宫水三叶的刷题日记

最小生成树

Author ：宫水三叶
Date ：2021/10/07
QQ Group：703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「图论：最小生成树」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「图论：最小生成树」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「图论：最小生成树」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题'
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **778. 水位上升的泳池中游泳** ，难度为 **困难**。

Tag : 「最小生成树」、「并查集」、「Kruskal」、「二分」、「BFS」

在一个 N x N 的坐标方格 grid 中，每一个方格的值 grid[i][j] 表示在位置 (i,j) 的平台高度。

现在开始下雨了。当时间为 t 时，此时雨水导致水池中任意位置的水位为 t 。

你可以从一个平台游向四周相邻的任意一个平台，但是前提是此时水位必须同时淹没这两个平台。

假定你可以瞬间移动无限距离，也就是默认在方格内部游动是不耗时的。

当然，在你游泳的时候你必须待在坐标方格里面。

你从坐标方格的左上平台 (0，0) 出发，最少耗时多久你才能到达坐标方格的右下平台 (N-1, N-1)？

示例 1:

```
输入: [[0,2],[1,3]]

输出: 3

解释:
时间为0时，你位于坐标方格的位置为 (0, 0)。
此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时的水位。
等时间到达 3 时，你才可以游向平台 (1, 1). 因为此时的水位是 3，坐标方格中的平台没有比水位 3 更高的，所以你可以游向坐标方格中
```

示例2:

```
输入: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]
输出: 16

解释:
 0  1  2  3  4
             5
12 13 14 15 16
11
10  9  8  7  6
```

提示:

- 2 <= N <= 50.
- grid[i][j] 是 `[0, ..., N*N - 1]` 的排列。

Kruskal

由于在任意点可以往任意方向移动，所以相邻的点（四个方向）之间存在一条无向边。

边的权重 w 是指两点节点中的最大高度。

按照题意，我们需要找的是从左上角点到右下角点的最优路径，其中最优路径是指**途径的边的最大权重值最小**，然后输入最优路径中的最大权重值。

我们可以先遍历所有的点，将所有的边加入集合，存储的格式为数组 $[a, b, w]$ ，代表编号为 a 的点和编号为 b 的点之间的权重为 w（按照题意，w 为两者的最大高度）。

对集合进行排序，按照 w 进行从小到达排序。

当我们有了所有排好序的候选边集合之后，我们可以对边从前往后处理，每次加入一条边之后，使用并查集来查询左上角的点和右下角的点是否连通。

当我们的合并了某条边之后，判定左上角和右下角的点联通，那么该边的权重即是答案。

这道题和前天的 1631. 最小体力消耗路径 几乎是完全一样的思路。

你甚至可以将那题的代码拷贝过来，改一下对于 w 的定义即可。

代码：

```java
class Solution {
    int n;
    int[] p;
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    boolean query(int a, int b) {
        return find(a) == find(b);
    }
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    public int swimInWater(int[][] grid) {
        n = grid.length;

        // 初始化并查集
        p = new int[n * n];
        for (int i = 0; i < n * n; i++) p[i] = i;

        // 预处理出所有的边
        // edge 存的是 [a, b, w]：代表从 a 到 b 所需要的时间为 w
        // 虽然我们可以往四个方向移动，但是只要对于每个点都添加「向右」和「向下」两条边的话，其实就已经覆盖
        List<int[]> edges =  new ArrayList<>();
        for (int i = 0; i < n ;i++) {
            for (int j = 0; j < n; j++) {
                int idx = getIndex(i, j);
                p[idx] = idx;
                if (i + 1 < n) {
                    int a = idx, b = getIndex(i + 1, j);
                    int w = Math.max(grid[i][j], grid[i + 1][j]);
                    edges.add(new int[]{a, b, w});
                }
                if (j + 1 < n) {
                    int a = idx, b = getIndex(i, j + 1);
                    int w = Math.max(grid[i][j], grid[i][j + 1]);
                    edges.add(new int[]{a, b, w});
                }
            }
        }

        // 根据权值 w 升序
        Collections.sort(edges, (a,b)->a[2]-b[2]);

        // 从「小边」开始添加，当某一条边别应用之后，恰好使用得「起点」和「结点」联通
```

```java
        // 那么代表找到了「最短路径」中的「权重最大的边」
        int start = getIndex(0, 0), end = getIndex(n - 1, n - 1);
        for (int[] edge : edges) {
            int a = edge[0], b = edge[1], w = edge[2];
            union(a, b);
            if (query(start, end)) {
                return w;
            }
        }
        return -1;
    }
    int getIndex(int i, int j) {
        return i * n + j;
    }
}
```

节点的数量为 $n * n$，无向边的数量严格为 $2 * n * (n - 1)$，数量级上为 n^2。

- 时间复杂度：获取所有的边复杂度为 $O(n^2)$，排序复杂度为 $O(n^2 \log n)$，遍历得到最终解复杂度为 $O(n^2)$。整体复杂度为 $O(n^2 \log n)$。
- 空间复杂度：使用了并查集数组。复杂度为 $O(n^2)$。

注意：假定 Collections.sort() 使用 Arrays.sort() 中的双轴快排实现。

二分 + BFS/DFS

在与本题类型的 1631. 最小体力消耗路径中，有同学问到是否可以用「二分」。

答案是可以的。

题目给定了 $grid[i][j]$ 的范围是 $[0, n^2 - 1]$，所以答案必然落在此范围。

假设最优解为 min 的话（恰好能到达右下角的时间）。那么小于 min 的时间无法到达右下角，大于 min 的时间能到达右下角。

因此在以最优解 min 为分割点的数轴上具有两段性，可以通过「二分」来找到分割点 min。

> 注意：「二分」的本质是两段性，并非单调性。只要一段满足某个性质，另外一段不满足某个性质，就可以用「二分」。其中 33. 搜索旋转排序数组 是一个很好的说明例子。

接着分析，假设最优解为 min，我们在 $[l, r]$ 范围内进行二分，当前二分到的时间为 mid 时：

1. 能到达右下角：必然有 $min \leqslant mid$，让 $r = mid$
2. 不能到达右下角：必然有 $min > mid$，让 $l = mid + 1$

当确定了「二分」逻辑之后，我们需要考虑如何写 $check$ 函数。

显然 $check$ 应该是一个判断给定 时间/步数 能否从「起点」到「终点」的函数。

我们只需要按照规则走特定步数，边走边检查是否到达终点即可。

实现 $check$ 既可以使用 DFS 也可以使用 BFS。两者思路类似，这里就只以 BFS 为例。

代码：

```java
class Solution {
    int[][] dirs = new int[][]{{1,0}, {-1,0}, {0,1}, {0,-1}};
    public int swimInWater(int[][] grid) {
        int n = grid.length;
        int l = 0, r = n * n;
        while (l < r) {
            int mid = l + r >> 1;
            if (check(grid, mid)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return r;
    }
    boolean check(int[][] grid, int time) {
        int n = grid.length;
        boolean[][] visited = new boolean[n][n];
        Deque<int[]> queue = new ArrayDeque<>();
        queue.addLast(new int[]{0, 0});
        visited[0][0] = true;
        while (!queue.isEmpty()) {
            int[] pos = queue.pollFirst();
            int x = pos[0], y = pos[1];
            if (x == n - 1 && y == n - 1) return true;

            for (int[] dir : dirs) {
                int newX = x + dir[0], newY = y + dir[1];
                int[] to = new int[]{newX, newY};
                if (inArea(n, newX, newY) && !visited[newX][newY] && canMove(grid, pos, to
                    visited[newX][newY] = true;
                    queue.addLast(to);
                }
            }
        }
        return false;
    }
    boolean inArea(int n, int x, int y) {
        return x >= 0 && x < n && y >= 0 && y < n;
    }
    boolean canMove(int[][] grid, int[] from, int[] to, int time) {
        return time >= Math.max(grid[from[0]][from[1]], grid[to[0]][to[1]]);
    }
}
```

- 时间复杂度：在 $[0, n^2]$ 范围内进行二分，复杂度为 $O(\log n)$；每一次 BFS 最多

有 n^2 个节点入队，复杂度为 $O(n^2)$。整体复杂度为 $O(n^2 \log n)$

- 空间复杂度：使用了 visited 数组。复杂度为 $O(n^2)$

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **1631. 最小体力消耗路径** ，难度为 **中等**。

Tag : 「最小生成树」、「并查集」、「Kruskal」

你准备参加一场远足活动。

给你一个二维 `rows x columns` 的地图 heights ，其中 `heights[row][col]` 表示格子 `(row, col)` 的高度。

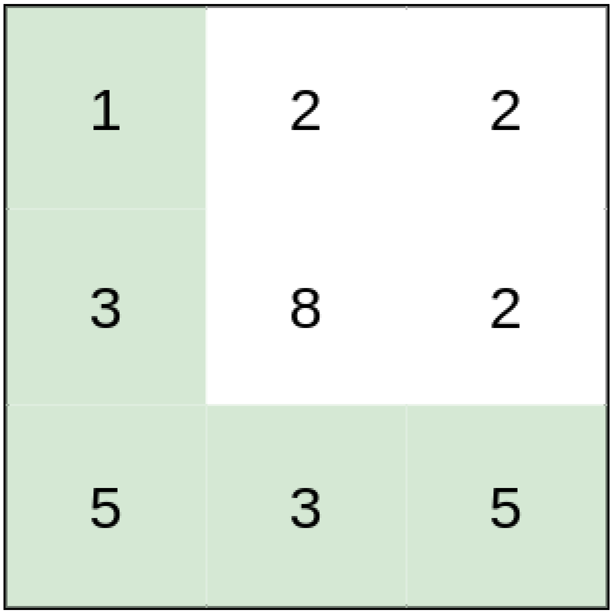
一开始你在最左上角的格子 `(0, 0)` ，且你希望去最右下角的格子 `(rows-1, columns-1)` （注意下标从 0 开始编号）。

你每次可以往 上，下，左，右 四个方向之一移动，你想要找到耗费 体力 最小的一条路径。

一条路径耗费的「体力值」是路径上相邻格子之间「高度差绝对值」的「最大值」决定的。

请你返回从左上角走到右下角的最小 体力消耗值 。

示例 1：




```
输入：heights = [[1,2,2],[3,8,2],[5,3,5]]

输出：2

解释：路径 [1,3,5,3,5] 连续格子的差值绝对值最大为 2 。
这条路径比路径 [1,2,2,2,5] 更优，因为另一条路径差值最大值为 3 。
```

示例 2：

```
输入：heights = [[1,2,3],[3,8,4],[5,3,5]]

输出：1

解释：路径 [1,2,3,4,5] 的相邻格子差值绝对值最大为 1 ，比路径 [1,3,5,3,5] 更优。
```

示例 3：

```
输入：heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]

输出：0

解释：上图所示路径不需要消耗任何体力。
```

提示：

- rows == heights.length

- columns == heights[i].length
- 1 <= rows, columns <= 100
- 1 <= heights[i][j] <= 10^6

基本分析

对于这道题，可能会有同学想这是不是应该用 DP 呀？

特别是接触过「路径问题」但又还没系统学完的同学。

事实上，当题目允许往任意方向移动时，考察的往往就不是 DP 了，而是图论。

从本质上说，DP 问题是一类特殊的图论问题。

那为什么有一些 DP 题目简单修改条件后，就只能彻底转化为图论问题来解决了呢？

这是因为修改条件后，导致我们 DP 状态展开不再是一个拓扑序列，也就是我们的图不再是一个拓扑图。

换句话说，DP 题虽然都属于图论范畴。

但对于不是拓扑图的图论问题，我们无法使用 DP 求解。

而此类看似 DP，实则图论的问题，通常是最小生成树或者最短路问题。

Kruskal

当一道题我们决定往「图论」方向思考时，我们的重点应该放在「如何建图」上。

因为解决某个特定的图论问题（最短路/最小生成树/二分图匹配），我们都是使用特定的算法。

由于使用到的算法都有固定模板，因此编码难度很低，而「如何建图」的思维难度则很高。

对于本题，我们可以按照如下分析进行建图：

因为在任意格子可以往「任意方向」移动，所以相邻的格子之间存在一条无向边。

题目要我们求的就是**从起点到终点的最短路径中，边权最大的值**。

我们可以先遍历所有的格子，将所有的边加入集合。

存储的格式为数组 $[a, b, w]$ ，代表编号为 a 的点和编号为 b 的点之间的权重为 w。

按照题意，w 为两者的高度差的绝对值。

对集合进行排序，按照 w 进行从小到大排序（Kruskal 部分）。

当我们有了所有排好序的候选边集合之后，我们可以对边进行从前往后处理，每次加入一条边之后，使用并查集来查询「起点」和「终点」是否连通（并查集部分）。

当第一次判断「起点」和「终点」联通时，说明我们「最短路径」的所有边都已经应用到并查集上了，而且由于我们的边是按照「从小到大」进行排序，因此最后一条添加的边就是「最短路径」上权重最大的边。

代码：

```java
class Solution {
    int N = 10009;
    int[] p = new int[N];
    int row, col;
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    boolean query(int a, int b) {
        return p[find(a)] == p[find(b)];
    }
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }
    public int minimumEffortPath(int[][] heights) {
        row = heights.length;
        col = heights[0].length;

        // 初始化并查集
        for (int i = 0; i < row * col; i++) p[i] = i;

        // 预处理出所有的边
        // edge 存的是 [a, b, w]：代表从 a 到 b 的体力值为 w
        // 虽然我们可以往四个方向移动，但是只要对于每个点都添加「向右」和「向下」两条边的话，其实就已经覆
        List<int[]> edges = new ArrayList<>();
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                int idx = getIndex(i, j);
                if (i + 1 < row) {
                    int a = idx, b = getIndex(i + 1, j);
                    int w = Math.abs(heights[i][j] - heights[i + 1][j]);
                    edges.add(new int[]{a, b, w});
                }
                if (j + 1 < col) {
                    int a = idx, b = getIndex(i, j + 1);
                    int w = Math.abs(heights[i][j] - heights[i][j + 1]);
                    edges.add(new int[]{a, b, w});
                }
            }
        }

        // 根据权值 w 降序
        Collections.sort(edges, (a,b)->a[2]-b[2]);

        // 从「小边」开始添加，当某一条边别应用之后，恰好使用得「起点」和「结点」联通
        // 那么代表找到了「最短路径」中的「权重最大的边」
```

```
        int start = getIndex(0, 0), end = getIndex(row - 1, col - 1);
        for (int[] edge : edges) {
            int a = edge[0], b = edge[1], w = edge[2];
            union(a, b);
            if (query(start, end)) {
                return w;
            }
        }
        return 0;
    }
    int getIndex(int x, int y) {
        return x * col  + y;
    }
}
```

令行数为 r，列数为 c，那么节点的数量为 $r * c$，无向边的数量严格为 $r * (c - 1) + c * (r - 1)$，数量级上为 $r * c$。

- 时间复杂度：获取所有的边复杂度为 $O(r * c)$，排序复杂度为 $O((r * c)\log{(r * c)})$，遍历得到最终解复杂度为 $O(r * c)$。整体复杂度为 $O((r * c)\log{(r * c)})$。
- 空间复杂度：使用了并查集数组。复杂度为 $O(r * c)$。

证明

我们之所以能够这么做，是因为「跳出循环前所遍历的最后一条边必然是最优路径上的边，而且是 w 最大的边」。

我们可以用「反证法」来证明这个结论为什么是正确的。

我们先假设「跳出循环前所遍历的最后一条边必然是最优路径上的边，而且是 w 最大的边」不成立：

我们令循环终止前的最后一条边为 a

1. *假设 a 不在最优路径内*：如果 a 并不在最优路径内，即最优路径是由 a 边之前的边构成，那么 a 边不会对左上角和右下角节点的连通性产生影响。也就是在遍历到该边之前，左上角和右下角应该是联通的，逻辑上循环会在遍历到该边前终止。与我们循环的决策逻辑冲突。

2. *a 在最优路径内，但不是 w 最大的边*：我们在遍历之前就已经排好序。与排序逻辑冲突。

因此，我们的结论是正确的。`a` 边必然属于「最短路径」并且是权重最大的边。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「图论：最小生成树」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：


"给作者手机充个电"
YOLO 的赞赏码