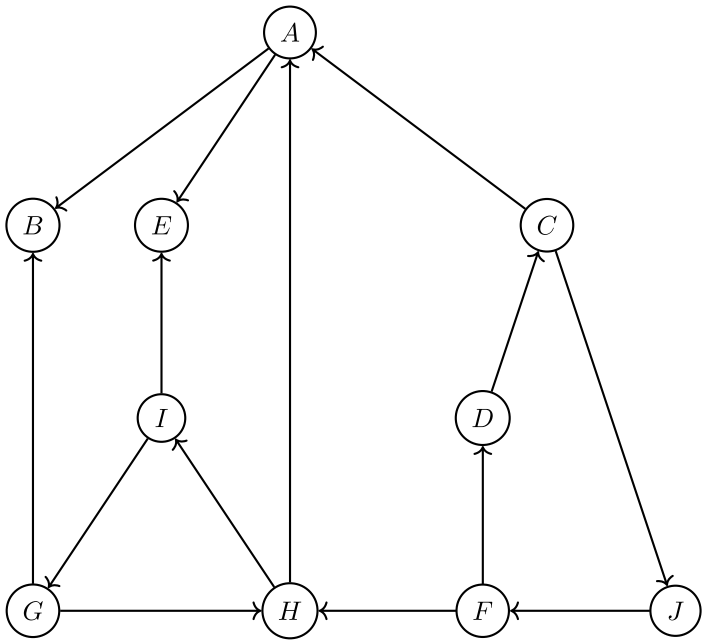
Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Graph Traversal



(a) Recall that given a DFS tree, we can classify edges into one of four types:

- Tree edges are edges in the DFS tree,
- Back edges are edges (u, v) not in the DFS tree where v is the ancestor of u in the DFS tree
- Forward edges are edges (u, v) not in the DFS tree where u is the ancestor of v in the DFS tree
- Cross edges are edges (u, v) not in the DFS tree where u is not the ancestor of v, nor is v the ancestor of u.

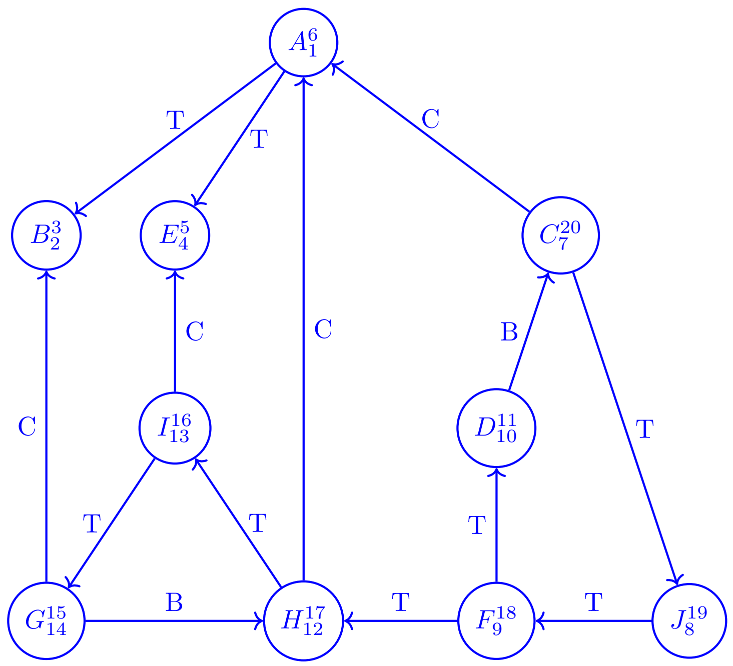
For the directed graph above, perform DFS starting from vertex A, breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as **T**ree, **B**ack, **F**orward or **C**ross.

(b) A strongly connected component (SCC) is defined as a subset of vertices in which there exists a path from each vertex to another vertex. What are the SCCs of the above graph?

(c) Collapse each SCC you found in part (b) into a meta-node, so that you end up with a graph of the SCC meta-nodes. Draw this graph below, and describe its structure.

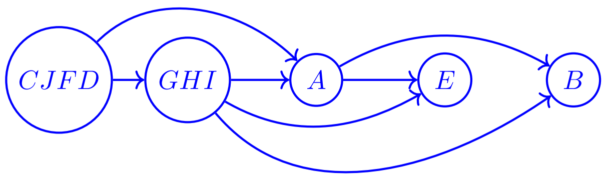
Solution:

(a)



(b)

$$\{A\}, \{B\}, \{E\}, \{G, H, I\}, \{C, J, F, D\}$$



(c)

2 Graph Short Answer

For each of the following, either prove the statement is true or give a counterexample to show it is false. Note that pre(v) and post(v) denote that pre-order and post-order values of v.

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

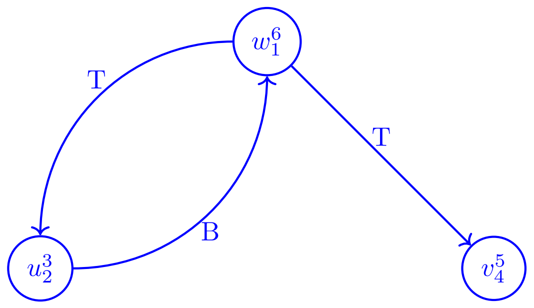
(c) We can modify the SCC algorithm from lecture, so that, the DFS on G is done in decreasing pre-order of G instead of decreasing post-order of G^R. This modified algorithm is also correct.

(d) We can modify the SCC algorithm from lecture so that the first DFS is run on G and the second DFS is run on G^R. This modified algorithm is also correct.

Solution:

(a) True. There are two possible cases: pre$(u) <$ pre$(v) <$ post$(v) <$ post(u) or pre$(v) <$ post$(v) <$ pre$(u) <$ 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 where we DFS starting from w:



(c) False. Consider a 2-vertex graph with a single directed edge, i.e. $G = (V = \{v_1, v_2\}, E = \{(v_1, v_2)\})$. Then if we start DFS from v_2, the vertex with the largest pre-order in G would be v_1. However, we know that v_1 is not the sink SCC and so this modified algorithm would return the wrong answer.

(d) True. The SCCs of G are the same as the SCCs of G^R.

3 Finding Clusters

We are given a directed graph $G = (V, E)$, where $V = \{1, \dots, n\}$, i.e. the vertices are integers in the range 1 to n. For every vertex i we would like to compute the value $m(i)$ defined as follows: $m(i)$ is the smallest j such from which you can reach vertex i. (As a convention, we assume that i is reachable from i.)

(a) Show that the values $m(1), \dots, m(n)$ can be computed in $O(|V| + |E|)$ time. **Please provide an algorithm description and runtime analysis; proof of correctness is not required.**

Solution: The algorithm is as follows.

```
procedure DFS-Clusters($G$)
    while there are unvisited nodes in $G$ do
        Run DFS on $G$ starting from the numerically-first unvisited node $j$
        for $i$ visited by this DFS do $m(i) := j$
```

To see that this algorithm is correct, note that if a vertex i is assigned a value then that value is the smallest of the nodes that can reach it in G, and every node is assigned a value because the loop does not terminate until this happens.

The running time is $O(|V| + |E|)$ since the algorithm is just a modification of DFS.

(b) Suppose we instead define $m(i)$ to be the smallest j that can be reached from i, instead of the smallest j from which you can reach i. How should you modify your answer to part (a) to work in this case? **Solution:** We start by reversing all edges in G, i.e. replacing each edge (u, v) with the edge (v, u), and then just using our algorithm from the previous part. This works because in the reversed graph, we can reach j from i if and only if we can reach i from j in the original.

4 Not So Exciting Scheduling

PNP University requires students to finish all prerequisites for a certain class before taking it; however, they made some mistakes when assigning prerequisites. Thus, some classes at PNP University are potentially *NP (Not Possible to take)* because it may be impossible to take all its prerequisite classes whilst following proper prerequisite rules. For example, if CS 61A is a prereq of CS 61B, CS 61B is a prereq of CS 170, and CS 170 is mistakenly listed as a prereq of CS 61A, then all three classes would be NP.

Due to these mistakes, students wish to figure out whether their classes can all be taken or not while following prerequisites. Their n classes are labelled with unique identifiers $\{c_1, c_2, \ldots, c_n\}$, and the set of m prerequisites in the form $[c_i, c_j]$ indicate that c_i must be taken before c_j.

Design an algorithm that outputs a potential scheduling of classes (i.e. an ordering of classes to take) if there are no NP classes; return false otherwise. **Please provide an algorithm description and a runtime analysis; proof of correctness is not required.**

Solution:

Algorithm: Create a node for each class. For each prerequisite $[c_i,\ c_j]$, construct a directed edge from c_j to c_i. Run DFS on the resulting graph while keeping track of pre and post numbers. (Cycle detection) If a back edge exists, then output false. Otherwise sort the nodes on post numbers, from highest to lowest (topological sort on DAG) or use a stack to keep track of visited nodes so that you don't need to do extra sorting at the end. And this will create a valid scheduling.

Proof of Correctness: Proof for cycle detection and topological sort will be the same as seen in lecture. A directed graph is a DAG if and only if there is no back edge. And because DAG has no back edge, if we order the vertices from highest post number to lowest post number, it's guaranteed to be topologically sorted by definition.

Runtime Analysis: Constructing the graph will take $O(n + m)$ time since we have n nodes and m directed edges. Running DFS will take again $O(n+m)$. We can do linear scan to find back edge. If we used stack to keep track of visited nodes, then topological sort will simply take linear time because we can just read off the nodes from the back the stack. Therefore, the overall runtime will be $O(n + m)$.

5 Biconnected Components

Consider any undirected connected graph $G = (V, E)$. We say that an edge $(u, v) \in E$ is *critical* if removing it disconnects the graph. In other words, the graph $(V, E \setminus (u, v))$ is no longer connected. Similarly, we call a vertex $v \in V$ critical if removing v (and all its incident edges) leaves the graph disconnected.

(a) Can you always find a vertex $v \in V$ that is **not** critical? What about an edge that is not critical?

(b) Give a linear time algorithm to find all the critical edges of G.

(c) Modify your algorithm above to find all the critical vertices of G.

(d) A biconnected component of G is a connected subgraph that does not contain critical vertices; in other words, if you remove a vertex from the component, then it will remain connected. If we collapse all the biconnected components of G into a meta-node (similar to what we did with SCCs), what does the resulting graph look like?

Solution:

(a) Consider running DFS from any vertex on the graph, and any leaf in the resulting DFS tree. The leaf can be removed without disconnecting the graph, since the remaining vertices are connected using the DFS tree edges. But we can't always find such an edge. For example, every edge in a tree is critical.

(b) Perform DFS on G while keeping track of $pre[v]$ for each vertex. We also maintain a low value for each vertex, where $low[v]$ denotes the smallest $pre[u]$ such that there is a back edge to u

from the subtree of v. The only potential critical edges are the tree edges in the DFS. An edge between v and its parent p is critical if and only if $low[v] > pre[p]$ (i.e. there does not exist a back edge from v to p or any of its ancestors).

(c) The root of the DFS tree is critical iff it has more than one child. A non-root vertex v is critical iff for at least one of its children c, $low[c] > pre[v]$.

(d) The resulting graph is a tree.

To prove this, show that the if there is a cycle in the resulting graph then all the meta nodes in the cycle will belong to the same biconnected component. This gives a contradiction.