

一些记录

1.我做的事情:

A*算法

详见/A-star/astar2.py, main.py

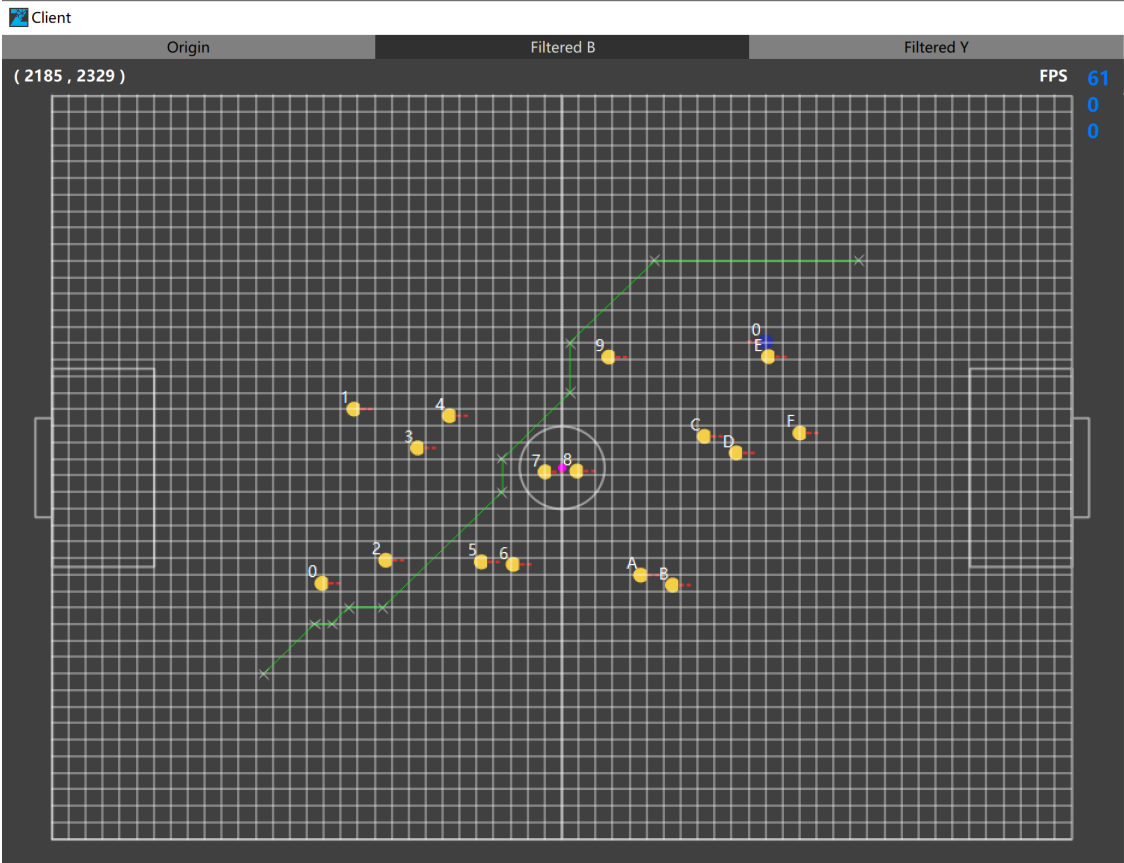
A*算法是我机器人学第一个大作业所写的内容


我的做法就是：从起点开始按照栅格搜索，根据A*算法的规则构建一个启发式函数（这里h采用了横纵坐标差值绝对值之和），然后操作openlist、closelist进行搜索，最后构造出路径。其实就是照着老师的课件上的那个规则，然后自己去实现。

然后这么做出来的“碎点”很多，为了轨迹规划的方便，对路径进行了简化，主要方式为：如果相邻三点共线，那么中间点被删去，如果相邻三点距离小于栅格大小的某个阈值，那么中间点被删去（主要针对可能出现的很小的三角形）

实现效果

实现效果如下图所示：



 C:\Windows\System32\cmd.exe - main.py

```
本次A*算法用时: 0.014941930770874023
Path found!
本次A*算法用时: 0.01494288444519043
Path found!
本次A*算法用时: 0.013962507247924805
Path found!
本次A*算法用时: 0.014961481094360352
Path found!
本次A*算法用时: 0.012964248657226562
Path found!
本次A*算法用时: 0.013961315155029297
Path found!
本次A*算法用时: 0.012964963912963867
Path found!
本次A*算法用时: 0.013967037200927734
Path found!
本次A*算法用时: 0.012964725494384766
Path found!
本次A*算法用时: 0.014961004257202148
Path found!
本次A*算法用时: 0.016954421997070312
Path found!
本次A*算法用时: 0.019945383071899414
Path found!
本次A*算法用时: 0.02094292640686035
Path found!
本次A*算法用时: 0.014942169189453125
Path found!
本次A*算法用时: 0.014941215515136719
Path found!
本次A*算法用时: 0.015957355499267578
```

我认为A*算法的表现是非常稳定的，表现平均水平基本在二十多毫秒

用500次测试来统计：

A-star用时平均值：0.02552624034881592

A-star用时最小值：0.00997304916381836

A-star用时最大值：0.036900997161865234

改进A*算法

在写好的大作业基础上，我在网上查阅到资料，启发式函数的不同选择有可能影响A-star算法的性能，有文章说在启发式函数部分，可以再加上父节点的h的值，这样能提高搜索的方向性，减少搜索的节点的个数，我改动了如下部分代码，进行了尝试：

```
nearby.g = tg
nearby.parent = self.find_index_of_e_in_closetlist(e)
nearby.f = nearby.g + nearby.h + self.closetlist[nearby.parent].h
```

用500次测试来统计：

A-star用时平均值：0.015598354339599609

A-star用时最小值：0.011968612670898438

A-star用时最大值：0.022938251495361328

可以看出，性能得到了改善

改变一些A*算法的参数

我认为比较重要的参数是栅格地图划分的大小

```
self.Grid_Size = 250
```

A-star用时平均值: 0.012922431945800781

A-star用时最小值: 0.0070037841796875

A-star用时最大值: 0.02200150489807129

```
self.Grid_Size = 100
```

A-star用时平均值: 0.0610319995880127

A-star用时最小值: 0.04699277877807617

A-star用时最大值: 0.1909940242767334

```
self.Grid_Size = 200
```

A-star用时平均值: 0.018881831645965578

A-star用时最小值: 0.010003089904785156

A-star用时最大值: 0.06999897956848145

虽然这些时间和测试当时电脑性能有关系, 比如我不同时候测试得到的时间就会不同, 无法完全的控制变量, 但是这也可以体现不同的栅格大小一定会影响搜索过程, 过大容易找不到终点, 可能无法完成路径规划; 过小的话也会浪费一定时间, 我推测可能会有某个最值。

RRT

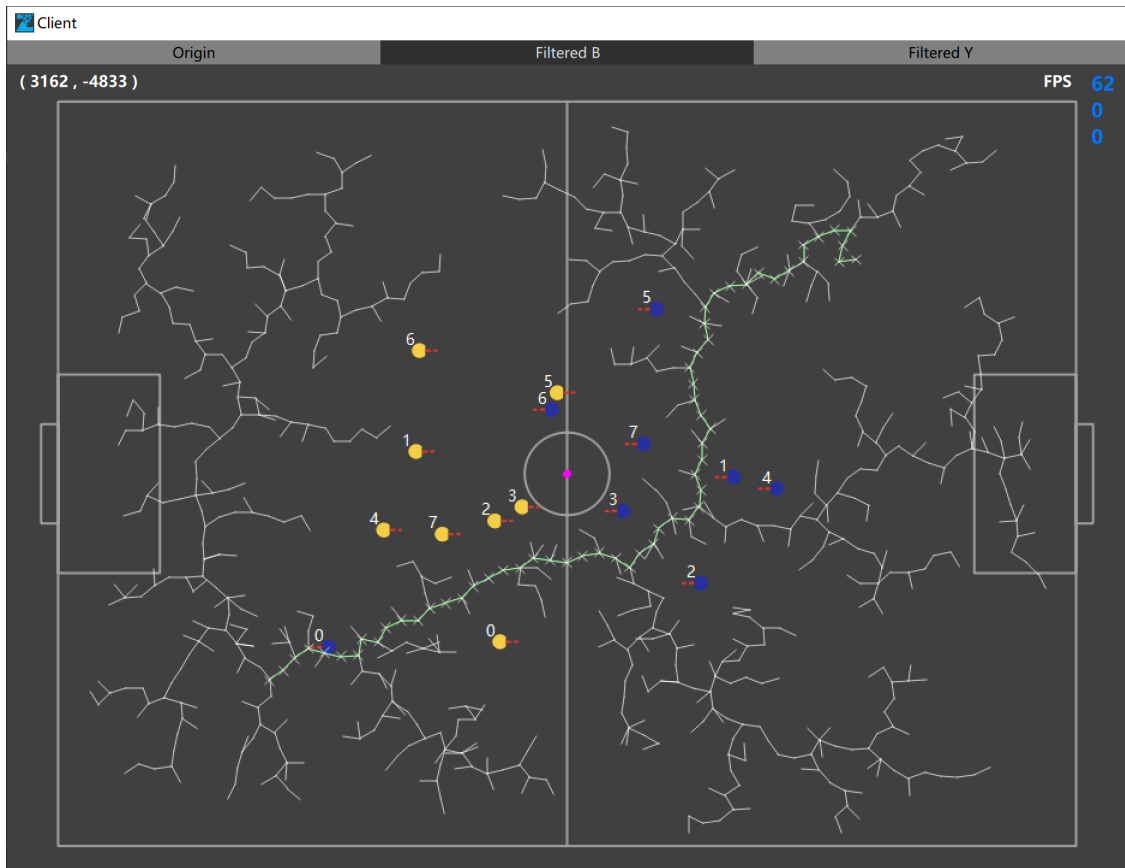
详见/RRT/rrt.py, main2.py

我就是按照这样的规则完成RRT的

Algorithm 3: RRT.

```
1  $V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset;$   
2 for  $i = 1, \dots, n$  do  
3    $x_{\text{rand}} \leftarrow \text{SampleFree}_i;$   
4    $x_{\text{nearest}} \leftarrow \text{Nearest}(G=(V, E), x_{\text{rand}});$   
5    $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}});$   
6   if  $\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$  then  
7      $V \leftarrow V \cup \{x_{\text{new}}\}; E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\};$   
8 return  $G=(V, E);$ 
```

实现效果



我测试了每一次路径规划的时间：

```
本次RRT用时： 0.14661121368408203
Path Found!
本次RRT用时： 1.2935419082641602
Path Found!
本次RRT用时： 0.6791896820068359
Path Found!
本次RRT用时： 1.5678057670593262
Path Found!
本次RRT用时： 0.08976054191589355
Path Found!
本次RRT用时： 0.6023902893066406
Path Found!
本次RRT用时： 2.452441692352295
Path Found!
本次RRT用时： 0.09175491333007812
Path Found!
本次RRT用时： 0.13563847541809082
Path Found!
本次RRT用时： 0.2593057155609131
Path Found!
本次RRT用时： 1.4481277465820312
Path Found!
本次RRT用时： 0.42386579513549805
Path Found!
本次RRT用时： 0.42685651779174805
```

在100次路径规划中，结果为：

RRT用时平均值： 0.4737211179733276

RRT用时最小值： 0.011996984481811523

RRT用时最大值： 3.384000301361084

我发现这表现主要有两个问题：一是时间表现不稳定，有几十毫秒级别的，有一两秒级别的，方差很大，这个我认为可能是由RRT算法的随机取点特性带来的；第二是平均水平时间过长，这个我分析可能是我写的程序中有那些参数或者过程或者数据结构导致用时过长。或者也有可能和电脑不太行有关系？

极端情况测试

这里测试一些极端情况

1.离障碍物距离很近为100多左右：

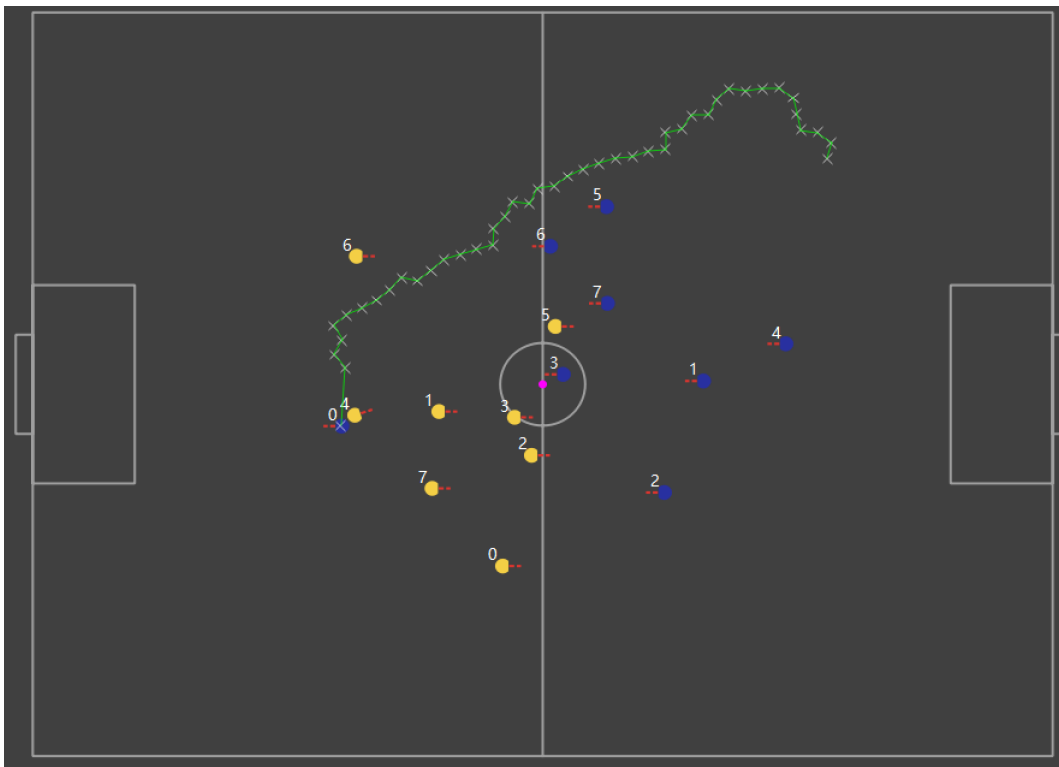
发现这时候程序会直接在最大采样次数后结束，没有找到路径。

我认为问题在于我的expand_length不能做的很大，而初始点和障碍物距离很近的情况下无论怎么扩展都还是在障碍物的那个圆的范围之内，这样新生成的节点永远无法添加，自然找不到路径。

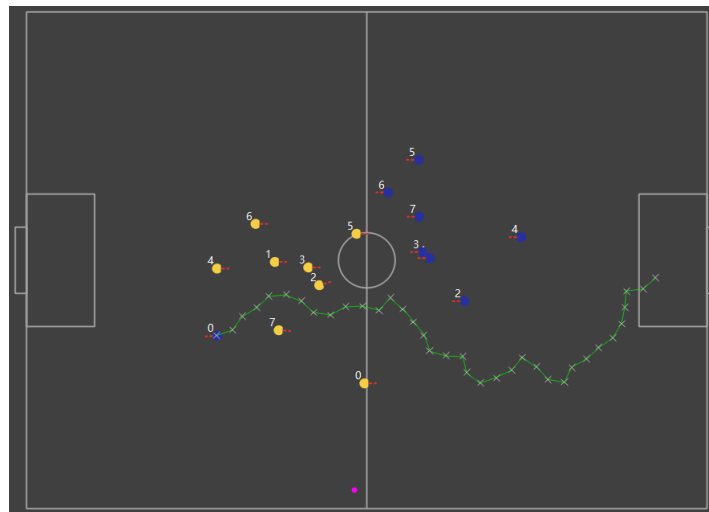
我想的解决办法是如果检测到初始零号的位置和某个障碍物很近，那么就在取点的时候用一个很大的expand_length，这样就跳出那个障碍物的范围，就可以进行正常的RRT过程。

```
for i in range(len(self.obstacle_x)):
    if self.distance(vision.my_robot.x, vision.my_robot.y, self.obstacle_x[i],
self.obstacle_y[i]) < 200:
        flag = 1
        break
if flag == 1 and min_index == 0:
    new_x = nearest_node.x + 700 * math.cos(theta)
    new_y = nearest_node.y + 700 * math.sin(theta)
    new_node = node(new_x, new_y, min_index)
else:
    new_x = nearest_node.x + self.expand_length * math.cos(theta)
    new_y = nearest_node.y + self.expand_length * math.sin(theta)
    new_node = node(new_x, new_y, min_index)
```

这样就会使得零号与某个障碍物距离很近的时候会有一个很大的expand的距离“冲出去”，从而正常完成RRT的路径规划过程。如下图所示：



2.把 (3500, 2500) 添加到障碍物列表中去:



就像上图这样，程序在最大次数之后跳出结束了，就没一次成过。

问题在于检查碰撞的时候， $\text{robot_size} + \text{avoid_size}$ 是400，这是一个不小的圆，而我的精度threshold是300乃至更小，所以所有在这个圆内的新节点都不行，所以即使到达了终点附近，这个节点也不会被添加进去，自然永远找不到终点。

那从这个角度我认为可以做一下几个事情：

(1) 简单粗暴，直接把终点附近的障碍物无视掉，选择一个合适的eps来表示终点附近就好

```
if self.distance(self.obstacle_x[index], self.obstacle_y[index], self.goal_x,
                 self.goal_y) < eps:
    del self.obstacle_x[index]
    del self.obstacle_y[index]
    continue
```

验证一下，这样确实可以找到路径了。反正总是要到终点，即使有障碍物也要怼过去，这样我觉得比较粗暴，也比较直接。但是就是有可能撞到障碍物。

(2) 改变到点精度，大于robot_size+avoid_size，这就看需求如何，如果对精度要求高，那么就不行了

(3) 检测终点附近有没有障碍物，有的话就把avoid_size改小，把精度稍改大，这样可能稳妥保守一些

```
for d in dist:
    if d < self.robot_size + self.avoid_distance and i not in end_obstacle :
        return True
    elif i in end_obstacle:
        if d < self.robot_size + 50:
            return True
        i += 1
return False
```

这样子测试了一下，也可以

改变参数

我认为印象比较大的参数是expand_length，就是每次在最近节点扩展的距离大小，太小了最后收敛速度慢，太大了容易碰到障碍物且容易找不到终点，也不太好，我进行了尝试。

```
self.expand_length = 400
```

RRT用时平均值: 0.33850192070007323
RRT用时最小值: 0.005999565124511719
RRT用时最大值: 2.9939987659454346

```
self.expand_length = 300
```

RRT用时平均值: 0.28003148555755614
RRT用时最小值: 0.006998538970947266
RRT用时最大值: 1.8140039443969727

```
self.expand_length = 200
```

RRT用时平均值: 0.5884209585189819
RRT用时最小值: 0.01099395751953125
RRT用时最大值: 3.2079997062683105

同时，最大采样个数会影响找到路径的成功率：

在同一个expand_length下进行改变N_sample：

```
self.N_sample = 1000
```

RRT找到路径的成功率为 0.6

```
self.N_sample = 3000
```

RRT找到路径的成功率为 0.98

当然我这里成功率很低也和我设置的到点精度高有关系，改低一点成功率就高了

Bidirectional-RRT

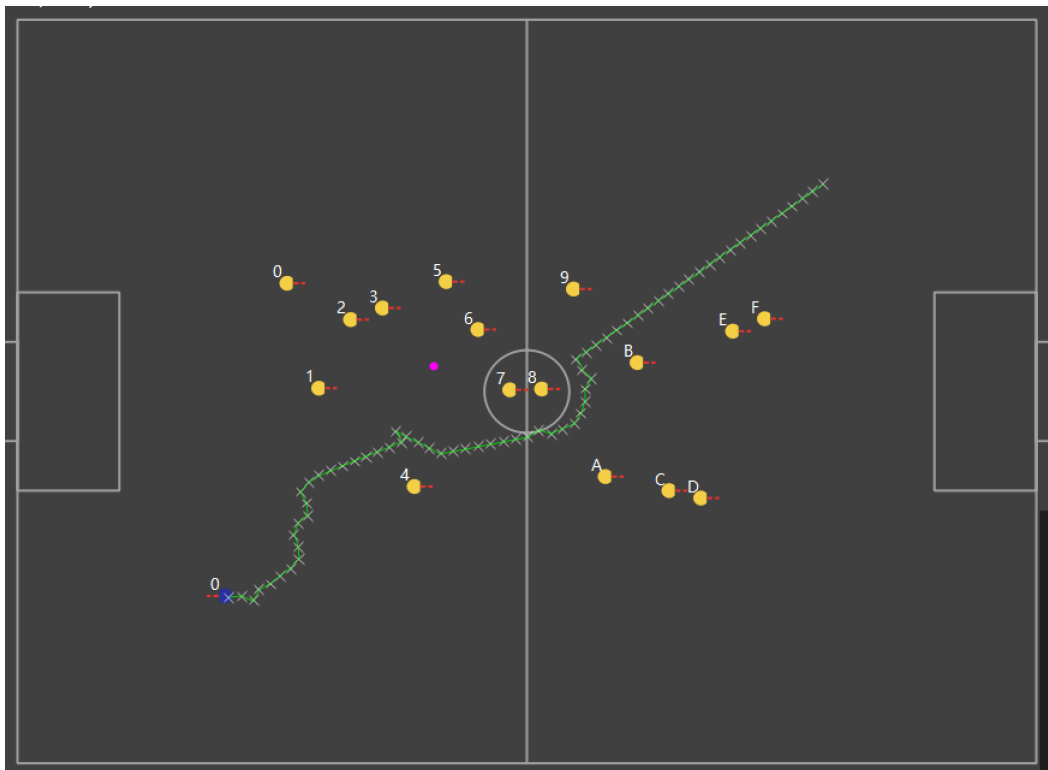
详见/RRT/bi_rrt.py, main3.py

双向搜索这样更具有方向性，提高了搜索的效率。我认为会对平均用时有缩短的作用。

我就是按照下面这样的伪代码去完成的：

```
1.  $V_1 \leftarrow \{q_{init}\}; E_1 \leftarrow \emptyset; G_1 \leftarrow (V_1, E_1);$ 
2.  $V_2 \leftarrow \{q_{goal}\}; E_2 \leftarrow \emptyset; G_2 \leftarrow (V_2, E_2); i \leftarrow 0;$ 
3. while  $i < N$  do
4.    $q_{rand} \leftarrow \text{Sample}(i); i \leftarrow i + 1;$ 
5.    $q_{nearest} \leftarrow \text{Nearst}(G_1, q_{rand});$ 
6.    $q_{new} \leftarrow \text{Steer}(q_{nearest}, q_{rand});$ 
7.   if  $\text{ObstacleFree}(q_{nearest}, q_{new})$  then
8.      $V_1 \leftarrow V_1 \cup \{q_{new}\};$ 
9.      $E_1 \leftarrow E_1 \cup \{(q_{nearest}, q_{new})\};$ 
10.     $q'_{nearest} \leftarrow \text{Nearst}(G_2, q_{new});$ 
11.     $q'_{new} \leftarrow \text{Steer}(q'_{nearest}, q_{new});$ 
12.    if  $\text{ObstacleFree}(q'_{nearest}, q'_{new})$  then
13.       $V_2 \leftarrow V_2 \cup \{q'_{new}\};$ 
14.       $E_2 \leftarrow E_2 \cup \{(q'_{nearest}, q'_{new})\};$ 
15.      do
16.         $q''_{new} \leftarrow \text{Steer}(q'_{new}, q_{new});$ 
17.        if  $\text{ObstacleFree}(q''_{new}, q'_{new})$  then
18.           $V_2 \leftarrow V_2 \cup \{q''_{new}\};$ 
19.           $E_2 \leftarrow E_2 \cup \{(q'_{new}, q''_{new})\};$ 
20.           $q'_{new} \leftarrow q''_{new};$ 
21.        else break;
22.      while not  $q'_{new} = q_{new}$ 
23.      if  $q'_{new} = q_{new}$  then return  $(V_1, E_1);$ 
24.      if  $|V_2| < |V_1|$  then  $\text{Swap}(V_1, V_2);$ 
```

得到的效果是：



测试路径规划的时间：

50次测试

RRT用时平均值: 0.02720062732696533
RRT用时最小值: 0.0010023117065429688
RRT用时最大值: 0.06899619102478027

100次测试

RRT用时平均值: 0.07872150421142578
RRT用时最小值: 0.0009989738464355469
RRT用时最大值: 0.25900745391845703

可以看出, 这样的双向的RRT使得总共节点数减少, 搜索效率提高很多, 而且时间的方差变小, 性能更加稳定, 而且在更小的N_sample下成功率不变, 这样的优化提高还是可以的。

数据结构改进

详见/RRT/rrt_ds.py, main4.py

我在想哪里浪费了时间, 有可能是寻找最近邻的过程中浪费了时间, 所以想试着改一改。

我看到当时大作业助教给的prm的代码里用到了KD Tree去寻找最近邻, 而寻找最近邻也是最费时间的一个过程, 所以我想试试。

```
def find_nearest_node(self, x, y, node):  
    tree = KDTree(np.vstack((x, y)).T)  
    distance, index = tree.query(np.array([node.x, node.y]), eps=0.5)  
    return index
```

得到的结果:

RRT用时平均值: 0.47428073883056643
RRT用时最小值: 0.050998687744140625
RRT用时最大值: 1.8349909782409668

改变eps=5

RRT用时平均值: 0.3331661891937256
RRT用时最小值: 0.03999948501586914
RRT用时最大值: 1.792997121810913

我觉得可能可以起到加速的作用, 尤其是设置eps可以返回approximate最近邻, 但是可能我理解的还是不够深, 所以用的也不好, 就是照猫画虎, 总的来讲我认为可能一个好的数据结构会让性能改变很多的。