PRM 路径规划

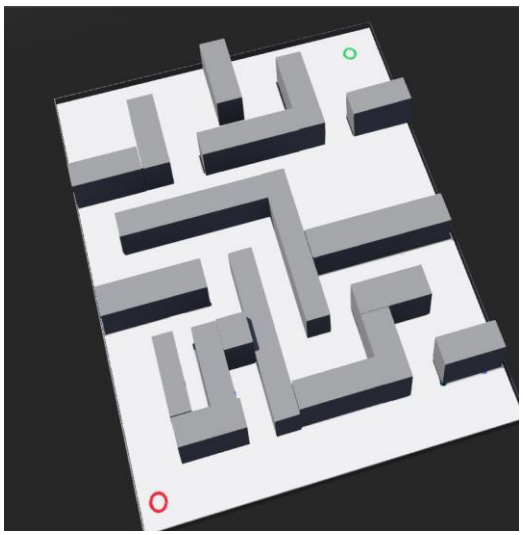
18329015 郝裕玮

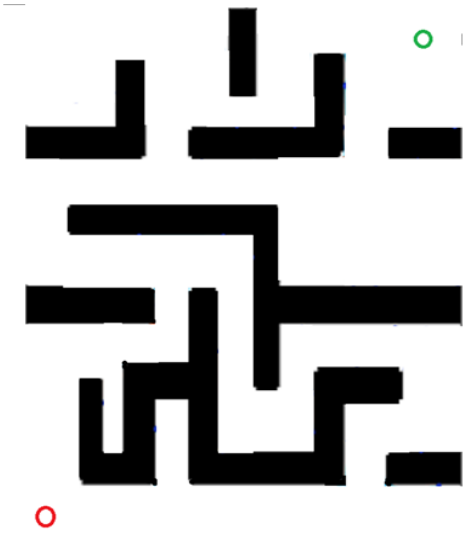
一、实验目标

使用 PRM 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：PIL，networkx 2.3

（2）使用 PRM 算法对 maze.png 进行处理，画出最优路径。

PRM 算法主要分为 3 个部分，学习阶段和搜索阶段。

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学习阶段可分为以下步骤：

1，图像预处理：

```
import math
from PIL import Image #用于对图像进行一系列处理
import numpy as np
import networkx as nx #对无向图进行相关处理
import copy

STAT_OBSTACLE='#' #障碍点用 # 表示
STAT_NORMAL='.' #普通点用 . 表示
#RoadMap 类：读入图片，将其二值化为有障碍物的二维网格化地图（即整张图只有 # 和 . ），并进行一系列操作
class RoadMap():

    #图像初始化处理
    def __init__(self,img_file):
        temp_map = [] #临时变量存储结果
        img = Image.open(img_file) #读取图片
        img_gray = img.convert('L') #将地图转换为灰度图像，每个像素用 8 个 bit 表示，0 表示黑，255 表示白，其他数字表示不同的灰度。
        img_arr = np.array(img_gray) #将该图像转换为数组形式存储
        img_binary = np.where(img_arr<127,0,255) #若像素值小于 127 则将其设置为黑色(0)，反之设置为白色(0)

        #遍历地图将各像素点修改为'.'或者'#'
        for x in range(img_binary.shape[0]): #shape[0]代表该数组的行数
            temp_row = [] #每轮内部循环前先将 temp_row 清空
            for y in range(img_binary.shape[1]): #shape[1]代表该数组的列数
                if img_binary[x,y] == 0:
                    status = STAT_OBSTACLE #若当前像素点为黑色则设置其为障碍点'#'
                else:
                    status = STAT_NORMAL #若为白色则设置其为普通点'.'
                temp_row.append(status) #将内部循环结果加入 temp_row 数组
            temp_map.append(temp_row) #内部循环结束后将当前 temp_row 的结果加入 temp_map
```

```
        #给成员变量赋值
        self.map = temp_map
        self.rows = img_binary.shape[0]
        self.cols = img_binary.shape[1]
```

2，随机撒点并对每两点间连线进行碰撞检测，最后构造无向图：

```
#PRM 类：使用 PRM 算法计算出最优路径、
#PRM 类继承 RoadMap 类，所以可以使用 RoadMap 类中的实例方法
class PRM(RoadMap):

    #初始化无向图并初始化相关数据
    def __init__(self, img_file, **param):
        # 随机路线图算法(Probabilistic Roadmap, PRM)
        # **param: 参数以字典形式传入
        # num_sample: 采样点个数
        # distance_neighbor: 邻域距离

        RoadMap.__init__(self,img_file) #先继承父类，再构造子类，否则就是重构了（这样的话子类就无法继承父类）

        self.num_sample = param['num_sample']
        self.distance_neighbor = param['distance_neighbor']
        self.G = nx.Graph() # 构造无向图，保存构型空间的完整连接属性


    #学习阶段，构造无碰撞的无向图
    def learn(self):
        # 随机“撒点”
        while len(self.G.nodes) < self.num_sample: #在地图上洒满 num_sample 个样点
            XY = (np.random.randint(0, self.rows),np.random.randint(0, self.cols)) # 在地图上随机取点
            #以下 if 的判定内容为：
            #（1）确保判定点仍在地图内（因为涉及到加减，可能使源位置超出地图范围）
            #（2）确保样点周围一定距离内没有障碍物点（这样可保证最终路线不会太靠近障碍物，避免小车行进时撞倒迷宫墙壁）
            #（3）这里设置（2）中的一定距离为 20
            if self.is_valid_xy(XY[0],XY[1]) \
            and self.is_valid_xy(XY[0]+20,XY[1]) \
            and self.is_valid_xy(XY[0],XY[1]+20) \
            and self.is_valid_xy(XY[0]+20,XY[1]+20) \
```

```
            and self.is_valid_xy(XY[0]-20,XY[1]) \
            and self.is_valid_xy(XY[0],XY[1]-20) \
            and self.is_valid_xy(XY[0]-20,XY[1]-20) \
            and self.not_obstacle(XY[0],XY[1]) \
            and self.not_obstacle(XY[0]+20,XY[1]) \
            and self.not_obstacle(XY[0],XY[1]+20) \
            and self.not_obstacle(XY[0]+20,XY[1]+20) \
            and self.not_obstacle(XY[0]-20,XY[1]) \
            and self.not_obstacle(XY[0],XY[1]-20) \
            and self.not_obstacle(XY[0]-20,XY[1]-20):
                self.G.add_node(XY) #满足条件则将该点加入无向图节点

        # 检测邻域范围内的连线是否与障碍物发生碰撞，若无碰撞，则将该连线加入到无向图中。
        # 遍历所有节点
        for node1 in self.G.nodes:
            for node2 in self.G.nodes:
                if node1 == node2:
                    continue
                dis = self.EuclidenDistance(node1,node2) #计算两点间欧氏距离（直线距离）

                #若两节点间距离在邻域范围内且连线间无障碍物点，则将该连线加入无向图
                if dis < self.distance_neighbor and self.check_path(node1,node2):
                    self.G.add_edge(node1,node2,weight=dis) #边的权重为欧氏距离（直线距离）
```

②搜索阶段可分为以下步骤：

1，寻找最短路径：

```
    #利用 learn 中得到的无碰撞无向图进行寻路
    def find_path(self):

        # 寻路时再将起点和终点添加进图中，以便一次学习多次使用
        temp_G = copy.deepcopy(self.G) #对无向图进行深拷贝
        start = (30,550) #迷宫起点坐标
        end = (750, 10)  #迷宫终点坐标
        #将起点和终点加入无向图节点
        temp_G.add_node(start)
        temp_G.add_node(end)
```

```
        # 将起点和终点与之前无向图中已存在的节点进行连线，并将满足同样条件的连线加入无向图
        for node1 in [start, end]:
            for node2 in temp_G.nodes:
                dis = self.EuclidenDistance(node1,node2)
                if dis < self.distance_neighbor and self.check_path(node1,node2):
                    #若两节点间距离在邻域范围内且连线间无障碍物点，则将该连线加入无向图
                    temp_G.add_edge(node1,node2,weight=dis) #、#边的权重为欧氏距离（直线距离）

        # 直接调用 networkx 中求最短路径的方法 shortest_path 来求出无向图中起点 start 和终点 end 间的最短路径
        path = nx.shortest_path(temp_G, source=start, target=end)

        return self.construct_path(path)
```

2，在图上画出最短路径并计算路径长度：

```
    #将最短路径上的各节点之间的连线所经过的像素点全部加入结果数组，便于绘图
    def construct_path(self, path):
        out = []
        print('PRM 最优路径长度为:',int(path_length(path))) #输出路径长度
        for i in range(len(path)-1):
            x1 = path[i][0]
            y1 = path[i][1]
            x2 = path[i+1][0]
            y2 = path[i+1][1]
            steps = max( abs(x1-x2), abs(y1-y2) ) #取横向、纵向较大值，确保经过的每个像素都被检测到
            #将以两点之间连线为对角线的矩阵的长|x1-x2|和宽|y1-y2|进行均分
            x_matrix = np.linspace(x1,x2,steps+1)
            y_matrix = np.linspace(y1,y2,steps+1)
            for i in range(0, steps+1): #将路径上等分的节点也加入结果数组
                #对于路径上的点的周围 2*2 范围内的点均设置为路径点，使得路径加粗，便于后续小车巡线
                for j in range (-2,2):
                    for k in range (-2,2):
                        out.append((math.ceil(x_matrix[i]+j), math.ceil(y_matrix[i]+k)))
```

```
        return out #返回最后的路径结果数组


#独立于两个类以外的函数
def path_length(path): #计算最短路径长度
    dis = 0
    for i in range(len(path)-1):
        x1 = path[i][0]
        y1 = path[i][1]
        x2 = path[i+1][0]
        y2 = path[i+1][1]

        dis+= pow( (x1-x2)**2+(y1-y2)**2 , 0.5 ) #依次累加连续两点间的距离
    return dis
```

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 类用于图像预处理并包含部分功能性函数（如判断像素点是否为障碍点，计算两点间欧氏距离和曼哈顿距离，碰撞检测等等），PRM 类继承 RoadMap 类的实例方法，用于计算最短路径。

全体代码如下（主函数在代码最下方）：

```
import math
from PIL import Image #用于对图像进行一系列处理
import numpy as np
import networkx as nx #对无向图进行相关处理
import copy

STAT_OBSTACLE='#' #障碍点用 # 表示
STAT_NORMAL='.' #普通点用 . 表示

#RoadMap 类：读入图片，将其二值化为有障碍物的二维网格化地图（即整张图只有 #
和 . ），并进行一系列操作
class RoadMap():

    #图像初始化处理
```

```
    def __init__(self,img_file):
        temp_map = [] #临时变量存储结果
        img = Image.open(img_file) #读取图片
        img_gray = img.convert('L') #将地图转换为灰度图像，每个像素用 8 个
bit 表示，0 表示黑，255 表示白，其他数字表示不同的灰度。
        img_arr = np.array(img_gray) #将该图像转换为数组形式存储
        img_binary = np.where(img_arr<127,0,255) #若像素值小于 127 则将其设
置为黑色(0)，反之设置为白色(0)

        #遍历地图将各像素点修改为'.'或者'#'
        for x in range(img_binary.shape[0]): #shape[0]代表该数组的行数
            temp_row = [] #每轮内部循环前先将 temp_row 清空
            for y in range(img_binary.shape[1]): #shape[1]代表该数组的列
数
                if img_binary[x,y] == 0:
                    status = STAT_OBSTACLE #若当前像素点为黑色则设置其为障
碍点'#'
                else:
                    status = STAT_NORMAL #若为白色则设置其为普通点'.'
                temp_row.append(status) #将内部循环结果加入 temp_row 数组
            temp_map.append(temp_row) #内部循环结束后将当前 temp_row 的结果
加入 temp_map

        #给成员变量赋值
        self.map = temp_map
        self.rows = img_binary.shape[0]
        self.cols = img_binary.shape[1]


    #判断当前点是否在地图范围内
    def is_valid_xy(self,x,y):
        if x < 0 or x >= self.rows or y < 0 or y >= self.cols: #防止随机
点越出地图范围
            return False
        return True


    #判断当前点是否为障碍点#
    def not_obstacle(self,x,y):
        if self.map[x][y] != STAT_OBSTACLE:
            return True
        return False
```

```
    #计算两点间欧氏距离（即直线距离）
    def EuclidenDistance(self, xy1, xy2):
        temp = (xy1[0]-xy2[0])**2 + (xy1[1]-xy2[1])**2
        dis = pow(temp,0.5)
        return dis


    #计算两点间曼哈顿距离
    def ManhattanDistance(self,xy1,xy2):
        dis = abs(xy1[0]-xy2[0]) + abs(xy1[1]-xy2[1])
        return dis


    #碰撞检测，检查两点之间连线是否经过障碍物
    def check_path(self, xy1, xy2):
        steps = max(abs(xy1[0]-xy2[0]), abs(xy1[1]-xy2[1])) # 取横向、纵向较大值，确保经过的每个像素都被检测到

        #将以两点之间连线为对角线的矩阵的长|xy1[0]-xy2[0]|和宽|xy1[1]-xy2[1]|进行均分
        x_matrix = np.linspace(xy1[0],xy2[0],steps+1)
        y_matrix = np.linspace(xy1[1],xy2[1],steps+1)

        # 第一个节点(x_matrix[0],y_matrix[0])和最后一个节点(x_matrix[steps+1],y_matrix[steps+1])分别是 xy1，xy2，不需要检查
        for i in range(1, steps):
            if not self.not_obstacle(math.ceil(x_matrix[i]), math.ceil(y_matrix[i])): #若连线中出现障碍物点，则立刻终止循环返回 False
                return False
        return True


    #画出路线图
    def plot(self,path):
        out = []

        #开始遍历整个地图
        for x in range(self.rows):
            temp = [] #每轮内部循环前先将 temp 清空
            for y in range(self.cols):
                if self.map[x][y]==STAT_OBSTACLE: #若为障碍物点则设置该点为黑色(0)
                    temp.append(0)
```

```
                elif self.map[x][y]==STAT_NORMAL: #若为普通点则设置该点为
白色(255)
                    temp.append(255)
            out.append(temp) #内部循环结束后将当前 temp 的结果加入 out

        for x,y in path:
            out[x][y] = 0 #将 path 数组中的所有位置点的像素值设置为 127（灰
色）

        #先将其转为数组再转为 image 格式
        out = np.array(out)
        img = Image.fromarray(np.uint8(out)) #不将数组转换为 uint8 格式会导
致图片保存时全黑，无法保存原图（尽管调用 show 时没有问题）
        img.show() #展示结果
        img.save("345.png") #保存结果




#PRM 类：使用 PRM 算法计算出最优路径、
#PRM 类继承 RoadMap 类，所以可以使用 RoadMap 类中的实例方法
class PRM(RoadMap):

    #初始化无向图并初始化相关数据
    def __init__(self, img_file, **param):
        # 随机路线图算法(Probabilistic Roadmap, PRM)
        # **param: 参数以字典形式传入
        # num_sample: 采样点个数
        # distance_neighbor: 邻域距离

        RoadMap.__init__(self,img_file) #先继承父类，再构造子类，否则就是重
构了（这样的话子类就无法继承父类）

        self.num_sample = param['num_sample']
        self.distance_neighbor = param['distance_neighbor']
        self.G = nx.Graph() # 构造无向图，保存构型空间的完整连接属性



    #学习阶段，构造无碰撞的无向图
    def learn(self):
        # 随机“撒点”
        while len(self.G.nodes) < self.num_sample: #在地图上洒满
num_sample 个样点
            XY = (np.random.randint(0, self.rows),np.random.randint(0,
self.cols)) # 在地图上随机取点
```

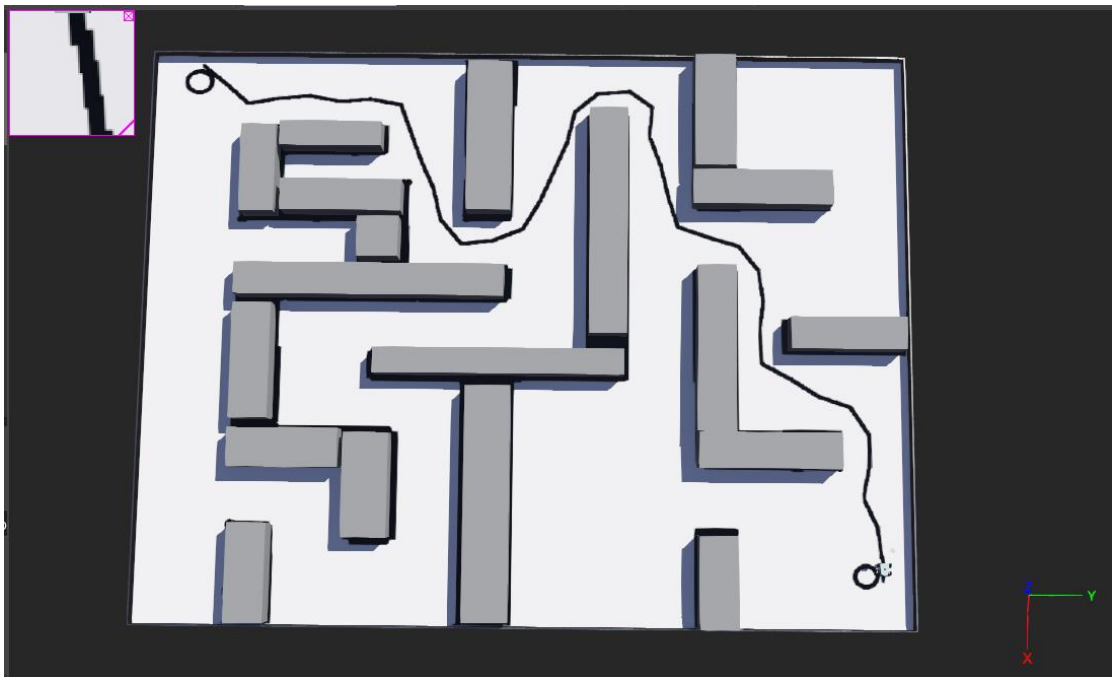
```
            #以下 if 的判定内容为：
            #（1）确保判定点仍在地图内（因为涉及到加减，可能使源位置超出地图范围）
            #（2）确保样点周围一定距离内没有障碍物点（这样可保证最终路线不会太靠近障碍物，避免小车行进时撞倒迷宫墙壁）
            #（3）这里设置（2）中的一定距离为 20
            if self.is_valid_xy(XY[0],XY[1]) \
            and self.is_valid_xy(XY[0]+20,XY[1]) \
            and self.is_valid_xy(XY[0],XY[1]+20) \
            and self.is_valid_xy(XY[0]+20,XY[1]+20) \
            and self.is_valid_xy(XY[0]-20,XY[1]) \
            and self.is_valid_xy(XY[0],XY[1]-20) \
            and self.is_valid_xy(XY[0]-20,XY[1]-20) \
            and self.not_obstacle(XY[0],XY[1]) \
            and self.not_obstacle(XY[0]+20,XY[1]) \
            and self.not_obstacle(XY[0],XY[1]+20) \
            and self.not_obstacle(XY[0]+20,XY[1]+20) \
            and self.not_obstacle(XY[0]-20,XY[1]) \
            and self.not_obstacle(XY[0],XY[1]-20) \
            and self.not_obstacle(XY[0]-20,XY[1]-20):
                self.G.add_node(XY) #满足条件则将该点加入无向图节点

        # 检测邻域范围内的连线是否与障碍物发生碰撞，若无碰撞，则将该连线加入到无向图中。
        # 遍历所有节点
        for node1 in self.G.nodes:
            for node2 in self.G.nodes:
                if node1 == node2:
                    continue
                dis = self.EuclidenDistance(node1,node2) #计算两点间欧氏距离（直线距离）

                #若两节点间距离在邻域范围内且连线间无障碍物点，则将该连线加入无向图
                if dis < self.distance_neighbor and self.check_path(node1,node2):
                    self.G.add_edge(node1,node2,weight=dis) #边的权重为欧氏距离（直线距离）


    #利用 learn 中得到的无碰撞无向图进行寻路
    def find_path(self):

        # 寻路时再将起点和终点添加进图中，以便一次学习多次使用
```

```
        temp_G = copy.deepcopy(self.G) #对无向图进行深拷贝
        start = (30,550) #迷宫起点坐标
        end = (750, 10)  #迷宫终点坐标
        #将起点和终点加入无向图节点
        temp_G.add_node(start)
        temp_G.add_node(end)

        # 将起点和终点与之前无向图中已存在的节点进行连线，并将满足同样条件的连线加入无向图
        for node1 in [start, end]:
            for node2 in temp_G.nodes:
                dis = self.EuclidenDistance(node1,node2)
                if dis < self.distance_neighbor and self.check_path(node1,node2):
                    #若两节点间距离在邻域范围内且连线间无障碍物点，则将该连线加入无向图
                    temp_G.add_edge(node1,node2,weight=dis) #、#边的权重为欧氏距离（直线距离）

        # 直接调用 networkx 中求最短路径的方法 shortest_path 来求出无向图中起点 start 和终点 end 间的最短路径
        path = nx.shortest_path(temp_G, source=start, target=end)

        return self.construct_path(path)


    #将最短路径上的各节点之间的连线所经过的像素点全部加入结果数组，便于绘图
    def construct_path(self, path):
        out = []
        print('PRM 最优路径长度为:',int(path_length(path))) #输出路径长度
        for i in range(len(path)-1):
            x1 = path[i][0]
            y1 = path[i][1]
            x2 = path[i+1][0]
            y2 = path[i+1][1]
            steps = max( abs(x1-x2), abs(y1-y2) ) #取横向、纵向较大值，确保经过的每个像素都被检测到
            #将以两点之间连线为对角线的矩阵的长|x1-x2|和宽|y1-y2|进行均分
            x_matrix = np.linspace(x1,x2,steps+1)
            y_matrix = np.linspace(y1,y2,steps+1)
            for i in range(0, steps+1): #将路径上等分的节点也加入结果数组
                for j in range (-2,2):
                    for k in range (-2,2):
```

```
                        out.append((math.ceil(x_matrix[i]+j),
math.ceil(y_matrix[i]+k)))

        return out #返回最后的路径结果数组


#独立于两个类以外的函数
def path_length(path): #计算最短路径长度
    dis = 0
    for i in range(len(path)-1):
        x1 = path[i][0]
        y1 = path[i][1]
        x2 = path[i+1][0]
        y2 = path[i+1][1]

        dis+= pow( (x1-x2)**2+(y1-y2)**2 , 0.5 ) #依次累加连续两点间的距离
    return dis


#======= 主函数 ==============

prm =
PRM('C:\\Users\\93508\\Desktop\\maze.png',num_sample=10000,distance_nei
ghbor=35) #读入文件且设置样本点数量和邻域大小

#这 2 个变量的值需要与主函数第 1 行内部参数保持一致，便于后续打印信息
num_sample=10000
distance_neighbor=35

print("样本点数量为:{}  邻域距离
为:{}".format(num_sample,distance_neighbor))
prm.learn() #构建无障碍连通图
path = prm.find_path() #使用 PRM 算法寻找最优路径
prm.plot(path) #画出路径
```

（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)。

最短路径图片详见压缩包中 maze_final.png。



（4）重新编写控制器代码，删除部分不需要的内容即可。

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
```

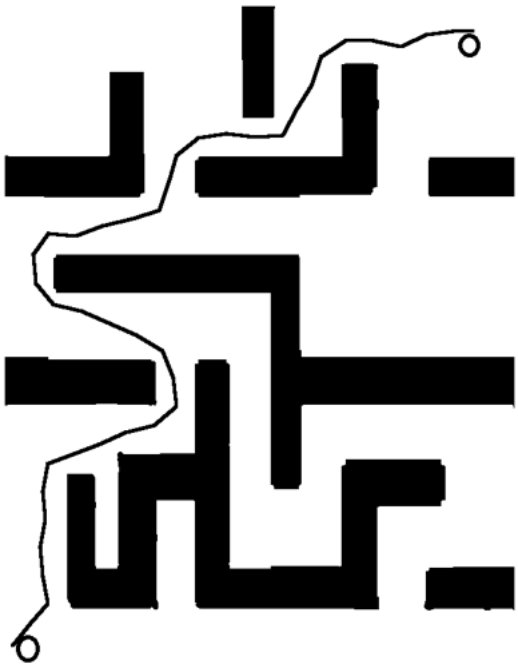
```
    Robot *robot=new Robot();//使用 webots 的机器人主体
    Camera *camera = robot->getCamera("camera");//获取相机，句柄名为
camera
    camera->enable(1);//设置相机每 1ms 更新 1 次

    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+5)*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 80(10-30 左右)
        if(a[b1]<80&&a[b2]>80&&a[b3]>80&&a[b4]>80){//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]>80&&a[b2]<80&&a[b3]<80&&a[b4]>80){//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];
            }
        }
        else if(a[b1]>80&&a[b2]>80&&a[b3]>80&&a[b4]<80){//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];
            }
        }
        else if(a[b1]>80&&a[b2]>80&&a[b3]>80&&a[b4]>80){//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];
            }
        }

        //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：

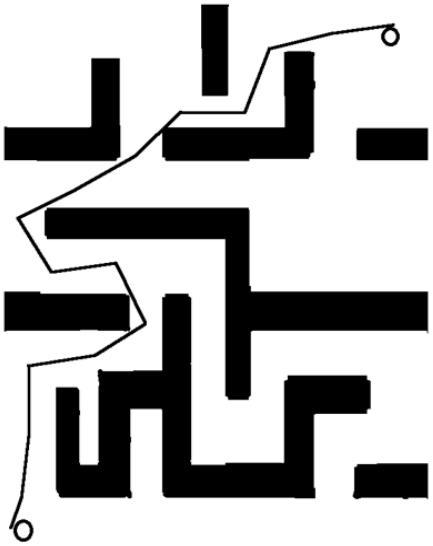
样本点 10000，邻域 35 的最优路径如下图所示：



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）一开始样本点设置过少且邻域过大，导致画出来的加粗路径过于贴近障碍物（如下图所示，该图样本点 1000，邻域 100）。增加样本点数量并缩小邻域后得到了满意的结果；



（2）程序运行时间较长，但为了路径的准确性，该问题无法略过。不同样本点和邻域的运行时间如下表所示：

样本点	邻域	运行时间（分:秒）
1000	100	0:10
5000	40	1:14
10000	35	6:35

（3）一开始不知道如何调整 Robot 的整体大小，后来发现只需调整其 scale 即可；

（4）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。