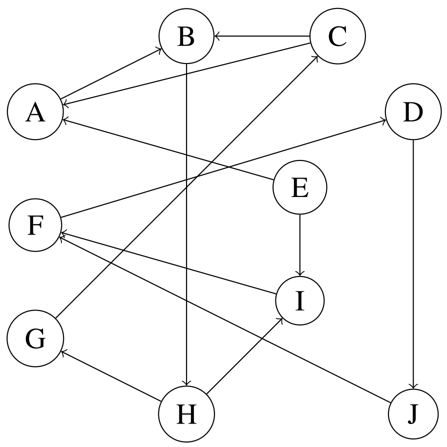
CMPSC 465 Data Structures & Algorithms
Spring 2024 Mehrdad Mahdavi and David Koslicki

Worksheet 6

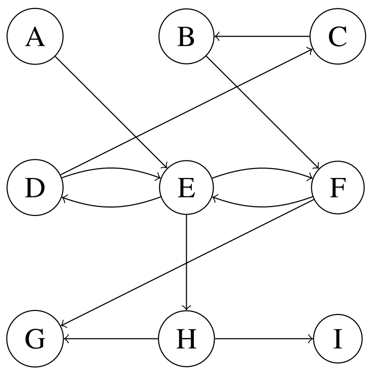
Wednesday, Feb 21, 2024

1. **Strongly Connected Components.** Run the strongly connected components algorithm on the following directed graphs and draw their SCC metagraphs. When doing DFS on the reverse graph G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

(i)

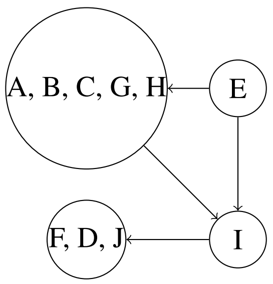


(ii)

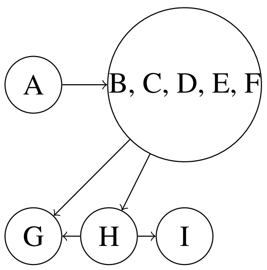


Solution

(i)



(ii)



2. **Award Ceremony.** 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 a directed edge (i, j) to G.

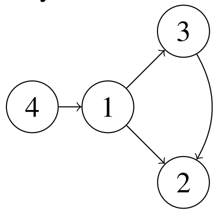
   Preparing the lineup would mean ordering the vertices. If there is an edge (i, j) in G, then i should appear before j in the order. Thus, we want G to be acyclic, i.e., a DAG. If G has a cycle, then no ordering is possible.

   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.

   Example

   For example, say, awardee 1 wants to receive before awardee 2 and 3, awardee 2 does not have a preference, awardee 3 before 2, and awardee 4 before 1. We can draw the corresponding graph this way :



   Now, we can run DFS, and find the topological sort order: $4, 1, 3, 2$. So, we line the awardee in order: $2, 3, 1, 4$ (awardee 4 is at the front of the line, then 1 and so on) and their conditions are met.

3. **Hamiltonian Path.** Given a directed acyclic graph G, write a linear-time algorithm to determine whether there exists a path that touches every vertex exactly once (a Hamiltonian Path).

   Solution
   Start by linearizing the DAG. Since the edges can only go in the increasing direction in the linearized order, and the required path must touch all the vertices, we simply check if the DAG has an edge $(i, i+1)$ for every pair of consecutive vertices labelled i and $i+1$ in the linearized order. Both, linearization and checking outgoing edges from every vertex, take linear time and hence the total running time is linear.

4. Odd Cycle. Give a linear-time algorithm to find an odd-length cycle in a *directed* graph. (*Hint:* First solve the problem under the assumption that the graph is strongly connected.)

Solution

Consider the case of a strongly connected graph first. The case of a general graph can be handled by breaking it into its strongly connected components since a cycle can only be present in a single SCC. We proceed by coloring alternate levels of the DFS tree nodes as red and blue. We claim that the graph has an odd cycle if and only if there is an edge between two vertices of the same color (which can be checked in linear time).

If there is an odd cycle, it cannot be two colored and hence there must be a monochromatic edge. For the other direction (monochromatic edge implying odd cycle), let u and v be two vertices having the same color and let (u,v) be an edge. Also, let w be their lowest common ancestor in the tree. Since u and v have the same color, the distances from w to u and v are either both odd or both even. This gives two paths p_1 and p_2 from w to v, one through u and one not passing through u, one of which is odd and the other is even.

Since the graph is strongly connected, there must also be a path q from v to w. Since the length of this path is either odd or even, q along with one of p_1 and p_2 will give an odd length tour (a cycle which might visit a vertex multiple times) passing through both v and w. Starting from v, we progressively break the tour into cycles whenever it intersects itself. Since the length of the tour is odd, one of these cycles must have odd length (as the sum of their lengths is the length of the tour).