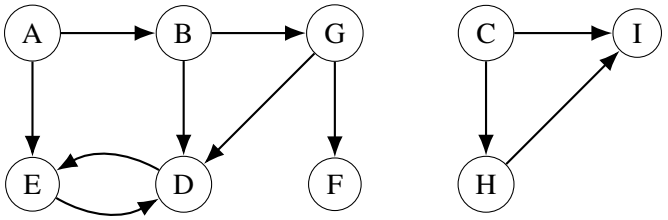
CMPSC 465 Data Structures & Algorithms
Fall 2025 Ke Chen and Yana Safonova

Worksheet 4

Monday, Sep 29, 2025

1. **Graph Basics.** Run DFS on the following graph, visit nodes alphabetically (e.g. given a choice between nodes D and F, visit D first).



   (a) List the nodes in the order you visit them (so each node should appear in the ordering exactly once).

   (b) List each node with its pre- and post-number. The numbering starts from 1 and ends at 18.

   (c) Label each edge as **T**ree, **B**ack, **F**orward or **C**ross.

   Solution

   (a) Ordering: ABDEGFCHI

   (b) The following table lists each node.

nodes	pre-visit	post-visit
A	1	12
B	2	11
D	3	6
E	4	5
G	7	10
F	8	9
C	13	18
H	14	17
I	15	16

   (c) Tree: AB, BD, DE, BG, GF, CH, HI Back: ED Forward: AE, CI Cross: GD

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.

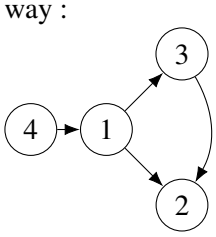
Preparing the lineup would mean ordering the vertices. If there is an edge (i, j) in G, then i should appear before j in the order. Thus, we want G to be acyclic, i.e., a DAG. If G has a cycle, then no ordering is possible.

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. **Bipartite Graph.** You are given an undirected graph $G = (V,E)$. Design an algorithm to determine if G is bipartite, i.e., its vertices can be colored with two colors such that every edge has endpoints of different colors. If it is bipartite, return such a coloring, otherwise return that it is not possible. Your algorithm should run in $O(|V|+|E|)$ time..

   Solution: This can be formulated as a graph problem. We want to check whether G contains an odd cycle. A graph is bipartite if and only if it has no odd cycle.

   We can use DFS with coloring. For every connected component, start DFS at an uncolored vertex u and assign it color "red". Whenever DFS explores an edge (u,v), assign v the opposite color of u if it is uncolored. If v is already colored and has the same color as u, then we have found an odd cycle and the graph is not bipartite. If the DFS completes without conflicts, then the graph is bipartite.

   DFS inspects each vertex and edge once, so the running time is $O(n+m)$.

   Example

   Consider $V = \{1,2,3,4\}, E = \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\}$. Start DFS at 1, color it red. Then 2 becomes blue, 3 red, 4 blue. All edges connect red to blue, so the graph is bipartite. If instead we had $V = \{1,2,3,4,5\}$ with edges forming a 5-cycle, DFS would eventually color both 1 and 5 red, producing a conflict on edge $\{1,5\}$, proving the graph is not bipartite.

4. **Pouring Water.** We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

1. Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.
2. What algorithm should be applied to solve the problem?
3. Find the answer by applying the algorithm.

Solution:

(a) Let $G = (V, E)$ be our (directed) graph. We will model the set of nodes as triples of numbers (a_0, a_1, a_2) where the following relationships hold: Let $S_0 = 10, S_1 = 7, S_2 = 4$ be the sizes of the corresponding containers. a_i will correspond to the actual contents of the i-th container. It must hold $0 \leq a_i \leq S_i$ for $i = 0, 1, 2$ and at any given node $a_0 + a_1 + a_2 = 11$ (the total amount of water we started from). An edge between two nodes (a_0, a_1, a_2) and (b_0, b_1, b_2) exists if both the following are satisfied:

- the two nodes differ in exactly two coordinates (and the third one is the same in both).
- if i, j are the coordinates they differ in, then either $a_i = 0$ or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

The question that needs to be answered is whether there exists a path between the nodes $(0, 7, 4)$ and $(*, 2, *)$ or $(*, *, 2)$ where $*$ stands for any (allowed) value of the corresponding coordinate.

(b) We can apply DFS on this graph, starting from node $(0, 7, 4)$ with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached and 'NO' if all the connected component of the starting node is exhausted and no desired vertex is reached.

(c) DFS can visit the graph this way: $(0, 7, 4) \to (7, 0, 4) \to (7, 4, 0) \to (10, 1, 0) \to (6, 1, 4) \to (6, 5, 0) \to (2, 5, 4) \to (2, 7, 2)$. So, we have reached our desired vertex, and the algorithm will answer "Yes".

5. DFS. You are given a binary tree $T = (V, E)$ (in adjacency list format), along with a designated root node $r \in V$. Recall that u is said to be an ancestor of v in the rooted tree, if the path from r to v in T passes through u. You wish to preprocess the tree so that queries of the form "is u an ancestor of v?" can be answered in constant time. The preprocessing itself should take linear time. How can this be done?

Solution: Do a DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. Thus, u is an ancestor of v if and only if $pre(u) < pre(v) < post(v) < post(u)$.

6. DFS. You are given a binary tree $T = (V, E)$ with a designated root node. In addition, there is an array $x[\cdot]$ with a value for each node in V. Define a new array $z[\cdot]$ as follows:
for each $u \in V$,
$z[u] =$ the maximum of the x-values associated with u's descendants.
Give a linear-time algorithm that calculates the entire z-array.
Solution:
We modify the explore procedure so that explore called on a node returns the maximum x value in the corresponding subtree. The parent stores this as its z value, and returns the maximum of this and its own x value.

```
function EXPLORE(G, u)
    visited(u) ← true
    z(u) ← −∞
    temp ← 0
    for each edge (u, v) ∈ E do
        if not visited(v) then
            temp ← explore(G, v)
            if temp > z(u) then
                z(u) ← temp
            end if
        end if
    end for
    postvisit(u)
    return max{z(u), x(u)}
end function
```