DOMAIN WINTER CAMP WORKSHEET

DAY-7 (26/12/2024)

Student Name :- Pratham Kapoor **University ID :- 22BCS10732**

Branch :- B.E. (C.S.E.) **Section/Group :- FL_ 603-A**

Problem-1 :- There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect center node with every other node. You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return center of the given star graph. **(Very Easy)**

Source Code

```
#include<iostream>
#include<vector>
#include<unordered_map>
using namespace std;

int FindCenter(vector<vector<int>>&edges) {
    unordered_map<int,int>Frequency;
    for(auto&edge:edges) {
        Frequency[edge[0]]++;
        Frequency[edge[1]]++;
    }

    for(auto&entry:Frequency) {
        if(entry.second == edges.size()) {
            return entry.first;
        }
    }
```

```
    return -1;
}

int main() {
    int N;
    cout<<"Enter the Number of Edges :- ";
    cin>>N;
    vector<vector<int>>edges(N, vector<int>(2));
    for(int i = 0; i < N; i++) {
        cout<<"Enter the Edge "<<i+1<<" - ";
        cin>>edges[i][0]>>edges[i][1];
    }
    cout<<"\nThe Center of the Star Graph is "<<FindCenter(edges)<<endl;
    return 0;
}
```

Output

```
Enter the Number of Edges :- 3
Enter the Edge 1 - 1 2
Enter the Edge 2 - 2 3
Enter the Edge 3 - 4 2

The Center of the Star Graph is 2
```

Problem-2 :- There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself. You want to determine if there is a valid path that exists from vertex source to vertex destination. Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise. **(Easy)**

Source Code

```
#include<iostream>
#include<vector>
#include<unordered_map>
using namespace std;

bool DFS(int current, int destination, unordered_map<int, vector<int>>& graph, vector<bool>& visited) {
    if (current == destination) return true;
    visited[current] = true;
    for (int neighbor : graph[current]) {
        if (!visited[neighbor]) {
            if (DFS(neighbor, destination, graph, visited)) {
                return true;
            }
        }
    }
    return false;
}

bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
    unordered_map<int, vector<int>> graph;
    vector<bool> visited(n, false);
    for (auto& edge : edges) {
        int u = edge[0], v = edge[1];
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    return DFS(source, destination, graph, visited);
```

```
}

int main() {
    int n, m, source, destination;
    cout << "Enter the Number of Nodes :- ";
    cin >> n;
    cout << "Enter the Number of Edges :- ";
    cin >> m;
    cout << endl;
    vector<vector<int>> edges(m, vector<int>(2));
    for (int i = 0; i < m; i++) {
        cout << "Enter the Edge " << i + 1 << " - ";
        cin >> edges[i][0] >> edges[i][1];
    }
    cout << "\nEnter the Source Node :- ";
    cin >> source;
    cout << "Enter the Destination Node :- ";
    cin >> destination;
    bool result = validPath(n, edges, source, destination);
    if (result)
   {
        cout << "\nTrue, there is a valid path from Node " << source << " to Node
" << destination << endl;
    }

else
{
        cout << "\nFalse, there is no valid path from Node " << source << " to
Node " << destination << endl;
    }
    return 0;
}
```

Output

```
Enter the Number of Nodes :- 3
Enter the Number of Edges :- 3

Enter the Edge 1 - 0 1
Enter the Edge 2 - 1 2
Enter the Edge 3 - 2 0

Enter the Source Node :- 0
Enter the Destination Node :- 2

True, there is a valid path from Node 0 to Node 2
```

Problem-3 :- There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where the prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai. For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1. Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array. **(Medium)**

Source Code

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites)
{
    vector<int> result;
    vector<int> indegree(numCourses, 0);
    vector<vector<int>> graph(numCourses);
```

```
for (const auto& prereq : prerequisites) {
    int ai = prereq[0], bi = prereq[1];
    graph[bi].push_back(ai);
    indegree[ai]++;
}

queue<int> q;

for (int i = 0; i < numCourses; i++) {
    if (indegree[i] == 0) {
        q.push(i);
    }
}

while (!q.empty()) {
    int course = q.front();
    q.pop();
    result.push_back(course);

    for (int neighbor : graph[course]) {
        indegree[neighbor]--;
        if (indegree[neighbor] == 0) {
            q.push(neighbor);
        }
    }
}

if (result.size() == numCourses) {
    return result;
}

return {};
```

```
}

int main() {
    int numCourses;
    cout << "Enter the Number of Courses :- ";
    cin >> numCourses;

    int m;
    cout << "Enter the Number of Pre-Requisites :- ";
    cin >> m;

    vector<vector<int>> prerequisites(m, vector<int>(2));
    for (int i = 0; i < m; i++) {
        cout << "Enter the Pre-Requisite " << i + 1 << " - ";
        cin >> prerequisites[i][0] >> prerequisites[i][1]; }

    vector<int> order = findOrder(numCourses, prerequisites);

    if (order.empty()) {
        cout << "\nIt is Impossible to Finish All Courses" << endl;
    } else {
        cout << "\nThe Order of Courses to Take is [";
        for (int i = 0; i < order.size(); i++) {
            cout << order[i];
            if (i < order.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
    return 0;
}
```

Output

```
Enter the Number of Courses :- 2
Enter the Number of Pre-Requisites :- 1
Enter the Pre-Requisite 1 - 1 0

The Order of Courses to Take is [0, 1]
```

Problem-4 :- You are given an m x n grid where each cell can have one of three values :- 0 representing an empty cell, 1 representing a fresh orange, or 2 representing a rotten orange. Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten. Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1. **(Hard)**

Source Code

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size();
    int n = grid[0].size();
    queue<pair<int, int>> rottenOranges;
    int freshOranges = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 2) {
                rottenOranges.push({i, j});
            } else if (grid[i][j] == 1) {
                freshOranges++;
            }
```

```
        }
    }
    int minutes = 0;
    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    while (!rottenOranges.empty() && freshOranges > 0) {
        int size = rottenOranges.size();
        bool rotOccurred = false;
        for (int i = 0; i < size; i++) {
            auto [x, y] = rottenOranges.front();
            rottenOranges.pop();
            for (auto& dir : directions) {
                int newX = x + dir.first;
                int newY = y + dir.second;

                if (newX >= 0 && newX < m && newY >= 0 && newY < n &&
grid[newX][newY] == 1) {
                    grid[newX][newY] = 2;
                    freshOranges--;
                    rottenOranges.push({newX, newY});
                    rotOccurred = true;
                }
            }
        }
        if (rotOccurred) {
            minutes++; }
    }
    return freshOranges == 0 ? minutes : -1;
}

int main() {
    int m, n;
    cout << "Enter the Number of Rows :- ";
```

```
    cin >> m;
    cout << "Enter the Number of Columns :- ";
    cin >> n;

    vector<vector<int>> grid(m, vector<int>(n));
    cout << "\nEnter the Grid Values :- " << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cout << "Enter the Value for Cell [" << i << "][" << j << "] - ";
            cin >> grid[i][j];
        }
    }
    int result = orangesRotting(grid);
    if (result == -1) {
        cout << "\nIt is Impossible to Rot All the Fresh Oranges" << endl;
    } else {
        cout << "\nThe Minimum Number of Minutes are " << result << endl; }
    return 0; }
```

Output

```
Enter the Number of Rows :- 3
Enter the Number of Columns :- 3

Enter the Grid Values :-
Enter the Value for Cell [0][0] - 2
Enter the Value for Cell [0][1] - 1
Enter the Value for Cell [0][2] - 1
Enter the Value for Cell [1][0] - 1
Enter the Value for Cell [1][1] - 1
Enter the Value for Cell [1][2] - 0
Enter the Value for Cell [2][0] - 0
Enter the Value for Cell [2][1] - 1
Enter the Value for Cell [2][2] - 1

The Minimum Number of Minutes are 4
```

Problem-5 :- In this problem, a tree is an undirected graph that is connected and has no cycles. You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two different vertices chosen from 1 to n and was not an edge that already existed. The graph is represented as an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in graph. Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input. **(Very Hard)**

Source Code

```
#include <iostream>
#include <vector>
using namespace std;

class UnionFind {
public:
    vector<int> parent;

    UnionFind(int n) {
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
```

```
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            parent[rootX] = rootY;
        }
    }
};

vector<int> findRedundantConnection(vector<vector<int>>& edges) {
    int n = edges.size();
    UnionFind uf(n + 1);

    vector<int> redundantEdge;

    for (auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];

        if (uf.find(u) == uf.find(v)) {
            redundantEdge = edge;
        } else {
            uf.unionSets(u, v);
        }
    }

    return redundantEdge;
}

int main() {
    int n, m;
```

```
    cout << "Enter the Number of Edges :- ";
    cin >> m;
    cout << endl;
    vector<vector<int>> edges(m, vector<int>(2));

    for (int i = 0; i < m; i++) {
        cout << "Enter the Edge " << i + 1 << " :- ";
        cin >> edges[i][0] >> edges[i][1];
    }

    vector<int> result = findRedundantConnection(edges);
    cout << "\nThe Redundant Edge is [" << result[0] << "," << result[1] << "]"
<< endl;

    return 0;
}
```

Output

```
Enter the Number of Edges :- 3

Enter the Edge 1 :- 1 2
Enter the Edge 2 :- 1 3
Enter the Edge 3 :- 2 3

The Redundant Edge is [2,3]
```