***CITS2200 Project Report***

***By Adrian Bedford 22973676 & Siwei Lin 22967534***

allDevicesConnected:

This method uses BFS to achieve its results. A queue is populated with the starting node of the graph, after which the graph is looped over to begin checking for descendants. In doing this we remove and return the current node from the queue, allowing us to terminate the loop once no more connected descendants can be found. If the current node in the queue is said to have children according to adjlist, they are also added to the queue. We keep track of all reachable nodes in a separate Boolean array where if an index position is false, its relative node was not reachable and if it is true it is connected in some way to the rest of the graph. Once the BFS gets to a stage where it has found all connected nodes and the queue is empty, the while loop terminates, and a new loop starts. This loop checks the Boolean array to see if there are any nodes that remained disconnected and if so, it returns a final answer of false, otherwise true.

In the worst case, the breadth-first search must traverse the entire length of the adjlist to find out that all nodes are connected, thus it is directly related to the size of the input data, O(N).

numPaths [1]:

This method uses BFS to achieve its results. It follows a similar process to allDevicesConnected, where it populates a queue with the initial node, only breaking once there are no children left to add to the queue. In a nested for loop, numPaths iterates over all elements, adding children that have not been visited. If a node has not been visited yet we assign it the distance travelled to get there from its parent plus 1, and we also change the number of paths encountered for that node to be equal to that of its parent. If the distance held inside the array for the child node is equal to the parent node plus 1, we know that the answer is simply the combination of the number of paths held for the child and the parent. Then we simply return the numPaths array for the specific index of the node.

A

closestInSubnet

This method primarily uses BFS to achieve its results. It is separated into 3 sections where closestInSubnet calls on two other methods to do the bulk of its calculations. The minDist method begins to perform a BFS in a similar manner to the way in which allDevicesConnected operated, but additionally calls on the isSubnet method which serves entirely to compare the query and addrs arrays to confirm that the current IP address has a valid prefix. Once minDist has confirmed that the current address is valid, it continues to perform BFS. It loops through the elements of adjlist and checks whether each node is yet to be visited and if not edits a Boolean array, updating its respective index to say it has now been visited. The position of the current node is now updated as the BFS has moved through the graph, and iteration is performed on an integer array increasing the current total distance of a path with a specific parent node (dist). Once the queue is empty, all reachable nodes have been seen, the initial loop breaks and the array containing iterables is returned to the main closestInSubnet method.

A

maxDownloadSpeed [2] [3]:

This method makes use of the Ford-Fulkerson algorithm, implementing the Edmonds-Karp algorithm which is a BFS implementation. Two for loops populate an array to be used to hold the max capacity of flow from the starting to destination node, assigning respective speeds to each node. It calls dstReachable, which is the Edmonds-Karp algorithm. It goes by a similar process to the other methods where it populates a queue with the starting node, and then proceeds to add all possible connected descendants to determine if a specific destination is reachable. If the possible flow of a certain path is determined to be 0, it immediately returns that it is unreachable. Moving back to maxDownloadSpeed, within the while loop it finds the minimum capacities for all possible paths returned from dstReachable, and then proceeds to add them all up to establish a maximum flow.

The complexity of this method is O(V\*E2), where V is the number of nodes in the graph and E is the number of edges. The length of any simple path is bounded to the number of nodes, V, and paths to s will have one saturated edge, meaning that to not undercut flow a new and longer path must be found, hence E2.

***Bibliography***

[1] https://www.geeksforgeeks.org/number-shortest-paths-unweighted-directed-graph/

[2] https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson\_algorithm

[3] https://cp-algorithms.com/graph/edmonds\_karp.html