

# BLG372E Analysis of Algorithms

## PROJECT 3

Ozan Arkan Can

040090573

Instructor: Zehra Çataltepe

Teaching Assistant: Meryem Uzun-Per

Submission Date: 20.05.2012

CRN: 22534

## INTRODUCTION

Network flow is an advanced branch of graph theory. The problem revolves around a special type of weighted directed graph with two special vertices: the source vertex, which has no incoming edge, and the sink vertex, which has no outgoing edge. The problem is generally like that: there are a bunch of junctions and a bunch of pipes connection the junction. The pipe will only allow material to flow one way. Each pipe has also has a capacity, representing the maximum amount of material that can flow through the pipe. The problem is to find maximum amount of material that will flow to the sink. In this project, it is aim that designing an algorithm for maximum flow problem by scenario that given project document. According to the document, there are several transportation agencies which carry products from their agencies to the other agencies. Each agency sen trucks that have a capacity. If products that exceed total weight, the remaining product is wasted. In the first part of project, the purpose is optimizing the transportation system in order to carry maximum possible weighted products without waste. In the second part of project, there is an transportation schedule and it is aim that finding if it is maximum flow. If it is not, finding traces that maximize flow is other purpose.

## DEVELOPMENT AND OPERATING ENVIRONMENTS

### MS Windows

The Code::Blocks IDE has been used to write source code, compile and run the code.

The screenshot displays the Code::Blocks IDE interface. The main editor window shows the source code for a program named 'Deneme3'. The code is written in C++ and includes a header file 'Graph.h'. The main function 'getNetwork' reads input from a file 'capacity.txt' and processes it to find the maximum flow. The code uses a deque to store nodes and a list to store edges. It also includes a function to print the traces and capacities of the edges.

```
67 deque<Cell> getNetwork(char* fileName, unsigned short int& numberOfNodes)
68 {
69     ifstream inputFile(fileName);
70     Cell* newData;
71     deque<Cell> list;
72
73     if(!inputFile.is_open())
74         throw "File could not be opened.";
75
76     while(!inputFile.eof())
77     {
78         newData = new Cell();
79         inputFile >> newData->source >> newData->destination >> newData->capacity;
80
81         if(newData->source > numberOfNodes)
82             numberOfNodes = newData->source;
83         if(newData->destination > numberOfNodes)
84             numberOfNodes = newData->destination;
85
86         list.push_back(*newData);
87     }
88 }
89
```

The output window shows the execution results. It displays the mission list, the user's choice, the traces and capacities of the edges, and the maximum flow value.

```
1. Mission 1
2. Mission 2
3. Exit
your choice: 1
Traces and capacities:
1-2-3-4-5 : 3
1-2-3-5 : 5
1-3-4-5 : 5
1-3-5-5 : 3
1-5-5 : 5
Maximum flow: 21
1. Mission 1
2. Mission 2
3. Exit
your choice:
```

The status bar at the bottom indicates the current line and column: Line 79, Column 15. The encoding is UTF-8, and the font is default.

## UNIX

The source code has been also copied to Unix, then compiled and tested with GNU C++ Compiler. The following commands have been used;

To Compile:

```
▷ g++ main.cpp Graph.h Graph.cpp Cell.h Cell.cpp -o main
```

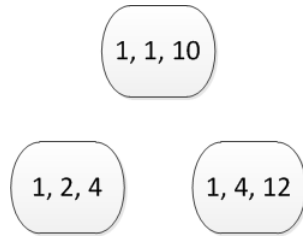
To Run:

```
▷ .\main filename1 filename2
```

## DATA STRUCTURE

### Node Representation

In this project nodes are represent by Cell class. The node has three data. Incoming node no("source" variable), Destination node no("destination" variable) and capacity ("capacity" variable). All nodes source and destination values are different except node that creates flow. Source and destination values of that node are same. Here some example:



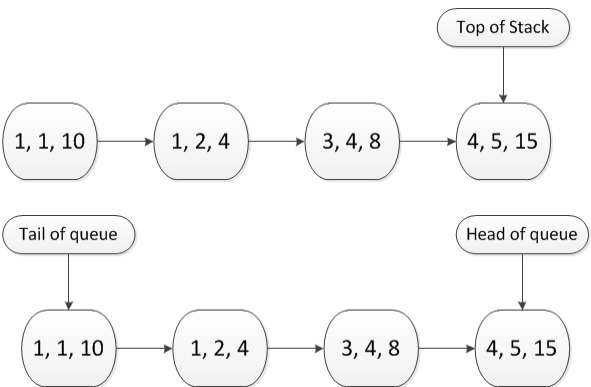
### Graph Representation

In this project adjacency matrix is chosen for representing graph. The reason why adjacency matrix is chosen that in depth first search algorithm neighbours of an is controlled and it is  $O(1)$  time with adjacency matrix. In adjacency matrix row is source, column is destination and if value of  $\text{matrix}[\text{row}][\text{source}]$  equals one , it means there is a directed edge between node(row) and node(column). In this project elements of adjacency matrix is Cell which represents node. Matrix is represented as a pointer of pointer to Cell in code and in the constructor gets memory place for that pointer dynamically. After creating graph, adjacency matrix is shown like that:

$$\begin{bmatrix} (1, 1, \text{capacity}) & (1, 2, \text{capacity}) & (1, 3, \text{capacity}) & \dots \\ (2, 1, \text{capacity}) & (2, 2, \text{capacity}) & (2, 3, \text{capacity}) & \dots \\ (3, 1, \text{capacity}) & (2, 3, \text{capacity}) & (3, 3, \text{capacity}) & \dots \\ \vdots & & & \end{bmatrix}$$

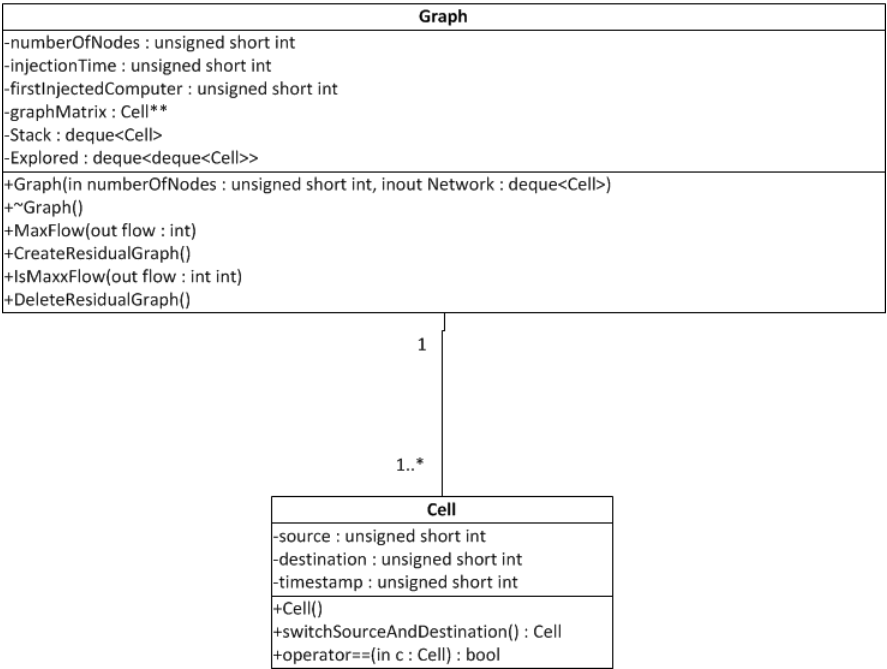
## Stack and Queue Representation

STL library is used for stack and queue data structure in the code. In STL library, there are several data structure and deque is chosen for this project. Almost all of these data structures have push-back(), push-front(), pop-back(), pop-front() methods, but all of them have not operator[]. deque structure has it and this property is used several time in graph traversing algorithm. Queue is used for holding trace and printing it. Nodes of stack are also represented by Cell objects. Nodes of queue is represented by pointer of Cell object for updating after proses.



### UML CLASS DIAGRAM

Program UML Class Diagram is like following chart:



## ANALYSIS OF ALGORITHM

In this chapter program flow is presented step by step and analyzed all functions. There are some abbreviations;

n : number of nodes

m : number of edges

### Getting Network :

Transportation network is taken from our source that name of file is given by user. File structure is like that:

source destination capacity

getNetwork method is used for this process in main source code. In this function each data is taken part by part. One Cell object is created for each triple (source, destination, capacity). And, it is added to list. Method also calculates number of nodes by finding node that its number is maximum. Method is processed until end of file. There are m triples so function running time is  $O(m)$ .

```
deque<Cell> getNetwork(char* fileName, unsigned short int& numberOfNodes)
{
    ifstream inputFile(fileName);
    Cell* newData;
    deque<Cell> list;

    if(!inputFile.is_open())
        throw "File could not be opened.";

    while(!inputFile.eof())
    {
        newData = new Cell();
        inputFile >> newData->source >> newData->destination >> newData->capacity;

        if(newData->source > numberOfNodes)
            numberOfNodes = newData->source;
        if(newData->destination > numberOfNodes)
            numberOfNodes = newData->destination;

        list.push_back(*newData);
    }

    return list;
}
```

### Creating graph :

Graph is created in the constructor of Graph Class. This constructor takes number of nodes and network info as parameters. Adjacency matrix. Adjacency matrix is filled by network info as directed graph. Number of elements in network info list is less or equal than number of edges, so running time of constructor is  $O(m)$ .

```
Graph::Graph(unsigned short int numberOfNodes, deque<Cell>& communicationNetwork)
{
    this->numberOfNodes = numberOfNodes; // set number of nodes
    Cell current;

    //Get memory place for graphMatrix and residualGraph pointer nxn.
    graphMatrix = new Cell*[numberOfNodes];

    for(int i = 0; i < numberOfNodes; i++)
        graphMatrix[i] = new Cell[numberOfNodes];
    //Iterate over network data
    for(unsigned int i = 0; i < communicationNetwork.size(); i++)
    {
        current = communicationNetwork[i];

        graphMatrix[current.source - 1][current.destination - 1] = current;
    }
}
```

## Finding maximum flow

To find maximum flow, Ford-Fulkerson Algorithm is used and depth first search algorithm is chosen for graph traversal. DFS algorithm modified for project. Here is pseudocode for algorithm:

```
1: Initialize S to be a stack with node that for source node.
2: flow  $\leftarrow 0$ , min  $\leftarrow Infinity$ 
3: While S is not empty
4:     Take a node u from S
5:     if Size of queue does not equal 0
6:         while destination of u smaller than destination of last element of queue
7:             Pop an element from end of queue
8:             if Size of queue equals 0
9:         if source and destination of u are not same
10:            Add u to queue
11:        if source of u equals number of nodes
12:            for i = 0 to size of queue
13:                cPtr  $\leftarrow Queue[i]$ 
14:                Calculate available capacity for cPtr
15:                if capacity less than min
16:                    min  $\leftarrow capacity$ 
17:            flow += min
18:            for i = 0 to size of queue
19:                Print source and update capacity
20:            min  $\leftarrow Infinity$ 
21:            Pop an element from end of queue
22:            continue
23:    For each node v that is neighbour of u
24:        if capacity of v is greater than 0
25:            Add v to the stack
```

When we consider given algorithm, it can be seen that while loop at line 3 iterates at most  $m$  times, because algorithm traverses all edge. Loops at line 6, 12 and 18 iterate at most  $n$  times, because trace data can have all node. Also, loop at line 23 iterates at most  $n - 1$  times, because a node can be neighbour of all other nodes, but there isn't an edge to itself. thus total running time is  $O(m * n)$ .

$m > n - 1 \rightarrow O(m^2)$ .

```
int Graph::MaxFlow()
{
    Cell current, sourceNode;
    int flow = 0;
    int min = INT_MAX;

    sourceNode.source = 1;
    sourceNode.destination = 1;
    sourceNode.capacity = 0;
    Stack.push_back(sourceNode);

    cout << "Traces and capacities:" << endl << endl;

    while(Stack.size() != 0)
    {
        current = Stack.back();
        current = current.switchSourceAndDestination();
        Stack.pop_back();

        if(Queue.size() != 0)
        {
            while(current.destination < Queue[Queue.size() - 1]->destination)
            {
                current = Stack.back();
                current = current.switchSourceAndDestination();
                Stack.pop_back();

                if(Queue.size() != 0)
                {
                    while(current.destination < Queue[Queue.size() - 1]->destination)
                    {
                        Queue.pop_back();
                        if(Queue.size() == 0)
                            break;
                    }
                }

                if(current.source != current.destination)
                {
                    Queue.push_back(&residualGraph[current.destination - 1][current.source - 1]);

                    if(current.source == numberOfNodes)
                    {
                        Cell* cPtr;
                        for(int i = 0; i < Queue.size(); i++)
                        {
                            int curCapacity;
                            cPtr = Queue[i];
                            curCapacity = graphMatrix[cPtr->source - 1][cPtr->destination - 1].capacity - cPtr->capacity;

                            if(curCapacity < min)
                                min = curCapacity;
                        }

                        flow += min;
                        for(int i = 0; i < Queue.size(); i++)//Show trace
                        {
                            cPtr = Queue[i];
                            cout << cPtr->source << "->";
                            cPtr->capacity += min;//Update capacity
                        }
                        cout << cPtr->destination << " : " << min << endl;
                        min = INT_MAX;
                        Queue.pop_back();

                        continue;
                    }
                }
            }
        }
    }
}
```

## Controlling schedule has maximum flow :

In this part, Residual graph is updated before controlling maximum flow. After this same procedure in the first part is processed. So running time is same  $O(m^2)$ .

### TESTING

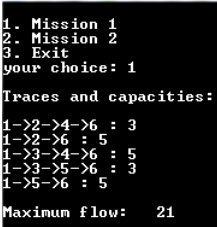
Program has tested with following two scenario:

#### Scenario 1:

Network Info:

```
1 2 10
1 3 8
1 5 5
2 4 3
2 6 5
3 2 3
3 4 10
3 5 3
4 6 8
5 4 3
5 6 10
```

Screen shots about execution of program:



```
1. Mission 1
2. Mission 2
3. Exit
your choice: 1
Traces and capacities:
1->2->4->6 : 3
1->2->6 : 5
1->3->4->6 : 5
1->3->5->6 : 3
1->5->6 : 5
Maximum flow: 21
```

#### Scenario 2:

In this, It is controlled that there is a maximum flow better than this situation with first scenario. Transportation Scheme:

```
1 2 5
1 3 8
1 5 5
4 2 8
2 6 5
3 4 8
4 6 8
5 6 5
```

In this part, the program does not work correctly.



## CONCLUSION

In this project,

- Graph theory and some greedy graph algorithms are investigated and learned.
- Maximum flow problem is investigated and learned.
- Ford-Fulkerson algorithm is learned.
- To experience analyzing algorithm.
- To experience developing efficient algorithm for a problem.
- To experience modifying known algorithm for a problem.
- It was observed that efficiency of well-chosen data structure on running time of algorithm.