

INFORMATICS INSTITUTE OF

TECHNOLOGY

In collaboration with

UNIVERSITY OF WESTMINSTER

Algorithms

5SENG002C

Coursework

Module Leader’s Name – Mr. Sudharshan Welihinda

Dinuka Piyadigama

UoW ID – 17421047

IIT ID – 2018373

# Algorithmic Approach taken

## Algorithmic Strategy

* Ford-Fulkerson algorithm was used to calculate the max flow of the flow network.
* Breadth First Search (BFS) was used to find whether a path exists from source to sink. The reason for choosing BFS was because BFS always picks up the path with the minimum number of edges. The worst-case time-complexity can be reduced as well. The **greedy** algorithmic approach was taken in this case. (Singh, n.d.)

## Chosen Data Structure & its Traversal Towards Solution

* A LinkedList (queue) has been used for the queue that is created in the Breadth First Search (BFS) method. In BFS *poll* method of the LinkedList was used to return the first element of the queue and remove it from the queue. (Anon., n.d.)
* I have used a 2-dimensional array ([][] graph) to represent the flow network’s graph as a matrix. The 1st index of the array gives the starting node, 2nd index gives the ending node of a link. The value at the 2nd index gives the capacity from the starting node to the ending node. If there is a capacity, a link exists between the two nodes.
* An array ([] parent) was used to store the residual path in BFS.
* When taking inputs from the user a HashMap was used instead of an ArrayList to get inputs because then, the order of entering inputs won’t matter. This was implemented for the ease of coding.

## Pseudocode in plain English

BEGIN

INPUT graph with capacities of links, source, sink of flow network

Initialize the Residual graph from the initial graph and the parent array

max\_flow = 0, max\_integer\_value = 2147483647

WHILE there’s a path from source to sink:

path\_flow = max\_integer\_value

path\_flow = MIN(path\_flow, capacity\_of\_residual\_link)

max\_flow = max\_flow + path\_flow

END WHILE

DISPLAY max\_flow

END

# Methodology for empirically analysing the performance of the algorithm

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input data size | | Time spent to produce the outcome  (in nanoseconds) | Ratio changes in time | log2 ratio of times |
| Nodes | Links |
| 6 | 10 | 6248500 |  |  |
| 12 | 20 | 10739800 | 1.71878 | 0.235 |
| 24 | 40 | 15172100 | 1.4 | 0.146 |
| 48 | 80 | 20936200 | 1.3799 | 0.1398 |

# Conclusions algorithmic performance

~~The log~~~~2~~ ~~ratio of the time spent seems to converge to a constant 0.14~~

According to the code, the highest complexity given is a double loop. This gives n2.

When the number of elements in both arrays in the 2D graph are equal, accessing the elements of the 2D array is n2 as well. This is because the run time is directly proportional to the elements in the array. (Anon., n.d.)

Therefore, Big O = O(n2)

## Graph (Discrete Fourier Transformation)

Time elapsed (in nanoseconds)

**Number of Inputs against Time (in nano seconds)**

Input data size (no. of nodes)

# References

Anon., n.d. *Geeksforgeeks.* [Online]   
Available at: https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/  
[Accessed 10 03 2020].

Anon., n.d. *StackOverFlow.* [Online]   
Available at: https://stackoverflow.com/questions/25512385/algorithm-complexity-in-2d-arrays  
[Accessed 20 03 2020].

Singh, N., n.d. *Geeksforgeeks.* [Online]   
Available at: https://www.geeksforgeeks.org/max-flow-problem-introduction/  
[Accessed 28 03 2020].