Moldova State University

Faculty of Mathematics and Informatics

Department of Informatics

Operations on graphs

Laboratory Report II

Author Ciobanu Stanislav

Group I2302

Date 23.10.23

Table of contents

Project Code

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

class Vertex;

class Edge
{
private:
	Vertex* _vertex1;
	Vertex* _vertex2;
	int _weight;

public:

	Edge(Vertex* v1, Vertex* v2, int weight)
	{
		_weight = weight;
		_vertex1 = v1;
		_vertex2 = v2;
	}

	int GetWeight()
	{
		return _weight;
	}

	Vertex* GetVertex1()
	{
		return _vertex1;
	}

	Vertex* GetVertex2()
	{
		return _vertex2;
	}
};

class Vertex
{
private:
	Edge** _edges;
	unsigned int _size;
	int _ID;

public:

	Vertex(unsigned int size, unsigned int ID)
	{
		_edges = new Edge * ();
		_size = size;

		_edges = (Edge**)calloc(_size, sizeof(Edge*));

		for (int i = 0; i < _size; i++)
		{
			_edges[i] = (Edge*)malloc(sizeof(Edge));
			_edges[i] = NULL;
		}

		_ID = ID;
```

```
	}

	void EnlargeEdgeList()
	{
		_size++;
		Edge* edge = (Edge*)malloc(sizeof(Edge));

		_edges = (Edge**)realloc(_edges, sizeof(Edge*) * _size);

		_edges[_size - 1] = edge;
	}

	void ShortenEdgeList(int ID)
	{
		_size--;

		for (int i = ID; i < _size; i++)
		{
			_edges[i] = _edges[i + 1];
		}

		Edge** tempEdges = (Edge**)calloc(_size, sizeof(Edge*));

		for (int i = 0; i < _size; i++)
		{
			tempEdges[i] = _edges[i];
		}
		_edges = tempEdges;
	}

	unsigned int GetID() {
		return _ID;
	}

	void ChangeID(int delta)
	{
		_ID += delta;
	}

	Edge* GetEdge(int ID)
	{
		return _edges[ID];
	}

	void SetEdge(Edge* edge, unsigned int ID)
	{
		_edges[ID] = edge;
	}

	Edge** GetEdgeArray()
	{
		return _edges;
	}

};

class Graph
{
private:
	Vertex** _verts;
	unsigned int _size;
	Edge** _allEdges;
	unsigned int _edgesArraySize;
public:

	Graph(int size)
```

```
	{
		_verts = new Vertex * ();
		_size = size;

		_verts = (Vertex**)calloc(_size, sizeof(Vertex*));

		for (int i = 0; i < _size; i++)
		{
			_verts[i] = new Vertex(_size, i);
		}

		_allEdges = (Edge**)malloc(sizeof(Edge*));
		_edgesArraySize = 0;
	}

	int GetSize() {
		return _size;
	}

	Vertex** GetVertexArray()
	{
		return _verts;
	}

	Edge** GetEdgeArray()
	{
		return _allEdges;
	}

	int GetEdgeArraySize()
	{
		return _edgesArraySize;
	}

	void CreateEdge(unsigned ID1, unsigned ID2, unsigned int weight)
	{
		Edge* edge = new Edge(_verts[ID1], _verts[ID2], weight);
		_verts[ID1]->SetEdge(edge, ID2);
		_verts[ID2]->SetEdge(edge, ID1);

		AddToEdgeList(edge);
	}

	void AddToEdgeList(Edge* edge)
	{

		if (_allEdges[0] == NULL)
		{
			_allEdges[0] = edge;
		}
		else
		{
			_edgesArraySize++;

			Edge** tempEdges = (Edge**)calloc(_edgesArraySize,
sizeof(Edge*));

			for (int i = 0; i < _edgesArraySize; i++)
			{
				tempEdges[i] = _allEdges[i];
			}

			tempEdges[_edgesArraySize - 1] = edge;

			_allEdges = tempEdges;
```

```
		}
	}

	void RemoveFromEdgeList(unsigned int ID1, unsigned int ID2)
	{

		for (int i = 0; i < _edgesArraySize; i++)
		{
			if (((_allEdges[i]->GetVertex1()->GetID() == ID1) or
(_allEdges[i]->GetVertex1()->GetID() == ID2))
				and ((_allEdges[i]->GetVertex2()->GetID() == ID1) or
(_allEdges[i]->GetVertex2()->GetID() == ID2)))
			{

				for (int j = i; j < _edgesArraySize - 1; j++)
				{
					_allEdges[j] = _allEdges[j + 1];
				}

				_edgesArraySize--;
				Edge** tempEdges = (Edge**)calloc(_edgesArraySize,
sizeof(Edge*));

				for (int j = 0; j < _edgesArraySize; j++)
				{
					tempEdges[j] = _allEdges[j];
				}

				_allEdges = tempEdges;

				break;
			}
		}
	}

	void AddVertex()
	{
		for (int i = 0; i < _size; i++)
		{
			_verts[i]->EnlargeEdgeList();
		}

		_size++;
		Vertex* vert = new Vertex(_size, _size - 1);

		Vertex** tempVerts = (Vertex**)calloc(_size, sizeof(Vertex*));

		for (int i = 0; i < _size; i++)
		{
			tempVerts[i] = _verts[i];
		}

		tempVerts[_size - 1] = vert;
		_verts = tempVerts;
	}

	void DeleteEdge(int ID1, int ID2)
	{
		for (int i = 0; i < _edgesArraySize; i++)
		{
			if ((_allEdges[i]->GetVertex1()->GetID() == ID1) and
(_allEdges[i]->GetVertex2()->GetID() == ID2))
			{
				_allEdges[i]->GetVertex1()->SetEdge(NULL, ID2);
				_allEdges[i]->GetVertex2()->SetEdge(NULL, ID1);
```

```
                        RemoveFromEdgeList(ID1, ID2);

                }
                else if (((_allEdges[i]->GetVertex2()->GetID() == ID1) and
(_allEdges[i]->GetVertex1()->GetID() == ID2)))
                {
                        _allEdges[i]->GetVertex1()->SetEdge(NULL, ID1);
                        _allEdges[i]->GetVertex2()->SetEdge(NULL, ID2);

                        RemoveFromEdgeList(ID1, ID2);
                }
        }
    }

    void DeleteVertex(int ID)
    {

        for (int i = 0; i < _edgesArraySize; i++)
        {
                int a = _allEdges[i]->GetVertex1()->GetID();
                int b = _allEdges[i]->GetVertex2()->GetID();

                if (a == ID)
                {
                        RemoveFromEdgeList(a, b);
                        i--;
                }
                else if (b == ID)
                {
                        RemoveFromEdgeList(b, a);
                        i--;
                }

        }

        for (int i = 0; i < ID; i++)
        {
                _verts[i]->ShortenEdgeList(ID);
        }

        _size--;

        for (int i = ID; i < _size; i++)
        {
                _verts[i] = _verts[i + 1];
                _verts[i]->ShortenEdgeList(ID);
                _verts[i]->ChangeID(-1);
        }

        Vertex** tempVerts = (Vertex**)calloc(_size, sizeof(Vertex*));

        for (int i = 0; i < _size; i++)
        {
                tempVerts[i] = _verts[i];
        }

        _verts = tempVerts;
    }

    void PrintAdjacenceMatrix()
    {

        std::cout << "\n\n\t[ ADJACENCE MATRIX ]\n";
        printf("\n");
```

```
		for (int i = 0; i < _size; i++)
		{
			Vertex vert = *_verts[i];

			for (int j = 0; j < _size; j++)
			{
				if (vert.GetEdge(j) != NULL) {
					std::cout << "\t" << 1 << "\t ";
				}
				else {
					std::cout << "\t" << 0 << "\t ";
				}
			}
			printf("\n\n");
		}
	}

	void PrintWeightMatrix()
	{
		std::cout << "\n\t[ WEIGHT MATRIX ]\n";
		printf("\n");
		for (int i = 0; i < _size; i++)
		{
			Vertex vert = *_verts[i];
			for (int j = 0; j < _size; j++)
			{
				if (vert.GetEdge(j) != NULL) {
					std::cout << "\t" << vert.GetEdge(j)->GetWeight() <<
"\t ";
				}
				else {
					std::cout << "\t" << 0 << "\t ";
				}

			}
			printf("\n\n");
		}
	}

	void PrintEdgesList()
	{
		std::cout << "\n\t[ ALL EDGES ]\n";

		for (int i = 0; i < _edgesArraySize; i++)
		{
			if (_allEdges[i] != NULL)
			{
				std::cout << "\n\t[ \t" << _allEdges[i]->GetVertex1()-
>GetID()
					<< "\t,\t" << _allEdges[i]->GetVertex2()->GetID() <<
"\t]\n";
			}
		}

		std::cout << "\n";
	}


	void PrintNeighboursList()
	{
		std::cout << "\n\t[ ALL NEIGHBOURS ]\n";
		for (int i = 0; i < _size; i++)
		{
			std::cout << "\n\t[" << _verts[i]->GetID() << "]\t";
```

```
                        Edge** edges = _verts[i]->GetEdgeArray();
                        for (int j = 0; j < _size; j++)
                        {
                                if (edges[j] != NULL)
                                {
                                        if (edges[j]->GetVertex1()->GetID() == i)
                                        {
                                                std::cout << edges[j]->GetVertex2()->GetID()
<< " \t";
                                        }
                                        else
                                        {
                                                std::cout << edges[j]->GetVertex1()->GetID()
<< " \t";
                                        }
                                }
                        }
                        std::cout << "\n";
                }
        }

        void PerformDFS(int start)
        {
                int vertsArrSize = _msize(_verts) / sizeof(Vertex*);
                std::cout << "\t[DFS]\n";
                bool** visited = (bool**)calloc(vertsArrSize, sizeof(bool*));
                for (int i = 0; i < vertsArrSize; i++)
                {
                        visited[i] = (bool*)malloc(sizeof(bool));
                        *visited[i] = 0;
                }

                DFS(start, visited);
        }

        bool** DFS(int current, bool** visited)
        {
                int vertsArrSize = _msize(_verts) / sizeof(Vertex*);
                *visited[current] = 1;
                std::cout << "\n\tVisited - " << current;
                for (int i = 0; i < vertsArrSize; i++)
                {
                        if ((_verts[current]->GetEdgeArray()[i] != NULL) and
(*visited[i] == 0))
                        {
                                visited = DFS(i, visited);
                        }
                }
                return visited;
        }

        void PerformBFS(int start)
        {
                int vertsArrSize = _msize(_verts) / sizeof(Vertex*);
                std::cout << "\n\t[BFS]\n";

                bool** visited = (bool**)calloc(vertsArrSize, sizeof(bool*));
                for (int i = 0; i < vertsArrSize; i++)
                {
                        visited[i] = (bool*)malloc(sizeof(bool));
                        *visited[i] = 0;
                }

                bool** checked = (bool**)calloc(vertsArrSize, sizeof(bool*));
                for (int i = 0; i < vertsArrSize; i++)
```

```
		{
			checked[i] = (bool*)malloc(sizeof(bool));
			*checked[i] = 0;
		}

		visited = VisitVertex(start, visited);
		BFS(start, visited, checked);
	}

	void BFS(int current, bool** visited, bool** checked)
	{
		int vertsArrSize = _msize(_verts) / sizeof(Vertex*);
		*checked[current] = 1;

		for (int i = 0; i < vertsArrSize; i++)
		{
			bool b = (*checked[i] == 0);
			bool d = (_verts[current]->GetEdgeArray()[i] != NULL);
			bool g = (*visited[i] == 0);

			if ((_verts[current]->GetEdgeArray()[i] != NULL) and
(*visited[i] == 0) and (*checked[i] == 0))
			{
				visited = VisitVertex(i, visited);
			}
		}

		for (int i = 0; i < vertsArrSize; i++)
		{
			if ((_verts[current]->GetEdgeArray()[i] != NULL) and
(*checked[i] == 0))
			{
				BFS(i, visited, checked);
			}
		}

	}

	bool** VisitVertex(int vertex, bool** visited)
	{
		*visited[vertex] = 1;
		std::cout << "\n\tVisited - " << vertex;
		return visited;
	}


};

	};

Graph CreateComplementaryGraph(Graph graph) // Not really a good code but it works
{
	int size = graph.GetSize();
	Graph complementaryGraph = *new Graph(size);

	for (int i = 0; i < size; i++)
	{
		for (int j = 0; j < size; j++)
		{
			if (graph.GetVertexArray()[i]->GetEdgeArray()[j] == NULL)
			{
				complementaryGraph.CreateEdge(i, j, 1);
			}
		}
	}
```

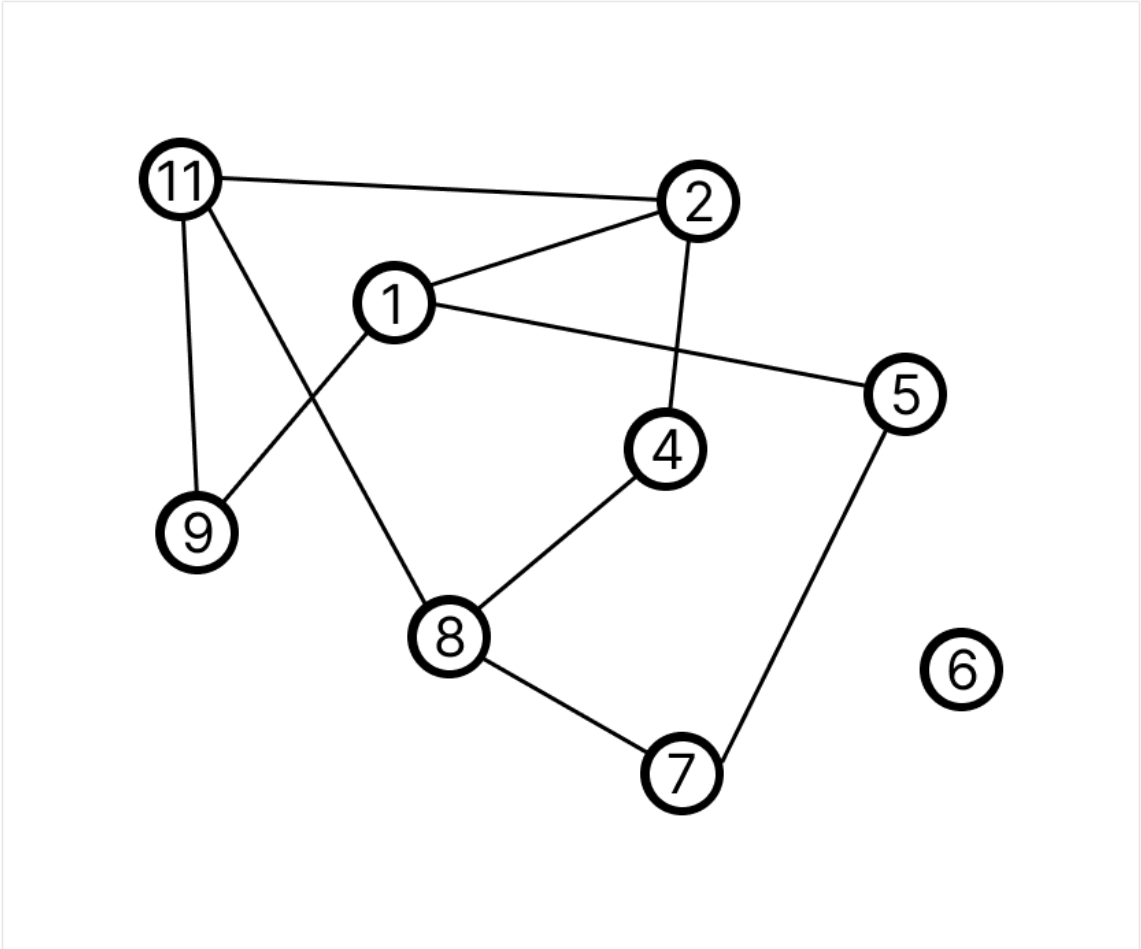
```
	return complementaryGraph;
}

Graph CreateAssociatedGraph(Graph graph) // Not really a good code too but it works
{
	int size = graph.GetEdgeArraySize();
	Graph asGraph = *new Graph(size);

	for (int i = 0; i < size - 1; i++)
	{
		graph.GetEdgeArray()[i];

		for (int j = i + 1; j < size; j++)
		{
			if ((graph.GetEdgeArray()[i]->GetVertex1()->GetID() ==
graph.GetEdgeArray()[j]->GetVertex1()->GetID()) or
				(graph.GetEdgeArray()[i]->GetVertex2()->GetID() ==
graph.GetEdgeArray()[j]->GetVertex1()->GetID()) or
				(graph.GetEdgeArray()[i]->GetVertex1()->GetID() ==
graph.GetEdgeArray()[j]->GetVertex2()->GetID()) or
				(graph.GetEdgeArray()[i]->GetVertex2()->GetID() ==
graph.GetEdgeArray()[j]->GetVertex2()->GetID()))
			{

				asGraph.CreateEdge(i, j, 1);
			}
		}
	}

	return asGraph;
}

int main()
{
	cout << "\n Created Graph\n";
	Graph graph = *new Graph(11);		//Task 1

	graph.CreateEdge(0, 1, 1);
	graph.CreateEdge(0, 4, 4);
	graph.CreateEdge(0, 8, 9);
	graph.CreateEdge(1, 3, 13);
	graph.CreateEdge(1, 10, 110);
	graph.CreateEdge(10, 8, 108);
	graph.CreateEdge(10, 7, 107);
	graph.CreateEdge(7, 3, 73);
	graph.CreateEdge(7, 6, 76);
	graph.CreateEdge(6, 4, 64);

	graph.PrintAdjacenceMatrix();

	cout << "\n Printed edges \n";
	graph.PrintEdgesList();			//Task 2
	graph.PrintNeighboursList();

	cout << "\n Created vertex \n";
	graph.AddVertex();			//Task 3
	graph.PrintAdjacenceMatrix();

	cout << "\n Deleted vertex \n";
	graph.DeleteVertex(5);			//Task 4
	graph.PrintAdjacenceMatrix();

	cout << "\n Created edge \n";
	graph.CreateEdge(0, 3, 3);		//Task 5
	graph.PrintAdjacenceMatrix();
```

```
        cout << "\n Deleted edge \n";
        graph.DeleteEdge(0, 3);                 //Task 6
        graph.PrintAdjacenceMatrix();

        cout << "\n Created complementary graph \n";
        Graph compGraph = CreateComplementaryGraph(graph); // Task 7
        compGraph.PrintAdjacenceMatrix();

        cout << "\n Created assiciated graph \n";
        Graph asGraph = CreateAssociatedGraph(graph);         // Task 8
        asGraph.PrintAdjacenceMatrix();

        cout << "\n Performed DFS \n";
        graph.PerformDFS(0);                    //Task 9

        cout << "\n Performed BFS \n";
        graph.PerformBFS(0);                    //Task 10


        return 0;
}
```

Task 0

1. Picture of graph

Section 1



2. Results

```
[ ADJACENCE MATRIX ]
0   1   0   0   1   0   0   0   1   0   0
1   0   0   1   0   0   0   0   0   0   1
0   0   0   0   0   0   0   0   0   0   0
0   1   0   0   0   0   0   1   0   0   0
1   0   0   0   0   0   1   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   1   0   0   1   0   0   0
0   0   0   1   0   0   1   0   0   0   1
1   0   0   0   0   0   0   0   0   0   1
0   0   0   0   0   0   0   0   0   0   0
0   1   0   0   0   0   0   1   1   0   0
```

Task 2

Results

```
[ ALL EDGES ]
[   1   ,   2   ]
[   1   ,   5   ]
[   1   ,   9   ]
[   11   ,   9   ]
[   11   ,   2   ]
[   11   ,   8   ]
[   8   ,   4   ]
[   2   ,   4   ]
[   5   ,   7   ]
[   7   ,   8   ]
[ ALL NEIGHBOURS ]
[1]   2   5   9
[2]   1   4   11
[3]
[4]   2   8
[5]   1   7
[6]
[7]   5   8
[8]   4   7   11
[9]   1   11
[10]
[11]   2   8   9
```

Task 3

Results

```
Created vertex

    [ ADJACENCE MATRIX ]
    0       1       0       0       1       0       0       0       1       0       0       1
    1       0       0       1       0       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       0       1       0       0       0       1
    1       0       0       0       0       0       1       0       0       0       0       1
    0       0       0       0       0       0       0       0       0       0       0       1
    0       0       0       0       1       0       0       1       0       0       0       1
    0       0       0       1       0       0       1       0       0       0       1       1
    1       0       0       0       0       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       0       1       1       0       0       1
    0       0       0       0       0       0       0       0       0       0       0       0
```

Task 4

Results

```
Deleted vertex

    [ ADJACENCE MATRIX ]
    0       1       0       0       1       0       0       1       0       0       1
    1       0       0       1       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       1       0       0       0       1
    1       0       0       0       0       1       0       0       0       0       1
    0       0       0       0       1       0       1       0       0       0       1
    0       0       0       1       0       1       0       0       0       1       1
    1       0       0       0       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       1       1       0       0       1
    0       0       0       0       0       0       0       0       0       0       0
```

Task 5

Results

```
Created edge

    [ ADJACENCE MATRIX ]
    0       1       0       1       1       0       0       1       0       0       1
    1       0       0       1       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       1
    1       1       0       0       0       0       1       0       0       0       1
    1       0       0       0       0       1       0       0       0       0       1
    0       0       0       0       1       0       1       0       0       0       1
    0       0       0       1       0       1       0       0       0       1       1
    1       0       0       0       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       1       1       0       0       1
    0       0       0       0       0       0       0       0       0       0       0
```

Task 6

Results

```
Deleted edge

    [ ADJACENCE MATRIX ]
    0       1       0       0       1       0       0       1       0       0       1
    1       0       0       1       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       1       0       0       0       1
    1       0       0       0       0       1       0       0       0       0       1
    0       0       0       0       1       0       1       0       0       0       1
    0       0       0       1       0       1       0       0       0       1       1
    1       0       0       0       0       0       0       0       0       1       1
    0       0       0       0       0       0       0       0       0       0       1
    0       1       0       0       0       0       1       1       0       0       1
    0       0       0       0       0       0       0       0       0       0       0
```

Task 7

Results

```
Created complementary graph

    [ ADJACENCE MATRIX ]
    1       0       1       1       0       1       1       0       1       1       1
    0       1       1       0       1       1       1       1       1       0       1
    1       1       1       1       1       1       1       1       1       1       1
    1       0       1       1       1       1       0       1       1       1       1
    0       1       1       1       1       0       1       1       1       1       1
    1       1       1       1       0       1       0       1       1       1       1
    1       1       1       0       1       0       1       1       1       0       1
    0       1       1       1       1       1       1       1       1       0       1
    1       1       1       1       1       1       1       1       1       1       1
    1       0       1       1       1       1       0       0       1       1       1
    1       1       1       1       1       1       1       1       1       1       1
```

Task 8

Results

```
Created assiciated graph

    [ ADJACENCE MATRIX ]
    0       1       1       1       1       0       0       0       0       0
    1       0       1       0       0       0       0       0       0       1
    1       1       0       0       0       1       0       0       0       0
    1       0       0       0       1       0       0       1       0       0
    1       0       0       1       0       1       1       0       0       0
    0       0       1       0       1       0       1       0       0       0
    0       0       0       0       1       1       0       1       1       0
    0       0       0       1       0       0       1       0       1       0
    0       0       0       0       0       0       1       1       0       1
    0       1       0       0       0       0       0       0       1       0
```

Task 9

Results

```
Performed DFS
        [DFS]

        Visited - 0
        Visited - 1
        Visited - 3
        Visited - 6
        Visited - 5
        Visited - 4
        Visited - 10
        Visited - 9
        Visited - 7
```

Task 10

Results