Report on Lab-08

DATA STRUCTURES LAB

Submitted by

Adid-Al-Mahamud Shazid

Student Id: 210042172

Department: CSE

Programme: SWE

Course Title: CSE 4304

Submitted to

Asaduzzaman Herok

Lecturer, Department of CSE

November 5, 2023

A - Back to Underworld

The Vampires and Lykans are fighting each other to death. The war has become so fierce that, none knows who will win. The humans want to know who will survive finally. But humans are afraid of going to the battlefield.

So, they made a plan. They collected the information from the newspapers of Vampires and Lykans. They found the information about all the dual fights. Dual fight means a fight between a Lykan and a Vampire. They know the name of the dual fighters, but don't know which one of them is a Vampire or a Lykan.

So, the humans listed all the rivals. They want to find the maximum possible number of Vampires or Lykans.

Solution

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <queue>
#include <cstring>

using namespace std;

const int SIZE = 20005;

list<int> adj[SIZE];
int color[SIZE];

enum { NOT_VISITED, BLACK, RED };

int main() {
    int tc, t = 0, n, mx, x, y, node;

    cin >> tc;
```

```
    while (t < tc) {
        t++;
        cin >> n;

        memset(color, 0, sizeof color);
        for (int i = 0; i < SIZE; i++) {
            adj[i].clear();
        }

        for (int i = 0; i < n; i++) {
            cin >> x >> y;
            adj[x].push_back(y);
            adj[y].push_back(x);
        }

        mx = 0;

        for (int i = 0; i < SIZE; i++) {
            if (!adj[i].empty() && color[i] == NOT_VISITED) {
                int black = 0, red = 0;
                queue<int> q;
                q.push(i);
                color[i] = BLACK;
                black++;

                while (!q.empty()) {
                    node = q.front();
                    q.pop();

                    for (list<int>::iterator it = adj[node].begin(); it !=
adj[node].end(); it++) {
                        if (color[*it] == NOT_VISITED) {
                            q.push(*it);
                            if (color[node] == BLACK) {
                                color[*it] = RED;
                                red++;
```

```
                        } else {
                            color[*it] = BLACK;
                            black++;
                        }
                    }
                }
            }
            mx += max(red, black);
        }
    }

    cout << "Case " << t << ": " << mx << endl;
}

return 0;
}
```

Analysis and Explanation

Through this task we have to find the maximum number of vampires or Lykans. We would try several test cases for which we're using tc. Then we'll read number of fights using n. The color array is used to determine whether the node is black, red or not visited yet where black denotes vampire and red lykan. Then through x and y involved characters in fight is taken. It starts iterating through all nodes (characters) in the adjacency list to identify connected components. For each unvisited character node, it performs a breadth-first search (BFS) to determine the number of Vampires (black) and Lykans (red) within the connected component. During the BFS, it uses a queue to explore the connected component while alternating between coloring characters as BLACK and RED. The idea is to ensure that neighboring characters (nodes) have different colors, representing opposing sides in the conflict. After processing the connected component using BFS, it calculates the maximum number

of characters that can be on either side (Vampires or Lykans) and adds it to the mx variable. Finally, the code prints the result for the current test case, which is the maximum number of Vampires or Lykans within the given dual fight information.

C - Oil Deposits

The GeoSurvComp geologic survey company is responsible for detecting underground oil deposits. GeoSurvComp works with one large rectangular region of land at a time, and creates a grid that divides the land into numerous square plots. It then analyzes each plot separately, using sensing equipment to determine whether or not the plot contains oil.

A plot containing oil is called a pocket. If two pockets are adjacent, then they are part of the same oil deposit. Oil deposits can be quite large and may contain numerous pockets. Your job is to determine how many different oil deposits are contained in a grid.

Solution

```
#include <iostream>
#include <vector>

using namespace std;

// Define the dimensions of the grid

// Function to perform flood fill
void floodFill(vector<vector<char>>& grid, int x, int y, int rows, int cols,char
targetColor, char replacementColor) {
    // Base cases for recursion
    if (x < 0 || x >= rows || y < 0 || y >= cols)
        return;
    if (grid[x][y] != targetColor)
        return;
```

```
    // Replace the current color with the replacement color
    grid[x][y] = replacementColor;

    // Recursive calls for neighboring cells
    floodFill(grid, x + 1, y, rows,cols, targetColor, replacementColor);
    floodFill(grid, x - 1, y, rows,cols, targetColor, replacementColor);
    floodFill(grid, x, y + 1, rows,cols, targetColor, replacementColor);
    floodFill(grid, x, y - 1, rows,cols, targetColor, replacementColor);
    floodFill(grid, x + 1, y + 1, rows,cols, targetColor, replacementColor);
    floodFill(grid, x - 1, y + 1, rows,cols, targetColor, replacementColor);
    floodFill(grid, x + 1, y - 1, rows,cols, targetColor, replacementColor);
    floodFill(grid, x - 1, y - 1, rows,cols, targetColor, replacementColor);
}

int main() {

    int m,n;

    while (cin >> m >> n && (m != 0 && n != 0))
    {

        vector<vector<char>> grid(m, vector<char>(n));

        for(int i=0 ; i<m ; i++){
            for(int j=0 ; j<n ; j++){
                cin >> grid[i][j];
            }
        }


        int count = 0;
        char targetColor= '@', replacementColor= '*';

        for(int i=0 ; i<m ; i++){
            for(int j=0 ; j<n ; j++){
                if(grid[i][j]==targetColor){
```

```
                floodFill(grid, i, j, m, n, targetColor, replacementColor);
                count++;
                }
            }
        }

        cout<< count << endl;
    }
    return 0;
}
```

Analysis and Explanation

The goal is to detect different oil deposits in a grid of squares and counting them. Here, grid represents a vector where it takes inputs if the elements of the vectors are pocket or not. Pocket is represented by @ and * denotes non-pocket. So, we use a flood-fill algorithm to accomplish the number of oil deposits. We start traversing from the beginning. If we find any @ we call the floodfill function. Thus, the @ with all its adjacent @ are replaced with *. The code then iterates through the grid and, for each cell that has the targetColor, it calls the floodFill function to mark and count the connected oil deposit. After each oil deposit is processed, the count is incremented. Finally, the code prints the value of count, which represents the number of different oil deposits in the grid.

D - Travelling cost

The government of Spoj_land has selected a number of locations in the city for road construction and numbered those locations as 0, 1, 2, 3, ... 500.

Now, they want to construct roads between various pairs of location (say A and B) and have fixed the cost for travelling between those pair of locations from either end as W unit.

Now, Rohit being a curious boy wants to find the minimum cost for travelling from location U (source) to Q number of other locations (destination).

Solution

```
#include<bits/stdc++.h>
typedef long long int ll;
using namespace std;
const int N=505;
const int INF=1e9+10;

vector<pair<int,int>>g[N];
vector<int>dist(N,INF);
vector<int>vis(N,0);

void bfs(int source)
{

    set<pair<int,int>>st;

    st.insert({0,source});
    dist[source]=0;
```

```
    while(st.size()>0)
    {
        auto node=*st.begin();
        int v=node.second;
        int v_dist=node.first;
        st.erase(st.begin());
        if(vis[v]) continue;

        vis[v]=1;
        for(auto child:g[v])
        {
            int child_v=child.first;
            int wt=child.second;

            if(dist[v]+wt <dist[child_v])
            {
                dist[child_v] = dist[v]+wt;
                st.insert({dist[child_v],child_v});
            }
        }
    }
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int m;
    cin>>m;
    for(int i=0;i<m;i++)
    {
        int x,y,wt;
        cin>>x>>y>>wt;
        g[x].push_back({y,wt});
```

```
        g[y].push_back({x,wt});
    }

    int u;
    cin>>u;
    bfs(u);

    int q;
    cin>>q;
    while(q--)
    {
        int v;
        cin>>v;
        if(dist[v]==INF)
        {
            cout<<"NO PATH\n";
        }
        else
        {
            cout<<dist[v]<<"\n";
        }
    }
}
```

Analysis and Explanation

In this problem the government constructs roads between various locations in a city, and the goal is to find the minimum cost for traveling from a source location U to several other destination locations V. The code uses Dijkstra's algorithm to find the minimum cost for each destination. It initializes a graph represented by an adjacency list, where each location is a node, and the edges represent the roads constructed with their associated costs. The bfs function performs Dijkstra's algorithm. It starts by initializing a set st to maintain the minimum distances from the source to each

location. It inserts the source location with a distance of 0 into the set and initializes the dist array with INF (infinity) for all locations. The algorithm iteratively explores the locations. In each iteration, it selects the location with the minimum distance from the set, updates the distance to that location, and explores its neighboring locations. If a shorter path is found to a neighbor, it updates the distance and adds the neighbor to the set for further exploration. For each query, it reads the destination location V and checks if the minimum distance to V from the source U is still INF. If it's INF, it means there is no path from U to V, so it prints "NO PATH." Otherwise, it prints the minimum distance.