Assignment 3 Report

Amelie Löwe, Håkan Johansson

November 17, 2019

Depth First Search and Breadth First Search

The time complexity of the Depth First Search(DFS) and Breadth First Search algorithms are : $O(|V| + |E|)$ where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph. This would be the worst case for our methods since it means that we have to traverse each vertex of the graph. Depending on the size of the graph, this could be an inefficient operation.

Exercise 1

isAcyclic()

In the directed and undirected graph we used the Depth First Search(DFS) approach to solve the traversing problem. By using a boolean array we could keep track of which path was already visited (those we set to true), and which ones still where untouched(false). This was done to avoid traversing graph multiple times. If a Vertice was visited before and the path led back to it again a cycle was found and the isAcyclic method returns false .

The only difference between the directed and undirected case is that we are using a stack in the directed case, and no extra data structure in the undirected. The reason why a stack is used to see whether or not the directed graph is acyclic is that the stack keep track of the previous vertex.

Time Complexity Analysis

The worst case scenario to if the graph is acyclic is to check all the vertices. That implies that we have a time complexity of $O(|V|+|E|)$. The best case is if we find a path of length three which starts and stops at the same vertex. That will give us $O(3) = O(1)$.

connectedComponents()

We start off by describing the case for the undirected graph, since there are quite some differences between undirected and directed. We traverse the undirected graph by using a DFS search. Whenever we can reach another vertex, we add it to a list. This implies that each list in the list of lists, is a connected component.

To solve the connected components of a directed graph we first need to look at the definition of a strongly connected component. That is, if we can reach a vertex b from a vertex a, we must also check so that we can reach a starting from b. Similarly to the undirected graph, we start by doing a DFS search of the list of vertices. The next step would be to traverse the transpose of the graph, which is due to the definition of a strongly connected component. The last step is to take the intersection of the lists of components that we traversed in original direction, and when we switched directions, and add the rest that is not in the intersection as separate strongly connected components.

isConnected()

Both undirected and directed graph call upon the connected components method to check if the graph has any connections. Because the connected components method returns a list of lists, and each of these lists must be a connected component. Furthermore we check if the size of the list i.e. connected components is equal to one. That is, if all components are connected to each other, and by the definition of a connected component, it implies that the size must be equal to one.

Time Complexity Analysis

Time complexity for the undirected graph is in worst case $O(|V| + |E|)$. That is due to, we visit all the nodes and at most using all the edges with the DFS algorithm.

In the case for the directed graph we will get time complexity $O(N^2)$. Which is based on the fact that we need to transpose the graph which runs in $O(N^2)$, and the rest is $O(|V| + |E|)$.

Exercise 2

This exercise is only for the undirected graph.

hasEulerPath()

Initially we used the isConnected method to check if the graph has any connections. If the graph is not connected, we instantly return false, since it cannot have an Euler path or circuit. In this solution we used the Euler path theorem. The theorem says that if there is more than two edges of odd degree, the graph does not contain an Euler path. This we check by simply iterating through the adjacency list and by using modulo two on each vertex we find out how many there are with an odd degree.

eulerPath()

Since we already have a hasEulerPath function, it was used to check the basic condition of containing a path or not . To avoid a path that leads to a dead end, we start by visiting a vertex with an odd degree. We call the find Euler method to recursively iterate over the graph. The method makes use of several different helper methods. These helper methods use a hashmap to store the deleted paths *i.e.* used edges. So when we traverse the graph we cannot go back on the same path, we've already visited.

Time Complexity Analysis

The time complexity for the euler path depends on a few different steps. The euler path calls the hasEuler method which uses the isConnected method, a

DFS based solution. This would make it linear. However the euler path method should not be of quadratic time complexity since they do not use any nested loops *i.e.* the time complexity is $O(N^2)$.

Exercise 3

For all of the methods in the third exercise, a Breadth First Traversal (or Search) approach was used to traverse the different friendship graphs. In the numberOfPeopleAtFriendshipDistance() method we save this information in an Integer array. When iterating through the array, we increase the friends variable every time a friend at the specified distance is found, this is then returned.The furthestDistanceInFriendshipRelationships() method is very similar. The main difference is that instead of looking for the friend at a distance, we look for friends with the most amountOfHopps from the start-vertex, *i.e.* the ones furthest away. The possibleFriends() method was solved in a slightly different way. After iterating over the adjacency list we find all of the friends that are at the distance of two, *i.e.* friends of a friend and them to the potentialFriendsFriends list. Then by using the iterator and the retainAll function we find the set that shares at least three friends. These we add to the eligibleFriendsFriends list and return it as the answer.

Time Complexity Analysis

Since we are using the BFS algorithm with adjacency lists the time complexity is $O(|V| + |E|)$. If an adjacency matrix would be used, the time complexity would be quadratic *i.e.* $O(N^2)$. That is because an adjacency matrix is always quadratic.

Considering the furthestDistanceInFriendshipRelationships method, which is similar to numberOfPeopleAtFriendshipDistance, the time complexity is the same for both.

Now for the possibleFriends method. We have two loops and we make use of the BFS algorithm. The loops are not nested so we will have $2 \cdot O(|V| + |E|) = O(|V| + |E|)$ by definition of time complexity.