**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

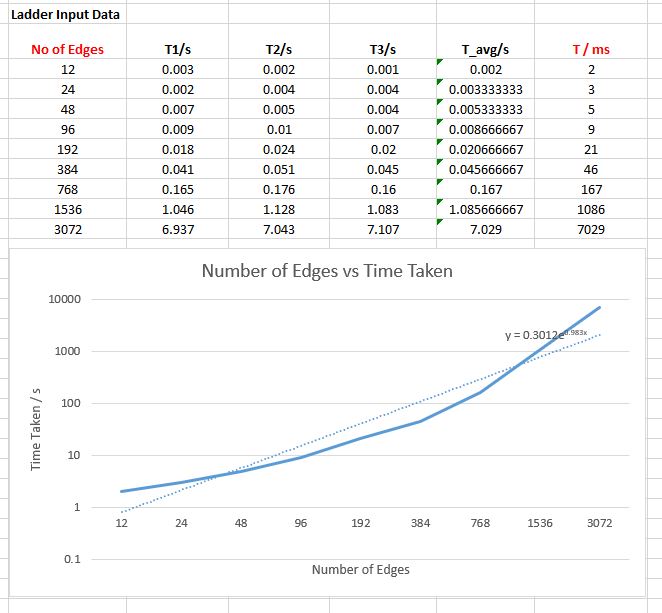
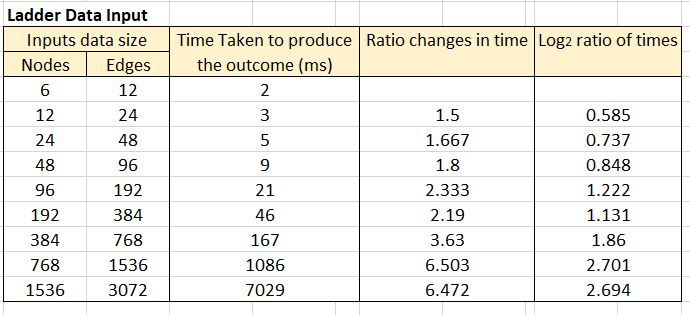
**Choice of Data Structure and Algorithm**

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**

* A Runner class and a FordFulkerson class are the two classes I've made. A method called poll() from the LinkedList Queue has been used in BFS, this is to get/return the first element of the queue and remove it’s from the queue as well.
* I built a method called "readDataFromFile()" in the Runner class, which reads all of the input lines from a file and adds them one by one to a list. Then using the list of lines collected I created the adjacent graph matrix in order to represent the capacities on each edge in between the vertices (network flow). Once the 2-dimensional graph matrix is created it is returned to the main program. It’s really important to note that the 1st index of the 2-D graph array represents the starting node and the 2nd index represents the ending node or vertex.
* The value at these indexes represents the capacity between the starting vertex and the ending vertex. If the value is 0 it means there is no edge which also means no capacity and if there is a non-zero value then it means that there is a capacity or an edge in between those starting and ending nodes.
* Moreover, the Runner class also contains logics such as asking the user to insert a new edge or remove an edge from the current graph as well and an option to quit the program as well.
* An Object of the FordFulkerson class has been instantiated in the Runner class and have passed the adjacent graph matrix as well as the source and target node to a method named as fordFulkerson.
* This fordFulkerson method is the main method where all the logic happens to find the maximum flow and then returns the value to the user (Runner class and then displays the result to the user).
* I initialize the residual graph as the original graph which I sent as a parameter to the fordFulkerson method. (This is because there is no initial flow hence the initial flow and initial residual capacity is equal to original capacity).
* The parent\_arr[] is used to store the found path and is filled by BFS.
* The maximum\_flow is set to 0 initially.
* In order to find the augmenting path I have used the BFS approach on the residual graph.
* We will use the BFS traversing method to see if there is a path from source to sink.
* The BFS has also built a parent\_arr[] array where we can traverse through the found path and find the possible flow through this path by finding the 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 2.7.

However, 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)