机器人导论Motion Planning作业

学号：21307046 姓名：梁根铭

一、作业要求

RRT的衍生算法的仿真实现

- RRT*：引入成本函数的概念，通过重新连接已有节点，以寻求更优的路径
- Informed RRT：引入启发式信息指导树的生长方向，以提高搜索效率
- Dynamic RRT：根据环境特征动态调整节点扩展的步长，提升算法在复杂环境中的性能

二、代码思路与实现

2.0 RRT

想要实现RRT（快速随机树）的衍生算法，首先需要将RRT算法的框架搭建起来，在RRT算法的基础上进行改进，以实现RRT*、Informed RRT和Dynamic RRT。RRT是一

种路径规划算法，其目的是找到一条可行路径。以下是RRT算法的实现步骤和对应代码：

1. **初始化**：定义节点类（包含父节点信息），初始化节点列表（树）、起始节点和目标节点，以及障碍物列表、步长等基本信息。

```
# 定义节点类
class Node:
    def __init__(self, x, y):
        self.x = x  # x坐标
        self.y = y  # y坐标
        self.parent = None  # 父节点
        self.cost = 0.0  # 当前路径开销
# 定义 RRT 算法类
class myRRT:
    def __init__(self, start, goal, map_dims, obstacle_list, expand_dis=1.0,
min_expand_dis=1.0,max_expand_dis=3.0,goal_sample_rate=0.1, max_iter=500):
        self.start = Node(start[0], start[1])  # 起始节点
        self.goal = Node(goal[0], goal[1])  # 目标节点
        self.map_width, self.map_height = map_dims  # 地图宽高
        self.obstacle_list = obstacle_list  # 障碍物列表
        self.expand_dis = expand_dis  # 步长（rrt_star, informed_rrt使用）
        self.min_expand_dis = min_expand_dis  # 最小步长（dynamic_rrt使用）
        self.max_expand_dis = max_expand_dis  # 最大步长（dynamic_rrt使用）
        self.goal_sample_rate = goal_sample_rate  # 采样目标点的概率
        self.max_iter = max_iter  # 最大迭代次数
        self.node_list = [self.start]  # 初始化节点列表
```

2. **随机采样**：在工作空间内随机生成一个点，根据采样目标点频率，使得随机时有小概率直接将目标节点作为采样目标，更能保证寻路时方向的正确性。

```
# 随机生成节点
 def get_random_node(self):
     # 是否在当前迭代中直接采样目标点，增加搜索空间的多样性，同时提高路径找到目标点的
概率
     if random.random() > self.goal_sample_rate:
         rand_node = Node(random.uniform(0, self.map_width), random.uniform(0,
self.map_height))
     else:
         rand_node = Node(self.goal.x, self.goal.y)
     return rand_node
```

3. **寻找最近节点**：在树中找到离该随机点最近的节点。

```
# 找到距离随机节点最近的节点索引
 def get_nearest_node_index(self, node_list, rand_node):
     dlist = [(node.x - rand_node.x) ** 2 + (node.y - rand_node.y) ** 2 for
```

```
node in node_list]
        min_idx = dlist.index(min(dlist))
        return min_idx
```

4. **生成新节点**：从最近节点向随机点方向延伸一个固定的步长，生成一个新节点。

```
# 扩展节点
 def steer(self, from_node, to_node, extend_length=float("inf")):
        # 初始化新节点
        new_node = Node(from_node.x, from_node.y)
        # 计算出from-to的方向和距离
        d, theta = self.calc_distance_and_angle(new_node, to_node)
        # 新节点需要加上初始节点的xy方向偏移量
        new_node.x += min(extend_length, d) * math.cos(theta)
        new_node.y += min(extend_length, d) * math.sin(theta)
        # 新节点的当前路径开销
        new_node.cost = from_node.cost + d
        # 新节点的父节点
        new_node.parent = from_node
        return new_node,theta,d
```

5. **碰撞检测**：检查新节点是否在障碍物内或是否在障碍物范围内延伸。如果没有碰撞，则将新节点添加到树中。

```
# 检查节点是否与障碍物碰撞
 def check_collision(self, node):
        if node is None:
            return False
        for ox, oy, size in self.obstacle_list:
            dx = ox - node.x
            dy = oy - node.y
            d = math.sqrt(dx ** 2 + dy ** 2)
            if d <= size:
                return False
        if node.x < 0 or node.x > self.map_width or node.y < 0 or node.y >
self.map_height:
            return False
        return True
 # 检查扩展的路径是否与障碍物碰撞
 def check_collision_extend(self, near_node, theta, d):
        tmp_node = Node(near_node.x, near_node.y)
        # self.expand_dis / 10可以视情况调整精度
        for i in range(int(10*d//self.expand_dis)):
            tmp_node.x += self.expand_dis / 10 * math.cos(theta)
            tmp_node.y += self.expand_dis / 10 * math.sin(theta)
            if not self.check_collision(tmp_node):
                return False
        return True
```

6. **检查目标**：如果新节点离目标点足够近，则认为路径找到，终止算法并将最终路径生成返回；否则，返回步骤2继续迭代。

```
# 计算到目标的距离
 def calc_dist_to_goal(self, x, y):
     dx = x - self.goal.x
     dy = y - self.goal.y
     return math.sqrt(dx ** 2 + dy ** 2)
# 生成最终路径
 def generate_final_course(self, goal_ind):
     path = [[self.goal.x, self.goal.y]]
     node = self.node_list[goal_ind]
     while node.parent is not None:
         path.append([node.x, node.y])
         node = node.parent
     path.append([self.start.x, self.start.y])
     return path
```

2.1 RRT*

2.1.1 RRT*原理

RRT* 旨在找到最优路径，而不仅仅是找到一条可行路径。其优化主要体现在找到新节点后，算法会尝试通过重新连接新节点和其邻近节点来找到更短路径，从而不断优化路径成本。RRT*代码中执行以下几个优化步骤：

1. **找到树中新节点附近的节点**：在寻找附近节点时采用动态范围，范围随着节点数量增加逐渐缩小以确保在保证计算效率的同时找到最优路径。

```
# 找到新节点附近的节点
 def find_near_nodes(self, new_node):
     nnode = len(self.node_list)
     # 动态范围r
     r = 50.0 * math.sqrt((math.log(nnode) / nnode))
     dlist = [(node.x - new_node.x) ** 2 + (node.y - new_node.y) ** 2 for node
in self.node_list]
     near_inds = [dlist.index(i) for i in dlist if i <= r ** 2]
     return near_inds
```

2. **选择父节点**：在找到邻近节点后，RRT* 会从中选择一个能使路径成本最小的节点作为其父节点，并更新新节点的信息。

```
# 选择新节点的父节点
 def choose_parent(self, new_node, near_inds):
     if not near_inds:
```

```
            return new_node
        dlist = []
        for i in near_inds:
            dx = new_node.x - self.node_list[i].x
            dy = new_node.y - self.node_list[i].y
            d = math.sqrt(dx ** 2 + dy ** 2)
            theta = math.atan2(dy, dx)
            if self.check_collision_extend(self.node_list[i], theta, d):
                dlist.append(self.node_list[i].cost + d)
            else:
                dlist.append(float("inf"))
        min_cost = min(dlist)
        min_ind = near_inds[dlist.index(min_cost)]
        if min_cost == float("inf"):
            return new_node
        new_node.cost = min_cost
        new_node.parent = self.node_list[min_ind]
        return new_node
```

3. **重新连线**：在新节点加入树后，RRT* 会尝试通过新节点重新连接其周围的节点，如果重新连接可以减少路径成本，则进行重新连线。

```
# 重新连接节点
 def rewire(self, new_node, near_inds):
        nnode = len(self.node_list)
        for i in near_inds:
            near_node = self.node_list[i]
            dx = new_node.x - near_node.x
            dy = new_node.y - near_node.y
            d = math.sqrt(dx ** 2 + dy ** 2)
            scost = new_node.cost + d
            if near_node.cost > scost:
                theta = math.atan2(dy, dx)
                if self.check_collision_extend(new_node, theta, d):
                    near_node.parent = new_node
                    near_node.cost = scost
```

2.1.2 RRT*路径规划实现

最终根据RRT*原理修改RRT算法，得到RRT*路径规划代码如下：

```
# rrt_star规划路径
def rrt_star_planning(self):
    for i in range(self.max_iter):
        rand_node = self.get_random_node()  # 随机采样节点
        nearest_idx = self.get_nearest_node_index(self.node_list, rand_node)  # 找到最近的节点索引
        nearest_node = self.node_list[nearest_idx]
        new_node,theta,d = self.steer(nearest_node, rand_node, self.expand_dis)  # 扩展新节点
```

```
        if self.check_collision_extend(nearest_node, theta, d):  # 检查路径是否与障碍物碰撞
            near_inds = self.find_near_nodes(new_node)  # 找到附近的节点
            new_node = self.choose_parent(new_node, near_inds)  # 选择新节点的父节点
            self.node_list.append(new_node)  # 将新节点添加到节点列表
            self.rewire(new_node, near_inds)  # 重新连接节点
        if self.calc_dist_to_goal(self.node_list[-1].x, self.node_list[-1].y) <= self.expand_dis:
            final_node,theta,d = self.steer(self.node_list[-1], self.goal, self.expand_dis)
            if self.check_collision(final_node):
                return self.generate_final_course(len(self.node_list) - 1)
    return None
```

2.2 Informed RRT

2.2.1 Informed RRT原理

Informed RRT 的优化旨在减少搜索空间，从而提高搜索效率。Informed RRT代码中主要将随机采样修改成启发式采样，如下：

1. **采样空间收缩以实现启发式采样**：在找到一条初始路径后，Informed RRT 会基于当前最优路径长度定义一个椭圆形的采样空间，只在该椭圆内进行随机采样，从而缩小搜索范围，提高搜索效率和收敛速度。

```
# 启发式生成节点
 def get_informed_random_node(self, c_max):
     if c_max == float("inf"):
         return self.get_random_node()
     c_min = np.linalg.norm([self.start.x - self.goal.x, self.start.y - self.goal.y])
     if c_min == 0:
         return self.get_random_node()
     while True:
         x = random.uniform(-1, 1)
         y = random.uniform(-1, 1)
         if x ** 2 + y ** 2 <= 1:
             break
     x *= (c_max / 2)
     y *= (math.sqrt(c_max ** 2 - c_min ** 2) / 2)
     theta = math.atan2(self.goal.y - self.start.y, self.goal.x - self.start.x)
     rot_x = x * math.cos(theta) - y * math.sin(theta) + (self.start.x + self.goal.x) / 2
     rot_y = x * math.sin(theta) + y * math.cos(theta) + (self.start.y + self.goal.y) / 2
     return Node(rot_x, rot_y)
```

2.2.2 Informed RRT路径规划实现

最终根据Informed RRT原理修改RRT算法，得到Informed RRT路径规划代码如下：

```
# informed_rrt规划路径
def informed_rrt_planning(self):
    best_cost = float("inf")
    best_path = None
    for i in range(self.max_iter):
        rand_node = self.get_informed_random_node(best_cost)    # 启发式采样
        nearest_idx = self.get_nearest_node_index(self.node_list, rand_node)  # 找
到最近的节点索引
        nearest_node = self.node_list[nearest_idx]
        new_node,theta,d = self.steer(nearest_node, rand_node, self.expand_dis)  #
扩展新节点
        if self.check_collision_extend(nearest_node, theta, d):  # 检查路径是否与障碍
物碰撞
            self.node_list.append(new_node)  # 将新节点添加到节点列表
        if self.calc_dist_to_goal(self.node_list[-1].x, self.node_list[-1].y) <=
self.expand_dis:
            final_node,theta,d = self.steer(self.node_list[-1], self.goal,
self.expand_dis)
            if self.check_collision(final_node):
                new_path = self.generate_final_course(len(self.node_list) - 1)
                new_cost = self.calc_cost(new_path)
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_path = new_path
    return best_path
```

2.3 Dynamic RRT

2.2.1 Dynamic RRT原理

Dynamic RRT 通过动态调整节点扩展的步长来提高算法的效率和适应性。Dynamic RRT代码中主要将固定步长修改成在最小步长和最大步长之间动态调整步长，如下：

1. **动态步长调整**：根据最近节点与障碍物的距离动态调整步长，在障碍物密集区域使用较小步长，在空旷区域使用较大步长。

```
# 计算当前动态步长
 def dynamic_expand_dis(self, node):
     min_dist_to_obstacle = float("inf")
     for ox, oy, size in self.obstacle_list:
         dx = ox - node.x
         dy = oy - node.y
         dist = math.sqrt(dx ** 2 + dy ** 2) - size
         if dist < min_dist_to_obstacle:
             min_dist_to_obstacle = dist
     # 确保步长在最小步长和最大步长之间
     dynamic_dis = max(self.min_expand_dis, min(self.max_expand_dis,
```

```
    min_dist_to_obstacle / 2))
        return dynamic_dis
```

2.2.2 Dynamic RRT路径规划实现

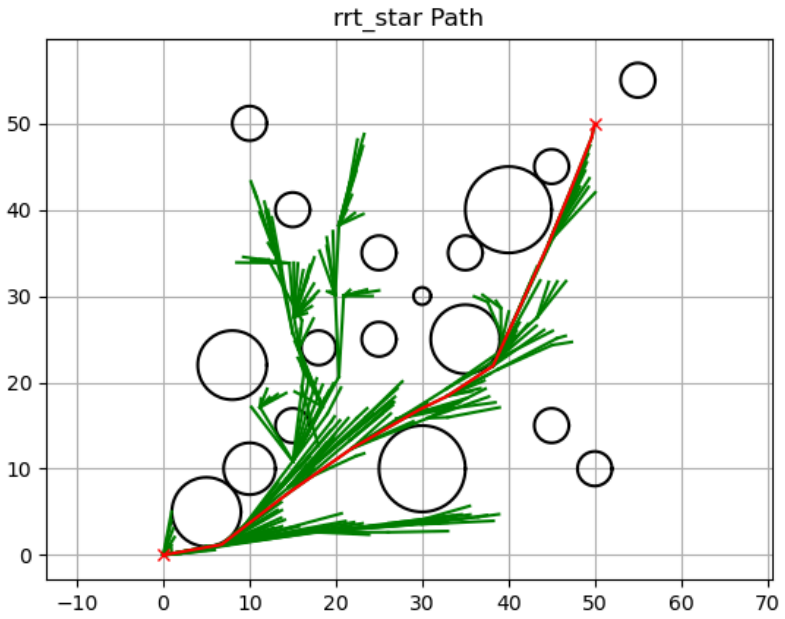
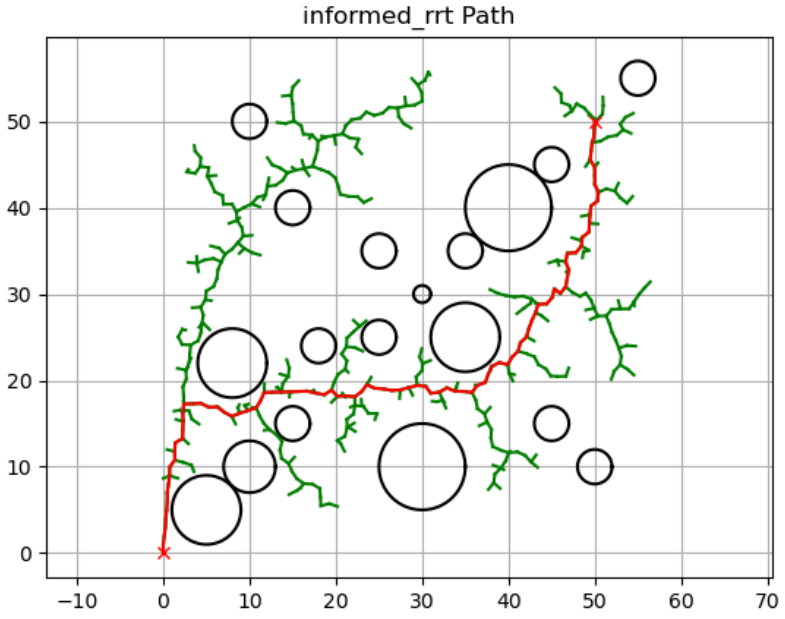
最终根据Dynamic RRT原理修改RRT算法，得到Dynamic RRT路径规划代码如下：

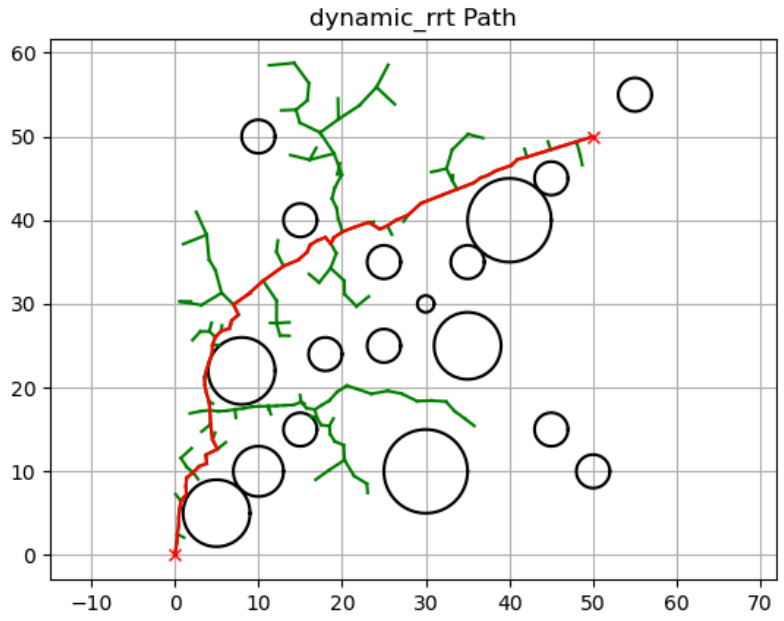
```
# dynamic_rrt规划路径
def dynamic_rrt_planning(self):
    for i in range(self.max_iter):
        rand_node = self.get_random_node()  # 随机采样节点
        nearest_idx = self.get_nearest_node_index(self.node_list, rand_node)  # 找
到最近的节点索引
        nearest_node = self.node_list[nearest_idx]
        dynamic_expand_dis = self.dynamic_expand_dis(nearest_node)  # 动态步长
        new_node,theta,d = self.steer(nearest_node, rand_node, dynamic_expand_dis)
# 扩展新节点
        if self.check_collision_extend(nearest_node, theta, d):  # 检查路径是否与障碍
物碰撞
            self.node_list.append(new_node)  # 将新节点添加到节点列表
        if self.calc_dist_to_goal(self.node_list[-1].x, self.node_list[-1].y) <=
self.min_expand_dis:
            final_node,theta,d = self.steer(self.node_list[-1], self.goal,
self.min_expand_dis)
            if self.check_collision(final_node):
                return self.generate_final_course(len(self.node_list) - 1)
    return None
```

三、实验结果与分析

3.1 实验结果

最终，在同一个地图环境中测试，三种RRT衍生算法分别生成的的可视化路径和路径规划时间如下：

RRT衍生算法	时间开销	可视化路径
RRT*	3.06s	
Informed RRT	0.71s	

RRT衍生算法	时间开销	可视化路径
Dynamic RRT	0.40s	

3.2 实验分析

- 明显看到，从路径规划时间开销上看，Dynamic RRT和Informed RRT路径规划时间远远低于RRT*，说明两者都能明显提高路径规划效率，而且从图中可以看出Dynamic RRT使用动态步长获取到的路径是优于Informed RRT的，并且时间开销也略低些。
- 从路径优化的角度看，RRT*算法得到的路径显然更短，其路径优于其他两个衍生算法。