CITS2200 Data Structures & Algorithms Project

Jake Lyell (22704832), Jordan Lee (22705507)

This report will detail the process the students Jake Lyell (22704832) and Jordan Lee (22705507) took to complete the CITS2200 Project.

Task 1: allDevicesConnected

For allDevicesConnected, we decided to utilise a modified Depth First Search to solve the problem. From each node, it adds the nodes it can transmit data to, given they aren't already visited or in the stack waiting to be checked. For every node that is connected, the counter "discovered" Is incremented, after all the nodes in the stack have been searched, the value of discovered is compared with the number of nodes in the graph. If the values are the same, all of the nodes are connected, and true is returned. if the values are different, then the nodes are not all connected and false is returned.

Time Complexity

In our solution, we utilise an adjacency list, in which there are nodes or devices (d) and links between them (l). For each device we find all neighbours by going through the adjacency list once. For a directed graph (which our adjacency list represents), we can represent the sum of the nodes, and their links as D and L respectively. So, we can understand that our time complexity is O(D) + O(E) = O(V+E). So, our size of the network (N) can be said to be N = (D+L). Therefore, we have a time complexity of O(N).

Task 2: numPaths

For this solution, we utilised a breadth-first search (BFS). We aim to 'determine the number of different paths a packet can take in the network to get from a transmitting device to a receiving device.' In other words we are looking to get the number of shortest paths between the given devices (src and dst). Our solution will firstly loop whilst there are still devices to be searched through. We analyse the top of the queue (through pop). For each device adjacent to the current device, if it has not been visited, we add it to the queue and it is then labelled as visited (we do not return to visited devices). If the distance to the parent of the device is shorter than the device's current distance, the devices distance is updated. Then we set the number of paths to the neighbour node to the number of paths to its parent. If the distance is instead equal, we set the length of paths to be the number of paths to its parent + itself. The number of paths to the destination device, is the answer returned.

Time Complexity

The time complexity of our approach is like that of other breadth-first search algorithms, that is O(N). We get this as, in a worst-case scenario, we traverse every vertex and edge (device and link). As such we have O(D+L). Therefore, our time complexity is O(N).

Task 3: closestInSubnet

This algorithm utilises a breadth-first search (BFS). It first searches the graph and creates a distance for every node from the source. It goes through each query in the query array; we compare the query to the first x elements of each address. If the address is indeed part of the subnet, we set the number of hops for that query to be the number of hops to that node. If no address matches the query, the value is left as 'max value'. There is also an included base case, where if the query is empty the number of hops required is set to zero. As all addresses are in the in a subnet of {}.

Time Complexity

As our solution implements a breadth-first search, we can initially see our complexity to be O(N) like our previous solution in task 2. However, our solution checks against our query structure, in the second for loop. As such we can see that we actually have a time complexity of O(N+Q). Q representing the sum of queries.

Task 4: maxDownloadSpeed

Our solution for this task implements an Edmonds-Karp algorithm. We look to perform a breadth-first search from the source to the destination, if there is still bandwidth available along the path to the destination, it returns a list of parent nodes which is used to build an augmenting path from source to destination. This path is then traced back from the destination to the source, to find the smallest bandwidth present. This bandwidth is added to the total bandwidth and subtracted from each node along the augmented path. This is repeated until there are no more paths from source to destination with bandwidth available. We then return the maximum flow of the graph.

Time Complexity

Our solution essentially implements an Edmonds-Karp algorithm. Which can be thought of as a version of the Ford-Fulkerson algorithm that instead uses breadth-first search. Our Edmonds-Karp algorithm results in a time complexity of O(D L^2). Essentially, each iteration runs O(L) times (like seen in BFS), and there are at most O(D x L) iterations. Therefore we get O(D L^2).

References:

Ford-Fulkerson Algorithm for Maximum Flow Problem - GeeksforGeeks (2013). Available at: https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/ (Accessed: 19 May 2021).

Sryheni, S. (2020) *Number of Shortest Paths in a Graph | Baeldung on Computer Science*, *Baeldung on Computer Science*. Available at: https://www.baeldung.com/cs/graph-number-of-shortest-paths (Accessed: 20 May 2021).

Edmonds-Karp Algorithm | Brilliant Math & Science Wiki (2021). Available at: https://brilliant.org/wiki/edmonds-karp-algorithm/ (Accessed: 21 May 2021).