1DV516 assignment 3

Björne Söderberg Laanemäe - bl222ey

8 november 2018

1 Exercise 1

1.1 Is Acyclic

1.1.1 Method for undirected graph

For an undirected graph, this is straight forward. I do a depth first search, iterating through the children of a vertex, marking them as visited as i go. If i somehow encounter a vertex that is already visited, there is a cycle, because i could get to the same place in two different ways, starting at the same vertex.

However, i cannot just start at the first node and call it a day. What if the graph is not connected? I must start at every node, in turn, to be sure that all possibilities are accounted for.

This can find a cycle in the first three vertices. It will then break and not bother looking for more cycles. The worst case though is that there is a cycle with the last vertex...

1.1.2 Complexity for undirected graph

1. First, i mark every vertex as *visited* $\rightarrow O(|V|)$
2. I then do a depth first search, begginning at the first vertex (first as in first created). I'm using adjacency lists $\rightarrow O(|V| + |E|)$
3. Then it's a little weird. I go through every other vertex linearly to check if they are visited. If not, i begin a new DFS at that vertex. This seems to me like $O(|V| \times (|V| + |E|))$ however i know this is not the case. I do not initiate a new DFS if the node is already visited.

   If the whole graph is connected, then this loop will only give an extra $O(|V|)$ because of the looping itself. If the graph is fractured, the first DFS didn't quite reach the $|V| + |E|$, because not all of the vertices have been visited yet - this loop finds the other subcomponents.

   What i'm trying to say is that all in all, regarding the whole *isAcyclic*, the sum of all the eventual DFSes i do amounts to $O(|V| + |E|)$ because i never reset who is visited. $\rightarrow$ this loop gives an extra $O(|V|)$.

In total: $O(|V| + |V| + |E| + |V|) = O(3|V| + |E|) = O(|V| + |E|)$.

1.1.3 Method for directed graph

A little trickier this time since the edges are directed. It does not suffice to do the DFS önceäs with the undirected graph - vertices can be *visited* but from the wrong direction, so to speak. A way to be sure is to actually redo the search from every vertex.

The code here looks similar, but everything is moved inside the loop.

1.1.4 Complexity for directed graph

1. Mark every vertex as *visited* $\rightarrow O(|V|)$
2. Do a DFS from the current node $\rightarrow O(|V| + |E|)$
3. Do these two steps for every vertex.

That last step brings us to $O(|V| \times (2|V| + |E|)) = O(|V|^2 + |V| \times |E|)$. Which of $O(|V|^2)$ and $O(|V| \times |E|)$ is more dominant depends on the graph.

1.2 Is Connected

1.2.1 Method for undirected graph

This one is quite similar to *isAcyclic*. I pick an arbitrary vertex, and do a DFS from it. Then i iterate through all vertexes and look if they are all visited.

1.2.2 Comlexity for undirected graph

1. more or less the same DFS as before $\rightarrow O(|V| + |E|)$
2. linear iteration $\rightarrow O(|V|)$

In total $O(2|V| + |E|) = O(|V| + |E|)$.

1.2.3 Method for directed graph

Again in the vein of the other solutions. Very bruteforce-ish. I iterate through all the vertices, and for each of them i see if a DFS leads back to itself.

The moment is does not, the search is over. Again, worst case scenario, this happens on the very last vertex i check.

1.2.4 Complexity for directed graph

Identical to *isAcyclic*.

1. Mark every vertex as *visited* $\rightarrow O(|V|)$
2. Do a DFS from the current node $\rightarrow O(|V| + |E|)$
3. Do these two steps for every vertex.

We already saw that this was $O(|V|^2 + |V| \times |E|)$.

1.3 Connected components

I use the same method for both directed and undirected graphs. The difference is in the recursive step of the DFS - in the undirected graph i need to keep track of the parentöf the current node as to not go back where i've already been. In the directed graph, if this happens it's explicit and should be considered the same way as any other edge.

The practical difference is an if-statement and an argument, and 1 vs 2 DFS's. The analysis will look the same, so i will not separate them.

1.3.1 Method

I iterate through all vertices. The first one automatically becomes its own group. For the other ones however, i see if they can reach the first one of any group (with a DFS). If they can, they are added. If not, they become their own group. This makes sure every vertex in a group can reach every other vertex in the same group - for new vertices, if they reach one of the vertices in a group, it reaches all of them.

Last i just return the list of groups.

1.3.2 Complexity

Very bad!

1. for every vertex $\rightarrow O(|V|)$
    (a) for every group $\rightarrow O(|V|)$
        i. depth first search $\rightarrow O(|V| + |E|)$
        ii. (optional) another depth first search $\rightarrow O(|V| + |E|)$

For every group is $O(|V|)$ because worst case nothing is connected $\rightarrow$ one group per vertex.

This gives us $O(|V| \times O(|V| \times O(2(|V| + |E|))$ which folds down to $O(|V|^3)$ or $O(|V|^2 + |E|)$.

2 Exercise 2

Full disclosure - i didn't read the book, i missed that you referred to it in the exercise description. Oh well.

2.1 Has euler path

First of all, if it's not connected it can't have a path. If it is connected, the next thing would be to check the degrees of vertices. If it has exactly two of an odd degree, it will contain an euler path. If it has only even degrees, it will contain an euler circuit, which is an euler path.

I'm reusing *isConnected* which has time complexity $O(|V| + |E|)$. Then, the counting of degrees is linear in $O(|V|)$. Of these two, $O(|V| + |E|)$ is the larger one and thus the final complexity.

2.2 eulerPath

You said to assume there is a path in the graph, so i'm not doing any checks for that.

First things first - this algorithm is destructive - it actually removes the edges. To remedy this, i have a loop to copy the adjacency lists. $\rightarrow O(|V|)$.

Next step. If there are vertices with an odd number of edges, we need to start in one of them, so we search for it. If we find nothing, we can start in any vertex. This search $\rightarrow O(|V|)$.

I have a large while-loop which runs until the current vertice has no more edges, which means it runs until we have removed all edges $\rightarrow O(|E|)$.

Inside this loop, if the currentVertex has only one edge, we traverse it and remove it $O(1)$. However if there are more, we can't just remove any edge. In the algoritm i'm using, an edge is called a *bridge* if it is the only edge between two parts of the graph. Such an edge can't be removed, so we need to check our edges before we traverse. This is done by counting the number of nodes you can reach (from the node you would traverse to) before and after you remove the edge. If the count differs, it's a bridge, and we need to select another node.

This count is done via DFS, which is $O(|V| + |E|)$. This is done (twice) per edge in the current vertex. It's possible that all edges are bridges but the last. Hence we get $O(|E|) \times O(|V| + |E|)$.

When a good edge is found, we remove it and traverse it, which is $O(1)$.

Thats the whole of it - in total we get $O(|V|) + O(|V|) + O(|E| \times |E| \times (|V| + |E|))$, which is, similar to before, $O(|E|^3)$ or $O(|E|^2 + |V|)$, where either of them could be dominant depending on the graph.

3 Exercise 3

Full disclosure pt. 2: i didn't particularly like my *MyUndirectedGraph* class, so i removed the extends here - i hope it's OK! The social network works in a similar fashion but it's more tailored (lots of noise removed).

3.1 The modified DFS

All these methods build on a modified version of the DFS function from earlier classes, but now it counts the depth and stores it in a table. To find the "correctdepth, even if we stumble upon a vertex that's already visited, if we could revisit it and give it a lower depth, we do that. The end result is something like a spanning tree, but we only keep the level of the vertices, not the relationships $\rightarrow$ a mapping of Person (vertex) to depth (from our guy).

The worst possible graph would be one where when we look at the first friend we give a depth to every node, then we look at the next friend and it's one step closer to everybody else so we need to update them all, and then for the next friend the same, and so on.

We would need to check all vertices the first time, all-1 the second time, all-2 the third time, and so on. This is a $O((N(N+1))/2)$ scenario, which can be generalized as $O(N^2)$. This is a graph, so we get $O((|V| + |E|)^2)$.

3.2 numberOfPeopleAtFriendshipDistance

I run my weird DFS thing once $\rightarrow O((|V| + |E|)^2)$. Then, since i have a table of distances from the specified person, i can go through it once in linear time, and count those with the correct friendship distance $\rightarrow O(|V|)$.

For this method, then, the complexity is $O((|V| + |E|)^2)$.

3.3 furthestDistanceInFriendshipRelationships

This one works in the exact same way! In the linear loop, instead of searching for a specific friendship distance i search for the most distant one.

For this method we get the exact same $O((|V| + |E|)^2)$.

3.4 possibleFriends

Once again, i run the DFS method, giving me a table of distances $\rightarrow O((|V| + |E|)^2)$. I go through the table linearly and pluck out those with distance 1, so i can look at their friends $\rightarrow O(|V|)$. I go through those friends at distance one, and for each i look at *their* friends. If those friends of friends have a distance of 2, i add them to a hashmap, with the value of how many times i've seen them $\rightarrow O(|V|^2)$.

Finally, i go through this hashmap, and if someone has a distance ≥ 3, they are added to the list that i return.

All in all, this *possibleFriends* gets a worst complexity of $O((|V| + |E|)^2)$.