Note: Your TA may not get to all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. The discussion worksheet is also a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Short Answer

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

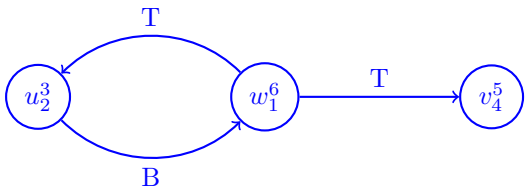
(a) If (u, v) is an edge in an undirected graph and during DFS, $\text{post}(v) < \text{post}(u)$, then u is an ancestor of v in the DFS tree.

(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:

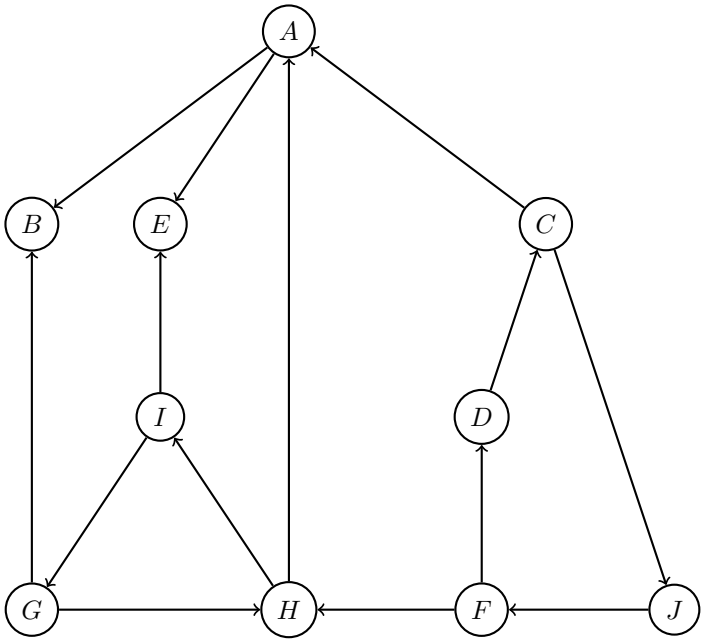
(a) True. There are two possible cases: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$ or $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since 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. 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.

2 Graph Traversal



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

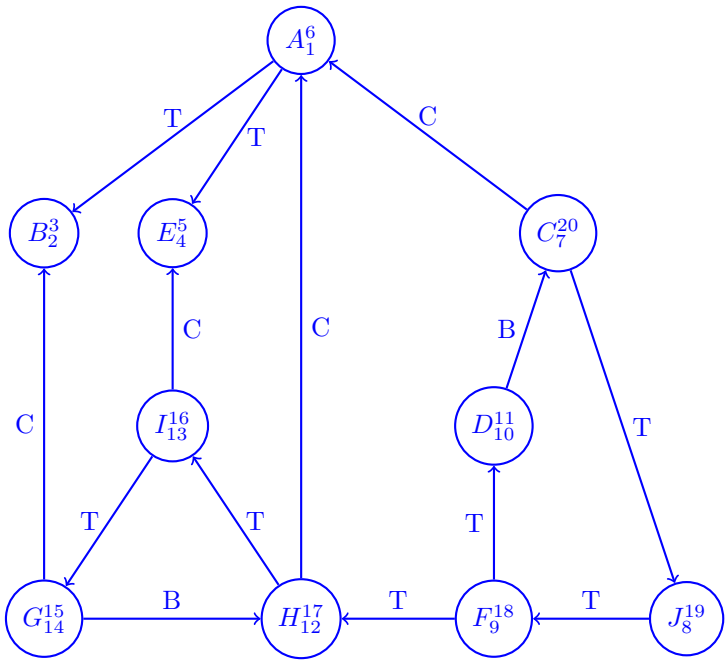
- Tree edges are edges in the DFS tree,
- Back edges are edges (u, v) not in the DFS tree where v is the ancestor of u in the DFS tree
- Forward edges are edges (u, v) not in the DFS tree where u is the ancestor of v in the DFS tree
- Cross edges are edges (u, v) not in the DFS tree where u is not the 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) What are the strongly connected components of the above graph?

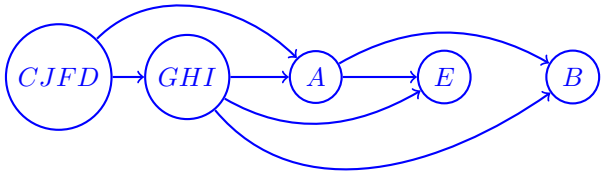
(c) Draw the DAG of the strongly connected components of the graph.

Solution:



(a)

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



(c)

3 Finding Clusters

We are given a directed graph $G = (V, E)$, where $V = \{1, \ldots, n\}$, 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 $m(1)$, ..., $m(n)$ can be computed in $O(|V| + |E|)$ time.

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 BFS Intro

In this problem we will consider the shortest path problem: Given a graph $G(V, E)$, find the length of the shortest path from s to every vertex v in V. For an unweighted graph, the length of a path is the number of edges in the path. We can do this using the *breadth-first search* (BFS) algorithm, which we will see again in lecture this week.

BFS can be implemented just like the depth-first search (DFS) algorithm, but using a queue instead of a stack. Below is pseudo-code for another implementation of BFS, which computes for each $i \in \{0, 1, \ldots, n-1\}$ the set of vertices distance i from s, denoted L_i.

```
Input: A graph G(V, E), starting vertex s
for all v ∈ V do
    visited(v) = False
visited(s) = True
L_0 → {s}
for i from 0 to n − 1 do
    L_{i+1} = {}
    for u ∈ L_i do
        for (u, v) ∈ E do
            if visited(v) = False then
                L_{i+1}.add(v)
                visited(v) = True
```

In other words, we start with $L_0 = \{s\}$, and then for each i, we set L_{i+1} to be all neighbors of vertices in L_i that we haven't already added to a previous L_i.

(a) Prove that BFS computes the correct value of L_i for all i (Hint: Use induction to show that for all i, L_i contains all vertices distance i from s, and only contains these vertices).

Solution: We claim that before we start iteration i of the for loop: (1) all vertices at exactly distance i from s are in L_i, and (2) All vertices at distance more than i from s have not been added to any L_j, $j \leq i$. We will prove this inductively holds for all i, which implies the algorithm is correct.

This holds for $i = 0$. Assume it holds for $i = k$. We will show it holds for $i = k + 1$. (1) holds for $i = k + 1$ because every vertex at distance $k + 1$ is adjacent to some vertex at distance k, and thus by inductive hypothesis (1) gets added to L_{k+1} in iteration k. (2) holds because no vertex at distance $k + 2$ or more can be adjacent to a vertex at distance k or less, and so the only vertices added to any L_{k+1} in iteration k are those at distance exactly $k + 1$.

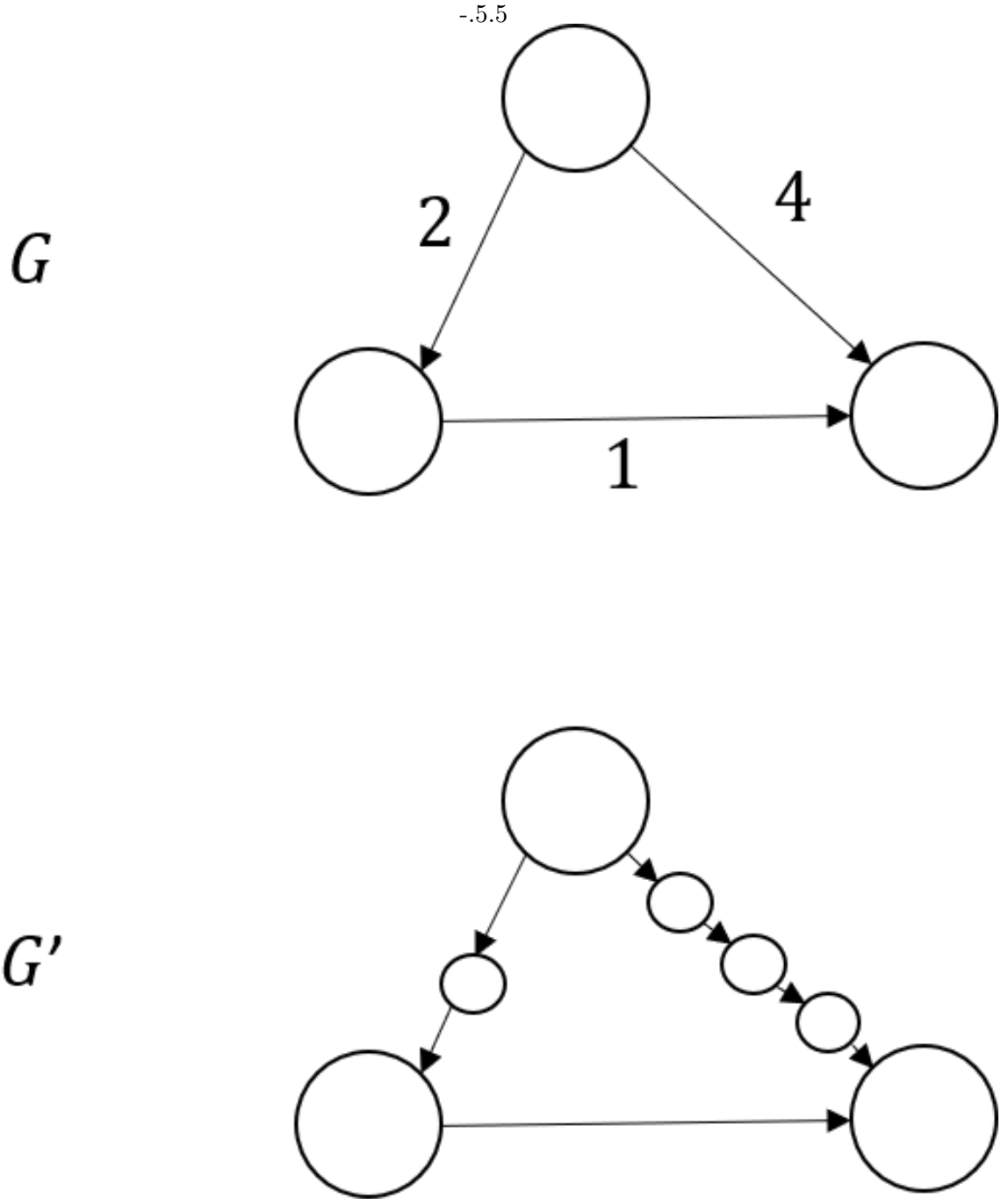
(b) Show that just like DFS, the above algorithm runs in $O(m + n)$ time.

Solution: Initializing *visited* takes $O(n)$ time. Iteration i of the for loop takes time $O(\sum_{v \in L_i} \delta(v))$, where $\delta(v)$ is the degree of v. Since no v appears in more than one L_i, the overall time is $O(n + \sum_{v \in V} \delta(v)) = O(n + m)$.

(c) We might instead want to find the shortest *weighted* path from s to each vertex. That is, each edge has weight w_e, and the length of a path is now the sum of weights of edges in the path. The above algorithm works when all $w_e = 1$, but can easily fail if some $w_e \neq 1$.

Fill in the blank to get an algorithm computing the shortest paths when w_e are integers: We replace each edge e in G with _____ to get a new graph G', then run BFS on G' starting from s. Justify your answer.

Solution: A path of w_e unweighted edges. See the below figure for e.g. a directed graph:



(d) What is the runtime of this algorithm as a function of the weights w_e? How many bits does it take to write down all w_e? Is this algorithm's runtime a polynomial in the input size?

Solution: The runtime is $O(\sum_{e \in E} w_e)$, since G' . The number of bits needed is $\Theta(\sum_{e \in E} \log w_e)$. So even though this algorithm's runtime looks like a polynomial, it takes time exponential in the input size when some w_e are large