RRT路径规划

18329015 郝裕玮

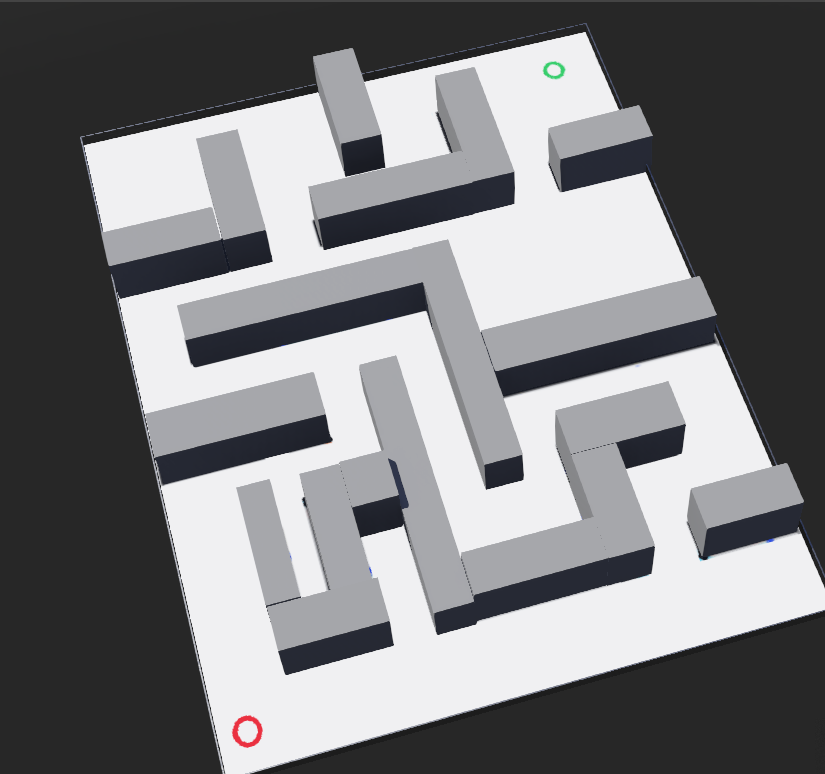
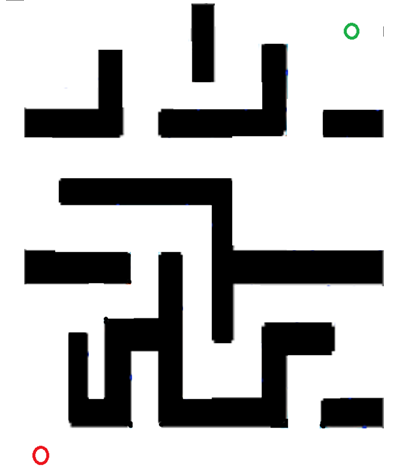
一、实验目标

使用RRT算法进行路径规划。

实验要求：绿色方块代表起始位置，红色方块代表目标位置，要求在已知地图全局信息的情况下，规划一条尽可能短的轨迹，控制机器人从绿色走到红色。

实验场景：给定了迷宫webots模型，地图的全局信息通过读取

maze.png这个图片来获取。

二、实验内容与步骤

（1）实验环境具体配置如下：

编程语言：Python 3.7.0

编辑器：Jupyter Notebook (anaconda 3)

第三方库：OpenCV（cmd执行pip install opencv-python即可）

（2）使用RRT算法对maze.png进行处理，画出最优路径。

RRT算法与之前实验的PRM算法十分类似，都是通过抽样来在已知的地图上建立无向图，进而通过搜索方法寻找相对最优的路径。

不同点在于，PRM算法在一开始就通过抽样在地图上构建出完整的无向图，再进行图搜索；而RRT算法则是从某个点出发，一边搜索一边抽样，并建图。

RRT算法主要分为以下3个阶段：

**注意：下述每个部分的代码注释中均已包含所有的思路分析和算法实现：**

1，运行主函数获取相关参数，并在RoadMap类中对图像进行预处理，使之二值化成为网格地图，同时构建RRT树的根节点：

①主函数

#主函数

if \_\_name\_\_=="\_\_main\_\_":

    #读取图像

    image\_path = "C:\\Users\\93508\\Desktop\\maze.png"

    img = cv2.imread(image\_path)

    #开始RRT路径规划

    print('开始RRT路径规划...')

    res = RoadMap(img)

    #这里可修改三项参数步长step，距离阈值dis，尝试次数cnt

    res.rrt\_planning(step = 10, dis = 10, cnt = 200000)

②图像预处理

#RoadMap类：读入图片，将其二值化为网格图，并进行一系列操作

class RoadMap(object):

    def \_\_init\_\_(self, map\_img):

        #读取图像尺寸

        self.length = map\_img.shape[0]

        self.width = map\_img.shape[1]

        #为方便后续操作，将图像尺寸的长宽均设置为更小的那个值（图片转为正方形）

        squad = min(self.length, self.width)

        map\_img = cv2.resize(map\_img,(squad,squad))

        self.map = map\_img

        #s设置图像的起点和终点

        self.point\_start = np.mat([550, 20]) #运动规划的起点

        self.point\_end = np.mat([20,565]) #运动规划的终点

    # RRT算法

    def rrt\_planning(self, \*\*param):

        ''' 快速扩展随机树算法（RRT算法）

        Args:

            \*\*param: 关键字参数，用以配置规划参数

                     step: 搜索步长，默认20。int

                     dis: 判断阈值，默认20。float

                     cnt: 尝试次数。默认20000。int

        Return:

            本函数没有返回值，但会根据计算结果赋值（或定义）以下属性变量：

                self.rrt\_tree: 所生成的rrt树。numpy.mat

                    数据含义: [[横坐标, 纵坐标, 父节点索引]]，其中第一个点（起点）为根，最后一个点（终点）为树枝

        Example:

            mr = RoadMap(img)

            mr.rrt\_planning(s=25, t=30, l=15000, pic='None')

        '''

        # 关键字参数处理

        if 'step' in param:

            step\_size = param['step'] #搜索步长step\_size

        if 'dis' in param:

            area = param['dis'] #判断阈值area

        if 'cnt' in param:

            limit\_try = param['cnt'] #尝试次数limit\_try

        # 地图灰度化

        image\_gray = cv2.cvtColor(self.map, cv2.COLOR\_BGR2GRAY)

        # 地图二值化

        # cv2.THRESH\_BINARY表示阈值的二值化操作，大于阈值使用maxval表示，小于阈值使用0表示

        # 大于127的像素点置为255（白色），小于127的像素点置为0（黑色）

        ret,img\_binary = cv2.threshold(image\_gray, 127, 255, cv2.THRESH\_BINARY)

        # 初始化 RRT 树:[横坐标，纵坐标，父节点索引]

        rrt\_tree = np.hstack((self.point\_start, [[0]]))

        # 初始化尝试次数

        num\_try = 0

        # 路径规划是否成功

        path\_found = False

2，开始RRT算法核心部分：

①随机采样：每次选择生长方向时，有一定的概率会向着目标点延伸，也有一定的概率会随机在地图内选择一个方向延伸一段距离。

        #开始limit\_try次随机尝试

        while num\_try <= limit\_try:

            #随机生成采样点

            #在每次选择生长方向时，有一定的概率会向着目标点延伸

            #也有一定的概率会随机在地图内选择一个方向延伸一段距离

            #在这里设置两种概率均为0.5

            if np.random.rand() < 0.5:

                #在地图范围内随机采样一个像素

                sample = np.mat(np.random.randint(0, img\_binary.shape[0] - 1, (1, 2)))

            else:

                sample = self.point\_end

②生长点选择与碰撞检测：假设我们采样了图像中的某个随机点，那么我们可从现有的RRT树中筛选出离该采样点最近的一个点，并向采样点生长一段距离（长为step\_size）。并对检验生长过程中是否发生碰撞（两点间连线有无障碍点），最后还要保证该采样点与RRT树中现有的所有点的距离均大于距离阈值area（若两点距离小于area则默认为两个点重合，是同一个点）。若以上2个条件均满足，则将该采样点加入RRT树中。

同理，当某个采样点与终点的距离 < area时，则认为已经到达终点，将其加入RRT树之后可终止循环。

            #计算各点与当前随机采样点的距离

            #找出rrt树中离当前随机采样点最近的点

            mat\_distance = EuclidenDistance(rrt\_tree[:, 0 : 2], sample)

            #argmin用于找出距离最小点的索引

            index\_close = np.argmin(mat\_distance, 0)[0, 0] #末尾索引用来取出数值，否则index\_close变为矩阵

            point\_close = rrt\_tree[index\_close, 0 : 2]

            #从距离最小点向当前采样点移动step\_size距离，并进行碰撞检测

            #计算出移动方向（角度）

            theta\_dir = math.atan2(sample[0, 0] - point\_close[0, 0], sample[0, 1] - point\_close[0, 1])

            #得到移动后的点point\_new

            point\_new = point\_close + step\_size \* np.mat([math.sin(theta\_dir), math.cos(theta\_dir)])

            #将坐标化为整数（矩阵转数组，元素int化，再转矩阵）

            point\_new = np.around(np.array(point\_new)).astype(int)

            point\_new = np.mat(point\_new)

            #若两点间连线失败（有障碍物），则继续下一次循环

            if not check\_path(point\_close, point\_new, img\_binary):

                num\_try = num\_try + 1

                continue

            #若连线成功，则先检验point\_new和终点point\_end的距离是否小于判断阈值area

            #若在范围内则代表两个点属于同一个点，默认路径规划成功，已到达终点

            if EuclidenDistance(point\_new, self.point\_end) < area:

                path\_found = True

                #将point\_new加入到rrt树，设置为新节点

                point\_new = np.hstack((point\_new, [[index\_close]]))

                rrt\_tree = np.vstack((rrt\_tree, point\_new))

                break

            #若point\_new尚未到达终点的邻域范围内，则计算rrt树中各点与point\_new的距离

            mat\_distance = EuclidenDistance(rrt\_tree[:, 0 : 2], point\_new)

            if np.min(mat\_distance, 0) < area:

                num\_try = num\_try + 1 #若存在距离小于area的，则point\_new与该点重合，直接继续下次循环

                continue

            #若均大于判断阈值area，则证明point\_new未与rrt树中任何一个点重合，可将其添加到rrt树中

            #设置离新点point\_new最近的节点为其父节点

            #父节点索引为index\_close

            else:

                point\_new = np.hstack((point\_new, [[index\_close]]))

                rrt\_tree = np.vstack((rrt\_tree, point\_new))

碰撞检测相关函数代码如下所示：

#碰撞检测，检验路径上的点是否越界或为障碍物点（与check\_path共同检验）

def check\_point(point, map\_img):

    point = np.mat(point) #先将这些点转换为矩阵

    not\_obstacle = True #验证是否为障碍点

    #若该点仍在图像范围内

    if (point[:, 0] < map\_img.shape[0] and

        point[:, 1] < map\_img.shape[1] and

        point[:, 0] >= 0 and

        point[:, 1] >= 0):

        #若在图像范围内但路径上有点的像素值为0（黑色，即该路径中间碰到了障碍）

        if map\_img[point[:, 1], point[:, 0]] == 0:

            not\_obstacle = False

    else:#路径上有某一点已经不在图像范围内

        not\_obstacle = False

    return not\_obstacle

#检验某点周围的邻域范围内是否无障碍点

def not\_obstacle\_in\_area(x, y, d, map\_img):

    #d代表邻域范围，检验的范围为与该点距离为d的4邻域（上下左右）和距离为根号2d的8邻域（4个斜向方向）

    if x < map\_img.shape[1] and x >= 0 and y < map\_img.shape[0] and y >= 0 and map\_img[x,y]!=0 \

    and x-d < map\_img.shape[1] and x-d >= 0 and y < map\_img.shape[0] and y >= 0 and map\_img[x-d,y]!=0 \

    and x < map\_img.shape[1] and x >= 0 and y-d < map\_img.shape[0] and y-d >= 0 and map\_img[x,y-d]!=0 \

    and x-d < map\_img.shape[1] and x-d >= 0 and y-d < map\_img.shape[0] and y-d >= 0 and map\_img[x-d,y-d]!=0 \

    and x+d < map\_img.shape[1] and x+d >= 0 and y < map\_img.shape[0] and y >= 0 and map\_img[x+d,y]!=0 \

    and x < map\_img.shape[1] and x >= 0 and y+d < map\_img.shape[0] and y+d >= 0 and map\_img[x,y+d]!=0 \

    and x+d < map\_img.shape[1] and x+d >= 0 and y+d < map\_img.shape[0] and y+d >= 0 and map\_img[x+d,y+d]!=0 \

    and x+d < map\_img.shape[1] and x+d >= 0 and y-d < map\_img.shape[0] and y-d >= 0 and map\_img[x+d,y-d]!=0 \

    and x-d < map\_img.shape[1] and x-d >= 0 and y+d < map\_img.shape[0] and y+d >= 0 and map\_img[x-d,y+d]!=0:

        return True #若8个邻域点和自身均不为障碍物点，则默认为该点邻域范围内无障碍点

    return False

#碰撞检测，检验路径上的点是否越界或为障碍物点（与check\_point共同检验）

def check\_path(point\_current, point\_other, map\_img):

    #首先确保连线的两点周围的邻域范围内无障碍点

    if not\_obstacle\_in\_area(point\_current[0,1], point\_current[0,0], 11, map\_img) \

    and not\_obstacle\_in\_area(point\_other[0,1], point\_other[0,0], 11, map\_img):

        #取横向、纵向较大值，确保经过的每个像素都被检测到

        step\_length = max(abs(point\_current[0, 0] - point\_other[0, 0]), abs(point\_current[0, 1] - point\_other[0, 1]))

        path\_x = np.linspace(point\_current[0, 0], point\_other[0, 0], step\_length + 1)

        path\_y = np.linspace(point\_current[0, 1], point\_other[0, 1], step\_length + 1)

        #检验路径连线上的点是否越界或为障碍点（调用check\_point函数）

        for i in range(int(step\_length + 1)):

            if check\_point([int(math.ceil(path\_x[i])), int(math.ceil(path\_y[i]))], map\_img):

                return True

    return False

两点间距离计算相关函数代码如下所示：

#计算两点间欧氏距离（即直线距离）

def EuclidenDistance(point\_a, point\_b):

    #point\_a可以是矩阵形式的点集，该函数将返回一个矩阵，每行对应各点与point\_b的直线距离

    distance = np.sqrt(np.sum(np.multiply(point\_a - point\_b, point\_a - point\_b), axis=1))

    return distance

3，画出RRT树形图（探索过程）以及最终结果：

#画出RRT树形图和最终路线图

def result\_plot(map\_img, rrt\_tree, length, width):

    # 首先绘制树形图

    # 设置树形图图相关参数：点的大小，颜色和线的粗细

    point\_size = 3

    point\_color = (0, 127, 0)

    thickness = 4

    #将矩阵转化为数组并转为整型，再转化为元组，以供cv2使用

    vertex = np.around(np.array(rrt\_tree)).astype(int)

    vertex\_tuple = tuple(map(tuple, vertex))

    map\_img1 = copy.deepcopy(map\_img) #需要进行深拷贝，而不是引用，否则会导致img1和img2一样

    #画出RRT树中所有点的节点并连线

    for point in vertex\_tuple:

        cv2.circle(map\_img1, point[0 : 2], point\_size, point\_color, thickness)

        if point[0] != 0:

            cv2.line(map\_img1, point[0 : 2], vertex\_tuple[point[2]][0 : 2], (255,150,150), 2)

    #通过回溯来绘制最优路径

    #并且通过优化连接点数量使得最终路径更加平滑

    #将离目标点最近的点a与其父节点b的父节点c进行连接，再从点c开始继续这个循环

    #三角形法则：(ac < ab + bc)，得到的路径比最初的路径会更短且更平滑

    point\_a\_index = -1 #用于定位rrt\_tree数组的最后一个元素（离目标点最近的点a）

    while point\_a\_index != 0: #直至遍历到rrt\_tree数组的第一个元素（即遍历结束）

        point\_b\_index = rrt\_tree[point\_a\_index, 2] #point\_b\_index为a的父节点索引

        point\_c\_index = rrt\_tree[point\_b\_index, 2] #point\_c\_index为b的父节点索引

        #连接a与c，(0,0,0)代表连线颜色的RGB(黑色)，3为线段粗细程度

        cv2.line(map\_img1,vertex\_tuple[point\_a\_index][0 : 2], vertex\_tuple[point\_c\_index][0 : 2],(0,0,0),3)

        point\_a\_index = point\_c\_index #将起始点转移到c，继续该循环

    img1 = cv2.resize(map\_img1,(width,length))#将图像尺寸变为初始尺寸

    cv2.imwrite('C:\\Users\\93508\\Desktop\\111.png', img1)#图像存储路径

    #去掉无关点，绘制用于小车巡线的最终路线图

    map\_img2 = copy.deepcopy(map\_img) #需要进行深拷贝，而不是引用，否则会导致img1和img2一样

    #相同的步骤绘制路线图

    point\_a\_index = -1 #用于定位rrt\_tree数组的最后一个元素（离目标点最近的点a）

    while point\_a\_index != 0: #直至遍历到rrt\_tree数组的第一个元素（即遍历结束）

        point\_b\_index = rrt\_tree[point\_a\_index, 2] #point\_b\_index为a的父节点索引

        point\_c\_index = rrt\_tree[point\_b\_index, 2] #point\_c\_index为b的父节点索引

        #连接a与c，(0,0,0)代表连线颜色的RGB(黑色)，3为线段粗细程度

        cv2.line(map\_img2,vertex\_tuple[point\_a\_index][0 : 2], vertex\_tuple[point\_c\_index][0 : 2],(0,0,0),3)

        point\_a\_index = point\_c\_index #将起始点转移到c，继续该循环

    img2 = cv2.resize(map\_img2,(width,length))#将图像尺寸变为初始尺寸

    cv2.imwrite('C:\\Users\\93508\\Desktop\\222.png', img2)#图像存储路径

    #将img1和img2放在同一个窗口下展示

    imgs = np.hstack([img1,img2])

    cv2.imshow("1", imgs)

    cv2.waitKey()#防止图像一闪而过

至此，代码的主要部分均已展示，接下来为体现代码整体逻辑，展示全体代码。

代码的逻辑关系为：RoadMap类用于图像预处理与RRT算法。在主函数和RoadMap类以外还有一些功能性函数（判断像素点是否为障碍点，计算两点间欧氏距离，碰撞检测，画图）。

全体代码如下（主函数在代码最下方）：

import cv2 #图像处理需要的库 OpenCV

import numpy as np

import math

import copy

#碰撞检测，检验路径上的点是否越界或为障碍物点（与check\_path共同检验）

def check\_point(point, map\_img):

    point = np.mat(point) #先将这些点转换为矩阵

    not\_obstacle = True #验证是否为障碍点

    #若该点仍在图像范围内

    if (point[:, 0] < map\_img.shape[0] and

        point[:, 1] < map\_img.shape[1] and

        point[:, 0] >= 0 and

        point[:, 1] >= 0):

        #若在图像范围内但路径上有点的像素值为0（黑色，即该路径中间碰到了障碍）

        if map\_img[point[:, 1], point[:, 0]] == 0:

            not\_obstacle = False

    else:#路径上有某一点已经不在图像范围内

        not\_obstacle = False

    return not\_obstacle

#检验某点周围的邻域范围内是否无障碍点

def not\_obstacle\_in\_area(x, y, d, map\_img):

    #d代表邻域范围，检验的范围为与该点距离为d的4邻域（上下左右）和距离为根号2d的8邻域（4个斜向方向）

    if x < map\_img.shape[1] and x >= 0 and y < map\_img.shape[0] and y >= 0 and map\_img[x,y]!=0 \

    and x-d < map\_img.shape[1] and x-d >= 0 and y < map\_img.shape[0] and y >= 0 and map\_img[x-d,y]!=0 \

    and x < map\_img.shape[1] and x >= 0 and y-d < map\_img.shape[0] and y-d >= 0 and map\_img[x,y-d]!=0 \

    and x-d < map\_img.shape[1] and x-d >= 0 and y-d < map\_img.shape[0] and y-d >= 0 and map\_img[x-d,y-d]!=0 \

    and x+d < map\_img.shape[1] and x+d >= 0 and y < map\_img.shape[0] and y >= 0 and map\_img[x+d,y]!=0 \

    and x < map\_img.shape[1] and x >= 0 and y+d < map\_img.shape[0] and y+d >= 0 and map\_img[x,y+d]!=0 \

    and x+d < map\_img.shape[1] and x+d >= 0 and y+d < map\_img.shape[0] and y+d >= 0 and map\_img[x+d,y+d]!=0 \

    and x+d < map\_img.shape[1] and x+d >= 0 and y-d < map\_img.shape[0] and y-d >= 0 and map\_img[x+d,y-d]!=0 \

    and x-d < map\_img.shape[1] and x-d >= 0 and y+d < map\_img.shape[0] and y+d >= 0 and map\_img[x-d,y+d]!=0:

        return True #若8个邻域点和自身均不为障碍物点，则默认为该点邻域范围内无障碍点

    return False

#碰撞检测，检验路径上的点是否越界或为障碍物点（与check\_point共同检验）

def check\_path(point\_current, point\_other, map\_img):

    #首先确保连线的两点周围的邻域范围内无障碍点

    if not\_obstacle\_in\_area(point\_current[0,1], point\_current[0,0], 11, map\_img) \

    and not\_obstacle\_in\_area(point\_other[0,1], point\_other[0,0], 11, map\_img):

        #取横向、纵向较大值，确保经过的每个像素都被检测到

        step\_length = max(abs(point\_current[0, 0] - point\_other[0, 0]), abs(point\_current[0, 1] - point\_other[0, 1]))

        path\_x = np.linspace(point\_current[0, 0], point\_other[0, 0], step\_length + 1)

        path\_y = np.linspace(point\_current[0, 1], point\_other[0, 1], step\_length + 1)

        #检验路径连线上的点是否越界或为障碍点（调用check\_point函数）

        for i in range(int(step\_length + 1)):

            if check\_point([int(math.ceil(path\_x[i])), int(math.ceil(path\_y[i]))], map\_img):

                return True

    return False

#计算两点间欧氏距离（即直线距离）

def EuclidenDistance(point\_a, point\_b):

    #point\_a可以是矩阵形式的点集，该函数将返回一个矩阵，每行对应各点与point\_b的直线距离

    distance = np.sqrt(np.sum(np.multiply(point\_a - point\_b, point\_a - point\_b), axis=1))

    return distance

#画出RRT树形图和最终路线图

def result\_plot(map\_img, rrt\_tree, length, width):

    # 首先绘制树形图

    # 设置树形图图相关参数：点的大小，颜色和线的粗细

    point\_size = 3

    point\_color = (0, 127, 0)

    thickness = 4

    #将矩阵转化为数组并转为整型，再转化为元组，以供cv2使用

    vertex = np.around(np.array(rrt\_tree)).astype(int)

    vertex\_tuple = tuple(map(tuple, vertex))

    map\_img1 = copy.deepcopy(map\_img) #需要进行深拷贝，而不是引用，否则会导致img1和img2一样

    #画出RRT树中所有点的节点并连线

    for point in vertex\_tuple:

        cv2.circle(map\_img1, point[0 : 2], point\_size, point\_color, thickness)

        if point[0] != 0:

            cv2.line(map\_img1, point[0 : 2], vertex\_tuple[point[2]][0 : 2], (255,150,150), 2)

    #通过回溯来绘制最优路径

    #并且通过优化连接点数量使得最终路径更加平滑

    #将离目标点最近的点a与其父节点b的父节点c进行连接，再从点c开始继续这个循环

    #三角形法则：(ac < ab + bc)，得到的路径比最初的路径会更短且更平滑

    point\_a\_index = -1 #用于定位rrt\_tree数组的最后一个元素（离目标点最近的点a）

    while point\_a\_index != 0: #直至遍历到rrt\_tree数组的第一个元素（即遍历结束）

        point\_b\_index = rrt\_tree[point\_a\_index, 2] #point\_b\_index为a的父节点索引

        point\_c\_index = rrt\_tree[point\_b\_index, 2] #point\_c\_index为b的父节点索引

        #连接a与c，(0,0,0)代表连线颜色的RGB(黑色)，3为线段粗细程度

        cv2.line(map\_img1,vertex\_tuple[point\_a\_index][0 : 2], vertex\_tuple[point\_c\_index][0 : 2],(0,0,0),3)

        point\_a\_index = point\_c\_index #将起始点转移到c，继续该循环

    img1 = cv2.resize(map\_img1,(width,length))#将图像尺寸变为初始尺寸

    cv2.imwrite('C:\\Users\\93508\\Desktop\\tree.png', img1)#图像存储路径

    #去掉无关点，绘制用于小车巡线的最终路线图

    map\_img2 = copy.deepcopy(map\_img) #需要进行深拷贝，而不是引用，否则会导致img1和img2一样

    #相同的步骤绘制路线图

    point\_a\_index = -1 #用于定位rrt\_tree数组的最后一个元素（离目标点最近的点a）

    while point\_a\_index != 0: #直至遍历到rrt\_tree数组的第一个元素（即遍历结束）

        point\_b\_index = rrt\_tree[point\_a\_index, 2] #point\_b\_index为a的父节点索引

        point\_c\_index = rrt\_tree[point\_b\_index, 2] #point\_c\_index为b的父节点索引

        #连接a与c，(0,0,0)代表连线颜色的RGB(黑色)，3为线段粗细程度

        cv2.line(map\_img2,vertex\_tuple[point\_a\_index][0 : 2], vertex\_tuple[point\_c\_index][0 : 2],(0,0,0),3)

        point\_a\_index = point\_c\_index #将起始点转移到c，继续该循环

    img2 = cv2.resize(map\_img2,(width,length))#将图像尺寸变为初始尺寸

    cv2.imwrite('C:\\Users\\93508\\Desktop\\result.png', img2)#图像存储路径

    #将img1和img2放在同一个窗口下展示

    imgs = np.hstack([img1,img2])

    cv2.imshow("1", imgs)

    cv2.waitKey()#防止图像一闪而过

#RoadMap类：读入图片，将其二值化为网格图，并进行一系列操作

class RoadMap(object):

    def \_\_init\_\_(self, map\_img):

        #读取图像尺寸

        self.length = map\_img.shape[0]

        self.width = map\_img.shape[1]

        #为方便后续操作，将图像尺寸的长宽均设置为更小的那个值（图片转为正方形）

        squad = min(self.length, self.width)

        map\_img = cv2.resize(map\_img,(squad,squad))

        self.map = map\_img

        #s设置图像的起点和终点

        self.point\_start = np.mat([550, 20]) #运动规划的起点

        self.point\_end = np.mat([20,565]) #运动规划的终点

    # RRT算法

    def rrt\_planning(self, \*\*param):

        ''' 快速扩展随机树算法（RRT算法）

        Args:

            \*\*param: 关键字参数，用以配置规划参数

                     step: 搜索步长，默认20。int

                     dis: 判断阈值，默认20。float

                     cnt: 尝试次数。默认20000。int

        Return:

            本函数没有返回值，但会根据计算结果赋值（或定义）以下属性变量：

                self.rrt\_tree: 所生成的rrt树。numpy.mat

                    数据含义: [[横坐标, 纵坐标, 父节点索引]]，其中第一个点（起点）为根，最后一个点（终点）为树枝

        Example:

            mr = RoadMap(img)

            mr.rrt\_planning(s=25, t=30, l=15000, pic='None')

        '''

        # 关键字参数处理

        if 'step' in param:

            step\_size = param['step'] #搜索步长step\_size

        if 'dis' in param:

            area = param['dis'] #判断阈值area

        if 'cnt' in param:

            limit\_try = param['cnt'] #尝试次数limit\_try

        # 地图灰度化

        image\_gray = cv2.cvtColor(self.map, cv2.COLOR\_BGR2GRAY)

        # 地图二值化

        # cv2.THRESH\_BINARY表示阈值的二值化操作，大于阈值使用maxval表示，小于阈值使用0表示

        # 大于127的像素点置为255（白色），小于127的像素点置为0（黑色）

        ret,img\_binary = cv2.threshold(image\_gray, 127, 255, cv2.THRESH\_BINARY)

        # 初始化 RRT 树:[横坐标，纵坐标，父节点索引]

        rrt\_tree = np.hstack((self.point\_start, [[0]]))

        # 初始化尝试次数

        num\_try = 0

        # 路径规划是否成功

        path\_found = False

        #开始limit\_try次随机尝试

        while num\_try <= limit\_try:

            #随机生成采样点

            #在每次选择生长方向时，有一定的概率会向着目标点延伸

            #也有一定的概率会随机在地图内选择一个方向延伸一段距离

            #在这里设置两种概率均为0.5

            if np.random.rand() < 0.5:

                #在地图范围内随机采样一个像素

                sample = np.mat(np.random.randint(0, img\_binary.shape[0] - 1, (1, 2)))

            else:

                sample = self.point\_end

            #计算各点与当前随机采样点的距离

            #找出rrt树中离当前随机采样点最近的点

            mat\_distance = EuclidenDistance(rrt\_tree[:, 0 : 2], sample)

            #argmin用于找出距离最小点的索引

            index\_close = np.argmin(mat\_distance, 0)[0, 0] #末尾索引用来取出数值，否则index\_close变为矩阵

            point\_close = rrt\_tree[index\_close, 0 : 2]

            #从距离最小点向当前采样点移动step\_size距离，并进行碰撞检测

            #计算出移动方向（角度）

            theta\_dir = math.atan2(sample[0, 0] - point\_close[0, 0], sample[0, 1] - point\_close[0, 1])

            #得到移动后的点point\_new

            point\_new = point\_close + step\_size \* np.mat([math.sin(theta\_dir), math.cos(theta\_dir)])

            #将坐标化为整数（矩阵转数组，元素int化，再转矩阵）

            point\_new = np.around(np.array(point\_new)).astype(int)

            point\_new = np.mat(point\_new)

            #若两点间连线失败（有障碍物），则继续下一次循环

            if not check\_path(point\_close, point\_new, img\_binary):

                num\_try = num\_try + 1

                continue

            #若连线成功，则先检验point\_new和终点point\_end的距离是否小于判断阈值area

            #若在范围内则代表两个点属于同一个点，默认路径规划成功，已到达终点

            if EuclidenDistance(point\_new, self.point\_end) < area:

                path\_found = True

                #将point\_new加入到rrt树，设置为新节点

                point\_new = np.hstack((point\_new, [[index\_close]]))

                rrt\_tree = np.vstack((rrt\_tree, point\_new))

                break

            #若point\_new尚未到达终点的邻域范围内，则计算rrt树中各点与point\_new的距离

            mat\_distance = EuclidenDistance(rrt\_tree[:, 0 : 2], point\_new)

            if np.min(mat\_distance, 0) < area:

                num\_try = num\_try + 1 #若存在距离小于area的，则point\_new与该点重合，直接继续下次循环

                continue

            #若均大于判断阈值area，则证明point\_new未与rrt树中任何一个点重合，可将其添加到rrt树中

            #设置离新点point\_new最近的节点为其父节点

            #父节点索引为index\_close

            else:

                point\_new = np.hstack((point\_new, [[index\_close]]))

                rrt\_tree = np.vstack((rrt\_tree, point\_new))

        #路径规划成功则开始画图，反之不画图

        if path\_found == True:

            print('规划成功！')

            self.rrt\_tree = rrt\_tree

            result\_plot(self.map, self.rrt\_tree, self.length, self.width) #绘图

        else:

            print('没有找到解。')

#主函数

if \_\_name\_\_=="\_\_main\_\_":

    #读取图像

    image\_path = "C:\\Users\\93508\\Desktop\\maze.png" #路径可修改

    img = cv2.imread(image\_path)

    #开始RRT路径规划

    print('开始RRT路径规划...')

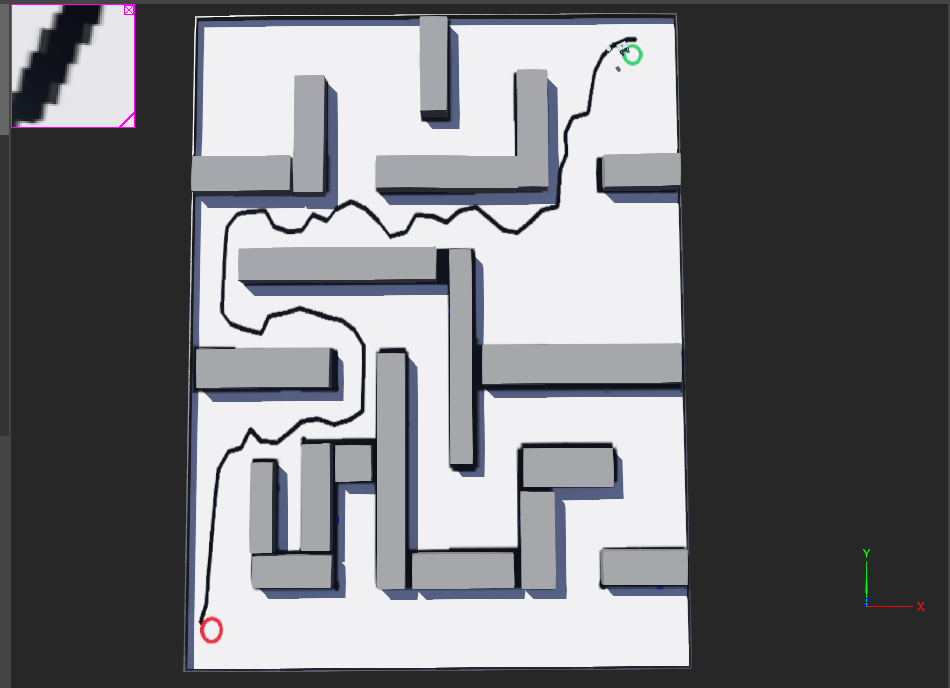
    res = RoadMap(img)

    #这里可修改三项参数步长step，距离阈值dis，尝试次数cnt

    res.rrt\_planning(step = 10, dis = 10, cnt = 200000)

（3）在画出最短路径后，将“巡线小车”实验中的Robot节点导入，并调整机器人的位置translation和大小scale，防止小车过大与障碍物发生碰撞。最后导入画有最短路径的图片，如下图所示（见下页）。

最优路径图片详见压缩包中result.png。



（4）重新编写控制器代码，对于修改方向部分额外添加两个判定条件，使得小车巡线更加准确（修改部分已在代码中进行了标注）。

控制器代码如下所示（**代码注释中已包含所有的思路分析和算法实现**）：

#include <webots/Robot.hpp>

#include <webots/Motor.hpp>

#include <webots/Keyboard.hpp>

#include <webots/Camera.hpp>

#include <webots/GPS.hpp>

#include <iostream>

#include <algorithm>

#include <iostream>

#include <limits>

#include <string>

#include<string.h>

using namespace std;

using namespace webots;

int main() {

    Motor \*motors[4];//电机和键盘都要用webots给的类型

    webots::Keyboard keyboard;

    char wheels\_names[4][8]={"motor1","motor2","motor3","motor4"};//对应RotationMotor里的句柄

    Robot \*robot=new Robot();//使用webots的机器人主体

    Camera \*camera = robot->getCamera("camera");//获取相机，句柄名为camera

    camera->enable(1);//设置相机每1ms更新1次

    keyboard.enable(1);//运行键盘输入设置频率是1ms读取一次

    double speed[4];//此数组会在后面赋值给电机以速度

    double velocity=15;//初速度

    int i;

    //初始化

    for(i=0;i<=3;i++){

        motors[i]=robot->getMotor(wheels\_names[i]);//按照你在仿真器里面设置的名字获取句柄

        motors[i]->setPosition(std::numeric\_limits<double>::infinity());

        motors[i]->setVelocity(0.0);//设置电机一开始处于停止状态

        speed[i]=3;//给予小车一个初速度

    }

    double speed\_forward[4]={velocity,velocity,velocity,velocity};//前进方向

    double speed\_leftCircle[4]={velocity,-velocity,-velocity,velocity};//左自旋(即左转弯)

    double speed\_rightCircle[4]={-velocity,velocity,velocity,-velocity};//右自旋(即右转弯)

    int timeStep=(int)robot->getBasicTimeStep();//获取你在webots设置一帧的时间

    while(robot->step(timeStep)!=-1){//仿真运行一帧

        const unsigned char \*a=camera->getImage();//读取相机抓取的最后一张图像。图像被编码为三个字节的序列，分别代表像素的红、绿、蓝

        int length=camera->getWidth();//图像长度

        int width=camera->getHeight();//图像宽度

        int b1,b2,b3,b4;

        b1=length\*3\*width/2;//图像中间一行的最左边的像素点

        b2=length\*3\*width/2+(width/2+3)\*3;//图像中间一行的中间靠左的某像素点

        b3=length\*3\*width/2+(width/2+7)\*3;//图像中间一行的中间靠右的某像素点

        b4=length\*3\*width/2+(length-1)\*3;//图像中间一行的最右边的某像素点

        //其中b1，b2代表图像中间的黑色轨迹的两侧(大致估计)

        //以rgb的r为标准，当颜色为黑时，r的值必定小于60(10-30左右)

        if(a[b1]<60 && a[b2]>60 && a[b3]>60 && a[b4]>60){//若只有最左边像素点为黑色，则小车需要右转弯使得轨道黑线向中间靠拢

            for(i=0;i<=3;i++){

                speed[i]=speed\_rightCircle[i];//速度方向为右转

            }

        }

        else if(a[b1]>60 && a[b2]<60 && a[b3]<60 && a[b4]>60){//若中间两个像素判断点为黑色，则小车可选择继续直行

            for(i=0;i<=3;i++){

                speed[i]=speed\_forward[i];//速度方向为直行

            }

        }

        //修改部分1

        else if(a[b1]>60 && a[b2]>60 && a[b3]>60 && a[b4]<60){//若只有最右边像素点为黑色，则小车需要左转弯使得轨道黑线向中间靠拢

            for(i=0;i<=3; i++){

                speed[i]=speed\_leftCircle[i];//速度方向为左转

            }

        }

        //修改部分2

        else if(a[b1]<60 && (a[b2]<60||a[b3]<60) && a[b4]>60){

            //若左边像素判断点为黑色加上中间两个像素判断点中有一个为黑色，则小车需要右转

            for(i=0;i<=3;i++){

                speed[i]=speed\_rightCircle[i];//速度方向为右转

            }

        }

        else if(a[b1]>60 && (a[b2]<60||a[b3]<60) && a[b4]<60){

            //若右边像素判断点为黑色加上中间两个像素判断点中有一个为黑色，则小车需要左转

            for(i=0;i<=3;i++){

                speed[i]=speed\_leftCircle[i];//速度方向为左转

            }

        }

        else if(a[b1]>60 && a[b2]>60 && a[b3]>60 && a[b4]>60){//若四个判断像素点全为白色，则小车可选择继续直行

            for(i=0;i<=3;i++){

                speed[i]=speed\_forward[i];//速度方向为直行

            }

        }

        //将速度赋值给电机

        for(i=0;i<=3;i++){

            motors[i]->setVelocity(speed[i]);

        }

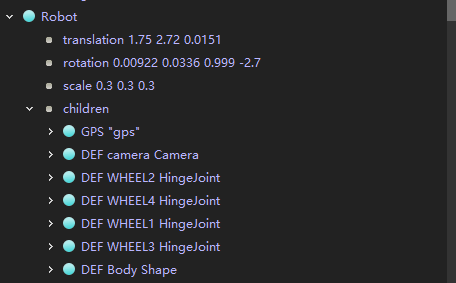
    }

    return 0;

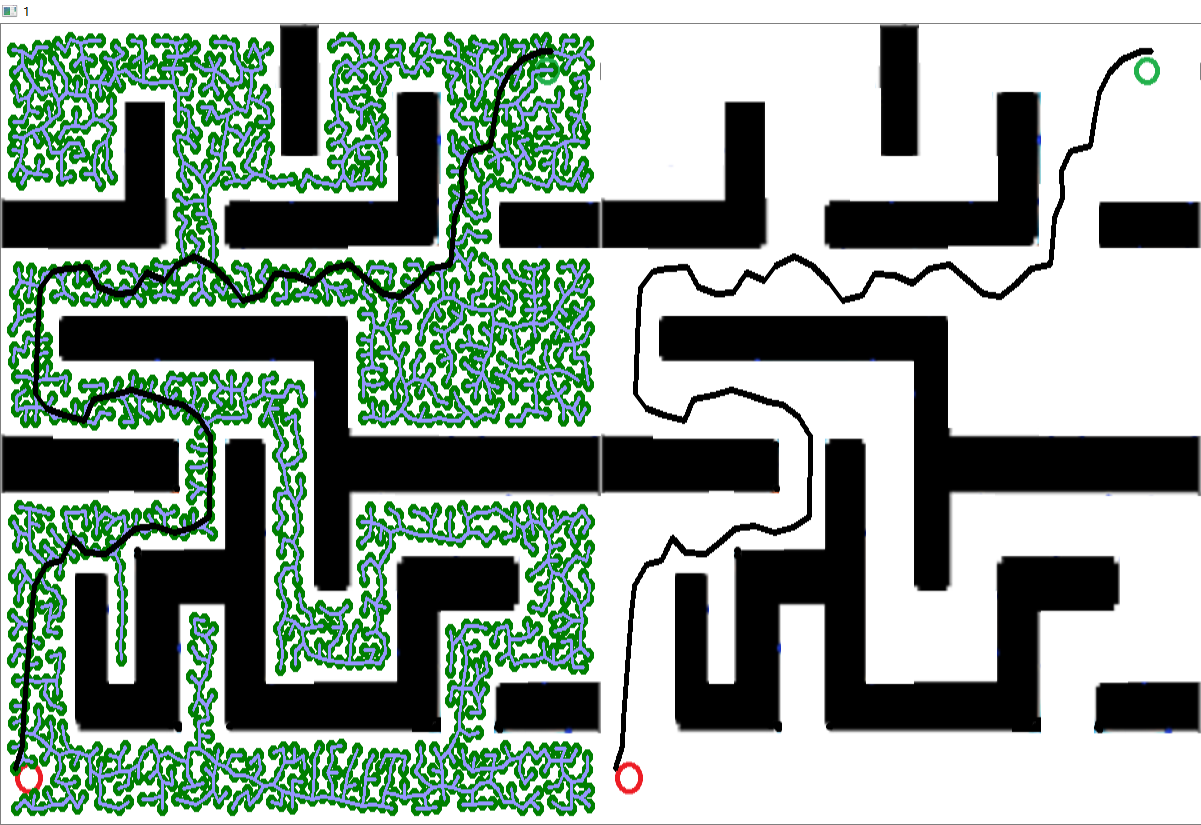
}

三、实验结果与分析

Robot的children节点结构如下图所示：



生长步长10，距离阈值10，随机采样次数200000的RRT树形图和最优路径图如下图所示（见下页）：



运行结果可见压缩包中的视频：“小车走迷宫”。

四、实验中的问题和解决方法

（1）由于RRT采样的随机性，导致生成的路线非常曲折，小车容易巡线失败，在修改了controller程序后，小车巡线的准确性得到了提升；

        //修改部分1

        else if(a[b1]>60 && a[b2]>60 && a[b3]>60 && a[b4]<60){//若只有最右边像素点为黑色，则小车需要左转弯使得轨道黑线向中间靠拢

            for(i=0;i<=3; i++){

                speed[i]=speed\_leftCircle[i];//速度方向为左转

            }

        }

        //修改部分2

        else if(a[b1]<60 && (a[b2]<60||a[b3]<60) && a[b4]>60){

            //若左边像素判断点为黑色加上中间两个像素判断点中有一个为黑色，则小车需要右转

            for(i=0;i<=3;i++){

                speed[i]=speed\_rightCircle[i];//速度方向为右转

            }

        }

（2）由于路线不够平滑，且本人的巡线程序不够精确，所以小车在行进过程中会有极少数事件略微偏移轨道，但仍能保证最终到达终点。