链滴

LeetCode #934 最短的桥

作者：matthewhan

原文链接：https://ld246.com/article/1597372684104

来源网站：链滴

Problem Description

在给定的二维二进制数组 A 中，存在两座岛。（岛是由四面相连的 1 形成的一个最大组。）

现在，我们可以将 0 变为 1，以使两座岛连接起来，变成一座岛。

返回必须翻转的 0 的最小数目。（可以保证答案至少是 1。）

note

- 1 <= A.length = A[0].length <= 100
- A[i][j] == 0 或 A[i][j] == 1

e.g.

- **示例 1：**
  - 输入： [[0,1],[1,0]]
  - 输出：1
- **示例 2：**
  - 输入： [[0,1,0],[0,0,0],[0,0,1]]
  - 输出：2
- **示例 3：**
  - 输入： [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]
  - 输出：1

原文链接：LeetCode #934 最短的桥

Solution

题目保证答案至少是1，那么一定是2个岛。岛的定义是上下左右相连所以斜着的不算。返回翻转0的小值，这个最小值就是桥的长度，也就是「最短路径」。

一般看到「最短路径」我们会想到 bfs、A*、Dijkstra算法，一般都是点对点的，这里是两个岛屿，以**只要将一个岛的所有边缘上的点都求一遍到第二个岛的路径**，返回最小的路径就可以了。

所以问题就是怎么找出这两个岛？岛屿只有上下左右相连，所以可以根据深度优先搜索，将某坐标的下左右进行搜索，如果是1则继续搜索，是0则终止。为了将两个岛区分开，我们需要将第一个找到的「涂色」，因为后面需要找第二个岛，如果都用1表示就没法区分开了。

注意的点：

1. bfs去找第二个岛，因为是第一个岛的每个节点去广搜，所以 visited访问标记每次循环需要初始。
2. bfs周围的点如果是 -1的话没必要入队，因为可能同是旁边的点（旁边的点会bfs一次，这里重了），或者是第一个岛的非边缘的点，一定不是「最短路径」。
3. bfs一定能找到第二个岛的点，所以一定是在迭代里 return的。
4. 优化点也有，比如在该题其实不需要 visited访问标记，每次初始化很浪费时间空间。

```
public class ShortestBridge {
    public static int shortestBridge(int[][] arr) {
        List<int[]> firstLand = new ArrayList<>();
        boolean[][] visited = new boolean[arr.length][arr[0].length];
        boolean flag = false;
        for (int i = 0; i < arr.length; i++) {
            if (flag) {
                break;
            }
             // 设置一个flag，找到后需要继续遍历这个arr了。
            for (int j = 0; j < arr[i].length; j++) {
                if (arr[i][j] == 1) {
                    dfs(arr, i, j, visited, firstLand);
                    flag = true;
                    break;
                }
            }
        }
        int ans = Integer.MAX_VALUE;
        for (int[] xy : firstLand) {
            // 【注意】每次初始化访问标记【注意】
            visited = new boolean[arr.length][arr[0].length];
            ans = Math.min(ans, bfs(arr, xy[0], xy[1], visited));

        }
        return ans;
    }

    /**
     *
     * @param arr
     * @param x
```

```
     * @param y
     * @param visited
     * @param firstLand
     */
    public static void dfs(int[][] arr, int x, int y, boolean[][] visited, List<int[]> firstLand) {
        if (x >= 0 && y >= 0 && x < arr.length && y < arr[0].length && !visited[x][y]) {
            visited[x][y] = true;
            if (arr[x][y] == 0) {
                return;
            }
            arr[x][y] = -1;
            firstLand.add(new int[]{x, y});
            // 仅仅只有上下左右
            dfs(arr, x - 1, y, visited, firstLand);
            dfs(arr, x + 1, y, visited, firstLand);
            dfs(arr, x, y - 1, visited, firstLand);
            dfs(arr, x, y + 1, visited, firstLand);
        }
    }


    /**
     * wcnd，写死我了
     *
     * @param arr
     * @param x
     * @param y
     * @param visited
     * @return
     */
    public static int bfs(int[][] arr, int x, int y, boolean[][] visited) {
        // 最后要减一
        int len = -1;
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[]{x, y});
        visited[x][y] = true;
        while (!queue.isEmpty()) {
            int limit = queue.size();
            for (int i = 0; i < limit; i++) {
                int[] curr = queue.poll();
                x = curr[0];
                y = curr[1];
                // 找到了1
                if (arr[x][y] == 1) {
                    return len;
                }
                // 如果周围的节点是-1的话，没必要入队
                if (x - 1 >= 0 && !visited[x - 1][y] && arr[x - 1][y] != -1) {
                    queue.offer(new int[]{x - 1, y});
                    visited[x - 1][y] = true;
                }
                if (x + 1 < arr.length && !visited[x + 1][y] && arr[x + 1][y] != -1) {
                    queue.offer(new int[]{x + 1, y});
                    visited[x + 1][y] = true;
```

```
                }
                if (y - 1 >= 0 && !visited[x][y - 1] && arr[x][y - 1] != -1) {
                    queue.offer(new int[]{x, y - 1});
                    visited[x][y - 1] = true;
                }
                if (y + 1 < arr[0].length && !visited[x][y + 1] && arr[x][y + 1] != -1) {
                    queue.offer(new int[]{x, y + 1});
                    visited[x][y + 1] = true;
                }
            }
            len++;
        }
        // 一定是可以找到最短路径的，所以这里返回的是错误的
        return Integer.MAX_VALUE;
    }
}
```