Number of Islands

题目描述

Given a 2d grid map of `'1'` s (land) and `'0'` s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
11110
11010
11000
00000
```

Answer: 1

Example 2:

```
11000
11000
00100
00011
```

Answer: 3

解题思路

拿到题目，如果对数据结构——“图”的内容比较熟悉，很容易就能想到用 `DFS` 或 `BFS` 来解决。通过这两种途径解决该问题相对不太难，并且网上有大量讲解，所以我不再赘述。今天要说的是种新的算法（对我来说）—— `union-find 算法` （见《算法》1.5节）。

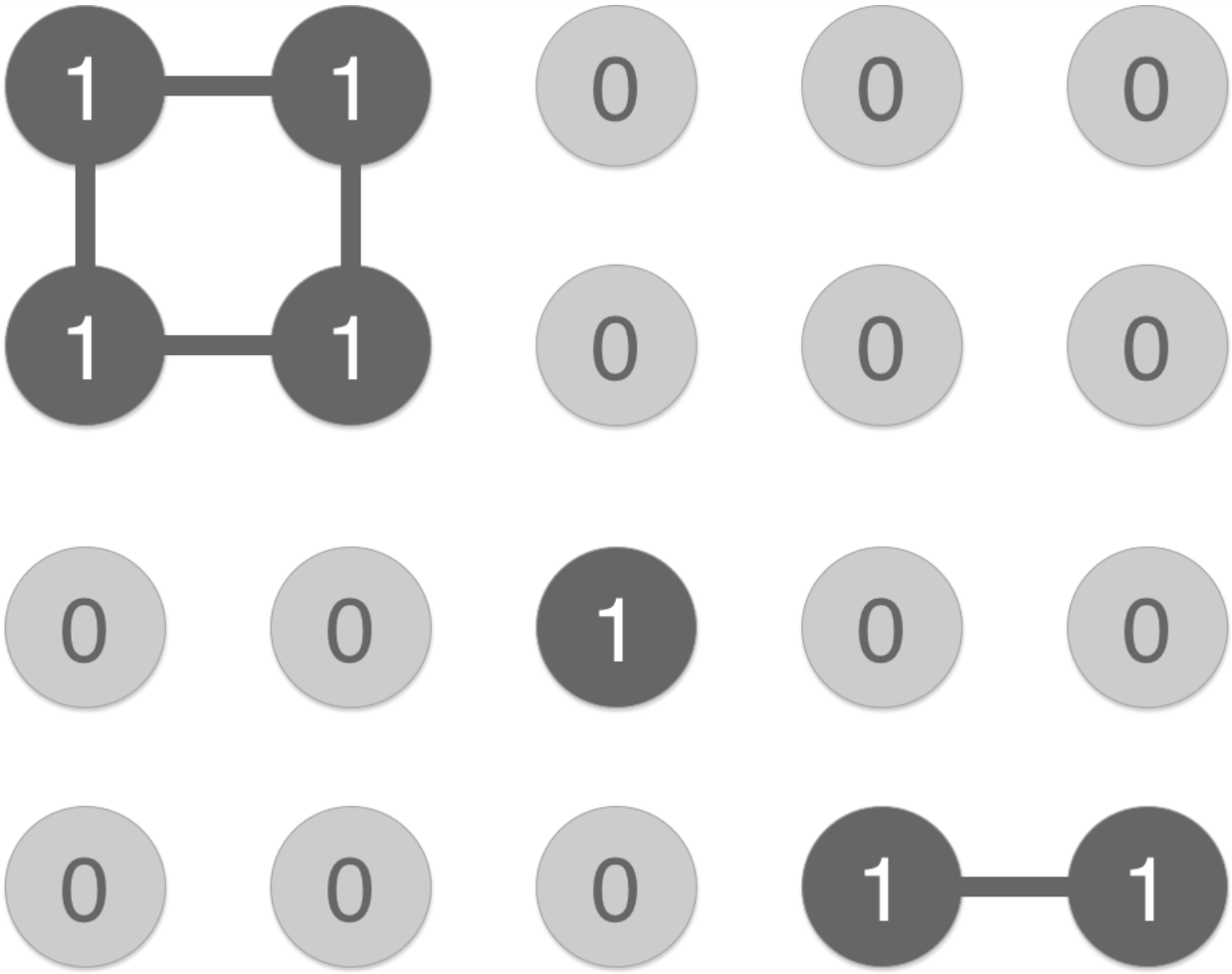
union-find 算法简介

顾名思义， `union` 即合并， `find` 为查找，核心就在这两部分。完整的算法的行为就是通过不断的“find”出某两个元素分别所属的集合，判断他们是否是同一个，然后将不属于同一集合的两个元素“union”到同一个集合中。

那它到底是用来干嘛的呢？嗯...书上说...它主要是用来解决 `动态连通性问题` 的。如果你不知道什么是 `动态连通性问题` ，那还是请你查阅书籍，我不觉得自己有能力解释的比书上更好起码在现阶段...

好在要解决今天这个问题，你还不需要明白那些复杂的东西，让我们进入正题。

UF 数据结构的实现（C++）



在此之前我们先来分析下题目：

将所给二维数组抽象成一个图（如上），标记 `1` 的为陆地，标记为 `0` 的是水域，相邻的陆地连接在一起成为岛屿，即图中的连通分量。所以本题就转换成为：**求所给图的连通分量总数**。

```
class UF {
public:
    int count = 0;  //用来记录总的连通分量数目
    int *id;        //数组的元素对应各顶点，存储的内容为它自身所属所属连通分量的名称
    //这里不太容易理解，我打个比方：
    //一群互不相识的小孩儿参加夏令营，老师把他们分成多组，每组选出一人为队长，并要求按组排队集合。集合站队时，队长站在最前方，其他人通过辨认自己的队长来选择自己的队伍。理论上只要每人都记住队长的样子就能站好队伍，但也并非必须如此。也许在第一次排队的时候小孩儿B没记住自己的队长，但他记住了自己前面的小朋友A，那只要A站对了位置，他就能跟着A站到正确的位置了
    //这里id数组的元素就像是一个小孩儿，它所存储的就是自己记住的那个跟自己在同一队的小伙伴的样子，当然这个人可能是队长，也可能是其他任何一个同队的人


  //构造函数
    UF(int m, int n, vector<vector<char>>& grid) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') count++;
                //初始化count值的时候，我们假设每块陆地起初都是孤立的，所以有多少块陆地，就有多少个连通分量
            }
        }
        int a[m*n];
```

```
        id = a;
        for (int i = 0; i < m * n; i++) {
            id[i] = i;
            //如上面所假设的，每块陆地都是孤立的（换成排队即指每个孩子除了自己谁都不认识，所以他们各自为营），所以自己的id里存储的就是自己的下标
        }
    }


    //寻找p所属的连通分量的名称（换成排队即寻找p小孩儿的队长）
    int find(int p) {
        while (p != id[p]) {
            id[p] = id[id[p]];
            p = id[p];
        }
        return p;
    }
    //当然队长只需要认识自己，站在原地不动就好，所以如果id[p]==p,那他就是队长，否则就说明他（id[p]）只是P小孩记住的那个同队的小伙伴。为了找到队长，需要再问id[p]小朋友他记住的那个同队的小伙伴（id[id[p]]）是不是队长了。一直这么问下去，总会找到队长本人的（毕竟不听话的捣蛋鬼只是少数呀）


    //判断p，q是否属于同一连通分量（即两块陆地是否相连...或者两个小孩是不是同一队的）
    bool isConnected(int p, int q) {
        int pRoot = find(p);     //p的队长
        int qRoot = find(q);     //q的队长
        if (pRoot != qRoot)
          return false; //若两个队长不是同一个人，则他们不同队
        else
          return true;  //否则同队
    }


    //合并p，q所属的两个连通分量（或者说把两队小孩组成一队）
    void myUnion(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if (pRoot == qRoot) return;
        id[pRoot] = qRoot;  //让p的队长（pRoot）认q的队长（qRoot）为自己的队长，就是说pRoot的职务被罢免了...
        count--;     //结果当然是总队伍数少一个～
    }
};
```

利用 UF 设计算法求岛屿数

实现数据结构的时候，我们假设一开始每块陆地是孤立的，就是把每块陆地都当成一个岛屿。这显然不符合题意，现在我们要做的就是找出所有相连的陆地，并把他们 `union` 到一起，直到所有相连的陆地都被包含在同一个岛屿中为止。

```
//求总的岛屿数（即其中的连通分量总数）
int numIslands(vector<vector<char>>& grid) {
    if (grid.size() == 0 || grid[0].size() == 0)
      return 0;      //如果没有陆地，当然就没有岛屿...
    int m = (int)grid.size(), n = (int)grid[0].size();
    UF uf = UF(m, n, grid);

    //从上向下，从左向右地遍历整个图，遇到陆地，就把它和自己周围的（上下左右四个方向）陆地or岛屿合并
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '0') continue;
            int p = i * n + j;
            int q;
            if (i < m - 1 && grid[i + 1][j] == '1') {   //右边相邻
                q = p + n;
                uf.myUnion(p, q);
            }
            if (j < n - 1 && grid[i][j + 1] == '1') {   //下边相邻
                q = p + 1;
                uf.myUnion(p, q);
            }
        }
    }
    //由于我们是按照从上向下，从左向右的顺序进行遍历，所以当我们遍历到某顶点时，它上面和左边的顶点一定已经遍历过了。所以事实上我们只需要对每个顶点作右和下方的判断即可

    return uf.count;    //每次union都会count--，所以当完整遍历整个图之后，count就是我们需要的岛屿数量了
}


int main(int argc, const char * argv[]) {
    vector<vector<char>> grid;
    string s[4] = {"11000", "11000", "00100", "00011"};
    int i = 0;
    while (i < 4) {
        vector<char> line(s[i].begin(), s[i].begin() + s[i].length());
        grid.push_back(line);
        ++i;
    }
    cout << numIslands(grid) <<endl;
    return 0;
}
```

总结

一开始我只是看到说 `union-find 算法` 可以解决岛屿问题，刚好手边的《算法》书里有详细讲解，就想学习后自己试着实现一下。可当我花了近两个小时终于似乎学会了这个算法，打算小试牛刀的时候，却完全找不到用它解决岛屿问题的思路。最终我还是放弃了，到 LeetCode 上查看了大佬的 solution，并醍醐灌顶，自叹不如。大佬是用 Java 实现的，看过之后为加深印象，也为温习 C++，我决定重新实现一下，便有了这篇文章。写过之后我发现自己对这个算法的理解更深了一层，希望能帮助更多的朋友。