Date: 08/05/2021

Practical 9:

2CSDE56 - Graph Theory

Name: Shrey Viradiya

Roll No: 18BCE259

Aim:

Write a program to find the maximum clique from a given graph.

Code:

Prac8_FindingMaxClique.cpp

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

int main(){
    using namespace std;

    UndirectedGraphMatrix K("Kirchoff", 6);

    K.addEdge(0,1);
    K.addEdge(0,2);
    K.addEdge(0,3);
    K.addEdge(0,4);
    K.addEdge(0,5);
    K.addEdge(1,2);
    K.addEdge(1,3);
    K.addEdge(1,4);
    K.addEdge(2,3);
    K.addEdge(2,4);
    K.addEdge(3,4);

    cout << "Maximum Clique Size: " << K.maxCliques(0,1) << endl;

    return 0;
}
```

UndirectedGraphMatrix.h

```cpp
#pragma once

#include<iostream>
#include<algorithm>
#include<map>
#include<cstring>
#include<vector>
#include"mincutsetutilities.h"

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

```cpp
public:
    int store[100] {0};
    UndirectedGraphMatrix(const char n[], int V);
    UndirectedGraphMatrix(const UndirectedGraphMatrix & obj);
    ~UndirectedGraphMatrix();

    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();
    int ** getGraphCopy();
    void displayGraph();
    bool is_clique(int b);
    int maxCliques(int i, int l);

    bool isSafe (int v, const int* color, int c);
    bool GraphColoringREC(int m, int* color, int v);
    int * SolveGraph(int m);



    static bool CheckIsomorphism(UndirectedGraphMatrix &graphA, UndirectedGrap
hMatrix &graphB);
    void minimumCutSet();
    void minimumCutVertex();
};

UndirectedGraphMatrix::UndirectedGraphMatrix(const char n[50], int V){
    noVertices = V;
    std::strcpy(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 << "\nGraph Created: " << name << endl;
}

// UndirectedGraphMatrix::UndirectedGraphMatrix(const UndirectedGraphMatrix &o
bj)
```

```
// {
//  noVertices = obj.noVertices;
//     strcpy_s(name, obj.name);
//     edges = obj.edges;
//     graph = (int *) obj.getGraphCopy();
//     degrees = new int [noVertices] {0};
//     for (int i = 0; i < noVertices; i++)
//     {
//         degrees[i] = obj.degrees[i];
//     }
// }

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

    using namespace std;
    cout << "\nMemory 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;
}

int ** UndirectedGraphMatrix::getGraphCopy(){
    int **graphCopy = new int *[noVertices];

    for (int i = 0; i < noVertices; i++)
```

```
    {
        graphCopy[i] = new int[noVertices];
        for (int j = 0; j < noVertices; j++)
        {
            graphCopy[i][j] = graph[i][j];
        }
    }
    return graphCopy;
}

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;
}

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

    cout << "\nGraph:" << name << "Cutset" << endl;
    cout << "=================" << endl;

    int * degS = getSortedDegrees();

    if(getNoVertices() < 2){
        cout << "This is a single vertex graph...Cutting not possible." << end
l;
        return;
    }

    if (degS[0] == 0) {
        cout << "Graph already disconnected....Cut set is empty." << endl;
        return;
    }

    int *visited = new int[getNoVertices()]{0};
    DFS(0, graph, visited, getNoVertices());
    for (int i = 0; i < getNoVertices(); i++)
    {
        if (visited[i] == 0)
        {
            cout << "Graph already disconnected....Cut set is empty." << endl;
            return;
        }
    }
```

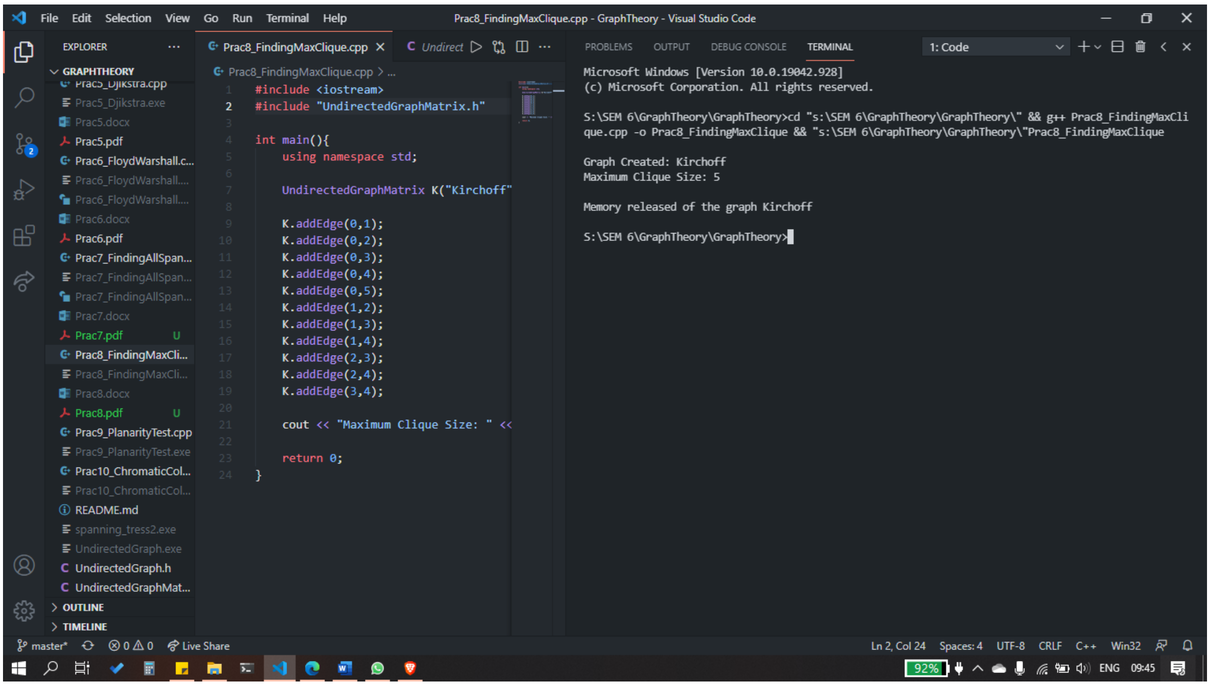
```
    delete[] visited;


    vector<pair<int, int>> edge_list;
    for (int i = 0; i < getNoVertices(); i++)
    {
        for (int j = 0; j < i; j++)
        {
            if(isEdge(i,j)){
                edge_list.push_back({i,j});
            }
        }
    }

    bool *check = new bool[edge_list.size()]{0};
    int *done = new int;
    *done = 0;
    for (int i = 1; i < degS[0]; i++)
    {
        int **graphCopy = getGraphCopy();
        // Code remaining for the removing edge and checking disconnectivity
        CombiEdges(done, edge_list, i, 0, 0, check, edge_list.size(), graphCop
y, getNoVertices());

        for (int i = 0; i < getNoVertices(); i++)
        {
            delete[] graphCopy[i];
        }
        delete[] graphCopy;
    }


    if(*done == 0)
    {
        for (int i = 0; i < getNoVertices(); i++)
        {
            if (getDegree(i) == degS[0])
            {
                for (int j = 0; j < getNoVertices(); j++)
                {
                    if(isEdge(i,j))
                        cout << i << "<->" << j << endl;
                }
                return;
            }
        }
    }
    delete[] check;
```

```
    delete done;
}

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

    cout << "\nGraph:" << name << " CutVertex" << endl;
    cout << "=================" << endl;

    int * degS = getSortedDegrees();

    if(getNoVertices() < 2){
        cout << "This is a single vertex graph...Cutting not possible." << end
l;
        return;
    }

    if (degS[0] == 0) {
        cout << "Graph already disconnected....Cut set is empty." << endl;
        return;
    }

    int *visited = new int[getNoVertices()]{0};
    DFS(0, graph, visited, getNoVertices());
    for (int i = 0; i < getNoVertices(); i++)
    {
        if (visited[i] == 0)
        {
            cout << "Graph already disconnected....Cut set is empty." << endl;
            return;
        }
    }
    delete[]visited;

    bool *check = new bool[getNoVertices()]{0};
    int *done = new int;
    *done = 0;
    for (int i = 1; i < getNoVertices(); i++)
    {
        int** graphCopy = getGraphCopy();
        int** graphBackup = getGraphCopy();
        // Code remaining for the removing edge and checking disconnectivity
        CombiVertices(done, i, 0, 0, check, getNoVertices(), graphCopy, graphB
ackup);

        for (int i = 0; i < getNoVertices(); i++)
        {
            delete[] graphCopy[i];
```

```
            delete[] graphBackup[i];
        }
        delete[] graphCopy;
        delete[] graphBackup;

    }
    delete[] check;
    delete done;
}

bool UndirectedGraphMatrix::is_clique(int b)
{

    // Run a loop for all set of edges
    for (int i = 1; i < b; i++) {
        for (int j = i + 1; j < b; j++)

            // If any edge is missing
            if (graph[store[i]][store[j]] == 0)
                return false;
    }
    return true;
}

int UndirectedGraphMatrix::maxCliques(int i, int l)
{
    // Maximal clique size
    int max_ = 0;

    // Check if any vertices from i+1
    // can be inserted
    for (int j = i + 1; j <= getNoVertices(); j++) {

        // Add the vertex to store
        store[l] = j;

        // If the graph is not a clique of size k then
        // it cannot be a clique by adding another edge
        if (is_clique(l + 1)) {

            // Update max
            max_ = max(max_, l);

            // Check if another edge can be added
            max_ = max(max_, maxCliques(j, l + 1));
        }
    }
    return max_;
```

```
}

bool UndirectedGraphMatrix::isSafe (int v, const int* color, int c)
{
    for (int i = 0; i < noVertices; i++)
        if (graph[v][i] && c == color[i])
            return false;
    return true;
}

bool UndirectedGraphMatrix::GraphColoringREC(int m, int* color, int v)
{
    if (v == noVertices)
        return true;

    for (int c = 1; c <= m; c++)
    {
        if (isSafe(v, color, c))
        {
            color[v] = c;

            if (GraphColoringREC (m, color, v+1) == true)
                return true;

            color[v] = 0;
        }
    }

    return false;
}

int * UndirectedGraphMatrix::SolveGraph(int m)
{
    int* color = (int *) calloc(noVertices, sizeof(int));

    if (GraphColoringREC(m, color, 0) == false)
    {
        return nullptr;
    }
    return color;
}
```

Snapshot of the output:



Prac8_FindingMaxClique.cpp - GraphTheory - Visual Studio Code
EXPLORER
GRAPHTHEORY
Prac5_Djikstra.exe
Prac5.docx
Prac5.pdf
Prac6.docx
Prac6.pdf
Prac7.docx
Prac7.pdf
Prac8.docx
Prac8.pdf
Prac9_PlanarityTest.cpp
Prac9_PlanarityTest.exe
README.md
spanning_tress2.exe
UndirectedGraph.exe
UndirectedGraph.h
OUTLINE
TIMELINE
#include <iostream>
#include "UndirectedGraphMatrix.h"
int main(){
using namespace std;
UndirectedGraphMatrix K("Kirchoff"
K.addEdge(0,1);
K.addEdge(0,2);
K.addEdge(0,3);
K.addEdge(0,4);
K.addEdge(0,5);
K.addEdge(1,2);
K.addEdge(1,3);
K.addEdge(1,4);
K.addEdge(2,3);
K.addEdge(2,4);
K.addEdge(3,4);
cout << "Maximum Clique Size: " <<
return 0;
}
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
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++ Prac8_FindingMaxClique.cpp -o Prac8_FindingMaxClique && "s:\SEM 6\GraphTheory\GraphTheory\"Prac8_FindingMaxClique
Graph Created: Kirchoff
Maximum Clique Size: 5
Memory released of the graph Kirchoff
S:\SEM 6\GraphTheory\GraphTheory>
master*
Live Share
Ln 2, Col 24
Spaces: 4
UTF-8
CRLF
C++
Win32