1. Prove or disprove: Any strongly connected undirected graph with n vertices and $(n-1)$ edges is a tree.

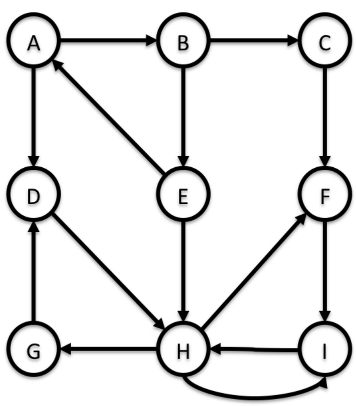
Solution: We will show that the statement is true using induction.

Let $P(n)$ denote the proposition "any strongly connected undirected graph with n vertices and $(n-1)$ edges is a tree". We will prove $\forall n, P(n)$ using induction.

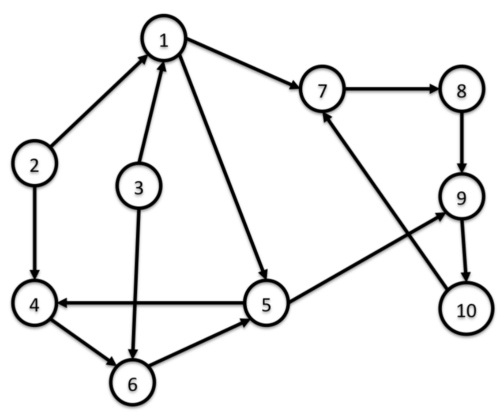
Basis step: $P(1)$ is true since a graph with 1 vertex and 0 edges is indeed a tree.

Inductive step: Assume that $P(1), P(2), ..., P(k)$ are true for an arbitrary k. We will show that $P(k+1)$ is true. Consider any strongly connected graph G with $k+1$ vertices and k edges. Then there is a vertex v with degree exactly 1. Otherwise the sum of degrees will be $\geq 2(k+1)$ but this is not possible since we know that sum of degrees is equal to $2|E|$ which in this case is $2k$. Consider the graph G' obtained by removing the vertex v and its connecting edge. Note that G' is still strongly connected and it has k vertices and $k-1$ edges. Using the induction hypothesis, we get that G' is a tree. This implies that G is a tree.

2. We know that the strongly connect components in any directed graph form a partition of vertices in the graph. So, the strongly connected components in a given graph can be represented as a partition of vertices. Consider the directed graphs G_1 and G_2 below and answer the questions that follow:



(a) Graph G_1



(b) Graph G_2

(a) Give the strongly connected components of graph G_1.

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

(b) Give the strongly connected components of graph G_2.

(b) $\{1\}, \{2\}, \{3\}, \{4, 5, 6\}, \{7, 8, 9, 10\}$

3. Given a directed graph $G = (V, E)$ and an edge $(u, v) \in E$, you want to determine if G has a cycle that contains this edge (u, v). Design an algorithm for this problem and discuss correctness and running time.

Solution: Here is the pseudocode for the algorithm:

`CheckCycle`$(G, (u, v))$
- Do a DFS in G starting with vertex v (i.e., execute ~~`DFS`(G, v)~~ `explore`(G, v))
- If (u was explored in the above DFS) return("yes")
- else return("no")

The correctness follows from the next two claims:

Claim 1: If there is a cycle in G containing the edge (u, v), then the algorithm outputs "yes".

Proof. Let $u, v, x_1, x_2, ..., x_l, u$ be a cycle in G that contains the edge (u, v). This means there exists is a path from v to u in G which is $v, x_1, ..., x_l, u$. This implies that u will be explored when doing DFS starting from v in G. In this case, the algorithms will output "yes". □

Claim 2: If there is no cycle in the graph containing (u, v), then the algorithm outputs "no".

Proof. We prove the contrapositive of the above statement. That is, we will show that if the algorithm outputs "yes", then there is a cycle in G containing (u, v). The algorithm outputs "yes" when u is explored when doing DFS starting from v in G. Consider the DFS tree with respect to the DFS execution and let $v, x_1, x_2, ..., x_l, u$ be the path from v to u in this DFS tree. Now, consider the sequence of vertices $u, v, x_1, ..., x_l, u$. This is a cycle in the graph G and contains the edge (u, v). □

Running time: Doing a DFS in G takes $O(n + m)$ time. So, the running time of the algorithm is $O(n + m)$.

4. You are given a directed acyclic graph $G = (V, E)$ in which each node $u \in V$ has an associated price, denoted by $price(u)$, which is a positive integer. The cost of a node u, denoted by $cost(u)$, is defined to be the price of the cheapest node reachable from u (including u itself). Design an algorithm that computes $cost(u)$ for all $u \in V$. Give pseudocode and discuss correctness and running time.

Solution:

Main idea: For any vertex u, let $v_1, ..., v_l$ be all the vertices to which u has an outgoing edge. Then note that $cost(u) = \min(price(u), cost(v_1), cost(v_2), ..., cost(v_l))$. So, if we somehow have the value of cost of vertices $v_1, ..., v_l$, then we can compute $cost(u)$. We also know that for a sink vertex v (*sink vertices are those vertices that do not have any outgoing edge*), we have $cost(v) = price(u)$ since the cheapest node reachable from a sink vertex is itself. The main challenge now is to figure out in what order to go about computing the value of the cost of the vertices.

An order that works for this problem is reverse topological ordering of the vertices. This is because if we compute in this order, then for any vertex u and its neighbours $v_1, ..., v_l$ as in the previous paragraph, we would have computed the value of $cost(v_1), ..., cost(v_l)$ before we get to u. We can easily convert this idea to the following pseudocode.

`ComputeCost(`$(V, E), price$`)`
- Compute the topological ordering of vertices using algorithm discussed in class.
- Let L denote the list of vertices in reverse topological ordering
- For $i = 1$ to $|V|$
    - Let u be the i^{th} element in the list L
    - $cost(u) \leftarrow price(u)$
    - For all v such that $(u, v) \in E$:
        - $cost(u) \leftarrow \min(cost(u), cost(v))$

We can prove the correctness using Mathematical Induction. Consider the proposition:
$P(i)$: The algorithm computes the cost of the i^{th} vertex in the list L correctly.

Basis step: $P(1)$ holds since the first vertex of the list L is a sink vertex and for any such vertex, its cost is the same as its price.

Inductive step: We assume that $P(1), P(2), ..., P(i)$ hold and show that $P(i + 1)$ holds. Consider the $(i + 1)^{th}$ vertex u in the list L. Let $v_1, ..., v_l$ denote all the vertices to which u has an out-going edge in the graph. Then from the induction hypothesis we know that the algorithm has correctly computed the cost of $v_1, ..., v_l$ since all these vertices are earlier in the list L than u. This means that the cost of u will be correctly computed.

Running time: The running time for computing the reverse topological ordering is $O(n + m)$ as discussed in class. After this, the algorithm simply iteratively considers the vertices of the graph and for every vertex u, it spends time proportional to the out-degree of u. So, the total time for this will be proportional to the number of vertices n plus the number of edges m. So, the running time of the algorithm is $O(n + m)$.