CITS2200 Project

by Ron Zatuchny (22984076) and Divyanshu Siwach (22912646)

Question 1

The Algorithm

The following question asks us to find an algorithm that would check whether all the devices are connected or not. By definition, if all the devices are connected, every device(vertex) should be able to transmit to any other device and receive from any device, therefore all the devices should be in one strongly connected component. This suggests that we need to find an algorithm that will check whether all the devices belong to a single strongly connected component.

This could be easily done using DFS, we should start the search from a given device (any device will do, in the code, we start from device 0), we will have two additional arrays, one is recording whether a vertex was visited (named visited), and the other will specify what is the vertex with the lowest index that belongs to the same strongly connected component as this vertex (named connected). All the visited array values will be initialised to 0(means not seen) except for index 0 which will have the value 1(means seen as we start from this index). For the connected array, we will initialise every value to its index in the array, as every vertex is a strongly connected component with itself. Then we will call recursively the DFS function on every element if it was not yet seen but connected to the vertex we are currently at. After we called (or did not call) the DFS function we will check whether that vertex has a visited value of 0, in case it does we will make the visited value of the current vertex 0 as well. After the process is over, we will check whether all the values in the array visited are 0, in case that is true, we will return true and false otherwise.

Why does this algorithm work?

-Every device can transmit to any other device only if all the devices are strongly connected. (by definition)

-We start our search from vertex 0, therefore every vertex we will see can be reached from 0, in case we get back to zero from any vertex, that means this vertex could both be reached from zero and also transmit from zero, therefore both devices belong to a single strongly connected component(and we will change the connected value to 0), that means that every vertex that we visited to get to that specific vertex also could be reached from zero(explained above) and could also transmit to zero(by transmitting to the vertex connected to vertex 0 either directly or indirectly), therefore all part of a single strongly connected component

-In the case above every device in the strongly connected component can transmit to zero(directly or indirectly) and then the ‘message’ could be transmitted to any other device in the strongly connected component from 0(directly or indirectly)

-In case we did not see a vertex, 0 could not transmit to it (neither directly nor indirectly), therefore could not be part of a strongly connected component with 0 and therefore not all devices are connected

-In case a device was seen but could not transmit to any device that could transmit to 0(either directly or indirectly), it cannot transmit to 0 and therefore not part of the strongly connected component and therefore not all devices are connected

Time complexity

DFS will take in the worst-case scenario the number of vertices + the number of edges, as we will iterate through all the vertices and all the edges(through the outer array in the output(every vertex) + the inner array for every outer array(every edge) then we will iterate through the array of the strongly connected components, to check whether all of its contents equal zero, we will iterate through the full array in the worst-case scenario(all the vertices except the last one will have to be 0) the complexity will be the number of vertices.

Therefore:

O(L+2D)=O(L+D)=O(N)

Question 2

The Algorithm

-To know to which devices a device should/shouldn't transmit, we need to know the shortest path from this device and the devices it is connected to to the destination, we will have to find those first and store all of those in an array(called distances for example) and initialise the value of the destination index(given as a parameter to the function) to 0 and the rest as the maximum value an integer could take(maximum distance, therefore no vertex will have a greater distance than that and therefore no vertex will try to transmit through that vertex in case there is no way to get from that vertex to the destination vertex) in the distances array.

To get all the minimum distances from all the vertices to the destination vertex, we will reverse the directions of all the edges and get a 'reversed' graph(represented as a list of integer stacks). This will be done by iterating through all the arrays inside the adjacency list given. In the new 'reversed' adjacency list, we will push a new value to the stack with an index of the value we just saw by iterating through an array inside the adjacency array, the value we will push will be the index of the array we are iterating through inside the adjacency matrix.

Now we can do a BFS over the reversed graph. We will first push the source vertex to a queue, then we will have a while loop with the condition that the queue is not empty. Every time the loop will execute, we will remove one vertex from the queue and store it in a variable, we will then iterate over the stack corresponding to that vertex in the reversed adjacency list and for each vertex not yet visited, we will change its corresponding distance value (in distances array) to the distance of the current vertex +1 and add the vertex to the queue.

Now, we will search the original graph recursively, we will use some arrays to store meaningful information, distances(the array of all the distances from the destination), the adjacency list, a Boolean array 'calculated' (initialised to false) storing weather the number of paths from a particular vertex to the destination in the graph was already calculated, and 'values', the number of paths from the given vertex to the destination, in case the number of paths have been calculated already(initialise this array to zero). For every vertex that the function is called on, we will check whether its number of paths was already calculated(using the calculated array), in case it was, we will just return the value stored in the array 'values'. If not, we will check whether its distance from the destination is zero or one, in which case we will return 1, otherwise, we will find the number of paths from this vertex by summing the number of paths from all of the vertices this vertex is connected to and have a smaller distance to the destination, so have to iterate through the array representing the edges of the current vertex, we will use a variable count to will store the number of paths found from the current vertex to the destination up till this point, we will check whether the distance of the connected vertex is less than that of the current vertex if that is the case, we will call the function recursively with that vertex as a parameter and add the returned value to count, we called the function with all of the vertices that

have a lower minimum distance, we will change the calculated value of the current vertex to true and the corresponding value from values array to count the as well as returning the count.

Why does this algorithm work?

-It is given that an edge that enables vertex I to transmit to vertex J is represented by having J in the array indexed I in the adjacency Array, therefore putting I in the stack indexed J will produce an edge that enables vertex J to transmit to I, which is exactly what we are doing to reeves the direction of the graphs

-Given that this graph is unweighted (as per project specifications) the minimum distance from a point to a destination is the minimum number of hops required to transmit from the source to the destination, and therefore, the minimum number of hops required to transmit from the destination to each point in the reversed graph, if we use BFS, we will first update all the nodes that could be reached directly from the destination in the reversed graph(set their value to 1), after that, for those noes that we will update the distance for all the vertices could be reached in two hops from the destination(update their values to 2) and so on. Therefore, we will visit each vertex ones, each time when it is the closest it gets to the destination, which means we will always get the right minimum distance from the destination to each value in the reversed graph, and therefore the minimum distance from this point to the destination in the original graph.

-By definition, if a point has distance 0 to the graph (the destination's distance to itself), there is only one way to get from the destination to itself(stay in that point), therefore returning 1 as the number of paths to get from that point the destination(in fact from the destination to itself), seems appropriate. In case the distance from the destination is 1, the vertex could only transmit to 0 (as a vertex is only allowed to transmit to a destination whose closest distance to the destination is less than that of the given vertex), therefore again returning one as the number of paths from that vertex to the destination should be correct. For all other vertices, the number of paths we could take to reach the destination is the sum of the number of paths to the destination from each vertex that the current vertex could directly transmit to, therefore adding all those values will produce the correct result. The graph does not change as we search through it, therefore if we already found the number of paths to reach the destination from a vertex we do not have to compute it again, we can just extract the computed value and return it, using this algorithm guarantees us to get the correct number of pathways from the source to the destination if we call this function and pass the source vertex as a parameter to that function(as well as all the lists specified above).

Time Complexity

- Reversing operation will take us O(D+L)(or O(N)), as we will iterate through all arrays inside the adjacency array(v times) and then through the inner arrays(additional e times) in the worst-case scenario.

- Finding the minimum distance from each vertex to the destination is done by a simple BFS over the reversed graph, it is well known that the time complexity for BFS in O(D+L) or O(N).

- Counting the number of paths is also done by searching over the graph, starting at the source vertex and going up till the elements that could transmit directly to the destination, only going to elements that the given vertex can transmit directly with lower distance(except for the case when the destination and the source are the same points, this case is irrelevant for asymptotic complexity), the path from each vertex is computed only ones, and stored, therefore the number of paths for each vertex is computed only ones and then could be retrieved by request. In the worst-case scenario, we will visit every node one time(except the destination) and then visit some nodes again via nodes that we can transmit to. Therefore, the asymptotic complexity is **O(L-1+D-1)=O(L+D) or O(N).**

- Given that all the steps we are doing have an O(n) complexity(and each step is done only one time), the algorithm has an O(3N)=O(N) complexity as well.

Question 3

The Algorithm

-We will start by getting the distance from the source with a very similar method to the method above (without reversing the edges of the graph, as we look at the distance of each point from the source and not from each point to the destination), storing all the distances in an array.

- Now we will enter all the subnet values to a HashMap(map), a map will represent the minimum distance from the source to each subnet, so, therefore, we will store a given subnet as a key in the map(represented by java.util.ArrayList), and the value will be an integer representing the minimum distance. For each vertex, we will enter the corresponding value from the distances array to its address key in the map, we will update the minimum path in the map of all the addresses of all the subnetworks the vertex belongs to in case such a value is not yet stored in the map for the given subnetwork, or the current value is greater than the value found.

- now we will iterate through the addresses array, convert every address to an ArrayList, and then get the minimum distance from the map, add the outputs from the map to an output array and return this output array, in case the query does not have a key in the map corresponding to it, we will return the biggest integer value, as that means we have not encountered any device that is part of this subnetwork and therefore does not have any minimum distance to it, in this case, we will return the biggest vale could be stored in an integer, as per project specifications.

Why Does This Algorithm Work?

-The first step is already explained in the previous question.

-For the vertices themselves, we just add their minimum distance from the source to the destination, this value is correct as it is explained why the step above is correct. For the subnetworks, by definition, the shortest path from the source to a subnetwork is the shortest path from the source to any device that belongs to that subnetwork, so therefore if we did not encounter a device yet any device that belongs to that subnetwork, we should add the minimum distance from that device as the minimum distance to that subnetwork, if we will later encounter a device that also belongs to this subnetwork but has a smaller distance from the source, we will change the value in the map corresponding to that subnet to the distance of the just encountered device, therefore we will always get the minimum distance to that subnet for all subnets that have at least one device inside them after checking all the devices.

-Given that the correct values are stored in the map for all subnetworks and all the subnetworks with at least one device in them are stored in the map as keys, by converting the queries into the right format(Integer ArrayList), we could just enter a query to the map and get the minimum distance from the source to that subnetwork, if such a key(subnetwork) does not exist in the map, we can safely conclude that none of the devices belong to that network and therefore return the biggest value could be stored in an integer type.

Time Complexity

-The first bit will be complexity O(D+L)=O(N) as explained in Q2

-We will update the values for each vertex ones in a map, we will have to update l+1(l is the length of the address array for each vertex) values in the subnet though, adding to that, each time we update a subnet we will have to convert the array to ArrayList, which takes linear time, therefore we get a complexity of O(D*l**2), fortunately, it is given that l is 4, therefore l and l^2 become constants and we are left with time complexity of O(D). Note adding an entry to a map is also a constant time operation.

-For the last part, we will have to convert every query to an ArrayList, which is a constant time operation as we know its maximum size (as discussed above) and look it up in a map(a constant operation as well), therefore the complexity is the number of queries(O(Q)).

- Now we can add the complexities, by doing that we will get
O(2D+L+Q)=O(D+L+Q)=O(N+Q).

Question 4

The Algorithm

The task in this section of this project was to find the maximum speed from a transmitting device to a receiving device. Because the download can take place through several paths simultaneously, we will have to consider all the paths to push as much speed as possible. It is also given that there is no traffic, hence, we can consider all the paths without worrying about any other interruptions. This makes it very similar to the maximum flow problems.

There are multiple Maximum Flow Algorithms in existence already but the one used in implementation here is Push-Relabel Algorithm. It is considered as one of the most efficient algorithms to solve flow problems.

We considered every device to have some height. We used this height to decide whether the flow of speed from a vertex through an edge to another vertex is possible or not. We only push speed if there exists an edge vertex whose height is lower than the height of the current vertex. If there are no adjacent devices with a height lower than the current vertex, we increase the height of the current vertex to the height of the adjacent vertex plus one.

Flow on an edge should not exceed the speed of the edge. Downloading speed is equal to the uploading speed for every device except for the source and destination devices.

We start with initializing flows and heights. The last loop in the code performs a push (push()) if it is possible to push the code otherwise, we relabel the heights. We must check for the active current devices with excess speed after we have performed these calls every time. We stop if there are no more active devices. The code returns -1 and we break the while loop.

Time Complexity

The time complexity of this algorithm can be calculated from the main loop of the method. To do this, we will have to calculate the number of times we have to use push and relabel methods. Now, the height of a device can not increase more than (2*D – 1) times. That is because in the worst-case, we can reach a height of (2*D – 1) but not more than that and the labelling value does not decrease. There are (D – 2) devices in the network after excluding the source and destination devices as stated in the above section, which gives a maximum of (2*D – 1) * (D – 2) operations.

The resulting worst-case complexity due to relabelling is, therefore, O((2*D – 1) * (D – 2)) complexity, i.e., $O(D^2)$ complexity.

For every saturated push, the complexity is of the order O(D*L) while for every unsaturated push, the complexity is of the order $O(D^2*L)$.

This leaves us with an overall complexity of **$O(D^2*L)$.**

This complexity can be further improved using FIFO method, i.e., deques, which will give a complexity of the order of $O(D^3)$.

Reference:
https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm