Directed Graphs

1. Given a DAG and two vertices v and w, describe an algorithm to find the lowest common ancestor (LCA) of v and w. The LCA of v and w is an ancestor of v and w that has no descendants that are also ancestors of v and w.

   Computing the LCA is useful in multiple inheritance in programming languages, analysis of genealogical data (find degree of inbreeding in a pedigree graph), and other applications.

   Hint: determine a way to find the height of vertices v and w as a distance from a root. Use that to find the lowest common ancestor of v and w.

Do a topological sort. Then, starting with first node ordered before them, set that node as a root in a breadth first search for one and then again for the other. If neither are found, repeat for next node before to be used as root for the next search.

Topological Sort

2. Describe a method that checks whether or not a given permutation of a DAG's vertices is a topological ordering of that DAG.

Run for loop iterating u over the vertices in order of the given permutation. Run an inner loop iterating v over the set vertices that the outer loop hasn't yet. In the inner loop, if there is an edge pointing from v to u, then print that it is not a topological ordering. After the second loop, print "done".

3. A colleague suggests an alternate method to find the topological ordering of a DAG: run BFS, and label the vertices by increasing distance to their respective source. Explain why your colleague's algorithm won't necessarily always produce a true topological ordering.

The simplest counterexample is a DAG of three vertices, each pair sharing one edge, say A->B, A->C, C->B. B and C are both the same distance from A, but whether the order is a topological ordering is not invariant under transposition of B and C.

4. Design an algorithm to determine whether a directed graph has a unique topological ordering.

   Hint: If the directed graph has multiple topological orderings, then a second topological order can be obtained by swapping a pair of consecutive vertices.

First do a topological sort. Then, run a for loop iterating u over all but the last element in the ordering. In this loop, if there is no edge from u to the element after it, the print that the topological ordering is not unique. After the loop, print "done".

Strongly Connected Components

5. Describe what the strongly connected components of a directed acyclic graph (DAG) are.

Strongly connected components are partitioned subsets of vertices, where you can reach each vertex of any of the subgraphs from any of the other vertices in the same sub by graph following arrows from either tail to head for the whole path or vice versa.

6. A colleague doesn't believe that the the strongly connected components in G^R are the same as in G. How can you prove your colleague wrong?

Each strongly connected component is combination of cycles that are all connected to each other. If you take any cycle and reverse all of the arrows, it is still a cycle. So, if you do this to all of the cycles in a stongly connected component they will still be cycles, and it doesn't effect how they are connected as that is only dependent on edges and vertices, not the direction of the edges.

7. Why do we need Kosaraju's algorithm? Why can't we simply use DFS or BFS on a DAG to determine strongly connected componets like we do in undirected graphs?

If strongly connected components are connected to others, then you can end up with more vertices than are actually in your strongly connected component.

8. The reverse postorder of a graph's reverse is the same as the postorder of the graph. Is there any truth to this statement? Why or why not? (There isn't necessarily one postorder.)This is equivalent to asking if the postorder of a graph's reverse is the same as the reverse postorder of said graph. This is true (but not limited to) when a graph is topologically equivalent (homeomorphic) to its reverse. But, here is the smallest counterexample I could think of: 1->2, 1->3, 4->2. Postorders (starting with vertices only on tails of arrows) will be (2,3,1,4), (3,2,1,4), or (2,4,3,1). The postorder of the graph's reverse will be (1,4,2,3), (4,1,2,3), or (1,3,4,2), meaning the reverse postorder of the graph's reverse will either be (3,2,4,1), (3,2,1,4) or (2,4,3,1). These don't match.

9. Why is it necessary to first transpose the graph (reverse the edges) in Kosaraju's algorithm? Could the algorithm have been designed to not require the transpose?

It doesn't have to be done first (which we know due to what we conclude in question 6). Since strongly connected components are subgraphs where each pair of vertices have at least one path going from one to the other and vice versa, the depth first search needs to be done in both directions (original and transpose).