
2) Get the unit vector from the q_{start} to q_{end}
3) Check if it is free space between q_{start} and q_{end} (incrementally add $delta\times$unit vector)
4) If yes, directly set q_{end} to q_{start} and put the q_{end} into $path\_smooth$ list. This iteration finishes
5) If no, update q_{start} to the next element in the *path* list, start from step 2) again

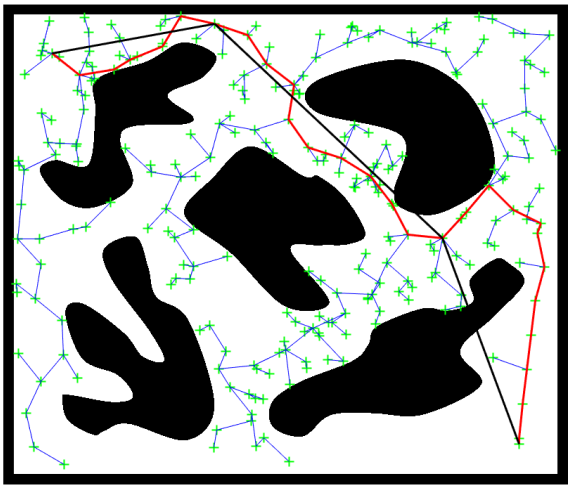


(a) `map.mat`

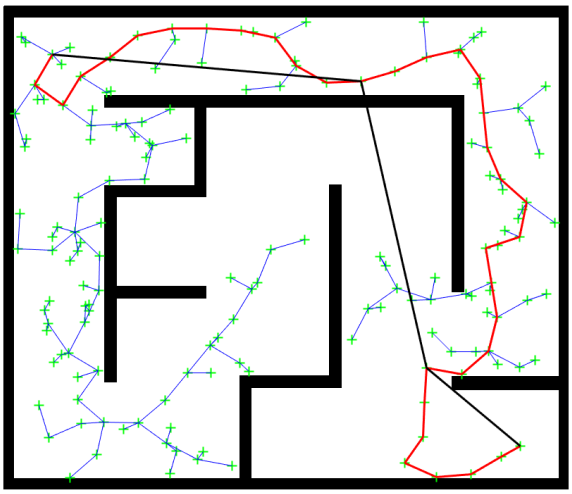


(b) `maze.mat`

Fig. 1. Illustrations for RRT algorithm. Green and black lines are the path acquired without and with smoothing step respectively



(a) `map.mat` with $delta = 20$



(b) `maze.mat` with $delta = 50$

Fig. 2. Illustrations for the wrong smooth path because of high *delta*. Green and black lines are the path acquired without and with smoothing step respectively

The smoothing path is the black lines shown in the Fig. 1.

C. Parameter tuning

In this part, we will discuss some interesting ideas when we played with two parameters.

The first tuning parameter that we played with was the incremental distance *delta* for the smoothing step. When *delta* is high, the algorithm can be speeded up, but there is a chance that the smooth path may be wrong when *delta* is too high (Fig. 2). For `map.mat`, we find that the wall width is 18 pixels so if the *delta* is larger than that, it is possible to directly cross the obstacles like in Fig. 2(a). For `maze.mat`, the obstacle sizes vary from one to the other, but the smallest width in the map is around 45 pixels, so the smooth path may be like Fig. 2(b) if *delta* is set to be 50. This indicates a fact that we need to choose *delta* wisely for each map.

The second parameter that we played with was probability p of choosing q_{goal} as q_{rand}. We notice that when the possibility is really high, fewer branches exist in the tree, but it also significantly increases the number of iterations. It is reasonable because now almost all the q_{near} points to the q_{goal} directly and it will fail to update q_{new} for

a huge amount of times because of the obstacles. However, once it is updated, q_{new} has a bigger chance that leads to a relatively optimized path. In contrast, if the p is really small, more branches appear on the map but q_{new} can be updated more often. Based on the discussion, p should also be chosen carefully depending on different cases.

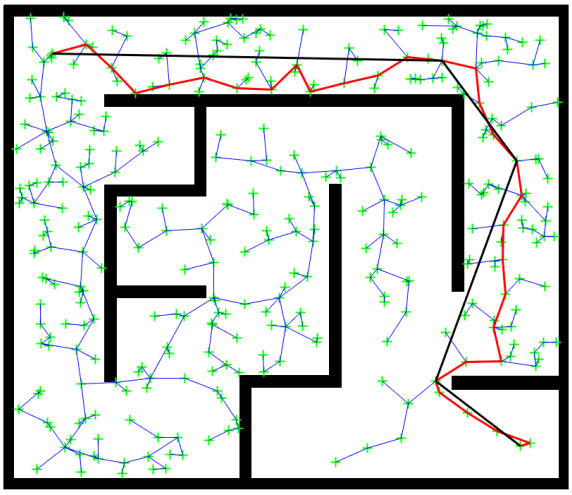
III. Conclusions

In this lab, we In this lab, we implemented the RRT algorithm as well as the smoothing step. Some parameters were also played with, which enabled us to understand the algorithm better.

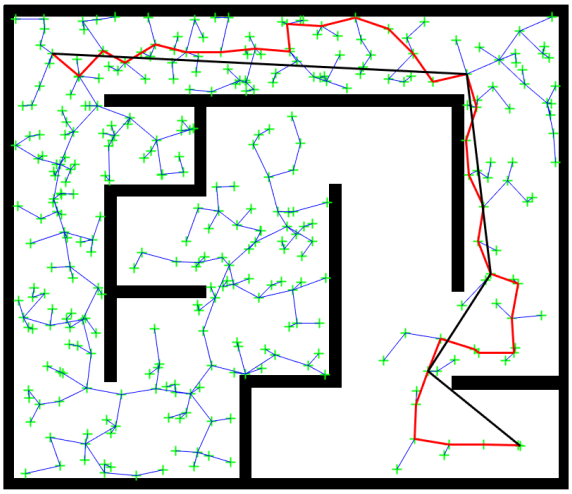
It should be noted that the RRT algorithm can always give a path to the goal point as long as the parameters are chosen wisely. However, even if the smooth step is applied, the path the path may not be the shortest one.

References

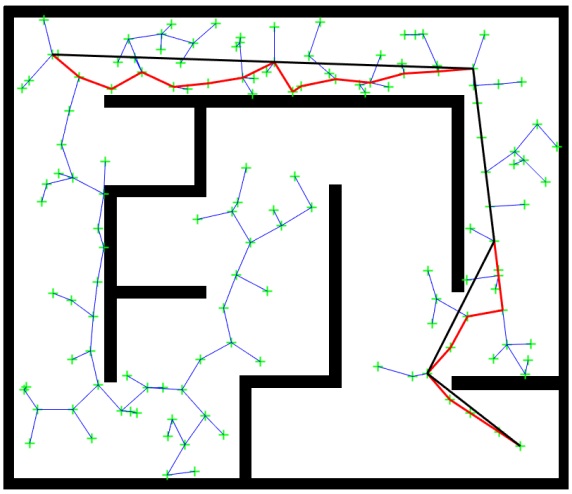
[1] Wikipedia, The Free Encyclopedia, s.v. "Rapidly exploring random tree" (accessed April 17, 2016), https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree



(a) $p = 0.05$, 350 branches, 500 iterations



(b) $p = 0.3$, 290 branches, 600 iterations



(c) $p = 0.95$, 130 branches, 5300 iterations

Fig. 3. Illustrations for the different possbility of q_{goal} as q_{rand}.