题目地址(1162. 地图分析)

https://leetcode-cn.com/problems/as-far-from-land-as-possible/

入选理由

1. 继续来一道常规的搜索，最后两天我们再来点不一样的

标签

- BFS

难度

- 中等

题目描述

你现在手里有一份大小为 N x N 的 网格 grid，上面的每个 单元格 都用 0 和 1 标记好了。
其中 0 代表海洋，1 代表陆地，请你找出一个海洋单元格，
这个海洋单元格到离它最近的陆地单元格的距离是最大的。

我们这里说的距离是「曼哈顿距离」（ Manhattan Distance）：
(x0, y0) 和 (x1, y1) 这两个单元格之间的距离是 |x0 - x1| + |y0 - y1| 。

如果网格上只有陆地或者海洋，请返回 -1。

示例 1:

1	0	1
0	0	0
1	0	1

image

输入：[[1,0,1],[0,0,0],[1,0,1]]
输出：2

解释：
海洋单元格 (1, 1) 和所有陆地单元格之间的距离都达到最大，最大距离为 2。
示例 2：

1	0	0
0	0	0
0	0	0

image

输入：[[1,0,0],[0,0,0],[0,0,0]]
输出：4
解释：
海洋单元格 (2, 2) 和所有陆地单元格之间的距离都达到最大，最大距离为 4。

提示：

1 <= grid.length == grid[0].length <= 100
grid[i][j] 不是 0 就是 1

公司

- 字节跳动

思路

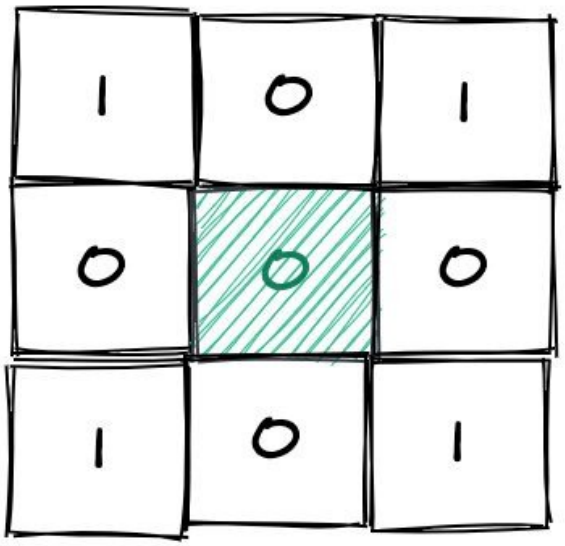
这里我们继续使用[上面两道题的套路](#)，即不用 visited，而是原地修改。由于这道题求解的是最远的距离，而距离我们可以使用 BFS 来做。

> 虽然 bfs 是求最短距离的，但是这道题其实就是让我们求最短距离最大的海洋单元格，因此仍然可以用 bfs 来解。

算法：

- 对于每一个海洋，我们都向四周扩展，寻找最近的陆地，每次扩展 steps 加 1。
- 如果找到了陆地，我们返回 steps。
- 我们的目标就是**所有 steps 中的最大值**。

实际上面算法有很多重复计算，如图中间绿色的区域， 向外扩展的时候，如果其周边四个海洋的距离已经计算出来了， 那么没必要扩展到陆地。实际上只需要扩展到周边的四个海洋格子就好了， 其距离陆地的最近距离就是 1 + 周边四个格子中到达陆地的最小距离。

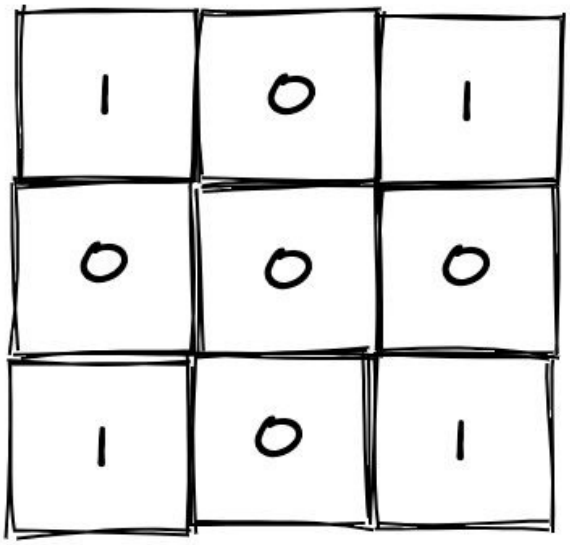


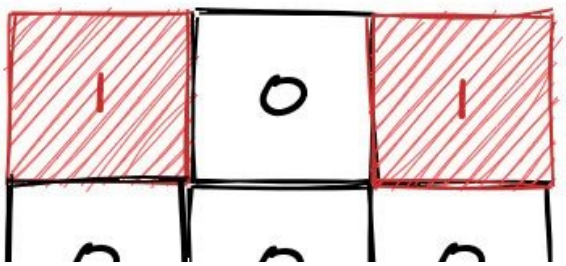
那么如何优化这种重复计算呢?

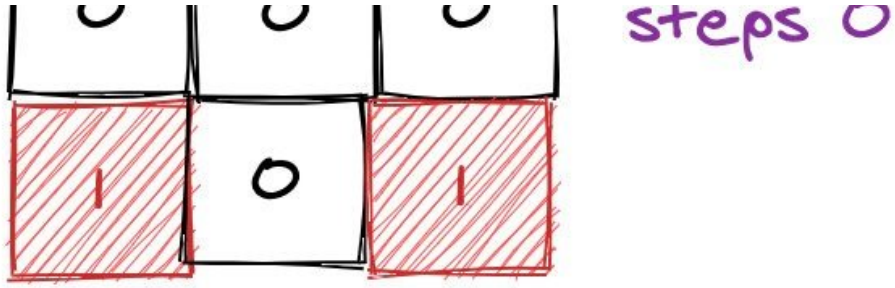
一种优化的方式是将所有陆地加入队列，而不是海洋。陆地不断扩展到海洋，每扩展一次就 steps 加 1，直到无法扩展位置，最终返回 steps 即可。

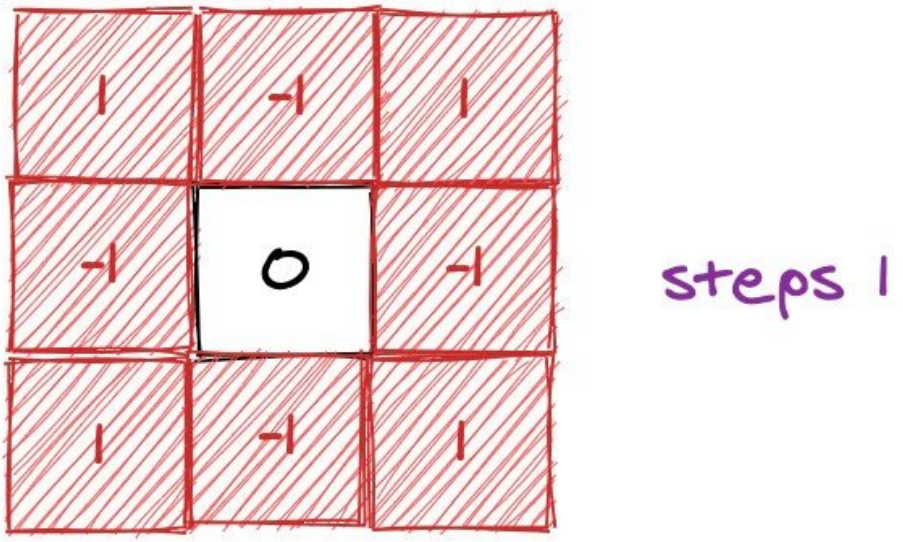
由于我们是陆地找海洋，因此从陆地开始找到的最后一次海洋就是答案。

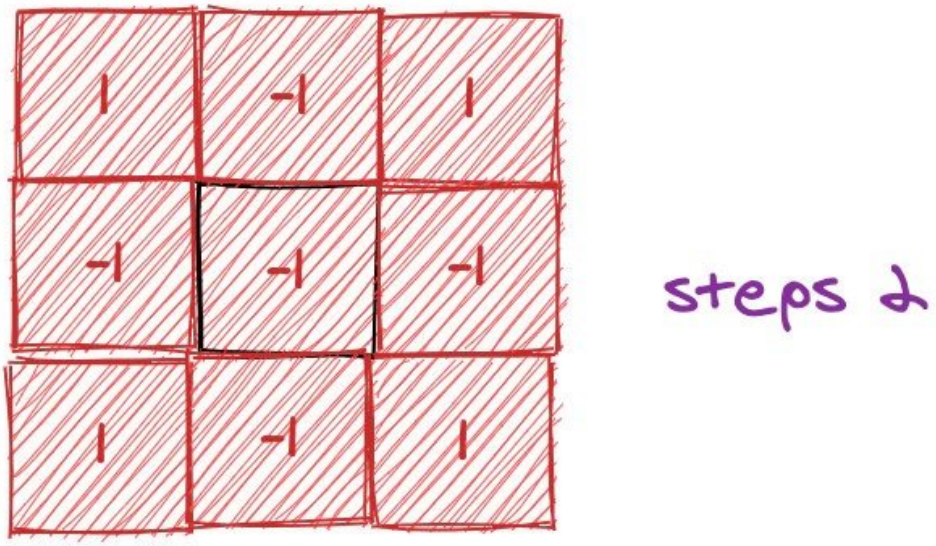
图解：











简单来说，我们的算法就是**从陆地边缘开始探索，探索到不能探索位置**。

关键点

- 陆地入队，而不是海洋入队

代码

- 语言支持：Python，CPP

```python
class Solution:
    def maxDistance(self, grid: List[List[int]]) -> int:
        n = len(grid)
        steps = -1
        queue = collections.deque([(i, j) for i in range(n) for j in range(n) if grid[i][j] == 1])
        if len(queue) == 0 or len(queue) == n ** 2: return steps
        while len(queue) > 0:
            for _ in range(len(queue)):
                x, y = queue.popleft(0)
                for xi, yj in [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]:
                    if xi >= 0 and xi < n and yj >= 0 and yj < n and grid[xi][yj] == 0:
                        queue.append((xi, yj))
                        grid[xi][yj] = -1
            steps += 1

        return steps
```

CPP Code:

```cpp
class Solution {
    const int dx[4] = {0, 1, 0, -1};
    const int dy[4] = {1, 0, -1, 0};
public:
    int maxDistance(vector<vector<int>>& grid) {
        int N = grid.size();
        queue<pair<int, int>> q; // queue存储坐标值, 即 pair of {x, y}
        vector<vector<int>> d(N, vector(N, 1000)); // 二维数组d[][]: 记录每个格子grid[i][j]的距离值
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
                if (grid[i][j] == 1)
                {
                    q.push(make_pair(i, j));
                    d[i][j] = 0;
                }
        }
        while (!q.empty())
        {
            auto kvp = q.front();
```

```
            q.pop();
            for (int i = 0; i < 4; i++)
            {
                int newX = kvp.first + dx[i], newY = kvp.second + dy[i];
                if (newX < 0 || newX >= N || newY < 0 || newY >= N) // 越界了，跳过
                    continue;

                if (d[newX][newY] > d[kvp.first][kvp.second] + 1) /* 如果从水域(值为0的格子)走到陆地(值为1的格子)或从陆地走到水域
                {
                    d[newX][newY] = d[kvp.first][kvp.second] + 1; /* 当前格子的上下左右4个方向之一走一步恰好使得曼哈顿距离增加1 *
                    q.push(make_pair(newX, newY));
                }
            }
        }
        int res = -1;
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                if (grid[i][j] == 0 && d[i][j] <= 200) /* 挑出访问过的水域位置(值为0的格子)，并维护这些格子中距离值d的最大值 */
                    res = max(res, d[i][j]);
            }
        }
        return res;
    }
};
```

复杂度分析

- 时间复杂度：由于 grid 中的每个点最多被处理一次，因此时间复杂度为 $O(N^2)$
- 空间复杂度：由于我们使用了队列，而队列的长度最多是 n^2，这种情况其实就是全为 1 的 grid，因此空间复杂度为 $O(N^2)$

上一页 下一页