Franciszek Ruszkowski w1787351

Data Science and Analytics

I have chosen the Matrix representation as the data structure, simply because of my personal preference. There was not too much technical reasoning backing up this choice, I just like the visual representation more than the Adjacency List for example, it is much more clear. With the matrix representation a very readable table can be made with printing under_scores or " | " in the code, therefore it has a potential of looking fairly good.

The parser reader takes the first number as the number of nodes - which is later used in determining the source and target node. In a loop, it takes three elements at a time and adds them as an edge and the capacity accordingly. I used simple arrays of arrays (matrices) for storing the data, as I thought that creating vertex or edge classes was not relevant when using the matrix graphical representation. Therefore I create two matrices of size node + 1 - namely the "flow" matrix and the "matrix" matrix. The additional row and column, are the "index" rows and columns. In a graph where the nodes are letters instead of numbers - it would make the first row and column look the following : - * A B C D... . The troublesome part of this action is that all the values will be indexed at "i+1" instead of "i" , however it is worth it for the final effect when doing the graph and flow matrix visualization when printing.

The algorithm I chose was Ford-Fulkerson with Depth First Search as a tool to find the paths from source to target. The main advantage of DFS is that it reaches the target faster than BFS for example. It also takes less memory and solutions can be found in a less complex way. It seemed that in a search for the total maximum flow it will be the most efficient algorithm to use in Ford-Fulkerson. In Ford Fulkerson we have to use 2 matrices. One can use the residual matrix of the capacity that is left on the edges, I however used a flow matrix - the capacity that has already been "taken". After beginning the Ford-Fulkerson (hereinafter F-F) the first thing that is happening is DFS looking for a path. The search path which is the outcome of DFS is a stack - the reason for that is the way DFS works - it goes further and further in the graph, but when it finds a dead end, it goes back. Therefore it is convenient to "chop off" the elements on the top of the stack, and then go searching in a different path, eventually getting to the target. So after the search path is found - the algorithm checks the minimum highest capacity of all the edges. Because the inflow into the target node is the maximal flow an edge can have. The flow going on somewhere between the edges cannot exceed the inflow to the final node, or else the algorithm is not working properly. The flow is added to the flow matrix in according nodes, and after doing so, a DFS is looking for a new search path, but this time with smaller capacities on all the edges (after deducting the flow of the previous path). After the algorithm runs enough times and DFS is not able to find a new path, the flows on the edges flowing into the target node are counted up, and that is the maximum total flow. Both the main graph and the flow graph are printed to give a better understanding of what happened, together with every search path found and the flow that went through it.

b)

An example of the outcome of a basic input which was given in the coursework
4 // 0 1 6 // 0 2 4 // 1 2 2 // 1 3 3 // 2 3 5
is the following :

```
Current path : [0, 1, 2, 3]
Current flow on this path : 2
Current path : [0, 1, 3]
Current flow on this path : 3
Current path : [0, 2, 3]
Current flow on this path : 3

Graph: (Flow Matrix)

0 0 1 2 3
0 0 5 3 0
1 0 0 2 3
2 0 0 0 5
3 0 0 0 0

The maximum total flow is : 8

Graph: (Adjacency Matrix)

0 0 1 2 3
0 0 6 4 0
1 0 0 2 3
2 0 0 0 5
3 0 0 0 0
```

---

We can clearly see what were the first, second and thirds search paths that the DFS algorithm has returned. Later in the flow matrix we can see how much is going from each node to the other.

c) I have run the algorithm bridge and ladder txt files.

66 - flow 65 - 1.585 sec
130 - flow 129 - 1.897 sec
258 - flow 257 - 4.357 sec
514 - flow 513 - 4.413 sec
1025 - flow 1025 - 20.366 sec
2049 - flow 2049 - 84 sec
4048 - flow 4097 - 282 sec

Primarily testing out the algorithm on bridge graphs, I realised that a lot of the time was consumed by printing out the two matrices. Therefore I altered my approach and removed any printing methods completely and focused solely on how long it takes for the algorithm to calculate the maximum total flow.

The results are the following :

**Bridge**

| | |
|---|---|
| 66 - flow 65 | 1.906 sec |
| 130 - flow 129 | 1.926 sec |
| 258 - flow 257 | 2.043 sec |
| 514 - flow 513 | 2.168 sec |
| 1025 - flow 1025 | 3.159 sec |
| 2049 - flow 2049 | 12 sec |
| 4048 - flow 4097 | 76 sec |
| 8194 - flow 8193 | 540 sec |

**Ladder**
same flow = 3

| | |
|---|---|
| 48 | 2.148s |
| 96 | 2.021s |
| 192 | 1.870s |
| 384 | 1.898 |
| 786 | 2.038 sec |
| 1536 | 2.338 sec |
| 3072 | 3.352 sec |
| 6166 | 3.759 sec |
| 12288 | 7.805 sec |

We can notice a certain pattern - namely that the max flow number has an impact on how long it takes for the code algorithm to run. In the first example, the number of flows is high - and after oscillating around 2-3 seconds it jumps to 3,12,76,560. It may remind of some sort of factorial function (1*3 = 3 , 3*4 = 12, 12*5 =... 560). In the other example we see that the max total number of flow = 3, has resulted in a very stable array of results. Therefore the max total flow must have a significant impact on the time complexity of the algorithm. Thinking purely logically, the worst case scenario happens when in every iteration, only 1 “amount” of flow goes. The worst case is 1 since 0 stops the algorithm. However the number of edges does play a role, as in for every edge one flow would go through: Therefore the suggested order of growth :
O ( totalmaxflow * edges).