1 Short Answer

For each of the following, either prove the statement is true or give a counterexample to show it is false.

(a) If \((u, v)\) is an edge in an undirected graph and during DFS, \(\text{post}(v) < \text{post}(u)\), then \(u\) is an ancestor of \(v\) in the DFS tree.

(b) In a directed graph, if there is a path from \(u\) to \(v\) and \(\text{pre}(u) < \text{pre}(v)\) then \(u\) is an ancestor of \(v\) in the DFS tree.

(c) In any connected undirected graph \(G\) there is a vertex whose removal leaves \(G\) connected.

Solution:

(a) True. There are two possible cases: \(\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)\) or \(\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)\). In the first case, \(u\) is an ancestor of \(v\). In the second case, \(v\) was popped off the stack without looking at \(u\). However, since there is an edge between them and we look at all neighbors of \(v\), this cannot happen.

(b) False. Consider the following case:

(c) True. Remove a leaf of a DFS tree of the graph.

2 Graph Traversal
(a) For the directed graph above, perform DFS starting from vertex A, breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as Tree, Back, Forward, or Cross.

(b) What are the strongly connected components of the above graph?

(c) Draw the DAG of the strongly connected components of the graph.

**Solution:**
3  Connectivity vs Strong Connectivity

(a) Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ such that removing $v$ from $G$ gives another connected graph.

(b) Give an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing $v$ from $G$ gives a directed graph that is not strongly connected.

(c) Let $G = (V, E)$ be a connected undirected graph such that $G$ remains connected after removing any vertex. Show that for every pair of vertices $u, v$ where $(u, v) \notin E$ there exist two different $u$-$v$ paths.

Solution:

(a) Let $T$ be a DFS tree on $G$, and let $v$ be a leaf of $T$. Then $T - v$ is a connected graph because any simple path $P$ from $u$ to $w$ $(u, w \neq v)$ in $T$ cannot pass through $v$ (since $v$ has degree 1). Since $T - v$ is a subgraph of $G - v$, $G - v$ is also connected.
(b) A directed cycle of three nodes is an example here (i.e. \( V = \{a, b, c\} \) and \( E = \{(a, b), (b, c), (c, a)\} \)).

(c) Let \( P \) be a \( u-v \) path in \( G \); then \( P \) contains a vertex \( w \) which is not \( u \) or \( v \). The graph \( G - w \) does not contain \( P \), but there exists a path \( P' \) connecting \( u \) and \( v \) since \( G - w \) is connected. \( P' \) exists in \( G \) and is different from \( P \).

4 Updating Labels

You are given a tree \( T = (V, E) \) with a designated root node \( r \), and a non-negative integer label \( l(v) \). If \( l(v) = k \), we wish to relabel \( v \), such that \( l_{\text{new}}(v) \) is equal to \( l(w) \), where \( w \) is the \( k \)th ancestor of \( v \) in the tree. We follow the convention that the root node, \( r \), is its own parent. Give a linear time algorithm to compute the new label, \( l_{\text{new}}(v) \) for each \( v \) in \( V \).

Slightly more formally, the parent of any \( v \neq r \), is defined to be the node adjacent to \( v \) in the path from \( r \) to \( v \). By convention, \( p(r) = r \). For \( k > 1 \), define \( p^k(v) = p^{k-1}(p(v)) \) and \( p^1(v) = p(v) \) (so \( p^k \) is the \( k \)th ancestor of \( v \)). Each vertex \( v \) of the tree has an associated non-negative integer label \( l(v) \). We want to find a linear-time algorithm to update the labels of all vertices in \( T \) according to the following rule: \( l_{\text{new}}(v) = l(p^l(v)) \).

**Solution:**

**Main Idea** When we implement DFS with a stack, the stack at any given moment will always contain all the ancestors of the current node we’re visiting. We want to maintain the labels of the relevant vertices currently on the stack, in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we’ll only add a vertex to our array (at index equal to the current depth) when we’ve actually visited it once (not when we first did it to the stack). Since a path can have at most \( n \) vertices, the length of this array is at most \( n \). Once we’ve processed all the children of a node, we can index into the array and set its label equal to the index of its \( k \)th ancestor. Notice that if we relabel the vertex before processing its children, we overwrite a label that the children of the vertex could depend on.

**Runtime Analysis** Since we add only a constant number of operations at each step of DFS, the algorithm is still linear time.