HARD

Maximum Connected Group

Intuition

Given : n*n grid, having 1 or 0

We can change at most one cell in a grid 0 to 1, we need to find the largest group of connected 1's after the operation. Adjacent cells are only said to be connected and if they have the same value.

Eg.

1	1	0	0
1	1	0	0
1	1	0	0
0	0	1	1

We can form a single component of size 9 connected by making 3,1 to 1

1	1	0	0
1	1	0	0
1	1	0	0
0	1	1	1

This is the largest connected component that we can get by converting at most 1 0 to 1 in this given configuration

Eg.

1	1	**1**	1	1
1	1	0	1	1
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
1	1	1	1	1

= 9

1	1	0	1	1
1	1	**1**	1	1
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
1	1	1	1	1

= 9

1	1	1	1	1
1	1	0	1	1
0	0	0	0	0
1	0	0	0	0
1	1	1	1	1
1	1	1	1	1

= 11 // we will return this, its not necessary that we need to connect the components we can just simply convert some 0 to 1 to make a single component having largest number of connected 1’s

The graph is dynamically changing therefore we need to use the disjoint set data structure

0…………(n*m)-1 | disjoint set representation

We can do this by using the simple formulae as row*m+col

Right Connected+left connected+1 // this can give us the results

Edge Case : We don't need to connect somewhere only on 4 directions

Approach

- Calculate the dimensions of the grid
- Create a disjoint set of size of grid
- Traverse through all components :
    - Check if the element is 0 :
        - Continue as there is nothing
    - Traverse for adjacent elements :
        - Calculate node number as row*n + col
        - Calculate adjacent node number as adjacent row * n + adjacent col
        - Perform the union operation on node number and adjacent node number
- Create a variable to store the maximum size of islands
- Traverse for all components :
    - Check if element is already 1 ie. we cannot convert it :
        - Continue
    - Create a set to store the components
    - Traverse for adjacent elements :
        - Check if the element is a 1 :
            - Insert the parent of adjacent node number into components
    - Create a variable to store the total size of components
    - Traverse for all components set :
        - Increment total size with individual component size
    - Update the max size variable with max (max variable, total size)
- Traversing for all disjoint set components :
    - Update the max variable with max (max variable, size of parent of the current component)
- Return max variable

Function Code

```
class DisjointSet
```

```
{
    public:
    vector<int> parent;
    vector<int> size;
    DisjointSet(int n)
    {
        size.resize(n+1);
        parent.resize(n+1);
        for(int i=0;i<=n;i++)
        {
            parent[i]=i;
            size[i]=1;
        }
    }
    int findParent(int node)
    {
        if(parent[node]==node)
        return node;
        return parent[node] = findParent(parent[node]);
    }
    void unionbysize(int u,int v)
    {
        int upu = findParent(u);
        int upv = findParent(v);
        if(upu==upv)return;
        if(size[upu]<size[upv])
        {
            parent[upu]=upv;
            size[upv]+=size[upu];
        }
        else
        {
            parent[upv] = upu;
            size[upu] +=size[upv];
        }
    }
};
class Solution {
  public:
    int MaxConnection(vector<vector<int>>& grid) {
        // calculating the dimensions of the grid
        int n = grid.size();
        // traversing for all grid elements and creating a disjoint set
```

```
        DisjointSet ds(n*n);
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                // checking ig the grid element is 0
                if(grid[i][j]==0)continue;
                // adding the element and its adjacent to disjoint set
                int delRow[] = {-1,0,1,0};
                int delCol[] = {0,1,0,-1};
                for(int k=0;k<4;k++)
                {
                    int nrow = i+delRow[k];
                    int ncol = j+delCol[k];
                    // checking for validity of indexes
                    if(nrow<n && ncol<n && nrow>=0 && ncol>=0)
                    {
                        // checking if the element is a 1
                        if(grid[nrow][ncol]==1)
                        {
                            // calculating the node and adjacent node
number

                            int nodenumber = i*n+j;
                            int adjnodenumber = nrow*n+ncol;
                            // performing union
                            ds.unionbysize(nodenumber,adjnodenumber);
                        }
                    }
                }
            }
        }

        int mx = 0;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                // checking if element is already 1 ie. we cant convert
                if(grid[i][j]==1)continue;
                 int delRow[] = {-1,0,1,0};
                int delCol[] = {0,1,0,-1};
                // creating a set to store the components
                set<int> components;
```

```
                for(int k=0;k<4;k++)
                {
                    int nrow = i+delRow[k];
                    int ncol = j+delCol[k];
                    // checking for validity of indexes
                    if(nrow<n && ncol<n && nrow>=0 && ncol>=0)
                    {
                        // checking if element is already 1
                        if(grid[nrow][ncol]==1)
                        {
                            // insert into component
                            components.insert(ds.findParent(nrow*n+ncol));
                        }
                    }
                }
                int sizeTotal = 0;
                for(auto it:components)
                {
                    sizeTotal+=ds.size[it];
                }
                mx = max(mx,sizeTotal+1);
            }
        }
        // traversing to check if we can have better option
        for(int i=0;i<n*n;i++)
        {
            mx = max(mx,ds.size[ds.findParent(i)]);
        }
        // returning the max size
        return mx;

    }
};
```

Time Complexity

O(N^2)