CIS 121 Fei Liu

Homework 06

Recitation 201

Collaborators: Vaughan Reale

Q1. The type of graph that we have is an undirected, maximally acyclic graph, or a tree. It's acyclic because there is only one path from one destination to any other destination. If there was a cycle, then this wouldn't be true. In a tree with n vertices, there are $n-1$ edges.

My algorithm will run BFS on an arbitrary node in the graph, then take the furthest node found and run BFS on that again. The two nodes with the greatest distance from each other will be the furthest node found in the first BFS and the farthest node found in the second BFS.

P.O.C.: I will prove this algorithm by contradiction. Assume towards a contradiction that my algorithm does not return the two nodes with the longest path. Then this means that there must exist another pair of nodes that make the longest path. Let's denote the arbitrary node that we ran our first BFS on x, the furthest node found from first BFS y, and the furthest node found from second BFS z. W.L.O.G., assume that y is further from x than z is. Let's denote the path from x to the first furthest node found with p, and denote the path from x to the second furthest node found with p'. There are multiple cases:

1. If the longest path was from y to another node other than z, denote this other node z', then the second BFS returned the wrong furthest node, which is a contradiction. If $p' > p$, then the first BFS would have returned z', not y, which means the first BFS returned the wrong furthest node, which is a contradiction.

2. If the longest path was from some other node y' to z, then BFS would have found y' as the furthest node from x, not y, which means BFS returned the wrong furthest node, which is a contradiction.

3. If the longest path was from some other node y' to some other node z', we have two cases: path from x to y' is shorter than path from x to y, or path from x to y' is longer. If the path from x to y' is shorter, p' must be longer to compensate for a shorter p. If p' becomes longer than p, then the first BFS would have found z', and if p' stays shorter than p, then the first BFS would have found y'. In both cases, BFS returned the wrong furthest node, which is a contradiction. If the path from x to y' is longer, then the we would have found y' from the first BFS, which means BFS returned the wrong furthest node, which is a contradiction.// // I used BFS twice in my algorithm. The run time of BFS on a graph with n vertices and m edges is $\theta(n+m)$. However, since the graph I'm working with is a tree, $m = n-1$, so the run time becomes $\theta(n)$.

Q2. This question is asking: find an algorithm that returns all nodes that have a path to all other nodes in a digraph.

My graph will take a digraph G as an input and here's what it does:

1. Run Kosaraju's algorithm on the input graph. This will return a graph with strongly connected components as nodes. Call this returned graph G'.

2. Toposort G'. There is a topological ordering because G' is a DAG. Call the list returned from toposort L. 3. Run DFS on G', starting on first element in L.

4. If the for loop of DFS in step 3 must iterate more than once, return an empty set, else, return all vertices that are in the source S.C.C. in G'.

We know that the the graph returned from Kosaraju's algorithm is an acyclic digraph. So there are two cases: the source node in G' has a path to all other nodes, or it doesn't. Toposort helps us find a source node. In the first case where the source node of G' has a path to all other nodes, DFS in step 2 will have a for loop that iterates once, since the helper function DFS-visit finishes all nodes in G'. Step 4 of my algorithm will then return the vertices in the source S.C.C. since those vertices lead to each other as well as every other S.C.C., which contains all other vertices.

In the case where the source node doesn't have a path to all of the other nodes in G', then DFS in step 2 will run more than once. This means that there are no vertices that lead to all other vertices because we know that the vertices in the source S.C.C. does not lead to all other (this is what I cased on), but there are no other vertices in any other S.C.C. that has a path back to this source S.C.C. because of the definition of a source S.C.C. Thus, step 3 of my algorithm returns an empty set.

Here's the run time analysis of my algorithm: we know that Kosaraju's algorithm runs in $\theta(m+n)$ time, where m is number of edges and n is number of vertices. Step 2 is toposort, which runs in $\theta(n+m)$ time. Step 3 simply uses DFS which also runs in $\theta(n+m)$ time and step 4 runs in constant time, so the time complexity of my algorithm is $\theta(n+m)$ time.

Q3. We can represent the villages and roads with an undirected graph. There are n vertices and m edges. Queen Ayyah's goal is to ensure that the graph is connected. If there are multiple connected components, return a minimal set of edges that would make the graph connected. If not, then return an empty set.

My algorithm's input is an undirected graph G. The algorithm will first initialize an empty array list of vertices. We can run a simple DFS on the vertices. The for loop in the DFS algorithm iterates once for every connected component; if the for loop terminates in one iteration, that means that the recursive helper function DFS-visit was able to visit every node, meaning there was just one connected component. In every iteration of the for loop in DFS, if the current vertex's color is white (meaning it has not been discovered yet), add the current vertex to the array list. Notice that this means that the number of vertices in the array list is equal to the number of connected components in the graph. Create an empty set of edges. If there's only one vertex in the array list, just return the empty set, else, for every vertex in the array list until the second last vertex, create an edge with the current vertex and the next vertex in the array list. Finally, return the set of edges.

Here's the P.O.C. (it's more of an in-depth explanation since the algorithm is very intuitive): The for loop in the DFS algorithm (which calls DFS-visit) will have as many iterations as connected components in the graph. At every iteration of this for loop, the current vertex the for loop is iterating on will be added to the array list. Since DFS-visit finishes all of the vertices in a connected component, there will be exactly one vertex for each connected component in the graph. The second part of the algorithm is a for loop that iterates through the array list. The first iteration will create an edge connecting the first and second connected component, the second iteration will create an edge connecting the second and third connected component, and so on until the last iteration will create an edge connecting the second to last connected component to the last connected component. The result is a path that runs through every connected component, making the graph connected, which is what Queen Ayyah wanted.

More formal P.O.C.: Assume towards a contradiction that my algorithm does not return the set of edges needed to connect the graph. The array list could contain more or less edges. Denote the number of connected components x. If the array list contains more edges (array list size $> x$), then that means the DFS ran on more connected components (of C.C $> x$), which is a contradiction, as there were x connected components in the graph. Similary, the if array list returned less edges, then that means the DFS ran on less connected components (of C.C. $< x$), which is a contradiction, as there were x connected components.

The array list could also return the wrong edges. Wrong edges would have to connect two vertices in the same S.C.C. This is a contradiction because we know that DFS-visit visits every vertex in the same S.C.C., so the for loop in DFS always enters DFS-visit on a new S.C.C. Vertices get added to the array list in the for loop in DFS, so these vertices are guaranteed to be in distinct S.C.C.s.

Here's a run time complexity analysis: The DFS algorithm takes $\theta(n+m)$ time. Adding a vertex to the array list for each connected component will take O(n) time. The second part of the algorithm which connects all of the connected components simply loops through the array list, which in the worst case scenario, contains n vertices, so this part takes

O(n) time. Everything else takes constant time. Thus, adding all of these running times together, the running time of my algorithm is big-theta($n + m$).