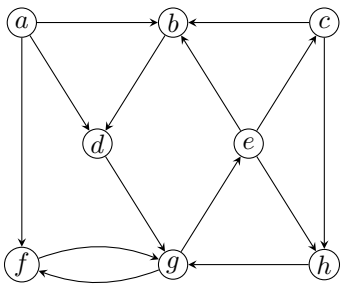
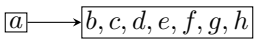
Problem 1 (10 points). What is the maximum number of edges a directed acyclic graph with n vertices can have? Prove your answer.

Solution. Answer is $n(n-1)/2$. Any directed graph has at most n^2 edges. However, since the DAG has no cycles it cannot contain a self loop, and for any pair of vertices x and y at most one edge, either (x,y) or (y,x), can be included. Therefore the number of edges can be at most $(n^2-n)/2$ as desired. It is possible to achieve $n \cdot (n-1)/2$ edges. Label n vertices $1,2,\ldots,n$ and add an edge (x,y) if and only if $x < y$. This graph has the appropriate number of edges and cannot contain a cycle (any path visits an increasing sequence of vertices).

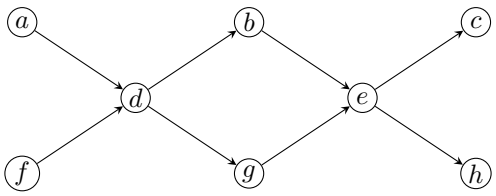
Problem 2 (10 points). Draw the meta-graph for the following directed graph.



Solution.



Problem 3 (10 points). How many different linearizations for the following graph?



Solution. The answer is 8. The linearization should be in the form of $(\{a,f\},d,\{b,g\},e,\{c,h\})$. So the number of possible lineariztion is $2^3 = 8$.

Problem 4 (20 points). Let $G = (V,E)$ be a DAG. Design an $O(|V|+|E|)$ time algorithm to decide whether there is only one possible linearization for G.

Solution. A DAG G has only one possible linearization if and only if there is a directed edge between each pair of consecutive vertices in one linearization. To see this, suppose that there exists an edge between any two consective vertices in linearization $L[1\cdots n]$. We now prove that L is the only possible linearization. Suppose conversely that there exists another linearization L'. Let k be the *first* different vertex between L and L', i.e., $L[i] = L'[i]$ for $1 \le i < k$ and $L[k] \neq L'[k]$. Let $u = L[k]$ and $v = L'[k]$. We have that v must be after u in L and u must be after v in L'. Since there exists edge among every consecutive edges in L, we know that there exists a path from u to v in G. This contradicts that L' is a linearization. Now we prove the other side. Suppose that there are two consecutive vertices in the linearization L which are not connected by an edge. Then we can swap them to get a new linearization.

The above statement immediately suggests an algorithm to decide whether a DAG has a unique linearization: We can use DFS to get one linearization, which takes $O(|E|+|V|)$ time. We then check every consecutive vertices u and v in the linearization whether $(u,v) \in E$. This again can be done in $O(|E|+|V|)$ time, as we only need to examine the adjacent edges for each vertex once (suppose that we use the adjacency list representation).

Problem 5 (20 points). Your job is to prepare a lineup of n awardees at an award ceremony. You are given a list of m constraints of the form: awardee i wants to receive his award before awardee j. Design an algorithm to either give such a lineup that satisfies all constraints, or return that it is not possible. Your algorithm should run in $O(m+n)$ time.

Solution. This can be formulated as a graph problem. We create a directed graph G with each awardee denoting a vertex. For every constraint "awardee i wants to receive an award before j", add an edge (i, j) to G. Preparing the lineup would mean ordering the vertices. If there is an edge (i, j) in G, then i should appear before j in the order. Thus, we want G to be acyclic, i.e., a DAG. If G has a cycle, then no ordering is possible. We can perform a DFS to check if there is a back edge (and a cycle). If we don't find a back edge, then the graph is a DAG. One possible ordering of vertices in a DAG is through a topological sort (specifically, decreasing order of post identifiers when performing a DFS). We can place the vertex appearing first in the topological sort at the head of the line, the second vertex at the second position, and so on. Since both DFS and topological sort are linear time, the overall approach takes $O(m+n)$ time.

Problem 6 (30 points). All streets in city X are one-way. Residents want to know whether there is a way to drive legally from any intersection in the city to any other intersection.

1. Formulate this problem as a graph problem and design a linear time algorithm for it.

   Solution. We create a directed graph $G = (V, E)$, where each vertex denotes one intersection and there is an edge between (v_i, v_j) only when there is an road going from intersection v_i to v_j. Clearly, we have that there is always a way from any intersection to any other intersection if and only if G is strongly connected, i.e., there is only one strongly connected component in G.

   In the lecture, we have introduced the algorithm (a variant of DFS) to find all strongly connected components in directed graph in linear time. We can call this algorithm, and verify if the returned number of strongly connected component (in variable num_cc) equals to 1.

   Here is an alternative algorithm (which is essentially the same with the above algorithm): starting from any vertex $s \in V$, we run explore (G, s) to find all vertices that can be reached from s. Then we build the *reverse graph* G_R of G and run explore (G_R, s) again. If both procedures can find all other vertices in G, then the whole graph is a single strongly connected component; otherwise, G is not strongly connected. The algorithm runs twice of explore and hence takes linear time.

2. Suppose that the answer for the above question is false. Residents further want to know whether the following weaker statement is true: if you start driving from city town hall (assume that it locates at one intersection), then no matter where you reach, there is always a way to drive legally back to the city town hall. Formulate this problem as a graph problem and design a linear time algorithm for it.

   Solution. We use the same graph in the last question and label the city town hall as vertex s. Then the statement can be formulated as "if vertex v can be reached by s, there is a way from v back to s". We can design the following algorithm: we first run explore (G, s) and store all explored vertices in set V_1. Then we create the reversed graph G_R of G and run explore (G_R, s). If the second explore can find all vertices in V_1, the statement is true; otherwise, the statement is false.

   In fact, the statement of "if vertex v can be reached by s, there is a way from v back to s" holds if and only if s belongs to a sink strongly connected component. This gives rise to an alternative algorithm: we can use the algorithm introduced in lecture to build the meta-graph G_M of G, which takes linear time. We then find the index of the component in which s locates (i.e., from cc array). Then we check in G_M whether the component containing s is a sink componenet, i.e., whether its out-degree is 0. If so the statement is true; otherwise, the statement is false. This algorithm runs in linear time as building meta-graph takes linear time.