January 3, 2023 ▪ Graph

Number of Islands – II – Online Queries – DSU: G-51

Problem Statement: You are given an n, m which means the row and column of the 2D matrix, and an array of size k denoting the number of operations. Matrix elements are 0 if there is water or 1 if there is land. Originally, the 2D matrix is all 0 which means there is no land in the matrix. The array has k operator(s) and each operator has two integers A[i][0], A[i][1] means that you can change the cell matrix[A[i][0]][A[i][1]] from sea to island. Return how many islands are there in the matrix after each operation. You need to return an array of size k.

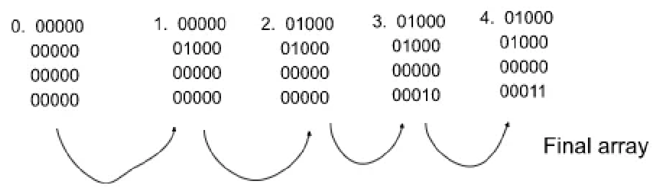
Note: An island means a group of 1s such that they share a common side.

Pre-requisite: Disjoint Set data structure

Example 1:

Input Format: n = 4 m = 5 k = 4 A = {{1,1},{0,1},{3,3},{3,4}} **Output:** 1 1 2 2

Explanation: The following illustration is the representation of the operation:



Example 2:

Input Format: n = 4 m = 5 k = 12 A = {{0,0},{0,0},{1,1},{1,0},{0,1},{0,3},{1,3},{0,4}, {3,2},{2,2},{1,2}, {0,2}} **Output:** 1 1 2 1 1 2 2 2 3 3 1 1 **Explanation:** If we follow the process like in example 1, we will get the above result.

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Problem Link.

Before moving on to the solution, let's quickly discuss some points about the question. First, we need to remember that an island means a group of 1s such that they share a common side. If we look into it from the matrix view, the statement actually means that two cells with value 1 are considered a single group if one of them is located in any of the four directions (Up, Down, Left, Right) of the other cell. But two

Recent Posts

Floor and Ceil in Sorted Array

Search Insert Position

Implement Upper Bound

Implement Lower Bound

2023 – Striver's SDE Sheet Challenge

diagonal adjacent cells will not be considered a single group rather they will be counted as different groups. The following illustration will depict the concept:

Here cells [0,0] and [0,1] are considered a single island as they share a common side but cells [0,1] and [1,2] must be considered two different islands as they do not have any common side.

Now, in the question, it is clearly stated that the operations are given in an array and we should find the number of islands after each operation. This fact actually indicates that after performing each operation the structure of the islands and the sea may change. If we assume the structure as a graph, the graph will be dynamic in nature. And there is also a concept of connecting two different islands if they share a common side.

So, from these observations, we can easily decide to choose the Disjoint Set data structure in order to solve this problem.

These types of problems are considered online query problems where we need to find the result after every query.

Let's discuss the following observations:

Observation 1: *What does each operation/query mean?*

In each operation/query, an index of a cell will be given and we need to add an island on that particular cell i.e. we need to place the value 1 to that particular cell.

Observation 2: *Optimizing the repeating same operations*

The same operations may repeat any number of times but it is meaningless to perform all of them every time. So, we will maintain a visited array that will keep track of the cells on which the operations have been already performed. If the operations repeat, by just checking the visited array we can decide not to calculate again, and instead, just take the current answer into our account. Thus we can optimize the number of operations.

Observation 3: *How to connect cells to include them in the same group or consider them a single island.*

Generally, a cell is represented by two parameters i.e. row and column. But to connect the cells as we have done with nodes, we need to first represent each cell with a single number. So, we will number them from 0 to n*m-1(from left to right) where n = no. of total rows and m = total no. of columns.

For example, if a 5X4 matrix is given we will number the cell in the following way:

Now if we want to connect cells (1,0) and (2,0), we will just perform a union of 5 and 10. The number for representing each cell can be found using the following formula: number = (row of the current cell*total number of columns)+column of the current cell for example, for the cell (2, 0) the number is = (2*5) + 0 = 10.

Observation 4: *How to count the number of islands.*

For each operation, if the given cell is not visited, we will first mark the cell visited and increase the counter by 1. Now we will check all four sides of the given cell. If any other islands are found, we will connect the current cell with each of them(If not already connected) decreasing the counter value by 1. While connecting we need to check if the cells are already connected or not. For this, we will first convert the cells' indices into numbers using the above formula and then we will check their ultimate parents. If the parents become the same, we will not connect them as well as we will not make any changes to the counter variable. Thus the number of islands will be calculated.

Approach:

The algorithm steps are as follows:

Initial Configuration:
Visited array: This 2D array should be initialized with 0.
Counter variable: This variable will also be initialized with 0.
Answer array: After performing the algorithm, this array will store the results after performing the queries.

1. First, we will iterate over all the queries selecting each at a time. Now, we can get the row and the column of the cell given in that query.
2. Then, we will check that cell in the visited array, if the cell is previously visited or not.

    1. ***If the cell is previously visited***, we will just take the current count into our account storing that count

value in our answer array and we will move on to the next query.

2. ***Otherwise,*** we will mark the cell as visited in the visited array and increase the value of the counter variable by 1.

    1. Now, it's time to connect the adjacent islands properly. For that, we will check all four adjacent cells of the current cell. If any island is found, we will first check if they(the current cell and the adjacent cell that contains an island) are already connected or not using the **findUPar()** method.
    2. For checking, we will first convert the indices of the current cell and the adjacent cell into the numbers using the specified formula. Then we will check their ultimate parents.
    3. ***If the ultimate parents are different***, we will decrease the counter value by 1 and perform the union(***either unionBySize() or unionByRank()***) between those two numbers that represent the cells.

    4. Similarly, checking all four sides and making the required changes in the counter variable, we will put the counter value into our answer array.
3. After performing step 2 for all the queries, we will get our final answer array containing the results for all the queries.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

C++ Code

```
#include <bits/stdc++.h>
using namespace std;

// User function Template for C++
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++)
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUP
    }
    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_
        }
    }
};
class Solution {
private:
    bool isValid(int adjr, int adjc,
        return adjr >= 0 && adjr < n
    }
public:
    vector<int> numOfIslands(int n,
                                vector<
        DisjointSet ds(n * m);
        int vis[n][m];
        memset(vis, 0, sizeof vis);
        int cnt = 0;
        vector<int> ans;
        for (auto it : operators) {
            int row = it[0];
            int col = it[1];
```

```
                if (vis[row][col] == 1)
                    ans.push_back(cnt);
                    continue;
                }
                vis[row][col] = 1;
                cnt++;
                // row - 1, col
                // row , col + 1
                // row + 1, col
                // row, col - 1;
                int dr[] = { -1, 0, 1, 0
                int dc[] = {0, 1, 0, -1}
                for (int ind = 0; ind <
                    int adjr = row + dr[
                    int adjc = col + dc[
                    if (isValid(adjr, ad
                        if (vis[adjr][ad
                            int nodeNo =
                            int adjNodeN
                            if (ds.findU
                                cnt--;
                                ds.union
                            }
                        }
                    }
                }
                ans.push_back(cnt);
            }
            return ans;
        }
};



int main() {

    int n = 4, m = 5;
    vector<vector<int>> operators =
        {0, 3}, {1, 3}, {0, 4}, {3,
    };


    Solution obj;
    vector<int> ans = obj.numOfIslan
    for (auto res : ans) {
        cout << res << " ";
    }
    cout << endl;
    return 0;
}
```

Output: 1 1 2 1 1 2 2 2 3 3 1 1

Time Complexity: O(Q*4α) ~ O(Q) where Q = no. of queries. The term 4α is so small that it can be considered constant.

Space Complexity: O(Q) + O(N*M) + O(N*M), where Q = no. of queries, N = total no. of rows, M = total no. of columns. The last two terms are for the parent and the size array used inside the Disjoint set data structure. The first term is to store the answer.

Java Code

```
import java.io.*;
import java.util.*;

//User function Template for Java
class DisjointSet {
    List<Integer> rank = new
ArrayList<>();
    List<Integer> parent = new
ArrayList<>();
    List<Integer> size = new
ArrayList<>();
    public DisjointSet(int n) {
        for (int i = 0; i <= n;
i++) {
            rank.add(0);
            parent.add(i);
            size.add(1);
        }
    }
    public int findUPar(int node)
{
        if (node ==
parent.get(node)) {
            return node;
        }
        int ulp =
findUPar(parent.get(node));
        parent.set(node, ulp);
        return parent.get(node);
    }

    public void unionByRank(int u,
int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v)
return;
        if (rank.get(ulp_u) <
rank.get(ulp_v)) {
            parent.set(ulp_u,
ulp_v);
        } else if (rank.get(ulp_v)
< rank.get(ulp_u)) {
            parent.set(ulp_v,
ulp_u);
        } else {
            parent.set(ulp_v,
ulp_u);
            int rankU =
rank.get(ulp_u);
            rank.set(ulp_u, rankU
+ 1);
        }
    }

    public void unionBySize(int u,
int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v)
return;
        if (size.get(ulp_u) <
size.get(ulp_v)) {
            parent.set(ulp_u,
ulp_v);
            size.set(ulp_v,
size.get(ulp_v) +
size.get(ulp_u));
        } else {
            parent.set(ulp_v,
ulp_u);
            size.set(ulp_u,
size.get(ulp_u) +
size.get(ulp_v));
        }
    }
}
class Solution {
    private boolean isValid(int
adjr, int adjc, int n, int m) {
        return adjr >= 0 && adjr <
n && adjc >= 0 && adjc < m;
    }
    public List<Integer>
numOfIslands(int n, int m, int[][]
operators) {
        DisjointSet ds = new
DisjointSet(n * m);
        int[][] vis = new int[n]
[m];
        int cnt = 0;
        List<Integer> ans = new
ArrayList<>();
        int len =
operators.length;
        for (int i = 0; i < len ;
i++) {
            int row = operators[i]
[0];
            int col = operators[i]
[1];
            if (vis[row][col] ==
1) {
                ans.add(cnt);
                continue;
            }
            vis[row][col] = 1;
            cnt++;
            // row - 1, col
            // row , col + 1
            // row + 1, col
            // row, col - 1;
            int dr[] = { -1, 0, 1,
0};
            int dc[] = {0, 1, 0,
-1};
            for (int ind = 0; ind
< 4; ind++) {
                int adjr = row +
dr[ind];
                int adjc = col +
dc[ind];
                if (isValid(adjr,
adjc, n, m)) {
                    if (vis[adjr]
[adjc] == 1) {
                        int nodeNo
= row * m + col;
                        int
adjNodeNo = adjr * m + adjc;
                        if
(ds.findUPar(nodeNo) !=
ds.findUPar(adjNodeNo)) {
                            cnt--;

ds.unionBySize(nodeNo, adjNodeNo);
                        }
                    }
                }
            }
            ans.add(cnt);
        }
        return ans;
    }

}

class Main {
    public static void main
(String[] args) {
        int n = 4, m = 5;
        int[][] operators = {{0,
0}, {0, 0}, {1, 1}, {1, 0}, {0,
1},
            {0, 3}, {1, 3}, {0,
4}, {3, 2}, {2, 2}, {1, 2}, {0, 2}
        };

        Solution obj = new
Solution();
        List<Integer> ans =
obj.numOfIslands(n, m, operators);

        int sz = ans.size();
        for (int i = 0; i < sz;
i++) {

System.out.print(ans.get(i) + "
");
        }
        System.out.println("");
    }
}
```

Output: 1 1 2 1 1 2 2 2 3 3 1 1

Time Complexity: O(Q*4α) ~ O(Q) where Q = no. of queries. The term 4α is so small that it can be considered constant.

Space Complexity: O(Q) + O(N*M) + O(N*M), where Q = no. of queries, N = total no. of rows, M = total no. of columns. The last two terms are for the parent and the size array used inside the Disjoint set data structure. The first term is to store the answer.

> Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

« Previous Post
Number of Provinces – Disjoint Set: G-48

Next Post »
Making a Large Island – DSU: G-52

Load Comments

Tree Series	Striver's A2Z DSA Sheet
Graph Series	SDE Core Sheet
DP Series	Striver's CP Sheet