**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**

**-** The data structure I choose is a LinkedList (queue) that is created in the BFS (Breadth First Search) method. Queue is a data structure with both ends open, indicating that one end is often used to enter data the other end is often used to exclude data. The reason for using Queues is due the searching or traversing algorithm used is BFS (Breadth First Search/Traversal).BFS is an algorithm which is used for traversing a graph and this uses queues to remember to capture the next vertex to start a search. The reason why *BFS* is used not *DFS* for finding the augmenting path is that BFS promises to find the shortest possible path from source to sink where as DFS doesn’t.

**-** Ford Fulkerson is the algorithm that was used. In a given graph, the Ford-Fulkerson algorithm is used to find the maximal flow from the start vertex to the sink vertex. Any edge in a graph has a capacity. Source and Sink are the two key vertices that are given to find the maximum flow between these vertices. The sink vertex will have all inward edges and no outward edges, while the root vertex will have all outward edges and no inward edges. There are also some important constraints to be followed which are 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.

**Explain of the Algorithm on the smallest benchmark example**

**-** I have created 6 java classes with it’s respective functions. BFS class to perform the Breadth first search, FordFulkerson class which contains the algorithm to find the maximum flow, Graph class used to create the Adjacent Matrix for given graph data, ReadDataFile class to read data from a txt file, Runner class where the main program lies and UserOption class which contains function for user menu functions

**-** The Graph class returns a 2-D Adjacent Matrix where 1st index represents the starting node and 2d index represents the ending node/vertex. The value at these indexes represents the capacity between the nodes. If it’s a 0 it means there is no edge else if it’s a positive integer it indicates an edge.

**-** Runner class contains additional logics where user is able to create and delete a node from the graph and also able to add and remove an edge from the nodes in the graph.

**-** 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.

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

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

**-** The maximum\_flow is set to 0 initially.

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

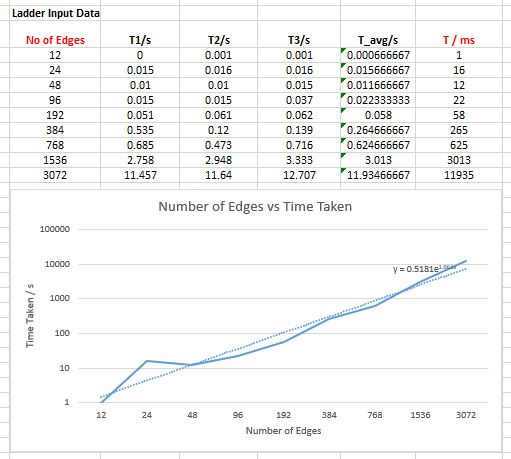
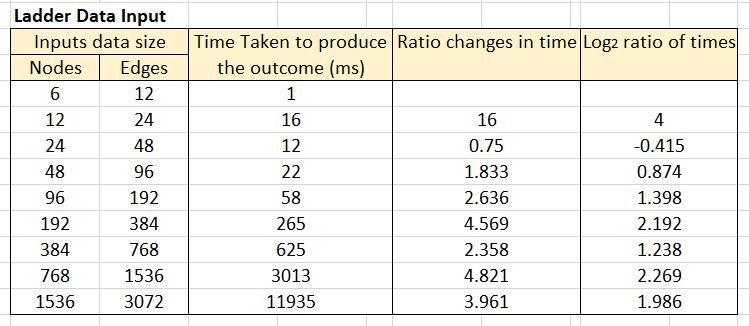
**-** The BFS uses the parent\_arr[] array where we can traverse through the found path and find the possible flow through this path by finding the (bottleneck capacity) minimum residual capacity along the path.

**-** Once the path is found I display the path to the user via the console to show the progress taken to get the maximum flow.

**-** I also update the residual capacities in the residual graph by subtracting path flow from all edges along the path and we add path flow along the reverse edges.

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

**-** Once the maximum overall flow is calculated it is then returned back to the Runner class to display it via the console.

**Performance Analysis of Algorithm.**

The log2 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 n2 and also according to the code the highest time complexity is given by the double loop, when accessing the elements of the 2D array you will be able to see a double nested for loop which means the time complexity comes down to a maximum of n2

Therefore, this will be the following Big O notation = O(n2)