Kristijan Korunoski 22966841

CITS2200 Project 2021 Report

Implementation of allDevicesConnected

First some variables are created. An empty queue. An integer, “*size*", being the number of devices in the network. A boolean array, “*visited*”, storing whether a device has been visited or not. A boolean, “*allVisited*”, storing false as long as any device has not yet been visited. And an integer, “visitedCount” initially set as 1. For all integers between 0 and “*size*” the value of the integer position in the array “*visited*” is set to false. The device indexed 0 is then added to the queue and the index 0 of “*visited*” is set to true, while the queue is not empty, an integer “*d*” is set as the integer at the front of the queue, and the front integer of the queue is removed, this means we are viewing device “*d*”. For each device “*d*” is connected to (accessible via “*adjlist*”), if the device has not been visited (checked via “*visited*” array) the device is added to the queue and the value of “*visited*” at index [connected device] is set to true, and visitedCount is incremented. This process loops with a new device giving “*d*” a new integer value. This is a breadth first search and visits all connected devices as the loop only ends once the queue is empty. We then check if the number of devices is equal to the number of visited devices and set allVisited to true if so. The boolean “*allVisited*” is then returned representing all devices being connected when the value of “*allVisited*” is true. This algorithm must be correct as all devices that can possibly be visited are visited and counted.

As this algorithm uses a breadth first search it completes the search in time proportional to the number of devices plus the number of connections in the worst case scenario. This means there is a time complexity of O(D + L) or O(N).

Implementation of numPaths

First the method checks if the source and destination are the same device, in that case the number of paths is returned as 1. If the source and destination are not the same device some variables are created. An empty queue. An integer, “*paths*”, set to 0. An integer, “*size*”, the number of devices in the network. And a boolean array, “*visited*”, storing whether a device has been visited or not. For all integers between 0 and size the value of the integer position in the array “*visited*” is set to false. The source vertex is then added to the queue and while the queue is not empty, an integer “*d*” is set as the integer at the front of the queue, and said front integer of the queue is removed, this means we are viewing device “*d*”. For each connection which device “*d*” can make to any other device if the receiving device (when “*d*” is the transmitting device) is the target destination the value of “*paths*” is incremented. Else if the receiving device has not yet been visited it is added to the queue and the receiving device is set to having been “*visited*”, this ensures that longer paths are not counted as required and the algorithm does not break if a loop is encountered. This is a breadth first search and visits all connected devices as the while loop only ends once the queue is empty. Once the loop ends “*paths*” is returned representing the total number of different paths which can be taken from the source to destination. This algorithm must be correct as all paths that can possibly be taken are taken and counted.

As this algorithm uses a breadth first search it completes the search in time proportional to the number of devices plus the number of connections in the worst case scenario. This means there is a time complexity of O(D + L) or O(N).

Implementation of closestInSubnet

First some variables are created. An integer “*numQ*", the number of queries which will be tested. An integer “*numD*”, the number of devices in the network. An empty queue. An integer array “*hops”*, storing how many hops for a query to find a device in it’s subnet from the source. An

integer array “num*Hops”*, which stores how many hops have been taken to reach a device indexed in the array. And a boolean array visited, storing whether or not a device has been visited. Each index of “*hops”* is set to Integer.MAX_VALUE and then a for loop is creating for each query. For each query the queue is reset to be empty and each index of visited and “*numHops*” is set to false and 0 respectively. The source device is added to the queue, and while the queue is not empty an integer “*v*” is set as the value of the first integer in the queue and the integer in the front of the queue is removed. The device is checked whether it is in the subnet and if so “*hops*” at the index of the current device is set to be equal to “*numHops”* at the same index. The while loop then breaks. In the case that the device is not in the subnet, for each device which can be reached by the current device, if said device has not already been visited it will be added to the queue and the visited value for that device is set to true. The value of “*numHops*” is incremented as well. The while loop ends and repeats until there is no more devices in the queue unless it was ended before when a device was located, and at this point the process repeats for the next query until all queries have been checked. This algorithm must be correct as the closest device to the source is located and every device is checked to be in the subset.

As this algorithm uses a breadth first search it completes the search in time proportional to the number of devices plus the number of connections in the worst case scenario. This means there is a time complexity of O(D + L) or O(N). This is nested in a for loop over the number of queries adding another layer of complexity of O(Q). This gives a total time complexity O(N + Q).

Implementation of maxDownloadSpeed

First the method checks if the source device is the same device as the destination, in that case -1 is returned as there can be no download speed between the same device. If the source is not equal to the destination some variables are created. An integer “*maxSpeed*” set by default to 0. An integer “*numD*”, the number of devices. An empty queue. An integer array “*key*” which will store the magnitude of the flow through a device. And a boolean array “*visited*” storing whether or not each device has been visited. All indexes of “*key” and “visited*” are set to 0 and false respectively. The source device is then added to the queue and its “*visited*” value is set to true. While the queue is empty the front integer in the queue (representing a device) is dequeued and stored as “*v*” we then check if there is no output from that device, if so the “*visited*” value of that device is set to true as it has been visited and no more flow can enter that device. If the “*v*” has a “*key*” value of -1 nothing occurs, else for each device connected to “*v*” if the device has not yet been visited if there is a speed greater than 0 from “*v*” to the device there is a number of outcomes depending on the situation. ___ If the “*speeds*” value representing the flow into the device is less than the “*key*” value representing the flow from “v” and the device is the destination then the value “*maxSpeed*” is incremented by the value “*speeds*” represents between the two devices, then “*key*” for “v” is decreased by the value of “*speeds*” between those devices and if “*key*” at “v” then equals 0 it is set to -1, and “*speeds*” for that connection is set to 0. If the “*speeds*” value is less than the “*key*” value but the device is not the destination, the “*key*” value for that device is incremented by the value of “*speeds*” for that connection and that device is added to the queue, then “*key*” for “v” is decreased by the value of “*speeds*” between those devices and if “*key*” at “v” then equals 0 it is set to -1, and “*speeds*” for that connection is set to 0. If “*key*” at “v” is 0 and the next device is the destination “*maxSpeed*” is incremented by the value of “*speeds*” between v and the device, then that value of “*speeds*” is set to 0. If “*key*” at “v” is 0 and the next device is not the destination, the “*key*” value for that device is incremented by the value of “*speeds*” for that connection, the device is added to the queue, and “*speeds*” for that connection is set to 0. If “*key*” at “v” is greater than “*speeds*” for the connection to the next device, and the next device is the destination, “*maxSpeed*” is incremented by the value of “*key*” at “v”, “*speeds*” for that connection is decreased by the value of “*key*” at “v”, and “*key*” at “v” is set to -1. f “*key*” at “v” is greater than “*speeds*” for the connection to the next device, but the next device is not the destination, the “*key*” value for that next device is set to the value of “*key*” at “v” and that device is added to the queue, the value of “*speeds*” for the connection is decreased by the value of “*key*” at “v” and “*key*” at “v” is set to -1. All connections will be checked and will satisfy one of the above conditions, then the while loop repeats until the queue is empty. Once the queue is empty the maximum flow algorithm is complete and so if -1 has not already been

returned due to the source being the destination then “*maxSpeed*” is returned, representing the maximum possible download speed. This algorithm must be correct as all flow paths are accounted for and the flow which exceeds the output limit is accounted for at each device as well as flow that is below the output limit. The algorithm takes into account both bottlenecks and sub par speeds to ensure that the correct maximum speed is returned.

As this algorithm uses a breadth first search it completes the search in time proportional to the number of devices plus the number of connections in the worst case scenario. This means there is a time complexity of O(D + L) or O(N). However inside the breadth first search a method with a for loop is called. This for loop has the size of the number of links for one device, over the entire breadth first search this for loop conceptually goes over all links giving complexity O(L). This O(L) nested in O(D + L) gives a complexity O(D L^2)