CITS2200 My Project Report

By: Fabian Scheffler (22987128) and Ching Chun Liu (22660829)

Public boolean allDevicesConnected(int[][] adjlist)

This method checks if all the devices are connected within the network. This uses a simple Breadth First Search (BFS) approach. We have made a queue that stores the vertex of devices yet to be checked. Queue being FIFO allowing the algorithm to check level by level without missing any devices. We have kept a Boolean of visited node that represents whether the node has been visited before or not. This stops same node form getting queued multiple times. Current_level integer keeps count of the number of devices visited with the BFS. If the number of devices visited does not equal to the total number of devices in the network then it returns false, otherwise true. We also added a boundary conditional if the adjacency list contains a device with int outside the whole network as well as if device integer is negative.

Space Complexity:

The space complexity is always smaller than the number of devices. As in the worst case, the queue will have D-1 length, where the first device is connected to every other device in the list.

Time Complexity:

The time complexity is O(D+L) as for most operation each device will be visited at least once. If we were given an adjacency matrix, the complexity will be different, however with adjacency list, the algorithm is traversing each device, only visiting links that are connecting the devices. Therefore, it is D + L, the number of devices plus the number of links for each operation. Since D + L = N the overall time complexity in terms of N will be O(N).

Public int numPaths(int[][] adjlist, int src, int dst)

Returns only the paths of the minimum distance from the source to the destination. Level-order traversal so it will find the minimum distance on the first path that it locates. Now that the minimum is set it will only consider paths with equal or smaller number of hops to reach the destination. Initially sets all the distances to -1 to indicate that all the devices haven't been visited. Distance of -1 represents an unvisited device. Paths keeps track of all the paths with the minimum number of hops and returns it at the end of the for loop.

Space Complexity:

Instead of using another visited Boolean we have integrated that function into the distances array by setting the distance to -1 if not visited and anything other than -1 to count as visited.

Space complexity will be O(N).

Time Complexity:

Will be in O(D + L) as for the worst case it will visit all the devices once. Similar to allDeviceconnected, this algorithm in the worst case will visit all devices and all the links to check for possible paths. Once it found the first possible path, it will set the minimum number

of hops as the amount it took for that path. However, it still has to check for all possible path with that minimum number of hops. This results in the complexity to be D + L and since D + L = N, the complexity will be O(N).

Public int[] closestInSubnet()

This method is once again a BFS but slightly modified as well as using a hash map in order to access the queries in constant time. In the first stage it creates a hashmap of all the queries as the key(in format of string) and the value is set to the index corresponding in the querydis array. In the second stage this algorithm checks each level and at each level it will queue all its adjacent devices. It keeps track of all the devices visited with a Boolean array called visited[], this stops the same device to be visited multiple times. There is also a if statement that only changes the distance to that device if the new distance will decrease the minimum distance to that node, or device. When the distance is changed it will update the querydis array using the helper method setquerydis. This setquerydis method will update the querydis by testing if each possibility that the device is a subnet of one or more of the queries(as I also included the case if we have multiple of the same queries). It then deletes this key if it does find a subnet in order to not repeat this.

Space complexity:

In order to reduce the time complexity it has significantly increased the space complexity. In order to cover the case that all the queries are not unique we have added an arrayList for each of the values in the hashmap.

O(N + N + Q + N) = O(3N + Q) = O(N + Q). As this accounts for the space required for querydis, distance, visited and the hash map.

Time complexity:

Since it is a level order traversal to find the minimum distances it will visit each node at least once which will be O(D + L) = O(N) as previously discussed . We also must include that the hashing process will take O(4Q) as it has to iterate through each query and each query will have a max of length 4. It can only iterate through the queries once during the getting shortest distances as we have deleted the key once it has been set into the querydis to avoid looking over the same queries. Then the setquerydis will have a time complexity of O(5) as it will iterate throught the length of the address as well as the case of a empty query, therefore 4+1 = 5. Putting this all together we get O(N + Q + 4Q + 5) but since it is time complexity we are able to drop all the constant as we are taking the “limit”. So this time complexity now becomes O(N + Q).

References: https://www.geeksforgeeks.org/shortest-path-unweighted-graph/

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.htm

Public int[] maxDownloadspeed()

This algorithm is a modified version of the relabel-to-front algorithm. Following the same invariants in a generic push and relabel algorithm, the source height will always be length of adjacency list and the destination will always be 0. Also, any data cannot flow too steep, meaning that at any step, device can only push to its neighbour if the device's height is neighbour's height plus 1. Each node has an associated height and excess amount. The height determines which node can be pushed to or not. The excess is the excess flow of the node. There is also a flows and capacity matrix. The flows matrix represents the flows between each of the edges and the capacity is the capacity between each of the edges. It first initializes by setting all the heights to 0 and the height of the source to the number of devices (D). It then makes a discharge list of size D – 2 as it does not include the source and destination. This algorithm has a discharge phase where it will continue to relabel and push till the excess of the target node is 0. The relabelling will consider all the adjacent nodes around that point, it will find the neighbour with the smallest value for height and adjust the current device to that value plus 1. For the push method it only can push if the height of the node it is pushing to is -1 of the height of the node we are pushing from. During the execution if a node is relabelled after the discharge phase, then we move this node to the front of the list, hence the relabel to front. At the end, the excess of the destination node will represent the max flow.

Space complexity:

Cap, flows and neighb matrixes will have a space complexity of D^2. Excess and height will have a upper bound of D. So this will mean that the in the worst case the space complexity will be $O(D^3)$.

Time complexity:

In the algorithm the relabel operation looks at the height of all the neighbours and compare all of them, it has the complexity of O(D) in the case where the current device is connected with all other devices available so this will be the upper bound for this instance. Discharge operation have to check all the neighbours again to see if there's possible places push and if there is no place to push the relabel operation is executed. Therefore, the complexity of the whole discharge cycle will be $O(D^2)$. Finally, there is a while loop that will look at all the nodes and will discharge each of those nodes and for worst case it will have a complexity of D. This is why including the relabel operation, discharge and the checking of each devices we will get O(D * D * D) which is $O(D^3)$.

References: https://www.geeksforgeeks.org/relabel-to-front-algorithm/#:~:text=Time%20Complexity%3A%20Runs%20in%20O,(V2E)%20time.

http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0165.html

The second references pseudo code was very helpful as well as the diagram.