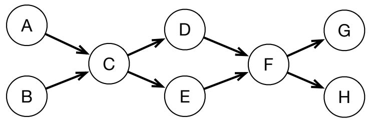
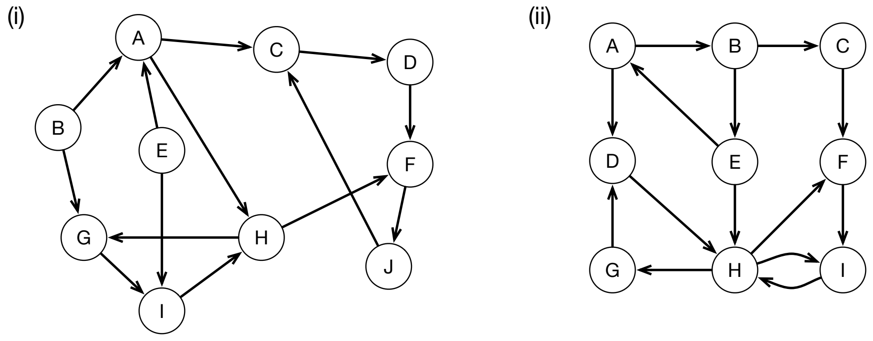
Basic Algorithms — Fall 2020 — Problem Set 3a
Due: Wed, Oct 21, 11am

1. **Kahn mechanics.** Run Kahn's topological sorting algorithm on the following graph. Whenever there is a choice of vertices, choose the one that is alphabetically first. Show the intermediate steps of the algorithm, as well as the resulting topological ordering of the vertices.

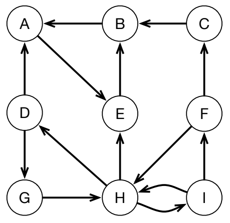


2. **DFS mechanics.** For each graph, show the "DFS forest" resulting from an execution of DFS. Whenever there is a choice of vertices, choose the one that is alphabetically first. Identify the cross, forward, and back edges, and label each vertex with its *discovery* and *finishing* time.



3. **DFS mechanics.** Run Tarjan's DFS-based topological sorting algorithm on the graph in Exercise 1. Whenever there is a choice of vertices, choose the one that is alphabetically first. Show the "DFS forest", including *discovery* and *finishing* times. Give the resulting topological ordering of the vertices.

4. **DFS mechanics.** Run the strongly-connected components algorithm as presented in class on this graph. Show the "DFS forest", including *discovery* and *finishing* times, for both runs of the DFS algorithm. Draw the resulting component graph. As usual, when faced with a choice among vertices, pick the one that is alphabetically first.



5. **Building the component graph.** Suppose you run an SCC algorithm that gives you the strongly connected components $C_1, \ldots, C_k$ for a graph G with with n vertices and m edges. Each C_i is given to you as a list of vertices.

   From this information, give the details of an algorithm that computes two things:

   (i) an array mapping $v \in V$ to $j \in \{1, \ldots, k\}$, where $v \in C_j$; that is, an array *Map*, indexed by $v \in V$, such that $Map[v] = i$ if $v \in C_i$;

   (ii) an adjacency list representation of the component graph G^{scc}; that is, an array $L[1..k]$ where each $L[i]$ is a list of indices j such that $i \to j$ is an edge in G^{scc}.

   Your algorithm should run in time $O(n + m)$.

   To be precise: your algorithm should take as input the graph G in adjacency list representation, along with sets $C_1, \ldots, C_k$. Each C_i is just a set of vertices (represented, for example, as a list).

   Note that in your adjacency list representation of G^{scc}, you should take care that edges in the component graph are not duplicated. For example, if there are two edges in G leading from a vertex in C_i to a vertex in C_j, the adjacency list $L[i]$ should contain just a *single copy* of j.

6. **Back to your roots.** Let $G = (V, E)$ be a directed graph. A *root* of G is a node from which every other node is reachable (i.e., r is a root means: for every $v \in V$ there is a path from r to V).

   Now suppose G is *acyclic*. Give an argument for each of the following:

   (i) if r is a root, then r must have in-degree 0;

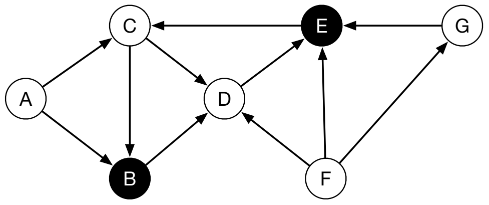
   (ii) if there is any root, then there can be *at most* one node of in-degree 0 in G;

   (iii) if there is exactly one node r of in-degree zero in G, then r is a root.

   Note: this says that in a DAG, there can be at most one root, and that the graph has a root if and only if there is a unique node of in-degree zero (in which case, that node is the root).

7. **Alternating paths.** Let $G = (V, E)$ be a directed graph. Suppose every node v has a color $c[v]$, which is either *black* or *white*. Let k be a positive integer. A path in G is called *k-alternating* if it changes color at least k times. Note that such a path need not be simple (that is, it may contain repeated nodes).

   For example, consider the following graph:



   The path $A \to C \to B \to D \to E$ is a 3-alternating path, as there are 3 alternations (from white to black as we go from C to B, from black to white as we go from B to D, and then from white to black as we go from D to E). The path $A \to C \to B \to D \to E \to C \to B$ is a 5-alternating path.

   (a) Design an algorithm that determines if an *acyclic* graph has a k-alternating path. Your algorithm takes as input a graph $G = (V, E)$ (in adjacency list representation), the color vector c, and a positive integer k. It outputs *true* or *false*. Your algorithm should run in time $O(|V| + |E|)$.

   Hint: start by running a topological sorting algorithm, and then compute for each vertex v the value $m[v]$, which is the largest k such that there is a k-alternating path starting at v.

   (b) Now generalize your algorithm to solve this problem on an *arbitrary* graph.

   Hint: start by running an SCC algorithm, and use the algorithm from part (a) as a subroutine.

8. **Carrying stones.** You are given a directed *acyclic* graph $G = (V, E)$ with n vertices and m edges, along with a node $t \in V$. At each vertex $v \in V$, there are a number of stones $q[v]$ (so $q[v]$ is a nonnegative integer). In addition, each edge $e \in E$ has a capacity $c(e)$, which is a positive integer.

   Now, you can start out at any node with an empty bag, and then can travel along a path picking up stones along the way, adding them to your bag as you go: if the path visits node v, you can pick up $q[v]$ stones and add them to your bag. However, whenever you traverse an edge along this path, the number of stones in your bag cannot exceed the capacity of this edge: if you have too many stones, you can drop some of them off before traversing the edge. You can think of each edge e as a "bridge" that cannot support the weight of more than $c(e)$ stones.

   Your goal is to find a path, starting at any node with empty bag, and ending at t, so that you arrive at t with as many stones as possible (if there are any stones at t itself, you can also add those to your bag).

   To simplify the problem, instead of computing an optimal path, just compute the number of stones that such an optimal path will yield.

   Show how to solve this problem in time $O(n + m)$.

   Hint: Start by running a topological sorting algorithm on the *reverse* of the graph G. Then compute for each node v the value $m[v]$, which is the maximum number of stones you can have in your bag when you arrive at node v (along any path in the *original* graph, starting at any node, including stones you pick up at v).

   Food for thought (will not be graded): Suppose you are also given a node $s \in V$ as input, and the goal is to find the number of stones on an optimal path that starts at s and ends at t. How would you modify your algorithm?

   More food for thought (will not be graded): Suppose you also want to print out the optimal path itself. How would you modify your algorithm?

Even more food for thought (will not be graded): Suppose you are not allowed to drop any stones, and you also want to print out the optimal path, including the number of stones to pick up at each node along the path. How would you modify your algorithm?

9. **Gotham City.** The mayor of Gotham City has made all the streets of the city one way. We can model the street layout as a directed graph $G = (V, E)$, where the vertices V of the graph correspond to street intersections (locations), and the edges E correspond the the roads connecting intersections. So for every pair of distinct vertices $u, v \in V$, if there is an edge from u to v, then there is no edge from v to u. Assume a sparse representation for G.

   (a) The mayor claims that one can legally drive from any one location in the city to any other. Show how to verify the mayor's claim using a linear-time algorithm that takes as input the graph G.

   (b) Suppose the mayor's claim is not necessarily true: that it may not be possible to legally drive from any one location to any other.

   Let us call a location *privileged* if you can legally drive from that location to any other location. Said another way, v is privileged if the following property holds: for every location w, one can legally drive from v to w.

   Give a linear-time algorithm that outputs *all* of the privileged locations, given the graph G as input.

   (c) Suppose again that the mayor's claim is not necessarily true.

   Let us call a location v a *safe space* if wherever you legally drive starting at v, you can always legally drive back to v. Said another way, v is a safe space if the following property holds: for every location w, if one can legally drive from v to w, then one can legally drive back from w to v.

   Give a linear-time algorithm that outputs *all* safe spaces, given the graph G as input.

   Hints: all of the above problems can be very easily solved by using standard algorithms as subroutines; for part (b), Exercise 6 may be useful.