数据结构第二次作业

陈小羽 2016060106020

目录

1 题目要求

- 从文件中读取迷宫数据，寻找并打印路径通路，储存迷宫数据到文件。
- 寻找多入口多出口地图的所有通路。
- 找到入口到出口最短的一条通路。
- 自动生成迷宫地图。

2 寻找路径

2.1 算法选择

按照题目的要求，我们需要找到**起点集合**到**终点集合**之间的一条最短的通路。因为迷宫中相邻的两个格子之间的距离始终为 1，所以我们可以简单的使用**宽度优先搜索 (BFS)** 来完成任务。

2.1.1 算法描述

BFS 算法有些类似现实中的声纳探测器。都从一点开始，向外辐射，寻找目标。不同的是，声纳探测器需要接受返回的声波才能确定是否发现目标，而 BFS 算法找到目标之后可以直接报告。

维护波 为了实现 BFS 算法，我们需要模拟现实中的波。因为我们没有必要走回头路，所以我们只需要维护整个波的**波前**, 也就是走在最前面的**波面**。

队列 因为我们知道，队列具有**先进先出**的优良性质。所以我们可以使用队列来维护**波前**。具体的操作如下:

1 将所有的起点入队。

2 将对首的位置出队。然后访问其周围的位置。

3 如果波访问到了一个位置, 且这个位置之前没有访问过，我们就将这个位置入队。

通过这种方式，我们就用程序模拟了现实中的波。其中，队列中的元素就是波的波前。我们可以发现，队列中的任意两个元素，到起点的距离之差不会超过 1。

寻路算法 我们将所有的起点入队，然后用 BFS 算法模拟若干个波前。这样，我们遇到的第一个终点，就一定是所有的起点到终点中最近的那一个。这样，我们就完成了题目的要求。我们在**算法 1** 中给出了伪代码。

2.1.2 算法分析

因为每一个点最多访问一遍，所以算法的时间复杂度是 $T \in \mathcal{O}(nm)$ 的，其中 n, m 分别表示迷宫地图的长和宽。空间复杂度和时间复杂度一致。

Algorithm 1 BFS 寻路算法

```
function BFS(起点集合 S, 终点集合 T, 地图 Map)
    for 地图中的每一个可能的位置 w_i do                          ▷ 初始化
        Distance[w_i] ← ∞
    end for
    Queue = ∅                                                    ▷ 初始化
    for 每一个在起点集合 S 中的元素 s_i do                       ▷ 起点入队
        Queue.tail ← s_i
        Distance[s_i] ← 0
    end for
    while Queue 非空 do
        Front ← Queue.pop                                        ▷ 对首元素出队
        if Front ∈ T then                                        ▷ 找到答案，直接返回
            return Distance[Front]
        end if
        for Front 的每个邻近位置 p do
            if p 没有障碍物且之前没有访问过 then                  ▷ 扩展波前
                Queue.tail ← p
                Distance[p] ← Distance[Front] + 1
            end if
        end for
    end while
    return ∞                                                     ▷ 没有找到答案
end function
```

3 生成地图

3.1 算法选择

按照 ppt 中的提示，我们可以将迷宫中的所有可以到达的点做成一棵连通的树。实现这个操作的一个很棒的算法就是**克鲁斯卡尔算法**

3.1.1 算法描述

主要的操作很简单，一开始，所有可以到达的点都是不连通的。然后我们考虑不断的打通一些墙，使得最终，所有的这些点都是连通的，且形成一棵树。为此，我们需要维护哪些点已经连通了，因为在这些点之间继续加边的话，会形成环。

维护连通集合 我们使用一种被称为**并查集**的数据结构来维护连通集合。并查集通过树的父亲表示法，表示了一个森林。其中，每个树的根节点的父亲就是这个点自己。这样，如果两个点在这个森林的同一棵树中，我们就说，这两个点在同一个集合中。我们可以快速的判断每个点所在树的根节点，并通过这个来判断两个点是否在同一棵树中。我们在**算法 2** 中给出了这种算法的伪代码。

Algorithm 2 并查集

```
function INITIAL                                          ▷ 初始化
    for 每个点 p_i do
        father[p_i] = p_i
    end for
end function
function FIND(点 p)                                       ▷ 找到点 p 所在树的根节点
    if father[p] = p then
        return p
    else
        return father[p] ← FIND(father[p])                ▷ 路径压缩
    end if
end function
function UNION(点 p_x,p_y)                                ▷ 将 p_x 和 p_y 加入同一个集合
    f_x ← FIND(p_x)
    f_y ← FIND(p_y)
    if f_x ≠ f_y then                                     ▷ 没有这个 if 有可能成环，产生死循环
        father[f_x] = f_y
    end if
end function
```

3.1.2 算法分析

执行算法时，我们枚举了每条边，且并查集每次操作的复杂度可以看作常数，所以，总的复杂度和边的数目同阶，故时间和空间复杂度都是 $\mathcal{O}(nm)$ 的，其中，m, n 分别表示地图的长和宽。

4 结论

4.1 结果

程序运行正常。

4.2 一些值得改进的地方

...