Date: 21/03/2021

Practical 4:

2CSDE56 - Graph Theory

Name: Shrey Viradiya

Roll No: 18BCE259

Aim:

Write a program to find the minimum cut-edges of the given graph.

Code:

Prac4_MinimumCutEdges.cpp

```cpp
// Implement the Code for finding Minimum
// cut edges and cut vertices for a graph

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

int main(){
    using namespace std;

    UndirectedGraphMatrix graphA("Euler", 8);

    // adding some edges in graph A

    //   1        5
    //  / \      / \
    // 0   2----4   6
    //  \ /      \ /
    //   3        7

    graphA.addEdge(0,1);
    graphA.addEdge(1,2);
    graphA.addEdge(2,3);
    graphA.addEdge(0,3);
    graphA.addEdge(4,5);
    graphA.addEdge(5,6);
    graphA.addEdge(6,7);
    graphA.addEdge(7,4);
    graphA.addEdge(2,4);

    graphA.displayGraph();

    graphA.minimumCutSet();

    UndirectedGraphMatrix graphB("Kirchoff", 8);

    // adding some edges in graph A

    //  0 - 1 --- 4 - 5
    //  | x |     | x |
    //  2 - 3 --- 6 - 7

    graphB.addEdge(0,1);
    graphB.addEdge(2,3);
    graphB.addEdge(0,3);
    graphB.addEdge(1,2);
```

```
    graphB.addEdge(0,2);
    graphB.addEdge(1,3);
    graphB.addEdge(4,5);
    graphB.addEdge(5,6);
    graphB.addEdge(6,7);
    graphB.addEdge(7,4);
    graphB.addEdge(4,6);
    graphB.addEdge(5,7);
    graphB.addEdge(1,4);
    graphB.addEdge(3,6);

    graphB.displayGraph();

    graphB.minimumCutSet();

    return 0;
}
```

UndirectedGraphMatrix.h

```
#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;

public:
    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();


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

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 << "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;
}

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;
        }
    }


    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()];
    int *done = new int;
    for (int i = 1; i < degS[0]; i++)
    {
        int **graphCopy = getGraphCopy();
        *done = 0;
```

```
        // 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;
            }
        }
    }
}
```

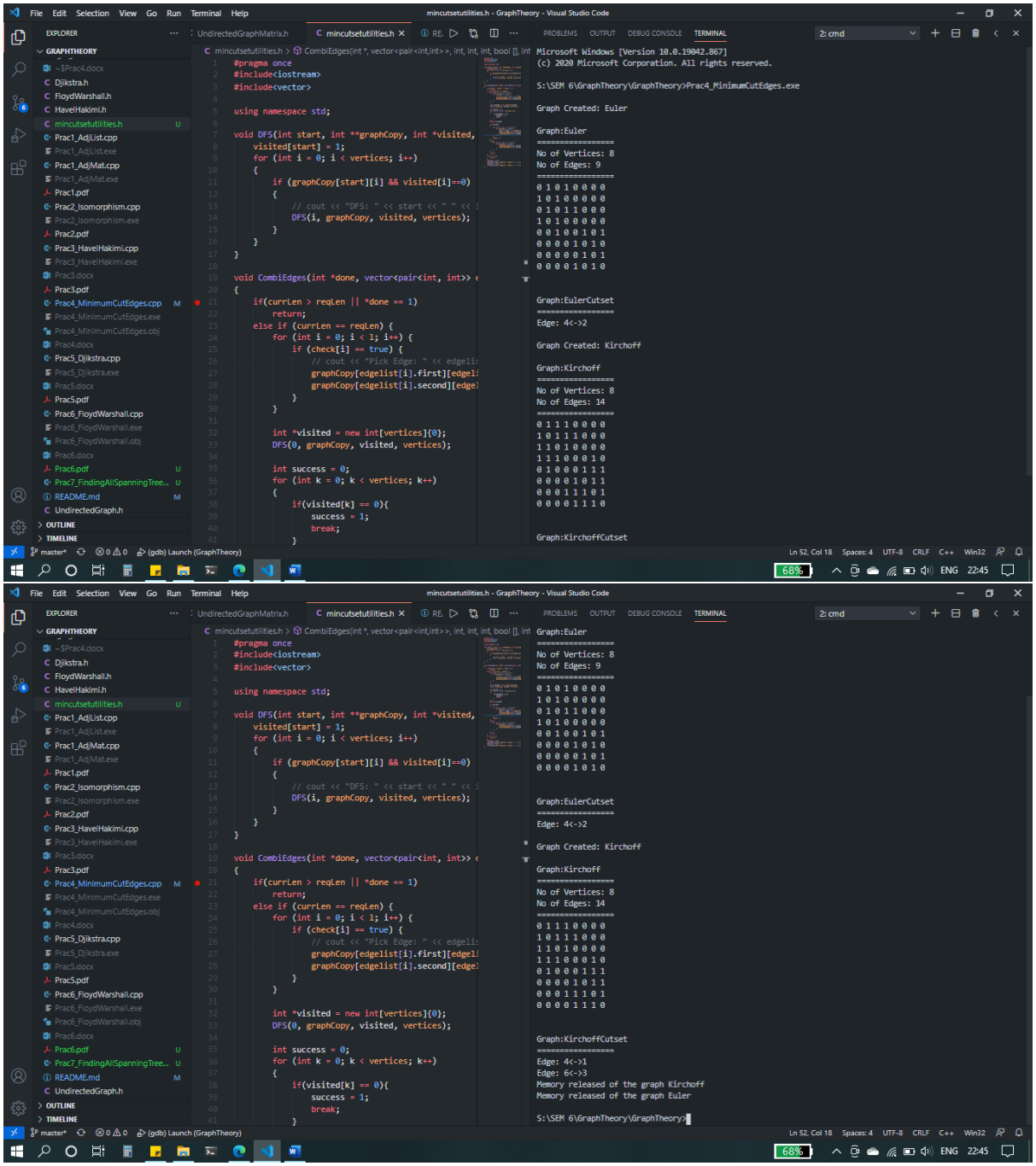
mincutsetutilities.h

```
#pragma once
#include<iostream>
#include<vector>

using namespace std;

void DFS(int start, int **graphCopy, int *visited, int vertices){
    visited[start] = 1;
    for (int i = 0; i < vertices; i++)
    {
        if (graphCopy[start][i] && visited[i]==0)
        {
            // cout << "DFS: " << start << " " << i << endl;
            DFS(i, graphCopy, visited, vertices);
        }
    }
}
```

```
void CombiEdges(int *done, vector<pair<int, int>> edgelist, int reqLen, int s,
 int currLen, bool check[], int l, int **graphCopy, int vertices)
{
    if(currLen > reqLen || *done == 1)
        return;
    else if (currLen == reqLen) {
        for (int i = 0; i < l; i++) {
            if (check[i] == true) {
                // cout << "Pick Edge: " << edgelist[i].first << ", " << edgel
ist[i].second << endl;
                graphCopy[edgelist[i].first][edgelist[i].second] = 0;
                graphCopy[edgelist[i].second][edgelist[i].first] = 0;
            }
        }

        int *visited = new int[vertices]{0};
        DFS(0, graphCopy, visited, vertices);

        int success = 0;
        for (int k = 0; k < vertices; k++)
        {
            if(visited[k] == 0){
                success = 1;
                break;
            }
        }
        delete[]visited;

        if (success)
        {
            for (int i = 0; i < l; i++) {
                if (check[i] == true) {
                    cout << "Edge: " << edgelist[i].first << "<-
>" << edgelist[i].second << endl;
                    graphCopy[edgelist[i].first][edgelist[i].second] = 1;
                    graphCopy[edgelist[i].second][edgelist[i].first] = 1;
                }
            }
            *done = 1;
        }
        else{
            for (int i = 0; i < l; i++) {
                if (check[i] == true) {
                    graphCopy[edgelist[i].first][edgelist[i].second] = 1;
                    graphCopy[edgelist[i].second][edgelist[i].first] = 1;
                }
            }
        }
```

```
        return;
    }
    if (s == l) {
        return;
    }
    check[s] = true;
    CombiEdges(done, edgelist, reqLen, s + 1, currLen + 1, check, l, graphCopy
, vertices);
    check[s] = false;
    CombiEdges(done, edgelist, reqLen, s + 1, currLen, check, l, graphCopy, ve
rtices);
}
```

Snapshot of the output:


File Edit Selection View Go Run Terminal Help
mincutsetutilities.h - GraphTheory - Visual Studio Code
EXPLORER
GRAPHTHEORY
mincutsetutilities.h
#pragma once
#include<iostream>
#include<vector>
using namespace std;
void DFS(int start, int **graphCopy, int *visited,
visited[start] = 1;
for (int i = 0; i < vertices; i++)
if (graphCopy[start][i] && visited[i]==0)
DFS(i, graphCopy, visited, vertices);
void CombiEdges(int *done, vector<pair<int, int>>
if(currLen > reqLen || *done == 1)
return;
else if (currLen == reqLen) {
for (int i = 0; i < 1; i++) {
if (check[i] == true) {
graphCopy[edgelist[i].first][edgeli
graphCopy[edgelist[i].second][edge
int *visited = new int[vertices]{0};
DFS(0, graphCopy, visited, vertices);
int success = 0;
for (int k = 0; k < vertices; k++)
if(visited[k] == 0){
success = 1;
break;
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2: cmd
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.
S:\SEM 6\GraphTheory\GraphTheory>Prac4_MinimumCutEdges.exe
Graph Created: Euler
Graph:Euler
No of Vertices: 8
No of Edges: 9
0 1 0 1 0 0 0 0
1 0 1 0 0 0 0 0
0 1 0 1 1 0 0 0
1 0 1 0 0 0 0 0
0 0 1 0 0 1 0 1
0 0 0 0 1 0 1 0
0 0 0 0 0 1 0 1
0 0 0 0 1 0 1 0
Graph:EulerCutset
Edge: 4<->2
Graph Created: Kirchoff
Graph:Kirchoff
No of Vertices: 8
No of Edges: 14
0 1 1 1 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 1 0 0 0 0
1 1 1 0 0 0 1 0
0 1 0 0 0 1 1 1
0 0 0 0 1 0 1 1
0 0 0 1 1 1 0 1
0 0 0 0 1 1 1 0
Graph:KirchoffCutset
master*
(gdb) Launch (GraphTheory)
Ln 52, Col 18
Spaces: 4
UTF-8
CRLF
C++
Win32
68%
ENG
22:45

File Edit Selection View Go Run Terminal Help
mincutsetutilities.h - GraphTheory - Visual Studio Code
EXPLORER
GRAPHTHEORY
mincutsetutilities.h
#pragma once
#include<iostream>
#include<vector>
using namespace std;
void DFS(int start, int **graphCopy, int *visited,
visited[start] = 1;
for (int i = 0; i < vertices; i++)
if (graphCopy[start][i] && visited[i]==0)
DFS(i, graphCopy, visited, vertices);
void CombiEdges(int *done, vector<pair<int, int>>
if(currLen > reqLen || *done == 1)
return;
else if (currLen == reqLen) {
for (int i = 0; i < 1; i++) {
if (check[i] == true) {
graphCopy[edgelist[i].first][edgeli
graphCopy[edgelist[i].second][edge
int *visited = new int[vertices]{0};
DFS(0, graphCopy, visited, vertices);
int success = 0;
for (int k = 0; k < vertices; k++)
if(visited[k] == 0){
success = 1;
break;
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2: cmd
Graph:Euler
No of Vertices: 8
No of Edges: 9
0 1 0 1 0 0 0 0
1 0 1 0 0 0 0 0
0 1 0 1 1 0 0 0
1 0 1 0 0 0 0 0
0 0 1 0 0 1 0 1
0 0 0 0 1 0 1 0
0 0 0 0 0 1 0 1
0 0 0 0 1 0 1 0
Graph:EulerCutset
Edge: 4<->2
Graph Created: Kirchoff
Graph:Kirchoff
No of Vertices: 8
No of Edges: 14
0 1 1 1 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 1 0 0 0 0
1 1 1 0 0 0 1 0
0 1 0 0 0 1 1 1
0 0 0 0 1 0 1 1
0 0 0 1 1 1 0 1
0 0 0 0 1 1 1 0
Graph:KirchoffCutset
Edge: 4<->1
Edge: 6<->3
Memory released of the graph Kirchoff
Memory released of the graph Euler
S:\SEM 6\GraphTheory\GraphTheory>
master*
(gdb) Launch (GraphTheory)
Ln 52, Col 18
Spaces: 4
UTF-8
CRLF
C++
Win32
68%
ENG
22:45