CITS2200 Project Report 2021 Semester 1

Jasmine Ngiam Yi Chen (22739879)

Tuguldur Gantumur (22677666)

CITS2200 - Data Structures and Algorithms

May 21, 2021

Part 1: allDevicesConnected()

Method explanation:

Determine if all of the devices in the network are connected to the network. Devices are considered to be connected to the network if they can transmit (including via other devices) to every other device in the network. If all devices in the network are connected, then return true , and return false otherwise.

Correctness:

This method is solved by applying Breadth-First Search algorithm (BFS). It uses Queue to find the connected vertices and its offer and poll method can be done in constant time. First, the length of the adjlist is checked whether there are no devices in the adjlist, return false if there are no devices in the adjlist since there is no device in the network. An array of keys is declared and each element in the array represents the device id for easy retrieval when checking whether each device can transmit to every other device in the network. Then, a boolean array is declared to determine whether a device has been visited and is initialised as false for all the devices. A Queue is declared to store the device to be searched for its connected devices and the BFS starts by inserting (offer) the first device (device 0 in this case) to the queue and assigning visited[0] as true. Then, the method goes into a while loop to find all the connected devices by taking out (poll) the front device (device 0) in the queue and traverse through the adjlist of each device. If the device taken out from the queue is present in the adjlist of other devices, it means that they are connected and the connected device id is inserted into the queue to check whether it is connected to every other device in the network. At the end of each while loop, the device taken out is assigned as visited. At last, a for-each loop is used to check whether each device is visited, allDevicesConnected return true if all the devices are visited, and return false if there is one unvisited device.

Complexity:

The first if-condition uses a constant time, O(1). Then to store every device id in the array keys, it uses O(D). The initialisation of each boolean array visited as false also takes O(D) time. In the while loop, traversing through all the devices in the network takes O(L) time (Datta 2021). The assignment of each device visited as true takes a constant time. At last, a for-each loop used to check whether each device is visited takes O(D) time. Therefore, the total time for this method is O(1+D+D+L+1+D) = O(D+L) = O(N).

Part 2: numPaths()

Method explanation:

Determine the number of different paths a packet can take in the network to get from a transmitting device to a receiving device. A device will only transmit a packet to a device that is closer to the destination, where the distance to the destination is the minimum number of hops between a device and the destination.

Correctness:

BFS is also used in this method. Its implementation is similar to allDevicesConnected to find its connected device but with a given source and destination to find the availability of path from the source to the destination. First, a condition is checked whether the source device is equal to the destination device, if it is, terminate the method and return 1 as there is no connected device involved. Next, an int numpaths is initialised to count the number of paths available from source device to destination device. A boolean array visited is declared to check whether a device has been visited and initialised as false for each device. A Queue is declared and the BFS starts by inserting (offer) the given source device and taking it out (poll) in the while loop. During the loop, each adjlist of the device taken out from the queue (int seen) is examined. If the device in the adjlist of seen is equal to the destination device, this indicates a path and

numpaths increases by one, else if the device has not been visited yet, it is assigned visited as true and inserted (offer) in the queue to examine its adjlist. The while loop continues until all the connected devices to the source device are examined.

Complexity:

The first if-condition to check whether the source device is equal to the destination device takes a constant time, O(1). The declaration and initialisation of numpaths takes a constant time, O(1). A boolean array visited is declared and initialised as false for each device takes O(D) time. The manipulation of Queue requires O(D+1) time where each device is offered and polled in constant time and all the devices are offered and polled into the queue. After a device is polled from the queue, its adjlist is checked for its connected devices, hence using an O(L) time to examine all the connected devices (Datta 2021). For each connected device, it is checked whether it is the destination device and increases the number of paths by one. This is done in constant time for an addition. Therefore, the total time for this method is O(1+1+D+D+1+L) = O(D+L) = O(N).

Part 3: closestInSubnet()

Method explanation:

Compute the minimum number of hops required to reach a device in each subnet query. Each device has an associated IP address. An IP address is here represented as an array of exactly four integers between 0 and 255 inclusive (for example, {192, 168, 1, 1}). Each query specifies a subnet address. A subnet address is specified as an array of up to four integers between 0 and 255. An IP address is considered to be in a subnet if the subnet address is a prefix of the IP address (for example, {192, 168, 1, 1} is in subnet {192, 168} but not in {192, 168, 2}). For each query, compute the minimum number of hops required to

reach some device in the specified subnet. If no device in that subnet is reachable, return Integer.MAX_VALUE .

Correctness:

First of all, an array named hoops is initialised with size queries.length, this array will store the hoops required to find the address IP which contains the query subnet from the source IP (src).
Another array choops is initialised to store the number of hoops required to reach each device in the network by calling the private method bfshoops.

Afterwards, we initialise a boolean variable called found which will indicate whether there is a match between the current address and the subnet query. Within the loop to traverse each item, a condition is implemented for the case when there is no subnet prefix within the query, in this case the index at hoops array will be set to 0. After the condition we will traverse through the array addresses to see if there is a match between the query and the address at each index. A private method isSubnet() is implemented, this method uses a the function `Arrays.equals(a, aFromIndex, aToIndex, b, bFromIndex, bToIndex)`, which is used to determine if the address contains the subnet prefix. If a match is found at the current device index(j), the number of hoops for queries[i] is equal to the choops[j] which is the number of hoops required to reach the device j from src.
If a match is not found at the current index and the current address has not been visited prior, according to the function specifications, we will increment the value of hoops within the private method bfshoops(). Alternatively, if all addresses have been visited and there still has not been a match for the query, the value of hoops at the index will be set to Integer.MAX_VALUE, and the final state for the variable found will be set as false.

Lastly, we will return the updated array hoops which will now contain the number of hoops required to reach each subnet from the source.

Complexity:

The solution implements a time complexity of O(NQ), where N represents the number of addresses and, Q the number of Queries.The first iteration through the array queries takes D(Q) complexity as it iterates through the array queries. Each conditional statement, boolean statements and arithmetic operations take a constant time, O(1). The iteration through the network will take O(N). Resulting in an overall time of: O(the length of address * the length of queries) = O(NQ)

Part 4: maxDownloadSpeed():

Method explanation:

Compute the maximum possible download speed from a transmitting device to a receiving device. The download may travel through more than one path simultaneously, and you can assume that there is no other traffic in the network. If the transmitting and receiving devices are the same, then you should return -1.

Correctness:

This method implements Edmond Karp's maximum flow algorithm, which is a variation of Ford Fulkersons' Max-Flow Min-Cut algorithm implemented using a BFS. The solution to this method was inspired by an article written by the website GeeksforGeeks (2021) which implemented the Edmond Karps' Algorithm using an adjacency matrix.

Firstly, an int variable max is initialised to be used and returned at the end of the function. Afterwards, we implement a condition `if (src == dst)` which will return -1 if true, this condition ensures that the maxspeed we are seeking is not derived from the speed it takes for the device to reach itself. We created a 2D array using which holds the capacities of [adjlist.length][adjlist.length], this array is implemented with the intention of using it as a residual adjacency list. The residual list is intended to indicate additional possible flow in the network, and we initialised it as empty as there is no initial flow.

To find all augmenting paths, all possible paths from the source device to the destination device, we have used a BFS from the private method: private boolean bfs() which is passed the adjacency list, the residual list, the source device id, the destination device id, and a parent array as parameters. The parent array is used to traverse through the found path and find possible speed by finding the minimum capacity along the path. Following this traversal, we need to update residual capacities in the residual list by subtracting the path flow from all edges from the discovered paths, and add the flow along the reverse edges as we may need it later to send flow in the reverse direction.

Lastly, every speed we have found from the augmenting paths will be added to the variable max, which is returned at the end of the function.

Complexity:

The above algorithm is referenced to the Ford-Fulkerson Algorithm for Maximum Flow Problem by GeeksforGeeks (2021) and it uses the time complexity of O(VE^2), where V is the number of Vertices and E is the number of Edges.

Each declaration, initialisation, conditional statements, boolean statements and arithmetic operations take a constant time, O(1). A 2D array, residual, is declared and initialized according to the 2D array, speeds, which takes O(DL). The while loop with condition returned from the private boolean bfs method takes O(L) time [The bfs method takes O(D) to initialise all devices visited as false, then executes a standard BFS with time complexity of O(DL); total time of O(DL)]. This while loop is repeated until there is no

residual path available which is at the worst case, in O(DL), the longest path that involves all the devices and going through all the connected devices. Then the for loop to get the maximum flow capacity from source to destination takes O(L) time. The updates of residual capacities take O(L). Overall, the time complexity for this method is O(DL+D(DL)+L+L) = O(DL^2+DL+2L) = O(DL^2).

Reference List:

Datta, A 2021, Dfs, lecture notes distributed in CITS2200 Data Structures and Algorithms. Available from: https://teaching.csse.uwa.edu.au/units/CITS2200/Lectures/newLectures/dfs.pdf.

GeeksforGeeks, 2021. *Ford-Fulkerson Algorithm for Maximum Flow Problem.* Available from: https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/. [19 May 2021].