Date: 08/05/2021

Practical 8:

2CSDE56 - Graph Theory

Name: Shrey Viradiya

Roll No: 18BCE259

Aim:

Write a program to check whether the graph is planar or not.

Code:

Prac9_PlanarityTest.cpp

```cpp
#include <iostream>
#include "UndirectedGraph.h"

int main(){
    using namespace std;

    UndirectedGraph K(4);

    K.addEdge(0,1);
    K.addEdge(0,2);
    K.addEdge(0,3);

    cout << "Plannarity: " << K.isPlanner() << endl;

    UndirectedGraph K2(5);

    K2.addEdge(0,1);
    K2.addEdge(0,2);
    K2.addEdge(0,3);
    K2.addEdge(0,4);

    K2.addEdge(1,2);
    K2.addEdge(1,3);
    K2.addEdge(1,4);

    K2.addEdge(2,3);
    K2.addEdge(2,4);

    K2.addEdge(3,4);

    cout << "Plannarity: " << K2.isPlanner() << endl;

    return 0;
}
```

UndirectedGraph.h

```cpp
#pragma once
#include<vector>
#include<set>
#include<unordered_set>
#include<iterator>
#include<iostream>
#include<algorithm>
```

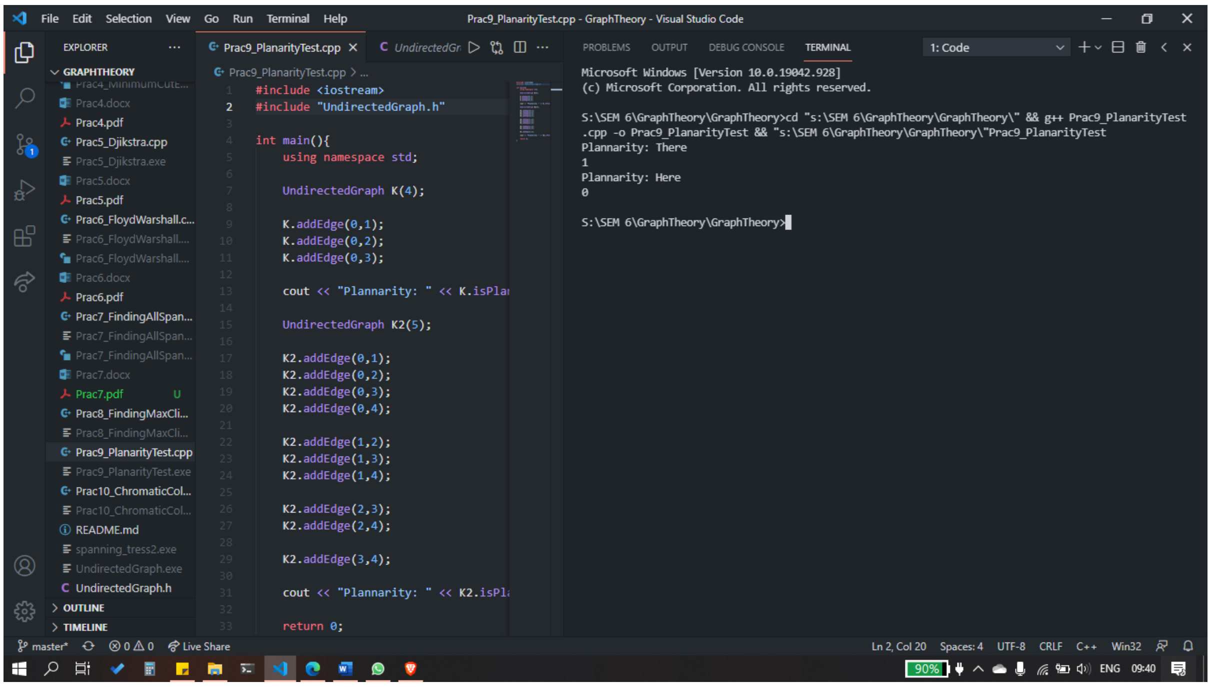
```cpp
using namespace std;

class UndirectedGraph
{
public:
    static int count;
    int id;
    int numVertices;
    int numEdges;
    vector <set <int>> graph;
    UndirectedGraph(int V);
    void addEdge(int src, int dest);
    void removeEdge(int src, int dest);
    int containsCycle();
    void displayGraph() const;
    int isEdge(int src, int dest);
    static int getNextID();
    static UndirectedGraph Union(UndirectedGraph graphA, UndirectedGraph graph
B);
    static UndirectedGraph Intersection(UndirectedGraph graphA, UndirectedGrap
h graphB);
    static UndirectedGraph Subtraction(UndirectedGraph graphA, UndirectedGraph
 graphB);
    static UndirectedGraph RingSum(UndirectedGraph graphA, UndirectedGraph gra
phB);
    UndirectedGraph Complement();
    bool isPlanner();

    void CyclicExchange();

    bool operator<(const UndirectedGraph& t) const
    {
        return id < t.id;
    }
};

int UndirectedGraph::count{ 0 };

int UndirectedGraph::getNextID() {
    return ++count;
}

UndirectedGraph::UndirectedGraph(int V) {
    id = getNextID();
    numVertices = V;
    numEdges = 0;
    for (int i = 0; i < numVertices; i++)
    {
        graph.push_back(set<int> {});
    }
}

void UndirectedGraph::displayGraph() const {
    int node = 0;
    for (auto i = graph.begin(); i != graph.end(); i++)
    {
        cout << (node++) << " -> ";
        for (auto j = (*i).begin(); j != (*i).end(); j++)
        {
            cout << *j << " -> ";
        }
        cout << endl;
    }
}

void UndirectedGraph::addEdge(int src, int dest) {
    auto it = find (graph[src].begin(), graph[src].end(), dest);
    if (it == graph[src].end()) numEdges++;
    graph[src].insert(dest);
    graph[dest].insert(src);
}

void UndirectedGraph::removeEdge(int src, int dest) {
    auto it = find (graph[src].begin(), graph[src].end(), dest);
    if (it != graph[src].end()) numEdges--;
    graph[src].erase(dest);
    graph[dest].erase(src);

}

int UndirectedGraph::containsCycle() {
    int* visited = new int[numVertices]();

    visited[0] = 1;
    for (int i = 0; i < numVertices; i++)
    {
        for (auto&& j : graph[i])
        {
            if (j > i && visited[j]) return 1;
            else visited[j] = 1;
        }
    }

    delete[] visited;
    return 0;
}
```

```cpp
int UndirectedGraph::isEdge(int src, int dest) {
    return (graph[src].find(dest) != graph[src].end());
}

UndirectedGraph UndirectedGraph::Union(UndirectedGraph graphA, UndirectedGraph
 graphB) {
    int V = max(graphA.numVertices, graphB.numVertices);
    UndirectedGraph union_graph(V);

    for (int i = 0; i < V; i++)
    {
        set_union(
            graphA.graph[i].begin(),
            graphA.graph[i].end(),
            graphB.graph[i].begin(),
            graphB.graph[i].end(),
            inserter(union_graph.graph[i], union_graph.graph[i].begin())
        );
    }

    return union_graph;
}

UndirectedGraph UndirectedGraph::Intersection(UndirectedGraph graphA, Undirect
edGraph graphB) {
    int V = max(graphA.numVertices, graphB.numVertices);
    UndirectedGraph intersection_graph(V);

    for (int i = 0; i < V; i++)
    {
        set_intersection(
            graphA.graph[i].begin(),
            graphA.graph[i].end(),
            graphB.graph[i].begin(),
            graphB.graph[i].end(),
            inserter(intersection_graph.graph[i], intersection_graph.graph[i].
begin())
        );
    }

    return intersection_graph;
}

UndirectedGraph UndirectedGraph::Subtraction(UndirectedGraph graphA, Undirecte
dGraph graphB) {
    int V = max(graphA.numVertices, graphB.numVertices);
    UndirectedGraph subtracted_graph(V);
    for (int i = 0; i < V; i++)
    {
        set_difference(
            graphA.graph[i].begin(),
            graphA.graph[i].end(),
            graphB.graph[i].begin(),
            graphB.graph[i].end(),
            inserter(subtracted_graph.graph[i], subtracted_graph.graph[i].begi
n())
        );
    }

    return subtracted_graph;
}

UndirectedGraph UndirectedGraph::RingSum(UndirectedGraph graphA, UndirectedGra
ph graphB) {
    return Subtraction(
        Union(graphA, graphB),
        Intersection(graphA, graphB)
    );
}

UndirectedGraph UndirectedGraph::Complement() {
    UndirectedGraph complement_graph(numVertices);

    set <int> allVer;
    for (int i = 0; i < numVertices; i++)
    {
        allVer.insert(i);
    }

    for (int i = 0; i < numVertices; i++)
    {
        allVer.erase(i);
        set_difference(
            allVer.begin(),
            allVer.end(),
            graph[i].begin(),
            graph[i].end(),
            inserter(complement_graph.graph[i], complement_graph.graph[i].begi
n())
        );
        allVer.insert(i);
    }

    return complement_graph;
```

```
}

void UndirectedGraph::CyclicExchange() {
    using namespace std;
    // creating a tree
    UndirectedGraph tree(numVertices);
    int n = 0;
    while (n < numVertices - 1)
    {
        for (int i = 0; i < numVertices; i++)
        {
            for (auto&& j : graph[i])
            {
                tree.addEdge(i, j);
                if (tree.containsCycle())  tree.removeEdge(i, j);
                else if (j > i) n++;
            }
        }
    }


    int c = 1;
    UndirectedGraph temp = tree;
    cout << "Tree: " << c << endl;
    cout << "================" << endl;
    tree.displayGraph();
    for (int src = 0; src < numVertices; src++)
    {
        for (auto&& dest : graph[src])
        {
            if (this->isEdge(src, dest) && !temp.isEdge(src, dest)) {

                temp.addEdge(src, dest);
                for (int treesrc = 0; treesrc < numVertices; treesrc++)
                {
                    for (auto&& treedest : tree.graph[treesrc])
                    {
                        int ti = treedest;
                        temp.removeEdge(treesrc, ti);
                        if (!temp.containsCycle()) {
                            ++c;
                            cout << "Tree: " << c << endl;
                            cout << "================" << endl;
                            temp.displayGraph();
                        }
                        temp.addEdge(treesrc, ti);
                    }
                }
```

```cpp
                temp.removeEdge(src, dest);
            }
        }
    }
}

bool UndirectedGraph::isPlanner(){

    if(containsCycle()){
        cout << "Here" << endl;
        if(numEdges<=(3*numVertices-6)){
            return true;
        }
        else{
            return false;
        }
    }
    else{
        cout << "There" << endl;
        if(numEdges<=(2*numVertices-4)){
            return true;
        }
        else{
            return false;
        }
    }
}
```

Snapshot of the output:



```cpp
#include <iostream>
#include "UndirectedGraph.h"

int main(){
    using namespace std;

    UndirectedGraph K(4);

    K.addEdge(0,1);
    K.addEdge(0,2);
    K.addEdge(0,3);

    cout << "Plannarity: " << K.isPla

    UndirectedGraph K2(5);

    K2.addEdge(0,1);
    K2.addEdge(0,2);
    K2.addEdge(0,3);
    K2.addEdge(0,4);

    K2.addEdge(1,2);
    K2.addEdge(1,3);
    K2.addEdge(1,4);

    K2.addEdge(2,3);
    K2.addEdge(2,4);

    K2.addEdge(3,4);

    cout << "Plannarity: " << K2.isPl

    return 0;
```

```
Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

S:\SEM 6\GraphTheory\GraphTheory>cd "s:\SEM 6\GraphTheory\GraphTheory\" && g++ Prac9_PlanarityTest
.cpp -o Prac9_PlanarityTest && "s:\SEM 6\GraphTheory\GraphTheory\"Prac9_PlanarityTest
Plannarity: There
1
Plannarity: Here
0

S:\SEM 6\GraphTheory\GraphTheory>
```