- Please note that one of the main goals of this course is to design efficient algorithms. So, there are points for efficiency, even though we may not explicitly state this in the question.
- Unless otherwise mentioned, assume that graphs are given in adjacency list representation.
- In the lectures, we have discussed an $O(V+E)$ algorithm for finding all the SCCs of a directed graph. We can extend this algorithm to design an algorithm `CreateMetaGraph(G)` that outputs the meta-graph of a given directed graph in $O(V+E)$ time. You may use `CreateMetaGraph(G)` as a sub-routine for this homework.
- The other instructions are the same as in Homework-1.

There are 6 questions for a total of 82 points.

---

1. A tree is defined as an undirected graph containing no cycles. An undirected graph is said to be connected iff, for every pair of vertices, there is a path between them. For this question, you have to show the following statement:

   Any connected undirected graph with n vertices and $(n-1)$ edges is a tree.

   We will prove the statement using Mathematical Induction. The first step in such proof is to define the propositional function. Fortunately, this is already given in the statement of the claim for this problem.

   $P(n)$: Any connected undirected graph with n vertices and $(n-1)$ edges is a tree.

   The base case is simple. $P(1)$ holds since any graph with 1 node and 0 edges is indeed a tree. For the inductive step, we assume that $P(1), P(2), ..., P(k)$ holds for an arbitrary $k \geq 1$, and then we will show that $P(k+1)$ holds. Consider any connected graph G with $(k+1)$ nodes and k edges. You are asked to complete the argument by doing the following:

   (a) (3 points) Show that G has a node v with degree 1.

   Solution: Let $G=(V,E)$. We will prove the statement by contradiction. Assume for the sake of contradiction that there does not exist any vertex that has degree 1. This implies that all vertices have a degree of at least 2. Moreover, we know that the sum of the degree of vertices of an undirected graph is precisely $2 \cdot |E|$. This gives us the following:

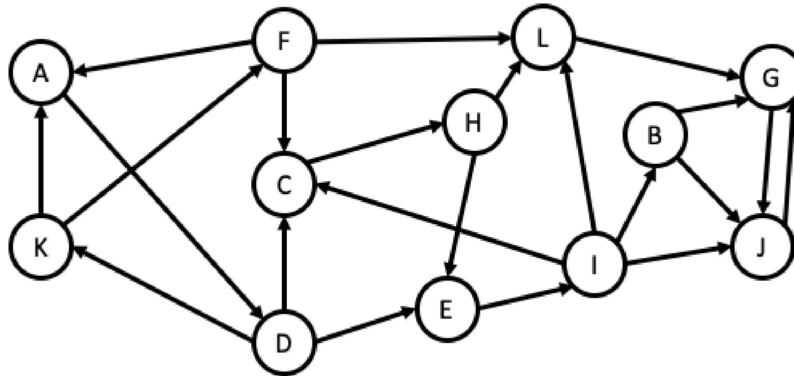
   $$2 \cdot |E| = \sum_{u \in V} deg(u) \geq \sum_{u \in V} 2 = 2 \cdot |V|$$

   This implies that $|E| \geq |V| = (k+1)$ which is a contradiction since G has $(k+1)$ nodes and k edges. Hence, there is a vertex in G with degree 1.

   (b) (2 points) Consider the graph G' obtained from G by removing vertex v and its edge. Now, use the induction assumption on G' to conclude that G is a tree.

   Solution: Removing the degree-1 vertex v and its edge from G does not disconnect the graph. This means that G' is a connected graph with k nodes and $(k-1)$ edges. From the induction assumption, G' is a tree. This implies that G is a tree since adding a node and connecting it to any node in G' does not create any cycles.

2. Consider the following directed graph and answer the questions that follow:



(a) (½ point) Is the graph a DAG?

(a) **No**

(b) (1 point) How many SCCs does this graph have?

(b) **5**

(c) (½ point) How many source SCCs does this graph have?

(c) **1**

(d) (1 point) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the pre-number of vertex F?

(d) **20**

(e) (1 point) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the post-number of vertex J?

(e) **10**

(f) (1 point) Is it possible to add a single edge to this graph so that the graph becomes a strongly connected graph? If so, which edge would you add?

(f) **Yes, add (J, K)**

3. You are given a directed 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 most expensive node reachable from u (including u itself). Your goal is to compute the cost of every node in the graph (this can be produced as an array *cost* of size $|V|$).

   (a) (9 points) Give a linear time algorithm that works for DAG's. (*Hint: Handle the vertices in a particular order*)

   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 $cost(u) = \max(price(u), cost(v_1), cost(v_2), ..., cost(v_l))$. 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. The ideas can be summarised as 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 ith element in the list L
     - $cost(u) \leftarrow price(u)$
     - For all v such that $(u, v) \in E$:
       - $cost(u) \leftarrow \max(cost(u), cost(v))$

   We can prove the correctness using Mathematical Induction. Consider the proposition:
   $P(i)$: The algorithm computes the cost of the ith 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 outgoing 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 $O(|V|+|E|)$ 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 $|V|$ plus the number of edges $|E|$. So, the running time of the algorithm is $O(|V| + |E|)$.

   (b) (9 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

   Solution: We will solve the problem on directed graphs by reducing it to the problem on DAGs (the algorithm for which we have already designed in the previous part). Let $V_1, ..., V_k$

denote the strongly connected components of a given directed graph $G = (V, E)$. Consider the meta-graph $G' = (\{1, ..., k\}, E')$ of G. We define the price of the meta-node i of the SCC V_i to be the price of the most expensive node in C. Now we can run the algorithm of the previous part on graph G' to compute the cost $cost[1], ..., cost[k]$ of all the meta-nodes $1, ..., k$. Finally, the cost of all nodes within an SCC V_i is set to be $cost[i]$. This idea can be summarised as the following pseudocode:

`ComputeCostGeneral(`$G, price$`)`
- Run `CreateMetaGraph(`G`)` to obtain SCCs $V_1, ..., V_k$ and meta graph $G' = (\{1, ..., k\}, E')$
- For all $i \in \{1, ..., k\}$:
  - $price'(i) \leftarrow \max_{v \in V_i} (price(v))$
- $cost' \leftarrow$ `ComputeCost(`$G', price'$`)`
- For all $i \in \{1, ..., k\}$:
  - For every $v \in V_i$: $cost(v) = cost'(i)$
- return($cost$)

Running time: Creating the meta-graph G' takes $O(|V| + |E|)$ time. Computing the price of meta node i takes $O(|V_i|)$ time. So, computing the price of all meta nodes takes $\sum_i O(|V_i|) = O(|V|)$ time. Since G' is a DAG with at most $|V|$ vertices and $|E|$ edges, the time for running `ComputeCost` is $O(|V| + |E|)$. Finally, computing the cost of all vertices $v \in V$ takes $O(|V|)$ time. So, the overall running time is $O(|V| + |E|)$.

Correctness of the algorithm follows from the following argument. Consider any node $u \in V$ and let c denote the cost of u. This means that the price of the most expensive node v that is reachable from u is c. We can do a case analysis based on which SCCs u and v belong to:

- Case 1: u and v are in the same SCC.
  In this case, the second line of the pseudocode sets the cost of u to be price of the most expensive node in the SCC of u which will be c.
- Case 2: u and v are in different SCCs.
  Let $u \in V_i$ and $v \in V_j$ for $i \neq j$. In this case $cost'(i) \geq cost'(v)$ since j is reachable from i in DAG G'. This means that cost of u will be correctly set to c in the last for-loop of the pseudocode

Give running time analysis and proof of correctness for both parts.

4. Given a directed graph $G = (V, E)$ that is not a strongly connected graph, you have to determine if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected. In other words, you must determine if there exists a pair of vertices $u, v \in V$ such that adding a directed edge from u to v in G converts it into a strongly connected graph. Design an algorithm for this problem. Your algorithm should output "yes" if such an edge exists and "no" otherwise.

   (a) (9 points) Give a linear time algorithm for DAGs.

Solution: The algorithm follows from the following main idea stated as a claim.

Claim 1: Let G be a DAG. There is a pair of vertices u, v such that adding a directed edge from u to v converts G into a strongly connected graph if and only if G has a unique source and sink vertex. Moreover, if G has a unique source and sink vertices, adding an edge from the unique sink vertex to the unique source vertex converts G into a strongly connected graph.

We break the if and only if part into the following two claims:

Claim 1.1: If a DAG G has a unique source s and a unique sink t, then adding a directed edge from t to s converts G to a strongly connected graph.

Proof. We claim that every vertex in G is reachable from the unique source s. Suppose, for the sake of contradiction, there is a vertex v which is not reachable from s. Let u_1 be the vertex with a directed edge to v. Note that such an edge exists; otherwise, v is also a source vertex. Again consider a vertex u_2 that has a directed edge to u_1 and keep traversing back along the incoming edges. This process will eventually terminate in a vertex u with no incoming edges. This vertex u cannot be equal to s since then v becomes reachable from s. Hence we get a contradiction with the fact that s is the unique source.

Similarly, we can argue that t is reachable from all vertices in G. So, adding a directed edge from t to s connects all pairs of vertices. Since then, one can reach t from any vertex u, take the edge (t, s) and then reach any other vertex v in the graph. □

Claim 1.2: If adding a directed edge from t to s in a DAG G converts it to a strongly connected graph, then s is the unique source and t is the unique sink of G.

Proof. Suppose, for the sake of contradiction, s is not the unique source (t not being the unique sink will be similar). Then there is a topological ordering of the vertices such that s is not the first vertex in this topological ordering. Consider the first vertex v in this topological ordering. Even after adding the edge (t, s), there is no path from s to v since otherwise, the same path will exist in G. □

Claim 1 suggests that all we need to check is whether the given DAG has a unique source and a unique sink. This can be done by checking if there is a unique vertex with no incoming edges (this is the unique source) and a unique vertex with no outgoing edges (this is the unique sink). So, the algorithm simply computes the in-degree and out-degree of all vertices. The algorithm is summarised in the following pseudocode:

`CheckConnectivity`($G(= (V, E))$)
- Compute $indegree(u)$ and $outdegree(u)$ for every vertex $u \in V$
- If there is a unique vertex with 0 indegree and a unique vertex with 0 outdegree
  return("yes")
- else
  - return("no")

Running time: The out-degree of a vertex v can be computed by counting the number of vertices in the linked list for v in the adjacency list of G. So, computing the out-degrees of all the vertices costs $O(V + E)$ time. Similarly, the in-degrees can be computed by doing the same

on the reverse graph, and we have seen that creating the reverse graph takes $O(V+E)$ time. So, the overall running time of the algorithm is $O(V+E)$.

(b) (9 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

Solution: Consider the meta-graph G' of the given directed graph. Note that G' is a DAG. If G' has a unique source and unique sink, then adding a directed edge from a node in the sink SCC to a source SCC converts G' and hence G to a strongly connected graph. G' does not have a unique source and sink; then, by the same argument as before (claim 1.2), we conclude that it is impossible to create a strongly connected graph by adding a single edge. The algorithm based on the above idea is now simple. We construct the meta-graph DAG G' and check if this has a unique source and a unique sink vertex. If so, the algorithm outputs "yes"; otherwise, it outputs "no". The pseudocode is given below.

```
CheckConnectivityGeneral(G)
  - Use the algorithm developed in class to construct the meta-graph G'
  - return(CheckConnectivity(G'))
```

Running time: Constructing G' takes $O(V+E)$ time. After this, we make a call to `CheckConnectivity(G')` that has $|V|$ vertices and $|E|$ edges in the worst case. So, the overall running time is $O(V+E)$.

Give running time analysis and proof of correctness for both parts.

5. (18 points) A directed graph $G = (V, E)$ is called one-way-connected if, for all pair of vertices u and v, there is a path from vertex u to v **or** there is a path from vertex v to u. Note that the "or" in the previous statement is a logical OR, not XOR. Design an algorithm to check if a given graph is one-way-connected. Give running time analysis and proof of correctness.

Solution: Let $V_1, V_2, ..., V_k$ be the vertex sets of the strongly connected components of the graph G. As discussed in class, a $O(n+m)$ algorithm exists to find the strongly connected components in any directed graph. We will construct another graph $G^{scc} = (V^{scc}, E^{scc})$ in which there is a node corresponding to each strongly connected component of G. Let the nodes in G^{scc} be denoted by $1, 2, ..., k$. As for the edges, $(i, j) \in E^{scc}$ iff there is a vertex $u \in V_i$ and a vertex $v \in V_j$ such that $(u, v) \in E$.

Claim 1: G^{scc} is a DAG.

Proof. Assume for the sake of contradiction that there is a cycle in G^{scc}. Let $i_1, i_2, ..., i_l, i_1$ denote this cycle in G^{scc}. Then note that this implies that for any pair of vertices $u, v \in V_{i_1} \cup V_{i_2} \cup ... \cup V_{i_l}$ there is a path from u to v and there is a path from v to u. We only prove the existence of a path in one direction and the other direction follows by symmetry. Suppose $u \in V_{i_p}$ and $v \in V_{i_q}$ such that $p < q$. Note that since there is an edge from i_p to i_{p+1} in G^{scc}, this means that there is a directed edge from a vertex $x \in V_{i_p}$ to a vertex $y \in V_{i_{p+1}}$. Since V_{i_p} is a strongly connected component, this means that there is a path from u to x in G, and this further means that there is a path from u to y in G. This further means that there is a path from u to all vertices in $V_{i_{p+1}}$ since $V_{i_{p+1}}$ is a strongly connected component. We can extend this argument to $V_{i_{p+2}}, ..., V_q$ proving that there is a path from u to any vertex in V_q and in particular path from u to v.

This gives us a contradiction since the above implies that $V_{i_1}, ..., V_{i_l}$ should form a single strongly connected component. □

Next, we give the crucial claim that will give us our algorithm.

Claim 2: G is one-way connected iff G^{scc} has a unique topological ordering.

Proof. The proof follows from the proof of the next two claims. □

Claim 2.1 If G^{scc} has a unique topological ordering, then G is one-way connected.

Proof. Let $i_1, i_2, ..., i_k$ be the unique topological ordering of G^{scc}. It follows that $(i_1, i_2), (i_2, i_3), ..., (i_{k-1}, i_k) \in E^{scc}$. Consider any two vertices $u, v \in V$. Let $u \in V_{i_p}$ and $v \in V_{i_q}$. WLOG assume that $p < q$. Then by the same arguments as that in the proof of Claim 1, we get that there is a path from u to v. □

Claim 2.2 If G^{scc} does not have a unique topological ordering, then G is not one-way connected.

Proof. Since G^{scc} does not have a unique topological ordering, it means that there are two orderings and two nodes i_p, i_q in V^{scc} such that i_p appears before i_q as per the first ordering and i_p appears later than i_q as per the second ordering. The former means that there is no path from i_q to i_p in G^{scc} and the latter means that there is no path from i_p to i_q in G^{scc}. This in turn implies that there is no path from any vertex in V_{i_p} to any vertex in V_{i_q} in G and vice versa. □

Now what remains to be done is to figure out a way to determine if there is a unique topological ordering of G^{scc}. This is easily done using the following algorithm.

```
OneWayConnected(G)
```

- Use algorithm discussed in class to find the strongly connected components and construct the graph $G^{scc} = (V^{scc}, E^{scc})$
- $G' \leftarrow G^{scc}$
- For $i = 1$ to $|V^{scc}|$
    - If there are more than two vertices with in-degree 0 in G'
        - return("G is not one-way connected")
    - Else let v be the vertex in G' with no incoming edges
    - Let G'' be the graph obtained from G' by removing v and its edges
    - $G' \leftarrow G''$
- return("G is one-way connected")

Proof of correctness: If the algorithm outputs "G is one-way connected", that means that at each step it found a unique vertex with 0 in-degree. This means that there is a directed edge from this unique vertex in the ith step to the unique vertex in the $(i+1)^{th}$ step which further implies that there is a unique topological ordering.

If in any iteration, there are at least two vertices with 0 in-coming edges, then that means that these vertices can be in any relative order with respect to each other. Which means that unique ordering is not possible. Now the correctness follows from Claim 2.

Running time: Tarjan's algorithm takes $O(n + m)$ and so does constructing G^{scc}. The time taken within the for loop is proportional to the size of G^{scc} which is at most $O(n + m)$. So, the total running time of the algorithm is $O(n + m)$.

6. (18 points) Design an algorithm that given a directed acyclic graph $G = (V, E)$ and a vertex $u \in V$, outputs all the nodes that have a simple path to u with a length that is a multiple of 3. *(Recall that a simple path is a path with all distinct vertices.)*

   Give running time analysis and proof of correctness.

Solution: We give the solution for the following problem variant (*the solution for the above version will follow similar ideas*).

> Given a directed acyclic graph $G = (V, E)$ and a vertex u, design an algorithm that outputs all vertices $S \subseteq V$ such that for all $v \in S$, there is an even length simple path from u to v in G.

Algorithm: Given a graph $G = (V, E)$, we construct a new graph $G' = (V', E')$ by "splitting" every vertex in V into two vertices. For a vertex $i \in V$ we create two copies of the vertex and call it i^{odd} and i^{even}. That is, for every vertex $i \in V$, V' has i^{odd} and i^{even}. As for edges, for every edge $(i, j) \in E$, we add two edges (i^{odd}, j^{even}) and (i^{even}, j^{odd}) to E'. Now, we just run $DFS(u^{even})$ on the graph G' and output all i such that i^{even} is visited while doing $DFS(u^{even})$.

`EvenPath`(G)
- Construct the graph G' as described above
- Execute `DFS`(G', u^{even})
- For all $i \in V$:
  - If (i^{even} is marked explored) append i to list L
- return(list L)

The following two claims prove the correctness of our algorithm.

Claim 1: For every vertex v such that there is an even length path from u to v, the above algorithm will output v.

Proof. Let $u, i_1, i_2, ..., i_k, v$ be an even length path from u to v. Then k is an odd number. Given this, note that there is a path $u^{even}, i_1^{odd}, i_2^{even}, ..., i_k^{odd}, v^{even}$ in the graph G'. So, v^{even} is reachable from u_{even}. So our algorithm will output v. □

Claim 2: If our algorithm outputs a vertex v, then there is an even length path from u to v in G.

Proof. Our algorithms outputs v iff v^{even} is reachable from u^{even} in G'. Consider the path from u^{even} to v^{even} in the DFS tree obtained during execution of $DFS(u^{even})$ on G'. Let this path be denoted by $u^{even}, i_1^{\ell_1}, i_2^{\ell_2}, ..., i_k^{\ell_k}, v^{even}$. Note that $\ell_1 = odd$ since all directed edges are from vertices labelled even to vertices labelled odd or vertices labelled odd to vertices labelled even. This further implies that $\ell_2 = even, \ell_3 = odd, ..., \ell_k = odd$. This means that k is odd. This implies an even length path in G from u to v since $u, i_1, i_2, ..., i_k, v$ is a path in G. □

Running time: The running time is proportional to the running time of DFS on a graph that is at most double the size of G. So, the running time is $O(|V| + |E|)$.