Date: 28/01/2021

Practical 2

2CSDE56 - Graph Theory

Name: Shrey Viradiya

Roll No: 18BCE259

Aim:

Write a program to check whether two graphs are isomorphic to each other or not.

Code:

Prac2_Isomorphism.cpp

```cpp
// Implement the CPP solution for checking isomorphic graphs

// Checking isomorphism is not solved
// In this practical only necessary conditions are checked
// and and additional check is performed which fails in
// some cases like Euler and Kirchhoff

#include "UndirectedGraphMatrix.h"

int main(){

    using namespace std;

    UndirectedGraphMatrix graphA("Euler", 10);
    UndirectedGraphMatrix graphB("Kirkman", 3);
    UndirectedGraphMatrix graphC("Kirchhoff", 10);
    UndirectedGraphMatrix graphD("Tesla", 6);
    UndirectedGraphMatrix graphE("Edison", 6);
    UndirectedGraphMatrix graphF("NG", 7);
    UndirectedGraphMatrix graphG("GS", 7);

    // adding some edges in graph A
    graphA.addEdge(0,1);
    graphA.addEdge(1,2);
    graphA.addEdge(2,3);
    graphA.addEdge(3,9);
    graphA.addEdge(0,8);
    graphA.addEdge(1,8);
    graphA.addEdge(2,7);
    graphA.addEdge(7,6);
    graphA.addEdge(5,7);
    graphA.addEdge(3,4);

    // adding some edges in graph B
    graphB.addEdge(0,1);
    graphB.addEdge(0,2);

    // adding some edges in graph C
    graphC.addEdge(0,1);
    graphC.addEdge(1,2);
    graphC.addEdge(2,3);
    graphC.addEdge(3,6);
    graphC.addEdge(3,9);
    graphC.addEdge(9,8);
```

```
    graphC.addEdge(7,9);
    graphC.addEdge(0,8);
    graphC.addEdge(1,5);
    graphC.addEdge(2,4);

    // adding some edges in graph D
    graphD.addEdge(0,1);
    graphD.addEdge(1,2);
    graphD.addEdge(2,3);
    graphD.addEdge(3,4);
    graphD.addEdge(2,5);

    // adding some edges in graph E
    graphE.addEdge(0,1);
    graphE.addEdge(1,2);
    graphE.addEdge(2,3);
    graphE.addEdge(3,4);
    graphE.addEdge(3,5);

    // adding some edges in graph F
    graphF.addEdge(0,1);
    graphF.addEdge(1,2);
    graphF.addEdge(2,3);
    graphF.addEdge(3,4);
    graphF.addEdge(1,5);
    graphF.addEdge(3,6);

    // adding some edges in graph G
    graphG.addEdge(0,1);
    graphG.addEdge(1,2);
    graphG.addEdge(2,3);
    graphG.addEdge(3,4);
    graphG.addEdge(1,5);
    graphG.addEdge(2,6);

    //printing the graphs
    // graphA.displayGraph();
    // graphB.displayGraph();
    // graphC.displayGraph();

    // Isomorphism checking
    cout << "--------" << endl;
    cout << "Isomorphism of " << graphA.getName() << " and " << graphB.getName
() << " : "<< UndirectedGraphMatrix::CheckIsomorphism(graphA, graphB) << endl;
    cout << "Isomorphism of " << graphA.getName() << " and " << graphC.getName
() << " : "<< UndirectedGraphMatrix::CheckIsomorphism(graphA, graphC) << endl;
    cout << "Isomorphism of " << graphD.getName() << " and " << graphE.getName
() << " : "<< UndirectedGraphMatrix::CheckIsomorphism(graphD, graphE) << endl;
```

```
    cout << "Isomorphism of " << graphF.getName() << " and " << graphG.getName
() << " : "<< UndirectedGraphMatrix::CheckIsomorphism(graphF, graphG) << endl;
    cout << "--------" << endl;

    return 0;
}
```

UndirectedGraphMatrix.h

```
#pragma once

#include<iostream>
#include<algorithm>
#include<map>
#include<cstring>

class UndirectedGraphMatrix
{
private:
    int noVertices, edges;
    char name[50];
    int **graph;
    int *degrees;

public:
    UndirectedGraphMatrix(const char n[], int V);
    void addEdge(int src, int dest);
    void deleteEdge(int src, int dest);
    int isEdge(int src, int dest);
    int getNoVertices();
    int getNoEdges();
    int getDegree(int src);
    int * getSortedDegrees();
    char * getName();
    void displayGraph();
    ~UndirectedGraphMatrix();
    static bool CheckIsomorphism(UndirectedGraphMatrix &graphA, UndirectedGrap
hMatrix &graphB);
};

UndirectedGraphMatrix::UndirectedGraphMatrix(const char n[50], int V){
    noVertices = V;
    strcpy_s(name, n);
    edges = 0;
    graph = new int *[noVertices];
    degrees = new int [noVertices] {0};
    for (int i = 0; i < noVertices; i++)
    {
```

```
        graph[i] = new int[noVertices] {0};
    }

    using namespace std;
    cout << "Graph Created: " << name << endl;
}

UndirectedGraphMatrix::~UndirectedGraphMatrix(){
    for (int i = 0; i < noVertices; i++)
    {
        delete[]graph[i];
    }
    delete[]graph;
    delete[]degrees;

    using namespace std;
    cout << "Memory released of the graph " << name << endl;
}

void UndirectedGraphMatrix::addEdge(int src, int dest){
    if(
        (src >= noVertices)
        ||
        (dest >= noVertices)
    ){
        return;
    }

    if (
        // (edges < (noVertices*(noVertices - 1)/2))
        // &&
        (graph[src][dest] == 0)
     )
    {
        ++edges;
        graph[src][dest] = 1;
        graph[dest][src] = 1;
        ++degrees[src];
        ++degrees[dest];
    }
}

void UndirectedGraphMatrix::deleteEdge(int src, int dest){

    if (
        // (edges > 0)
        // &&
        (graph[src][dest] == 1)
```

```
    )
    {
        --edges;
        graph[src][dest] = 0;
        graph[dest][src] = 0;
        --degrees[src];
        --degrees[dest];
    }
}

int UndirectedGraphMatrix::getNoVertices(){
    return noVertices;
}

int UndirectedGraphMatrix::getNoEdges(){
    return edges;
}

int UndirectedGraphMatrix::isEdge(int src, int dest){
    return graph[src][dest];
}

int UndirectedGraphMatrix::getDegree(int src){
    return degrees[src];
}

char * UndirectedGraphMatrix::getName(){
    char* arr = new char[50];
    strcpy(arr, name);
    return arr;
}

int * UndirectedGraphMatrix::getSortedDegrees(){
    int * sortedDegrees = new int[noVertices];
    std::copy(degrees, degrees+noVertices, sortedDegrees);
    std::sort(sortedDegrees, sortedDegrees+noVertices);
    return sortedDegrees;
}

void UndirectedGraphMatrix::displayGraph(){
    using namespace std;

    cout << "\nGraph:" << name << endl;
    cout << "=================" << endl;
    cout << "No of Vertices: " << noVertices << endl;
    cout << "No of Edges: " << edges << endl;
    cout << "=================" << endl;
    for (auto i = 0; i < noVertices; i++)
```

```
    {
        for (auto j = 0; j < noVertices; j++)
        {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

bool UndirectedGraphMatrix::CheckIsomorphism(UndirectedGraphMatrix &graphA, Un
directedGraphMatrix &graphB){

    // simple check for number of vertices and no if edges
    if(
        (graphA.getNoEdges() != graphB.getNoEdges())
        ||
        (graphA.getNoVertices() != graphB.getNoVertices())
    ){
        return false;
    }

    // next check for number of same degree vertices
    int *graphAdegrees = graphA.getSortedDegrees();
    int *graphBdegrees = graphB.getSortedDegrees();

    for (int i = 0; i < graphA.getNoVertices(); i++)
    {
        if (graphAdegrees[i] != graphBdegrees[i])
        {
            return false;
        }
    }

    // edge correspondence remaining
    std::map<std::pair<int, int>, int> EdgeDegreeData;
    for (int i = 0; i < graphA.getNoVertices(); i++)
    {
        for(int j = i; j < graphA.getNoVertices(); j++){
            if (graphA.isEdge(i,j))
            {
                std::pair<int, int> key;
                if ( graphA.getDegree(i) <= graphA.getDegree(j) )
                {
                    key = {graphA.getDegree(i), graphA.getDegree(j)};
                }
                else{
                    key = {graphA.getDegree(j), graphA.getDegree(i)};
```

```
                }

                auto it = EdgeDegreeData.find(key);

                if(it == EdgeDegreeData.end())
                    EdgeDegreeData[key] = 1;
                else
                    EdgeDegreeData[key] += 1;
            }
        }
    }

    // for(auto it = EdgeDegreeData.cbegin(); it != EdgeDegreeData.cend(); ++i
t)
    //     {
    //         std::cout << it->first.first << ", " << it->first.second << "-
>" << it->second << "\n";
    //     }

    for (int i = 0; i < graphB.getNoVertices(); i++)
    {
        for(int j = i; j < graphB.getNoVertices(); j++){
            if (graphB.isEdge(i,j))
            {
                std::pair<int, int> key;
                if ( graphB.getDegree(i) <= graphB.getDegree(j) )
                {
                    key = {graphB.getDegree(i), graphB.getDegree(j)};
                }
                else{
                    key = {graphB.getDegree(j), graphB.getDegree(i)};
                }

                auto it = EdgeDegreeData.find(key);

                if(it == EdgeDegreeData.end())
                    return false;
                else
                    EdgeDegreeData[key] -= 1;

                if (EdgeDegreeData[key] < 0) return false;
            }
        }
    }

    return true;
}
```

Snapshot of the output: