IT251 – DATA STRUCTURES & ALGORITHMS II

ASSIGNMENT 1

Name: **Sachin Prasanna**
Roll No.: **211IT058**

1)

Code Written:

```
#include <iostream>
#include <unordered_map>
#include <queue>
#include <vector>
#include <list>
#include <set>

using namespace std;

void bfs(unordered_map <int, list <int>> &adjList, unordered_map <int, bool>
&visited,  vector <int> &ans, int node){

    queue <int> q;
    q.push(node);
    visited[node] = true;

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

        ans.push_back(frontNode);
```

```
        for(auto i : adjList[frontNode]){
            if(!visited[i]){
                q.push(i);
                visited[i] = true;
            }
        }
    }
}

void printAdjacencyList(unordered_map <int, list<int>> &adjList){

    for(auto i : adjList){
        cout<<i.first<<" -> ";
        for(auto j : i.second){
            cout<<j<<", ";
        }
        cout<<"\n";
    }
}

vector <vector <int>> BFS (set <int> &uniqueNodes, vector <pair <int, int>>
edges){

    vector <int> ans;
    unordered_map <int, list <int>> adjList;
    unordered_map <int, bool> visited;

    vector <vector <int>> answer;

    for(int i = 0 ; i < edges.size() ; i++){
        int u = edges[i].first;
        int v = edges[i].second;

        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    cout<<"\nAdjacency List for the given edges is as follow:\n";
    printAdjacencyList(adjList);
    cout<<"\n\n";

    int component = 1;

    set<int>::iterator itr;
```

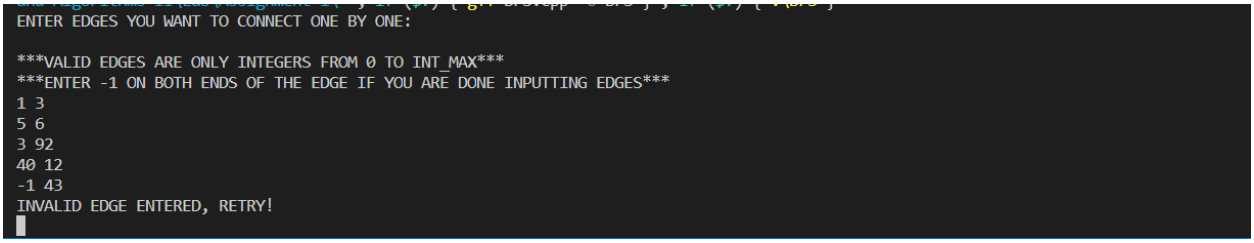
```
    cout<<"COMPONENT WISE BFS:\n\n";

    for (itr = uniqueNodes.begin(); itr != uniqueNodes.end(); itr++){
        if(!visited[*itr]){
            vector <int> ans;
            cout<<"Component Number "<<component++<<": ";
            bfs(adjList, visited, ans, *itr);

            for(int i = 0 ; i < ans.size() ; i++){
                cout<<ans[i]<<"  ";
            }
            cout<<endl;
            answer.push_back(ans);
        }
    }

    return answer;
}

bool isPresent(int edgeInitial, int edgeFinal, vector <pair <int, int>> &edges){

    for(int i = 0 ; i < edges.size() ; i++){
        if(edgeInitial == edges[i].first && edgeFinal == edges[i].second) return
true;
    }
    return false;
}


int main(){

    //Carrying out BFS to a disconnected undirected graph.

    vector <pair <int, int>> edges;

    cout<<"ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:\n\n";
    cout<<"***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***\n";
    cout<<"***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING
EDGES***\n";

    int edgeInitial;
    int edgeFinal;

    cin>>edgeInitial>>edgeFinal;
```

```
    while(edgeInitial != -1 || edgeFinal != -1){

        if(edgeInitial <= -1 || edgeFinal <= -1){
            cout<<"INVALID EDGE ENTERED, RETRY!\n";
            cin>>edgeInitial>>edgeFinal;
            continue;
        }

        else if(isPresent(edgeInitial, edgeFinal, edges)){
            cout<<"EDGE IS ALREADY PRESENT!, RETRY!\n";
            cin>>edgeInitial>>edgeFinal;
            continue;
        }

        pair <int, int> edge;
        edge.first = edgeInitial;
        edge.second = edgeFinal;

        edges.push_back(edge);

        cin>>edgeInitial>>edgeFinal;
    }

    set <int> uniqueNodes;

    for(int i = 0 ; i < edges.size() ; i++){
        uniqueNodes.insert(edges[i].first);
        uniqueNodes.insert(edges[i].second);
    }

    vector <vector <int>> bfsArray = BFS(uniqueNodes, edges);

    cout<<"\nFINAL BFS: ";

    for (int i = 0; i < bfsArray.size(); i++) {
        for (int j = 0; j < bfsArray[i].size(); j++) {
            cout << bfsArray[i][j] <<" ";
        }
    }
    cout<<endl<<endl;

    return 0;
}
```

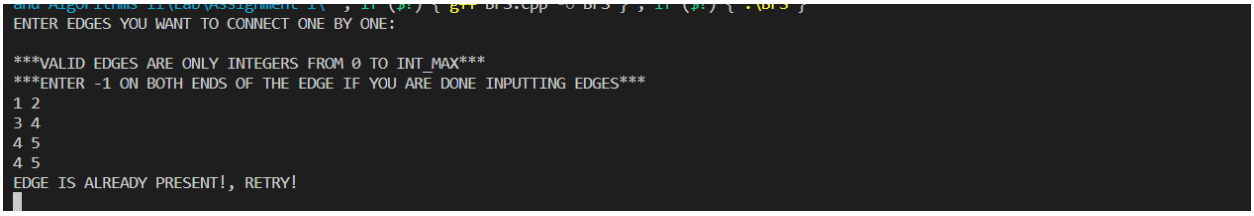
Edge Cases Handled:

1) The set data structure is used in order to keep track of all unique nodes from the edges entered in by the user. This is helpful in finding the Breadth First Search of the graph in case the graph is disconnected. In that case, Breadth First Search is carried out for each component.

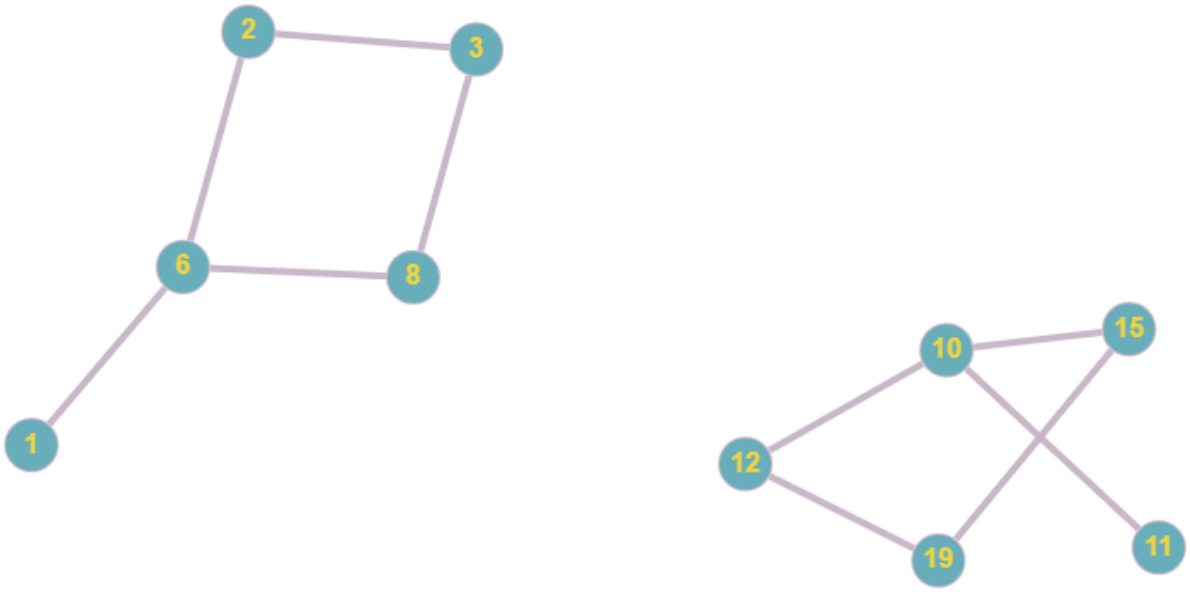
2) Any inputted edge which has either end or start vertex as below -1 will not be accepted by the program. Also, -1 as both end vertices is the marks the end of the edge acceptance phase and thereafter, Breadth First Search is performed.



3) Same edge inputted again will be rejected by the program and the user will be prompted to input a new edge again.



Sample graph used for BFS:



Output:

```
ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:

***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***
***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING EDGES***
1 6
6 2
2 3
3 8
6 8
10 15
10 11
10 12
15 19
19 12
-1 -1

Adjacency List for the given edges is as follow:
12 -> 10, 19,
11 -> 10,
10 -> 15, 11, 12,
8 -> 3, 6,
3 -> 2, 8,
15 -> 10, 19,
2 -> 6, 3,
19 -> 15, 12,
6 -> 1, 2, 8,
1 -> 6,


COMPONENT WISE BFS:

Component Number 1: 1  6  2  8  3
Component Number 2: 10  15  11  12  19

FINAL BFS: 1 6 2 8 3 10 15 11 12 19
```

2)

Code Written:

```cpp
#include <iostream>
#include <unordered_map>
#include <queue>
#include <vector>
#include <list>
#include <set>

using namespace std;

// CHECK FOR CYCLIC LOOP
bool checkCycleDFS(int node, unordered_map<int, bool> &visited,
unordered_map<int, bool> &dfsVisited, unordered_map<int, list<int>> &adjList)
{

    visited[node] = true;
```

```
    dfsVisited[node] = true;

    for (auto neighbour : adjList[node])
    {
        if (!visited[neighbour])
        {
            bool cycleDetected = checkCycleDFS(neighbour, visited, dfsVisited,
adjList);

            if (cycleDetected)
                return true;
        }
        else if (dfsVisited[neighbour])
        {
            return true;
        }
    }

    dfsVisited[node] = false;
    return false;
}

bool detectCycleDirected(set<int> &uniqueNodes, vector<pair<int, int>> &edges)
{

    unordered_map<int, list<int>> adjList;

    for (int i = 0; i < edges.size(); i++)
    {
        int u = edges[i].first;
        int v = edges[i].second;

        adjList[u].push_back(v);
    }

    unordered_map<int, bool> visited;
    unordered_map<int, bool> dfsVisited;

    set<int>::iterator itr;

    for (itr = uniqueNodes.begin(); itr != uniqueNodes.end(); itr++)
    {
        if (!visited[*itr])
        {
            bool cycleDetect = checkCycleDFS(*itr, visited, dfsVisited, adjList);
```

```
            if (cycleDetect)
                return true;
        }
    }

    return false;
}

// EDGE CLASSIFICATION

int t;
void classifyEdgeDFS(int node, vector<int> &startTime, vector<int> &endTime,
vector<bool> &visited, vector<int> &traversal, unordered_map<int, list<int>>
adjList)
{
    visited[node] = true;
    traversal.push_back(node);
    startTime[node] = t++;
    for (int neighbour : adjList[node])
    {
        if (!visited[neighbour])
        {
            cout << "Tree Edge: " << node << "->" << neighbour << endl;
            classifyEdgeDFS(neighbour, startTime, endTime, visited, traversal,
adjList);
        }
        else
        {
            if (startTime[node] < startTime[neighbour] && endTime[node] >
endTime[neighbour])
            {
                cout << "Forward Edge: " << node << "->" << neighbour << endl;
            }
            else
            {
                cout << "Cross Edge: " << node << "->" << neighbour << endl;
            }
        }
    }
    endTime[node] = t++;
}

void classifyEdges(set<int> uniqueNodes, unordered_map<int, list<int>> adjList)
{
```

```
    vector<int> startTime(uniqueNodes.size(), 0);
    vector<int> endTime(uniqueNodes.size(), 0);
    vector<bool> visited(uniqueNodes.size(), false);
    vector<int> traversal;

    set<int>::iterator itr;

    for (itr = uniqueNodes.begin(); itr != uniqueNodes.end(); itr++)
    {
        if (!visited[*itr])
        {
            classifyEdgeDFS(*itr, startTime, endTime, visited, traversal,
adjList);
        }
    }
}

// DEPTH FIRST SEARCH
void dfs(int node, unordered_map<int, bool> &visited, unordered_map<int,
list<int>> &adjList, vector<int> &ans)
{

    ans.push_back(node);
    visited[node] = true;

    for (auto i : adjList[node])
    {
        if (!visited[i])
        {
            dfs(i, visited, adjList, ans);
        }
    }
}

void printAdjacencyList(unordered_map<int, list<int>> &adjList)
{

    for (auto i : adjList)
    {
        cout << i.first << " -> ";
        for (auto j : i.second)
        {
            cout << j << ", ";
        }
        cout << "\n";
```

```
    }
}

vector<vector<int>> depthFirstSearch(set<int> &uniqueNodes, vector<pair<int,
int>> edges)
{
    unordered_map<int, list<int>> adjList;

    for (int i = 0; i < edges.size(); i++)
    {
        int u = edges[i].first;
        int v = edges[i].second;

        adjList[u].push_back(v);
    }

    vector<vector<int>> answer;

    unordered_map<int, bool> visited;

    cout << "\nAdjacency List for the given edges is as follow:\n";
    printAdjacencyList(adjList);
    cout << "\n\n";

    int component = 1;

    cout << "EDGE CLASSIFICATION:\n";

    classifyEdges(uniqueNodes, adjList);
    set<int>::iterator itr;

    cout << "\nCOMPONENT WISE DFS:\n\n";

    for (itr = uniqueNodes.begin(); itr != uniqueNodes.end(); itr++)
    {
        if (!visited[*itr])
        {
            vector<int> ans;
            cout << "Component Number " << component++ << ": ";
            dfs(*itr, visited, adjList, ans);

            for (int i = 0; i < ans.size(); i++)
            {
                cout << ans[i] << "  ";
            }
```

```
            cout << endl;
            answer.push_back(ans);
        }
    }

    return answer;
}

bool isPresent(int edgeInitial, int edgeFinal, vector<pair<int, int>> &edges)
{

    for (int i = 0; i < edges.size(); i++)
    {
        if (edgeInitial == edges[i].first && edgeFinal == edges[i].second)
            return true;
    }
    return false;
}

int main()
{

    // Carrying out DFS to a Directed Acyclic Graph (DAG)

    vector<pair<int, int>> edges;

    cout << "ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:\n\n";
    cout << "***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***\n";
    cout << "***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING
EDGES***\n";

    int edgeInitial;
    int edgeFinal;

    cin >> edgeInitial >> edgeFinal;

    set<int> uniqueNodes;

    while (edgeInitial != -1 || edgeFinal != -1)
    {

        if (edgeInitial <= -1 || edgeFinal <= -1)
        {
            cout << "INVALID EDGE ENTERED, RETRY!\n";
            cin >> edgeInitial >> edgeFinal;
```

```
            continue;
        }

        else if (isPresent(edgeInitial, edgeFinal, edges))
        {
            cout << "EDGE IS ALREADY PRESENT!, RETRY!\n";
            cin >> edgeInitial >> edgeFinal;
            continue;
        }

        pair<int, int> edge;
        edge.first = edgeInitial;
        edge.second = edgeFinal;

        edges.push_back(edge);

        bool detectCycle = detectCycleDirected(uniqueNodes, edges);

        if (detectCycle)
        {
            cout << "THE EDGE YOU INPUTTED WILL CAUSE A CYCLE, SO IT CANNOT BE
INPUTTED.\nPLEASE RE ENTER AN EDGE:\n";
            edges.pop_back();
        }

        uniqueNodes.insert(edge.first);
        uniqueNodes.insert(edge.second);

        cin >> edgeInitial >> edgeFinal;
    }

    vector<vector<int>> dfsArray = depthFirstSearch(uniqueNodes, edges);

    cout << "\nFINAL DFS: ";

    for (int i = 0; i < dfsArray.size(); i++)
    {
        for (int j = 0; j < dfsArray[i].size(); j++)
        {
            cout << dfsArray[i][j] << " ";
        }
    }

    cout << endl;
```

```
    return 0;
}
```

Edge Cases Handled:

1) The set data structure is used in order to keep track of all unique nodes from the edges entered in by the user. This is helpful in finding the Depth First Search of the graph in case the graph is disconnected. In that case, Depth First Search is carried out for each component.

2) Any inputted edge which has either end or start vertex as below -1 will not be accepted by the program. Also, -1 as both end vertices is marks the end of the edge acceptance phase and thereafter, Depth First Search is performed.

```
ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:

***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***
***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING EDGES***
1 2
2 4
-4 -32
INVALID EDGE ENTERED, RETRY!
```

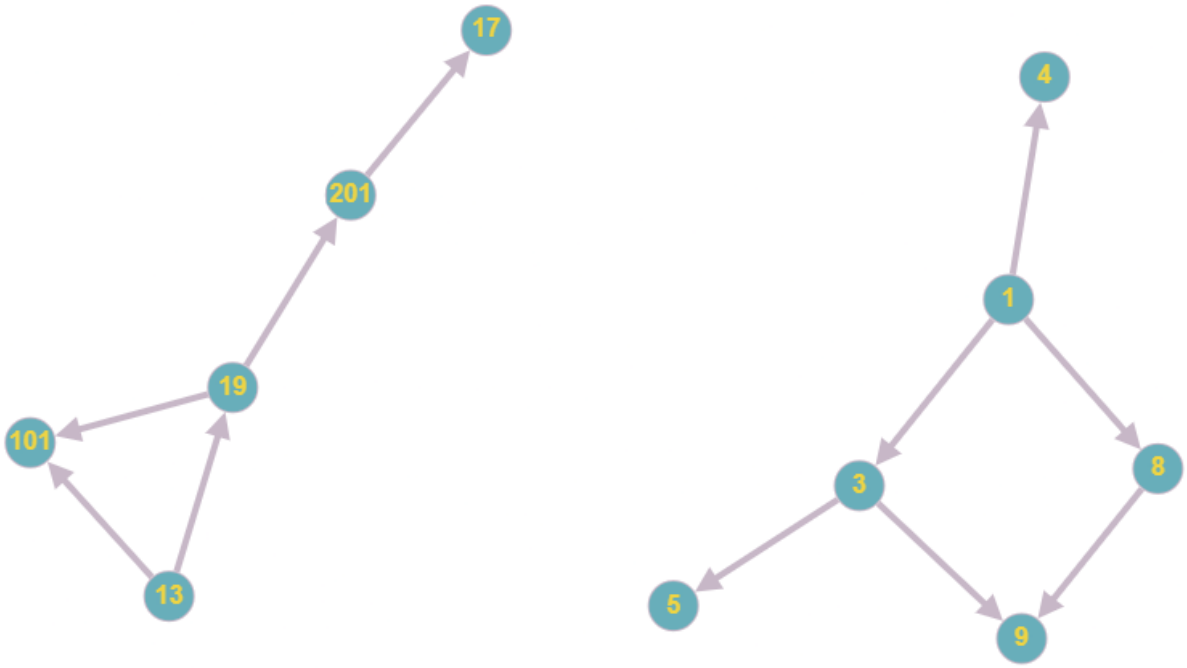
3) As the question demands an **ACYCLIC** graph, whenever the user inputs an edge that adds a cycle in the graph, is not added to the graph and informed to the user. Re input of another edge is therefore required.

```
ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:

***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***
***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING EDGES***
1 2
2 3
3 4
4 1
THE EDGE YOU INPUTTED WILL CAUSE A CYCLE, SO IT CANNOT BE INPUTTED.
PLEASE RE ENTER AN EDGE:
```

4) Same edge inputted again will be rejected by the program and the user will be prompted to input a new edge again.

```
ENTER EDGES YOU WANT TO CONNECT ONE BY ONE:

***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***
***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING EDGES***
1 2
3 4
5 6
4 5
4 5
EDGE IS ALREADY PRESENT!, RETRY!
```

Sample graph used for DFS:



Output:

```
***VALID EDGES ARE ONLY INTEGERS FROM 0 TO INT_MAX***
***ENTER -1 ON BOTH ENDS OF THE EDGE IF YOU ARE DONE INPUTTING EDGES***
1 4
1 8
1 3
8 9
3 9
3 5
13 19
13 101
19 101
19 201
201 17
-1 -1

Adjacency List for the given edges is as follow:
201 -> 17,
19 -> 101, 201,
13 -> 19, 101,
3 -> 9, 5,
8 -> 9,
1 -> 4, 8, 3,


EDGE CLASSIFICATION:
Tree Edge: 1->4
Tree Edge: 1->8
Tree Edge: 8->9
Tree Edge: 1->3
Cross Edge: 3->9
Tree Edge: 3->5
Tree Edge: 13->19
Forward Edge: 19->101
Forward Edge: 19->201
Cross Edge: 13->101

COMPONENT WISE DFS:

Component Number 1: 1  4  8  9  3  5
Component Number 2: 13  19  101  201  17

FINAL DFS: 1 4 8 9 3 5 13 19 101 201 17
PS C:\Users\91900\Desktop\Computer\Semester 4\IT251 - Data Structures and Algorithms II\Lab\Assignment 1>
```

****