AI-PROJECT 1 Cross-line 实验报告

陈之琪 2023010958

搜索算法设计思路详解

一、问题背景与目标

问题描述

在一个n×n的网格中，存在m条线路，每条线路有起点和终点。每条线路的起点需要通过横向或纵向移动（每次移动一格，可上下左右移动）到达终点，且移动路径不能经过其他线路的占用节点（除终点外）。算法需要找到一种移动策略，使得所有线路从起点到达终点，同时最小化路径成本（可包含转向惩罚等）。

设计原则

1. 设计状态表示与转移规则
2. 实现启发式搜索算法，利用启发函数引导搜索方向
3. 支持两种模式：基础模式（无转向惩罚）和改进模式（含转向惩罚）

二、状态表示与数据结构

状态定义（`state`）

状态由三元组构成：`[grid, lines, active_line]`

- `grid`：n×n矩阵，记录每个节点的占用情况（`0`表示空，`i+1`表示第i条线路占据）
- `lines`：线路列表，每个元素为`[start, end]`（坐标格式为`[行, 列]`）
- `active_line`：当前活动线路的索引（未完成线路中优先级最高的，按顺序查找下一个未完成线路）

节点（ Node 类）

节点类的设计思路与第一次编程作业的示例代码类似，而略有不同。

每个节点包含：

- `state`： 当前状态
- `parent`： 父节点（用于路径回溯）
- `action`： 从父节点到当前节点的动作（格式为 `[线路索引, 新位置]` ）
- `path_cost`： 总代价（ `g(n) + h(n)` ，其中 `g(n)` 为实际代价， `h(n)` 为启发函数值）
- `directions`： 各线路的最后移动方向（用于模式2的转向惩罚计算）

三、动作生成与状态转移

可用动作（ actions 方法）

对于当前活动线路，生成其起点的上下左右四个相邻位置中有效的移动位置：

1. 位置在网格内（ `0 ≤ 行/列 < n` ）
2. 位置未被其他线路占据（除非该位置是当前线路的终点）

状态转移（ move 方法）

1. 更新 `grid`： 将新位置标记为当前线路占用
2. 更新 `lines`： 将当前线路的起点移动到新位置
3. 确定下一个 `active_line`： 从当前线路的下一个开始查找，若所有后续线路已完成，则从头查找未完成线路

四、路径成本计算（ g 函数）

基础代价（模式1）

每次移动代价为1，与方向无关：

`g(n) = 父节点g值 + 1`

转向惩罚（模式2）

若当前移动方向与上一次该线路的移动方向不同，额外增加2点代价：

`g(n) = 父节点g值 + 1 + (转向时加2)`

（通过 directions 记录历史方向，判断是否转向）

五、启发函数设计

启发函数 h(n) 用于估计当前状态到目标状态的最小代价，设计了两种策略：

启发函数1：曼哈顿距离之和（ h_function_method1 ）

- **核心思想**：对每条未完成线路，计算起点到终点的曼哈顿距离，求和作为启发值
  曼哈顿距离公式： |start_row - end_row| + |start_col - end_col|
- **数学表达式**：

$$h(n) = \sum_{i=0}^{m-1} \text{Manhattan}(lines[i][0], lines[i][1])$$

- **优点**：计算简单，保证单调性（可采纳性），适用于基础搜索，保证找到最优解。
- **缺点**：未考虑路径中的障碍物，可能导致搜索效率较低

启发函数2：曼哈顿距离+障碍物惩罚（ h_function_method2 ）

- **核心思想**：在曼哈顿距离基础上，对路径中的障碍物（其他线路占据的节点）进行惩罚
  i. 生成横向优先和纵向优先两条曼哈顿路径
  ii. 统计每条路径中的障碍物数量（排除当前线路自身和终点）
  iii. 取障碍物较少的路径，每个障碍物增加2点惩罚
- **数学表达式**：

$$h(n) = \sum_{i=0}^{m-1} (\text{Manhattan}(s_i, e_i) + 2 \times \min(O_h, O_v))$$

  其中 O_h 和 O_v 为横/纵向路径的障碍物数量
- **优点**：结合环境信息，减少无效搜索，提升启发式准确性
- 可以发现，这种启发函数仍然从问题的子问题中去生成，因而仍然是一致的。它可以找到最优解。
- **实现细节**：具体代码请见 .py 文件。
  - generate_horizontal_path ： 先横向移动，再纵向移动
  - generate_vertical_path ： 先纵向移动，再横向移动
  - count_obstacles ： 统计路径中被其他线路占据的节点数

六、搜索算法流程（A*算法变体）

核心逻辑（ search_generator 函数）

1. **优先队列（Open集）**：存储待扩展节点，按 path_cost = g(n) + h(n) 排序，每次取出代价最小的节点
2. **闭合集（Closed集）**：记录已访问的状态，避免重复处理
3. **扩展节点**：对当前节点生成所有合法动作，创建子节点并更新队列：
   - 若子节点未被访问且不在队列中，加入队列
   - 若子节点已在队列中但新代价更低，更新队列中的节点

算法步骤

1. 初始化：将初始状态节点加入优先队列
2. 循环直到队列为空：
   - 取出当前代价最小的节点 current
   - 若 current 是目标状态，结束搜索
   - 生成所有合法动作，创建子节点
   - 对子节点进行剪枝（已访问或代价更高则跳过）
   - 将有效子节点加入优先队列
3. 目标检测：所有线路起点等于终点，且无活动线路

七、目标检测与终止条件

目标状态条件

1. 所有线路的起点等于终点（ start == end ）
2. active_line 为 None （无未完成线路）

终止条件

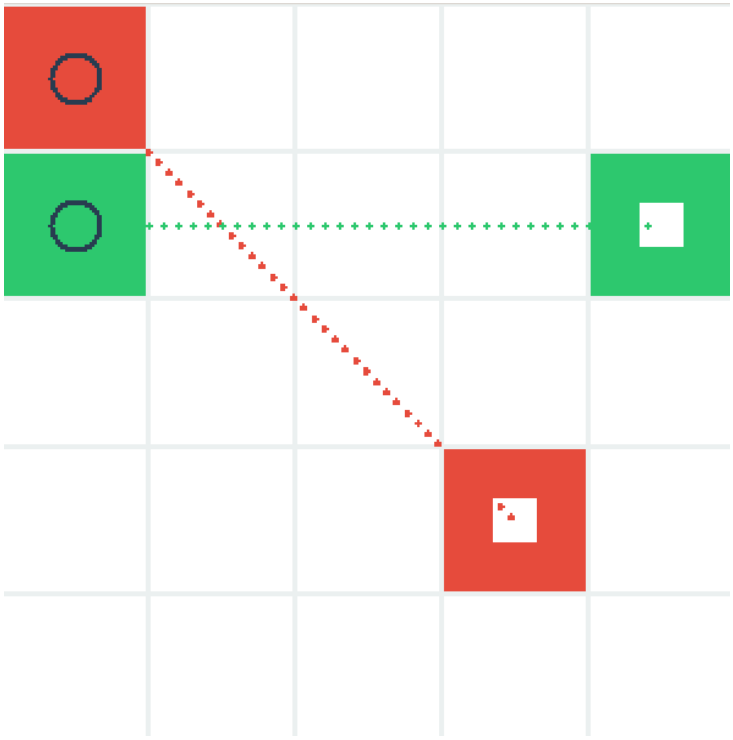
1. 找到目标状态：返回成功，路径成本为 path_cost
2. 队列为空：返回失败，无可行解

搜索算法的结果演示

八、结果示例

取 $n = 5, m = 2$
线路1：起点(1,1) 终点(4,4)
线路2：起点(2,1) 终点(2,5)

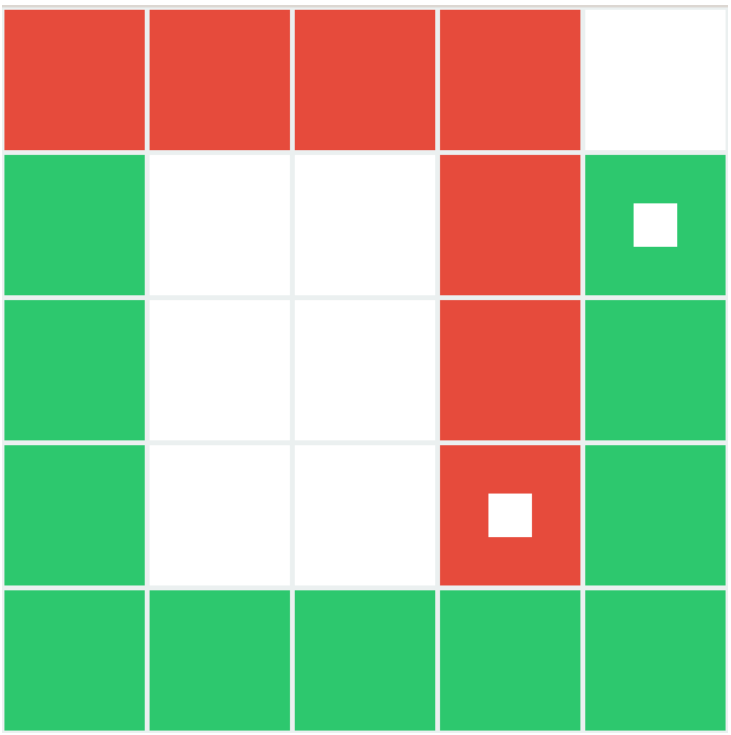
初始状态如下图所示：



1. mode1 基础代价模式搜索结果

代价 cost = 18

2. mode2 转向惩罚模式搜索结果



代价 cost = 24

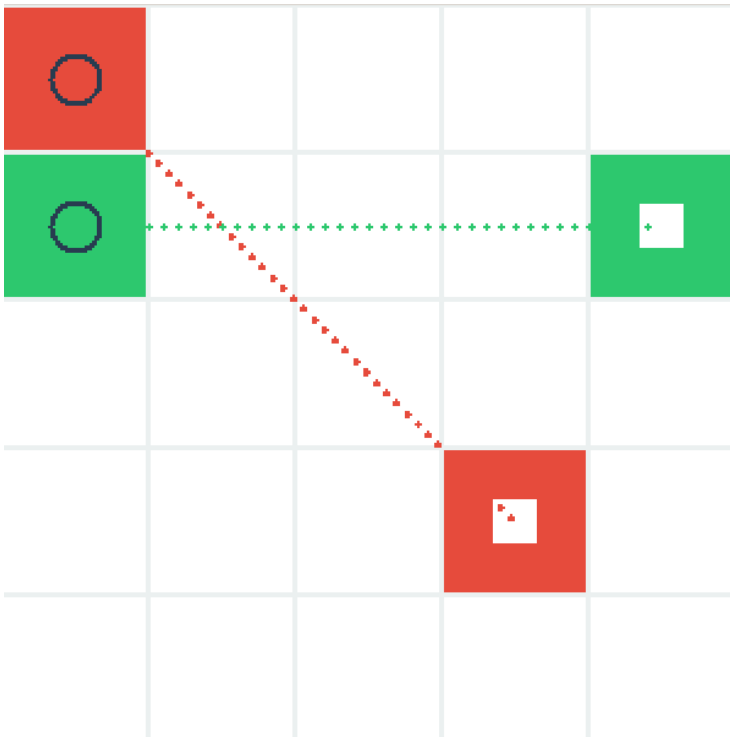
以上两种模式的搜索均找到了最优解。亦可尝试其他样例布局。

搜索算法的结果讨论

九、启发式函数分析

同样以 $n=5, m=2$ 为例，分析两种启发式函数的效果。

- 例1：



mode1 基础代价模式

启发式函数1：曼哈顿距离之和

搜索步长（即访问的状态节点数）$step = 762$

启发式函数2：曼哈顿距离+障碍物惩罚

搜索步长（即访问的状态节点数）$step = 485$

mode2 转向惩罚模式

启发式函数1：曼哈顿距离之和

搜索步长（即访问的状态节点数）$step = 261$

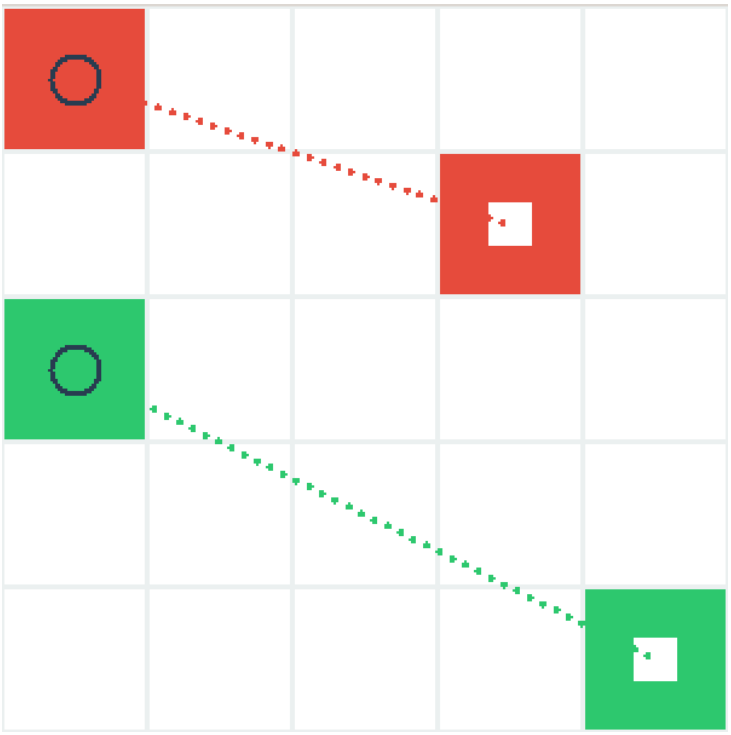
启发式函数2：曼哈顿距离+障碍物惩罚

搜索步长（即访问的状态节点数）$step = 203$

以上这个例子非常好地说明了第二种启发函数的优势。 设计第二种启发式函数的初衷，是考虑到有的情况下，需要匹配的两个点的曼哈顿距离很近，但是它们的曼哈顿连线被另外一条线的必经之路所截断。通过加入障碍物惩罚的机制，可以使得这种情况的h值增加，使得线实现尽早“绕路走”的效果。对比实验结果非常好地体现了这一设计效果。

但是，如果这两条线的两个端点之间的曼哈顿连线本就不存在交点，那第二种启发函数就不见得起到效果了。以下就是一个例子：

- 例2：



mode1 基础代价模式

启发式函数1：曼哈顿距离之和

搜索步长（即访问的状态节点数）$step = 203$

启发式函数2：曼哈顿距离+障碍物惩罚

搜索步长（即访问的状态节点数）$step = 203$

mode2 转向惩罚模式

启发式函数1：曼哈顿距离之和

搜索步长（即访问的状态节点数）$step = 93$

启发式函数2：曼哈顿距离+障碍物惩罚

搜索步长（即访问的状态节点数）$step = 93$

可以看到，在这两条线的两个端点之间的曼哈顿连线本就不存在交点的情况下，两种启发式函数的搜索步长相等，说明第二种启发函数的优势在这里不能体现。 实际上，在搜索过程中，任取一个过程中的状态节点，都会发现该状态下的两条曼哈顿连线都不存在交点。因此，这种情况下，第二种启发函数的优势就不能体现了。

另外，实验又发现，这时使用第二种启发函数的搜索总时长要大于第一种。这是因为每一次调用第二种启发函数的计算都比第一种耗时长。因此在这种情况下，第一种启发函数更适合。需要注意的是这样简单的更适合第一种启发函数发挥的情况是比较少见的，大多数的情况下，线不可避免地要“绕路走”，那第二种启发函数普遍表现更好。