Tristan Ward (22756187)
Burbukje Shakjiri (22579289)

CITS 2200 Project

The graph is directed and weighted.

1 public boolean allDevicesConnected(int[][] adjlist) (6 marks)

1.1 Problem

Iterates through the nodes of the graph and checks if all nodes are connected.

Returns a boolean, true if all nodes are connected, false otherwise.

Needs to make sure every node is visited once.

1.2 Implementation

AllDevicesConnected uses the Breath First Search to check if all nodes can be reached. If all nodes are reached then the graph is deemed to be connected.

The BFS implementation iterates through all the nodes of the adjacency list, inserts the element in a queue, it retrieves and removes the head element of the queue, iterates over the adjacency list of that element. It uses a helper parent array to keep track of the previous nodes. It then checks if the index of the loop is not the node whose adjacency list we're iterating through, if that is the case, it stores that node to the parent array and increases the vertex number. Finally, it checks if the vertex number has the same number of elements as the adjacency list. If that is true, means the graph is connected.

This method is calculated using Breadth-First-Search. In order to keep track on which nodes are to be visited, a data structure should be used. We have used queues. An array is used to make sure no node is visited twice.

1.3 Why it works

BFS effectively iterates though the nodes of the graph while marking the ones that are visited to avoid checking the same node twice. Hence, reducing the time necessary to get to all the nodes of the graph.

1.4 Complexity

BFS has a worst-case time complexity of O(V+E), where V-vertices, E-edges. As stated in the lecture slides, each vertex is enqueued and dequeued once, the operation works on constant time. The queue manipulation takes O(V) time, and O(E) time is taken for examining all the adjacency lists of the vertex dequeued.

1.5 Improvement

Complexity goals stated in the Project were reached so no improvements are necessary.

2 public int numPaths(int[][] adjlist, int src, int dst) (10 marks)

2.1 Problem. Needs to be done and pitfalls

Need to search through the graph and make sure the same path is not counted twice, nor longer and inefficient paths are being examined.

2.2 Implementation

NumPath calculates the shortest distance from all vertices to the destination vertex and then uses this to calculate the number of paths. The implementation uses the transpose of adjlist, Dijkstra's algorithm and BFS to count the number of paths from source to destination.

Transpose of adjlist is calculated to be able to use Dijkstra's algorithm to calculate the distance from all vertices to the dst vertex. Then, BFS uses the distances values to build a tree of vertices that get closer to the destination. During the BFS loop, the path number is incremented when the vertex found is equal to dst.

2.3 Why it works

BFS is the important loop in the algorithm. It creates a Minimum Spanning sub-Tree where the child is always closer to the destination compared to the parent. The leafs of the Tree are equal to the number of paths. In other words, when the destination is reached the path number is incremented and that branch stops being examined. However, we must know what nodes are closer to the destination.

Dijkstra's algorithm is used to calculate the distances of all vertices to the destination. Unfortunatelly, Dijkstra's algorithm returns the distance from a source to all other values instead of the distance from all vertices to the destination. Therefore, the transpose of the adjlist is calcuculated to flip all edges of the directed graph and use the dst vertex as the initial vertex for Dijkstra's algorithm. Dijkstra's algorithm calculates the distance from the dst vertex to all other vertices in the transpose graph. Hence, Dijkstra's algorithm and the transpose graph are calculating the distance from all vertices to the dst in the original graph.

2.4 Complexity

Transpose algorithm must visit all vertices and all edges to build the transpose matrix. Hence, it has a complexity as large as the size of the network O(N). Unfortunately, this implementation also increases the space complexity as a transpose matrix must be created. Space increase is an integer matrix of D rows and D columns.

Dijkstra's algorithm has a complexity of O(L lg D) as explained on the Shortest Path Algorithm Lecture.

BFS has a maximum complexity of O(N) as explained in the Tree and Traversals Lecture. This complexity is only reached in the worst-case scenario that all vertices are connected and closer to

the destination compared to the source. For other graphs, the time complexity will improve as only edges that get closer to the destination are searched.

The overall time complexity of this implementation is O(N).

2.5 Improvements

It is expected that Dijkstra's algorithm alone could be used to calculate the distances and the number of paths. This could be achieved by finding paths of increasing length and finishing the algorithm when the addition of an edge does not bring you closer to dst. However, implementation was not achieved before the due date.

3 public int[] closestInSubnet(int[][] adjlist, short[][] addrs, int src, short[][] queries) (10 marks)

3.1 Problem

This method iterates through the elements of the adjacency list, checks if their IP addresses contain the queries (if the queries are subsets of addresses).

3.2 Implementation

This method was implemented using BFS to traverse through the elements of the graph. The data structure used was a queue. We start by pushing the vertices to the queue, when we pop them to check for the adjacency vertices, we check if the query is subset of the IP address, by using a helper method that compares the elements of the query with the elements of the address.

3.3 Why it works

BFS searches effectively through all the nodes of the graph while keeping track of the ones visited by using a data structure, in this case- a queue.

3.4 Complexity

The time complexity of the BFS which is O(V+E). The time complexity of the rest of the code is O(Q) to go through all the addresses and check if the queries are subsets. The total time complexity is O(Q(V+E)).

3.5 Improvements

The time-complexity of this method could have been improved by using a different data structure, in this case a hashmap, which would run in an O(n) time in the worst-case, or alternatively we could have used tries, which would run in a worst case in O(M) time, M being the maximum length of the string, provided that we parse the address elements and queries to strings. However, none of these ways were used due to failure of implementation before the deadline.

4 public int maxDownloadSpeed(int[][] adjlist, int[][] speeds, int src, int dst) (14 marks)

4.1 Problem. Needs to be done and pitfalls

The implementation of this method falls into a Maximum Flow problem. The challenge of the problem is finding edges that increase the Download Speed which is dependent of edges prior and after the edge in question.

4.2 Implementation

Ford-Fulkerson algorithm was used to calculate the maximum download speed using a BFS implementation to find augmenting paths.

Apart from initializing variables, the implementation uses a range of while loops and for loops.

1. The first outer "while" loop runs until there is no more augmenting paths. Within it there is the four consecutive loops.
2. Second loop is used to initialize and reset all variables that are used for the BFS loop.
3. The BFS loop that grows a minimum spanning tree to find if there is an augmenting path. The loop finishes when the destination is found or when all vertices have been visited.
4. If a path was found the fourth loop finds the minimal residual edge of the path.
5. The last loop updates all residual values stored in the residual Graph.

4.3 Why it works

Ford-Fulkerson algorithm was described in the book recommended by this unit named "Introduction to Algorithms". The book explains how this algorithm works under the Maximum Flow chapter. The decision to use BFS to find paths was made after further research as the book was not easy to understand.

4.4 Complexity

The book proves that the complexity of the Ford-Fulkerson has a complexity of O(E |f*|) when using BFS. f* stands for the maximum flow in the transformed network and |f*| stands for the Integral of f*.

From the five loops explained in the section 4.2;

1. This loop has a worst-case complexity when Paths found by BFS increment the maximum download speed by one. Hence, O(|f*|)
2. Complexity of O(D) as all nodes and edges must be initialized within the visited and parent arrays which have a length equal to D.
3. The BFS loop has a maximum complexity of O(N) as explained in the Tree and Traversals Lecture. However, in most cases the complexity is lower as the loop finishes if the destination vertex has been found.
4. Updating all residual values has a complexity of O(L)

4.5 Improvements

Using Dijkstra's algorithm instead of BFS may improve the algorithm as the path with the greatest augmentation value would be found first. Hence, improving in complexity from O(N) to the better

O(L lg D). However, the increasing residual values of the reverse edges would affect Dijkstra's algorithm in ways that are not understood.