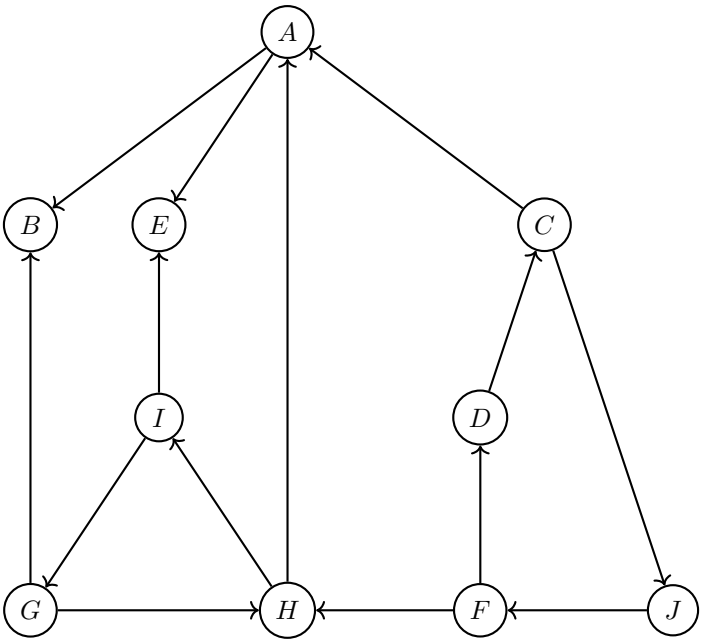
Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

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.

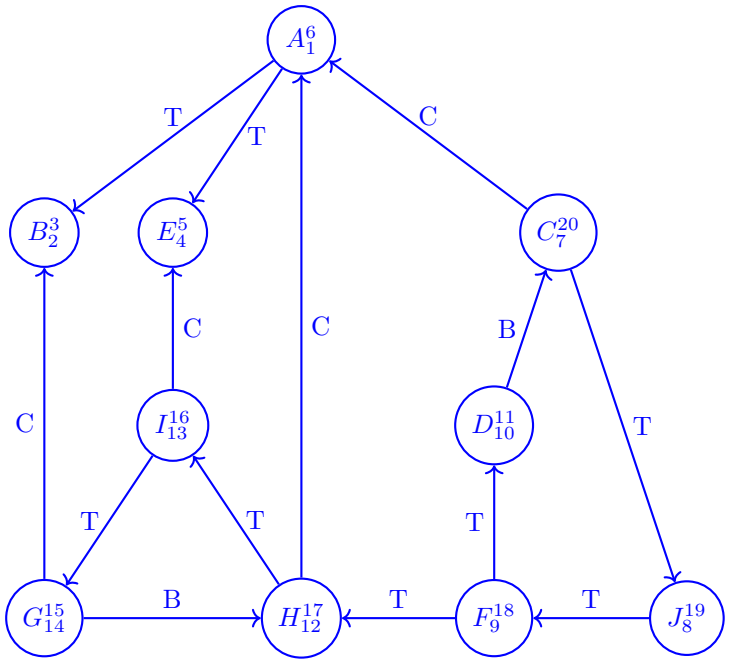
For the directed graph above, perform DFS starting from vertex A, breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as **T**ree, **B**ack, **F**orward or **C**ross.

(b) A strongly connected component (SCC) is defined as a subset of vertices in which there exists a path from each vertex to each other 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.

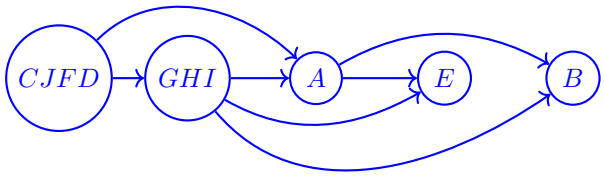
Solution:

(a)



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

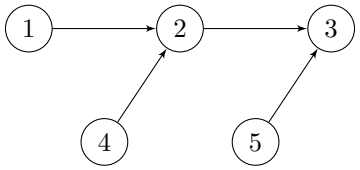
(c)



2 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.)

Example: Consider the following directed graph with 5 vertices:



The $m(i)$ values are:

- $m(1) = 1$: Only vertex 1 can reach vertex 1 (itself).
- $m(2) = 1$: Vertices 1, 2, and 4 can reach vertex 2. The smallest is 1.
- $m(3) = 1$: Vertices 1, 2, 3, 4, and 5 can all reach vertex 3. The smallest is 1.
- $m(4) = 4$: Only vertex 4 can reach vertex 4 (itself).
- $m(5) = 5$: Only vertex 5 can reach vertex 5 (itself).

(a) Show that the values $m(1), \ldots, m(n)$ 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. Can we use the same DFS approach from part (a) If not, what goes wrong, and how can we fix it?

Solution: We might try a forward DFS approach: run DFS and track the minimum vertex reachable from each vertex. When we finish exploring a vertex v, we set $m(v) = \min(v, \min_{(v,u)\in E} m(u))$.

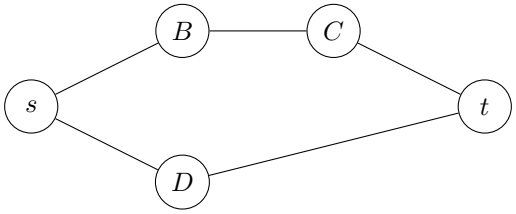
The problem is **back edges**. A back edge goes from a vertex to one of its ancestors in the DFS tree. When we're at vertex v and see a back edge to ancestor u, we know v can reach u. But u is still on the DFS stack, we haven't finished computing $m(u)$ yet! And $m(u)$ depends on $m(v)$ (since u can reach v via tree edges). This circular dependency means we can't resolve the minimum in a single forward pass.

The fix: Reverse all edges in G, i.e., replace each edge (u, v) with the edge (v, u), and then use our algorithm from part (a). This works because in the reversed graph G^R, we can reach j from i if and only if we can reach i from j in the original graph G. So finding the smallest vertex that can reach i in G^R is exactly finding the smallest vertex reachable from i in G.

3 Odd Shortest Path

Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, find the shortest path from s to t that uses an **odd number of edges**, or report that no such path exists. Your algorithm should run in $O(|V| + |E|)$ time.

Example: Consider the following undirected graph:



The shortest path from s to t overall is $s \to D \to t$ with length 2 (even). However, this path has an **even** number of edges, so it doesn't count!

The shortest **odd-length** path is $s \to B \to C \to t$ with length 3.

Solution: The key idea is to transform the problem using **state duplication**. We create a new graph $G' = (V', E')$ that tracks the parity (odd/even) of the number of edges traversed so far.

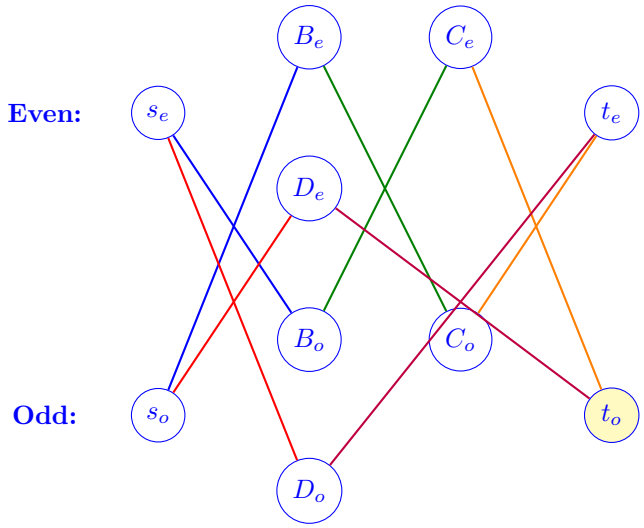
Construction of G':

- For each vertex $v \in V$, create two copies: v_{even} and v_{odd}.
- For each edge $(u, v) \in E$, add edges:
  - $(u_{\text{even}}, v_{\text{odd}})$ crossing an edge from even parity goes to odd parity
  - $(u_{\text{odd}}, v_{\text{even}})$ crossing an edge from odd parity goes to even parity

Algorithm: Run BFS from s_{even} in G'. The answer is the distance to t_{odd}.

(Note: We don't technically need s_{odd} or t_{even} since we always start at s_{even} and only care about reaching t_{odd}, but including them keeps the construction simple.)

Example:



Runtime: $|V'| = 2|V|$ and $|E'| = 2|E|$, so BFS runs in $O(|V| + |E|)$ time.

4 Bottleneck Spanning Tree

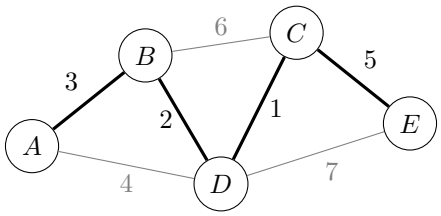
Recall that a spanning tree of a connected, undirected graph $G = (V, E)$ is a subgraph $T = (V, E_T)$ that:

- Contains all vertices of G
- Is a tree (connected and acyclic)
- Uses only edges from E

A **Minimum Spanning Tree (MST)** is a spanning tree that minimizes the *total* weight of all edges:

$$\text{MST minimizes } \sum_{e \in T} w(e)$$

Example: Consider the following weighted graph and its MST (bold edges):



A **Bottleneck Spanning Tree (BST)** is a spanning tree that minimizes the weight of the *heaviest* edge:

$$\text{BST minimizes } \max_{e \in T} w(e)$$

(a) Is every Bottleneck Spanning Tree also a Minimum Spanning Tree? Prove or give a counterexample.

Solution: No. Consider the example graph above. The MST is $\{A\text{-}B, B\text{-}D, D\text{-}C, C\text{-}E\}$ with weights $\{3, 2, 1, 5\}$, total weight 11, and max edge 5.

Now consider the alternative spanning tree $\{A\text{-}D, B\text{-}D, D\text{-}C, C\text{-}E\}$ with weights $\{4, 2, 1, 5\}$. This tree has total weight 12 and max edge 5.

Both trees have the same maximum edge weight (5), so both are Bottleneck Spanning Trees. However, the second tree has a larger total weight ($12 > 11$), so it is **not** a Minimum Spanning Tree.

(b) Is every Minimum Spanning Tree also a Bottleneck Spanning Tree? Prove or give a counterexample.

Solution: Suppose T is an MST but not a Bottleneck Tree. Then there exists some other tree T_{bottle} with a smaller max-weight edge. Let e be the max-weight edge in T. Removing e splits T into two components. T_{bottle} must have an edge e' connecting these two components. Since T_{bottle} has a smaller max weight, $w(e') < w(e)$. But then we could swap e for e' to get a tree with total weight less than T, contradicting that T was an MST.