INFORMATICS INSTITUTE OF TECHNOLOGY

IN COLLABORATION WITH

UNIVERSITY OF WESTMINSTER (UOW)

B.Eng. (Hons) Software Engineering

5SENG002C.2 – Algorithms: Theory Design and Implementation

Coursework

Module Leader: Sudharshan Welihinda

UOW ID: w1761265

Student ID: 2019281

Student Full Name: Mohammed Nazhim Kalam

Choice of Data Structure and Algorithm

- A LinkedList (queue) is created in the BFS (Breadth First Search) method by making use of the poll() method to remove and get the first element of the queue. Queue is a data structure with both ends open, one end is often used to enter data the other is used to exclude data. Reason for using Queues is due the searching or traversing algorithm used is BFS (Breadth First Search/Traversal). BFS is a searching algorithm which is used for traversing a graph and this uses queues to remember to capture the next vertex to start a search. Reason why *BFS* is used not *DFS* for finding the augmenting path is that BFS promises to find the shortest possible (least number of edges) path from source to sink where as DFS doesn't, this also reduces the worst-case time complexity.

- Algorithm used is Ford Fulkerson. Ford-Fulkerson algorithm is used to find the maximal flow from start vertex to sink vertex. Any edge in a graph has a capacity. Source and Sink are the two key vertices to find the maximum flow between them. The sink vertex will have all inward edges and no outward edges, the source vertex will have all outward edges and no inward edges. The flow on an edge cannot exceed its maximum capacity of flow through that edge and except for the source and sink, any edge's incoming and outgoing flow would be equal. The Greedy Algorithm approach was considered in this case.

Explain of the Algorithm on the smallest benchmark example

- I have created 6 java classes. BFS class to perform the Breadth First Search Operation, Ford Fulkerson class to compute the maximum flow of a graph. GraphADT abstract class containing all the abstract methods for the graph operation such as (generateGraph, visualizeGraph, existsEdges, insertEdges, deleteEdges, insertNode, deleteNode). Graph class which implements the GraphADT class. ReadDataFile class used to read data from the txt file. Runner class where the main program runs.

- Firstly, the main menu is display which asks user the following to perform insert an edge, delete an edge, insert a node, delete a node, check if edge exists, perform max flow search or quit the program

- A 2-D Adjacent Matrix where 1st index represents the starting node and 2nd index represents the ending node/vertex is created. The value at these indexes represents the capacity between the vertices. If it's a 0 it means there is no edge else if it's a positive integer it indicates an edge.

- An instance of the FordFulkerson class is created and graph data, source node, target node data are sent.

- A residual graph is been initialized as the original graph with its capacities since there is no flow in the beginning. (The residual graph is also displayed to the user via console)

- To find the augmenting path BFS is used on the residual graph.

- The created parent[] array is used to store the found path and is filled by BFS.

- The maximum_flow is set to 0 initially, because at start there is no flow considered.

- We will use the BFS traversing method to see if there is a (augmenting) path from source to sink.

- The BFS uses the parent[] array where we can traverse through the found path and find the possible flow through this path by finding the (bottleneck capacity) minimum residual capacity along the path.

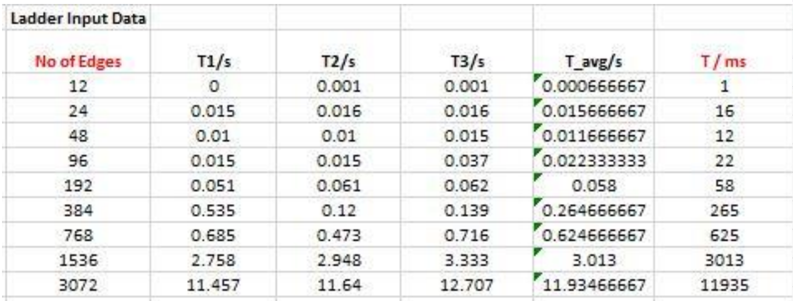
- The Augmenting path found is displayed to the user along with the bottleneck capacity for that path and also displays the updating maximum flow as it progresses to get the maximum flow.

- I also update the residual capacities in the residual graph by subtracting path flow from all (forward) edges along the path and we add path flow along the (reverse/ backward) edges. (at the same the updated residual graph is displayed to the user via console to show the progress).

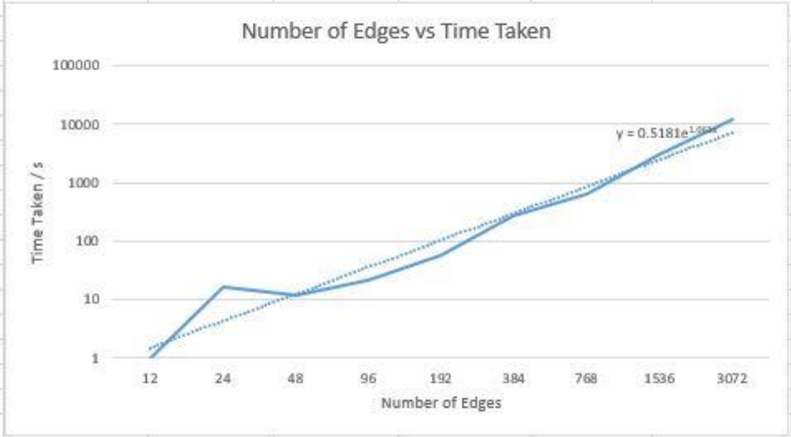
- Finally, we add the found path flow to overall flow.

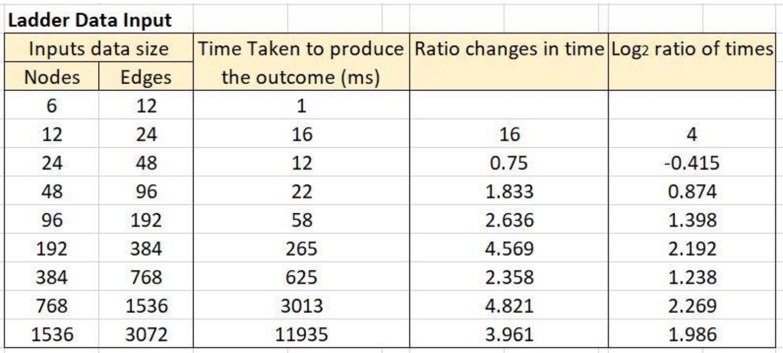
- Once the maximum flow is displayed via the console, the main menu is then displayed again.

Performance Analysis of Algorithm.



Ladder Input Data					
No of Edges	T1/s	T2/s	T3/s	T_avg/s	T / ms
12	0	0.001	0.001	0.000666667	1
24	0.015	0.016	0.016	0.015666667	16
48	0.01	0.01	0.015	0.011666667	12
96	0.015	0.015	0.037	0.022333333	22
192	0.051	0.061	0.062	0.058	58
384	0.535	0.12	0.139	0.264666667	265
768	0.685	0.473	0.716	0.624666667	625
1536	2.758	2.948	3.333	3.013	3013
3072	11.457	11.64	12.707	11.93466667	11935


Number of Edges vs Time Taken



Ladder Data Input				
Inputs data size		Time Taken to produce the outcome (ms)	Ratio changes in time	Log2 ratio of times
Nodes	Edges			
6	12	1		
12	24	16	16	4
24	48	12	0.75	-0.415
48	96	22	1.833	0.874
96	192	58	2.636	1.398
192	384	265	4.569	2.192
384	768	625	2.358	1.238
768	1536	3013	4.821	2.269
1536	3072	11935	3.961	1.986

The $\log_2$ ratio of the time spent seems to converge to a constant roughly around 2, which means that the time complexity comes down to a maximum of n^2 and also according to the code the highest time complexity is given by the double nested loop, when accessing the elements of the 2D array we will be able to see a double nested for loop, (when creating the residual graph) which means the time complexity comes down to a maximum of n^2

Therefore, this will be the following Big O notation = $O(n^2)$

[This applies for worst, average and best case]