机器人导论作业

使用 RRT 算法进行路径规划

中山大学计算机学院　计算机科学与技术

19335174　施天予

一、实验目标

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，使用 RRT 算法，规划一条尽可能短的轨迹，控制机器人从绿色走到红色。

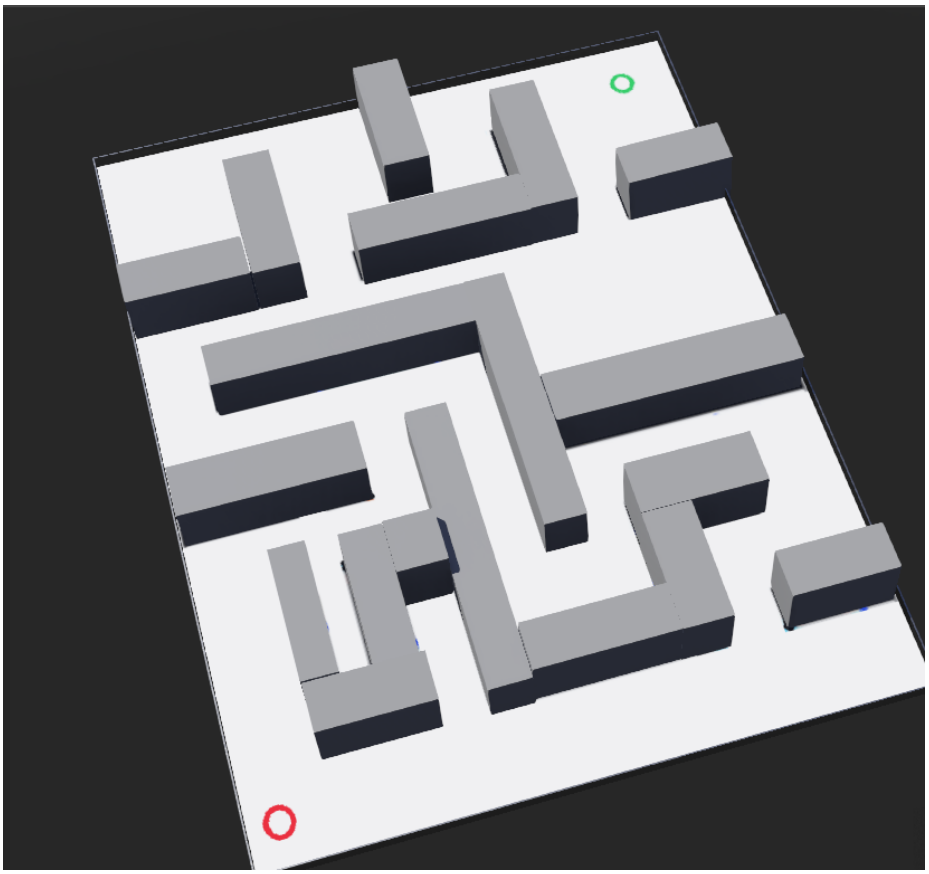

图 1: 迷宫地图

二、实验原理

基于快速扩展随机树（RRT / Rapidly exploring Random Tree）的路径规划算法，通过对状态空间中的采样点进行碰撞检测，避免了对空间的建模，能够有效地解决高维空间和复杂约束的路径规划问题。该方法的特点是能够快速有效地搜索高维空间，通过状态空间的随机采样点，把搜索导向空白区域，从而寻找到一条从起始点到目标点的规划路径，适合解决多自由度

机器人在复杂环境下和动态环境中的路径规划。与 PRM 类似，该方法是概率完备且不最优的。

所以 RRT 是一种多维空间中有效率的规划方法。它以一个初始点作为根节点，通过随机采样增加叶子节点的方式，生成一个随机扩展树，当随机树中的叶子节点包含了目标点或进入了目标区域，便可以在随机树中找到一条由从初始点到目标点的路径。

算法伪代码如下：

$V_1 \leftarrow \{q_{init}\}; E_1 \leftarrow \emptyset; G_1 \leftarrow (V_1, E_1);$
$V_2 \leftarrow \{q_{goal}\}; E_2 \leftarrow \emptyset; G_2 \leftarrow (V_2, E_2); i \leftarrow 0;$
while $i < N$ **do**
    $q_{rand} \leftarrow \text{Sample}(i); i \leftarrow i + 1;$
    $q_{nearst} \leftarrow \text{Nearst}(G_1, q_{rand});$
    $q_{new} \leftarrow \text{Steer}(q_{nearst}, q_{rand});$
    if $\text{ObstacleFree}(q_{nearst}, q_{new})$ **then**
        $V_1 \leftarrow V_1 \cup \{q_{new}\};$
        $E_1 \leftarrow E_1 \cup \{(q_{nearest}, q_{new})\};$
        $q'_{nearst} \leftarrow \text{Nearst}(G_2, q_{new});$
        $q'_{new} \leftarrow \text{Steer}(q'_{nearst}, q_{new});$
        if $\text{ObstacleFree}(q'_{nearst}, q'_{new})$ **then**
            $V_2 \leftarrow V_2 \cup \{q'_{new}\};$
            $E_2 \leftarrow E_2 \cup \{(q'_{nearst}, q'_{new})\};$
            do
                $q''_{new} \leftarrow \text{Steer}(q'_{new}, q_{new});$
                if $\text{ObstacleFree}(q''_{new}, q'_{new})$ **then**
                $V_2 \leftarrow V_2 \cup \{q''_{new}\};$
                $E_2 \leftarrow E_2 \cup \{(q''_{new}, q'_{new})\};$
                $q'_{new} \leftarrow q''_{new};$
                else break;
            while *not* $q'_{new} = q_{new}$
        if $q'_{new} = q_{new}$ **then return** $(V_1, E_1);$
    if $|V_2| < |V_1|$ **then** $\text{Swap}(V_1, V_2);$

RRT 算法的运行大致可以分为四个部分：初始化、随机采样、扩展和终止条件。初始化部分需要我们将事先构建好的地图读取进来，对图进行灰度处理和二值化处理，目的是使地图矩阵只包含两种元素，一种是障碍物，另一种则是非障碍物。随机采样部分需要我们在每次采样时，有一定的概率向着目标点延伸，也有一定的概率在地图上随机选择一个方向延伸一段距离。扩展部分顾名思义就是对树进行扩展，最重要的就是做碰撞检测，找出当前树距离目标最近的顶点后，根据步长走一段距离就要判定路线内是否有障碍物阻挡。终止条件就是当我们对树扩展后，若当前点到终点的距离已经足够小，就可以终止算法的运行并输出。

三、核心代码

在上次 PRM 算法的作业中，我没有使用 A* 搜索且不知道如何将绘制的线加粗，只好用笔描，十分愚蠢。后来得知其他同学可以用 OpenCV 设置线的粗细，于是这次作业我也使用了 OpenCV 进行绘图，并且自己实现了 A* 搜索。

因为 RRT 算法的整体代码过长，这里就展示几个关键函数，并加以文字辅助说明。部分说明与上次作业相同：

- 迷宫 maze.png 是 600*800（长 * 宽）像素的图片
- 起点位置约为（50，528），终点位置约为（730，30）
- 距离采用欧氏距离度量

首先构建一个 Map 的类，读取图片并将图片二值化。

```
class Map(object):
    def __init__(self, path, init=None, goal=None):
        super().__init__()
        # 读取图片
        img = cv2.imread(path)
        # 转换成灰度图
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # 灰度图二值化（0或255）
        ret, thresh = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY)
        self.img = img
        self.mat = thresh
        self.init = init
        self.goal = goal
```

检查两个点之间是否碰撞的函数

```
    def check_collision(self, pt1, pt2):
        # STRIDE = 100 是控制检查直线是否穿过障碍物x,y的步长
        if self.mat[pt1[0]][pt1[1]] == 0 or self.mat[pt2[0]][pt2[1]] == 0:
            return False
        stepx = (pt2[0] - pt1[0]) / STRIDE
        stepy = (pt2[1] - pt1[1]) / STRIDE
        nowx, nowy = pt1[0], pt1[1]
        for i in range(STRIDE):
            nowx += stepx
            nowy += stepy
            if self.mat[round(nowx)][round(nowy)] == 0:
                return False
        return True
```

检查给定点是否到达终点的函数

```
    def is_goal(self, pt):
        # EPS = 3 是控制检查是否为终点时容忍的像素差
        diffx, diffy = pt[0] - self.goal[0], pt[1] - self.goal[1]
        diff = (diffx ** 2 + diffy ** 2) ** 0.5
        if diff < EPS: return True
        return False
```

接下来是 RRT 算法。在随机生成采样点时，加入一定的优化使得生成过程更加具有贪婪

性而非完全随机，即以一定的概率随机生成采样点，而其余情况直接将目标点设置为终点，即贪婪地选择向终点扩展。

```
def RRT(mmap: Map, pixel_step, sample_prob, upper_limit):
    PIXEL_STEP = pixel_step  # 步长
    SAMPLE_PROB = sample_prob # 随机采取点的概率
    UPPER_LIMIT = upper_limit # 限制的采样点个数
    # 初始化空RRT树
    init_node = mmap.init
    vertices = [init_node]
    edges = {init_node: []}
    mmap.draw_dot(init_node)
    sample_count = 0
    while True:
        # 以一定概率随机生成采样点
        if np.random.random() < SAMPLE_PROB:
            samplex = np.random.randint(0, mmap.mat.shape[0])
            sampley = np.random.randint(0, mmap.mat.shape[1])
            sample = (samplex, sampley)
        else:
            sample = mmap.goal
        # 超过步长，计算角度根据步长生成新点
        dist, pt_nearest = get_nearest(vertices, sample)
        if dist > PIXEL_STEP:
            x = round(pt_nearest[0] + (sample[0] - pt_nearest[0]) / dist *
                ↪ PIXEL_STEP)
            y = round(pt_nearest[1] + (sample[1] - pt_nearest[1]) / dist *
                ↪ PIXEL_STEP)
            sample = (x, y)
        sample_count += 1
        # 如果路径合法，生成新的边
        if mmap.check_collision(pt_nearest, sample):
            vertices.append(sample)
            if pt_nearest not in edges: edges[pt_nearest] = [sample]
            else: edges[pt_nearest].append(sample)
            if sample not in edges: edges[sample] = [pt_nearest]
            else: edges[sample].append(pt_nearest)
            mmap.draw_dot(sample)
            mmap.draw_line(pt_nearest, sample)
        # 到达终点，算法停止返回结果
        if mmap.is_goal(sample):
            return vertices, edges, mmap
        # 超过限制的采样点个数，算法直接停止
        if sample_count >= UPPER_LIMIT:
            return None, None, mmap
```

最后使用 A* 搜索算法，通过启发式函数（当前节点到终点的欧氏距离），从起点到终点寻找一条最短路径。

```python
# A*算法搜索节点
class Node(object):
    def __init__(self, hist, goal):
        super().__init__()
        self.hist = hist.copy()
        self.goal = goal
        self.pt = self.hist[-1]
        self.gx = len(self.hist) - 1
        self.hx = cal_dist(self.hist[-1], self.goal)
        self.fx = self.gx + self.hx
    def __lt__(self, other):
        return self.fx < other.fx
# A*搜索算法
def AStar(vertices: list, edges: dict, mmap: Map):
    init_vert, goal_vert = vertices[0], vertices[-1]
    init_node = Node([init_vert], goal_vert)
    pq = queue.PriorityQueue()
    pq.put(init_node)
    while not pq.empty():
        node = pq.get()
        if mmap.is_goal(node.pt):
            for i in range(len(node.hist)-1):
                mmap.draw_dot(node.hist[i], color=(0,0,0), thickness=2)
                mmap.draw_line(node.hist[i], node.hist[i+1], color=(0,0,0),
                    ↪ thickness=8)
            mmap.draw_dot(node.pt, color=(0,0,0))
            return mmap
        for v in edges[node.pt]:
            if v in node.hist: continue
            new_hist = node.hist.copy()
            new_hist.append(v)
            new_node = Node(new_hist, node.goal)
            pq.put(new_node)
    return mmap
```

四、实验过程

1. 插入路径图：将 RRT 算法得到路径图插入 webots 中
   （为便于巡线，随机采样点没有画出）
2. 添加机器人：将之前的巡线小车复制到这次的 webots 中，调整起始位置
3. 添加代码：巡线的 C++ 代码与之前的作业完全相同，就不再展示

最后实现的机器人成功从起点到达了终点，完整视频在 video.mkv 中。

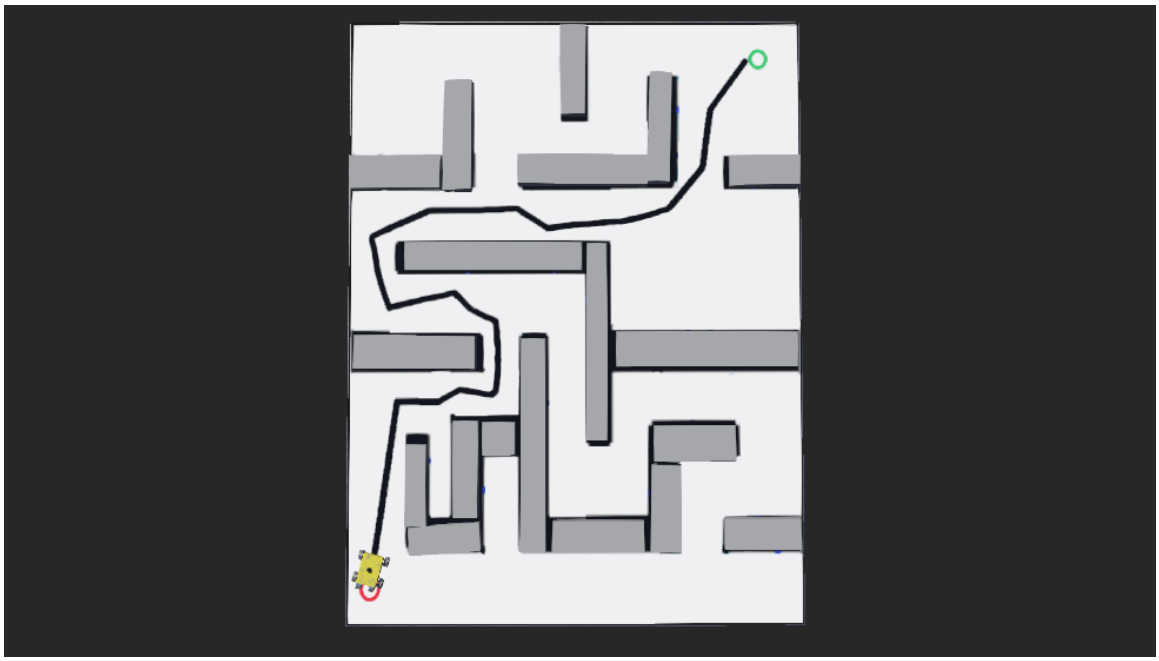

图 2: webots 中小车到达终点

五、结果分析

1. RRT 算法的步长

下面三张图的步长分别设置为 10、80、300，随机概率和采样上限均为 0.5 和 50000。

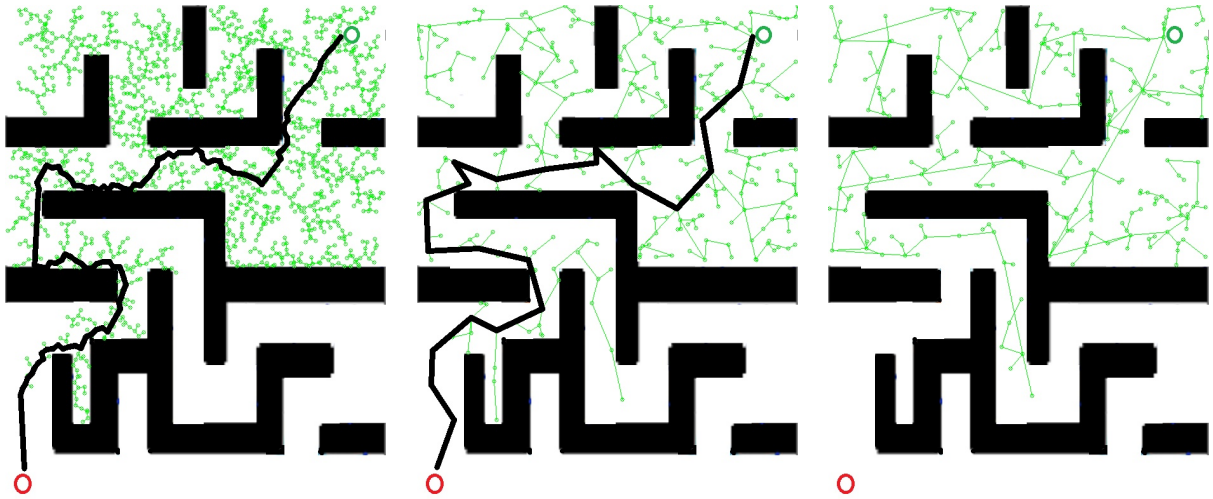

图 3: 步长为 10、80、300 的路径图

当步长过小时，会使得每次扩展的距离过小，从而大大地增加迭代扩展的次数，使得算法性能下降；而当步长过大时，显然此时每一步扩展的距离会很大，发生碰撞的概率也会很大，在

一些可行路径较窄小时甚至可能会找不到可行解。(算法本身具有随机性，每次运行的结果都大不相同，只能选出比较具有代表性的展示，随机性的存在其实让很多种情况都有机会求出解)

2. RRT 算法的随机概率

下面三张图的随机概率分别设置为 0.2、0.5、0.8，步长和采样上限均为 80 和 50000。

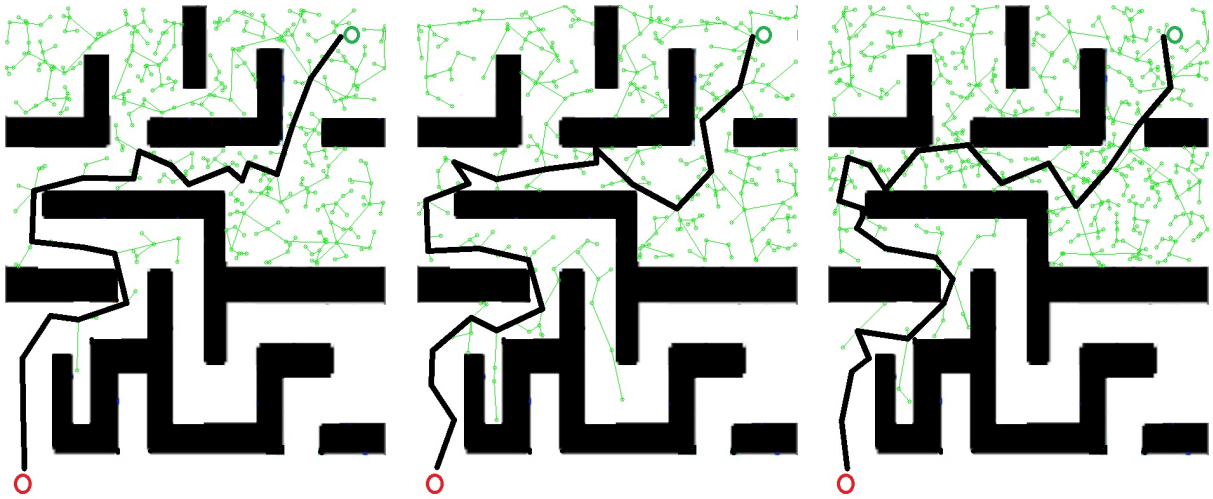

图 4: 随机概率为 0.2、0.5、0.8 的路径图

随机概率意味着每次采样在地图中随机采取点的概率。同样地可以看到，随着随机概率的上升，采样点在地图中的覆盖率也是逐步上升，但从理论上来说并不是越高越好，因为这建立在采样上限足够高的情况下才能成立，若采样上限降低一些，随着随机概率的上升很有可能出现无法求解的情况。

六、实验总结

经过上次 PRM 算法的作业后，本次 RRT 算法的实现我就熟练很多了，并且也弥补了上次作业画线太细和没有使用 A* 搜索的缺陷。

实验的核心内容为实现 RRT 算法，规划机器人的避障路径。通过整个实验的流程，可以深刻地认识到 RRT 算法的核心本质：RRT 是一种多维空间中有效率的随机规划方法，从初始状态点生长的快速扩展随机树来搜索整个状态空间从而寻找可行路径。另外，在编写和实现这些算法的时候，总会遇到大大小小的 bug 需要自己去发现和解决，比如说起初我使用 opencv 在地图上画线和画点时发现，搜索结果和打印结果完全对不上号，后来发现时横纵坐标 x 和 y 出现错位，所以在这个过程中我也总结了相关的一些经验。

总而言之，这次实验收获还是比较丰富的。让我对 RRT 算法有了更深入的理解，不仅仅是停留在理论上，而是将算法应用到具体问题中去，同时也给我带来了极大的趣味感和成就感。