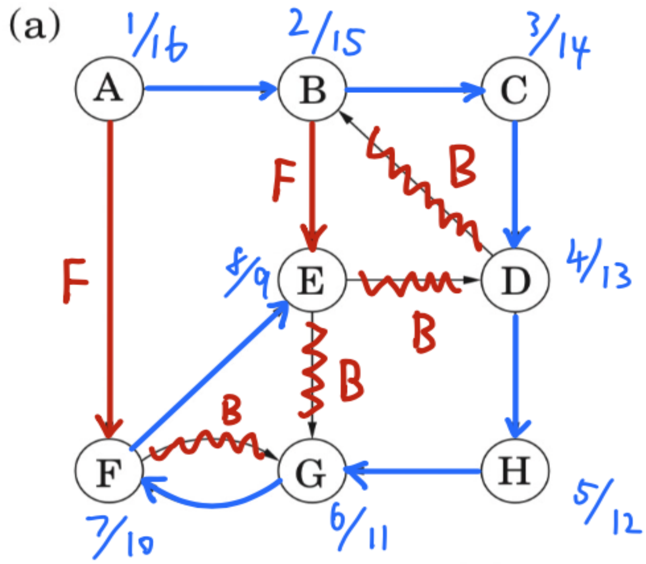
Student Name: Xinxi Zhang
NetID: XZ657
RUID: 219004759

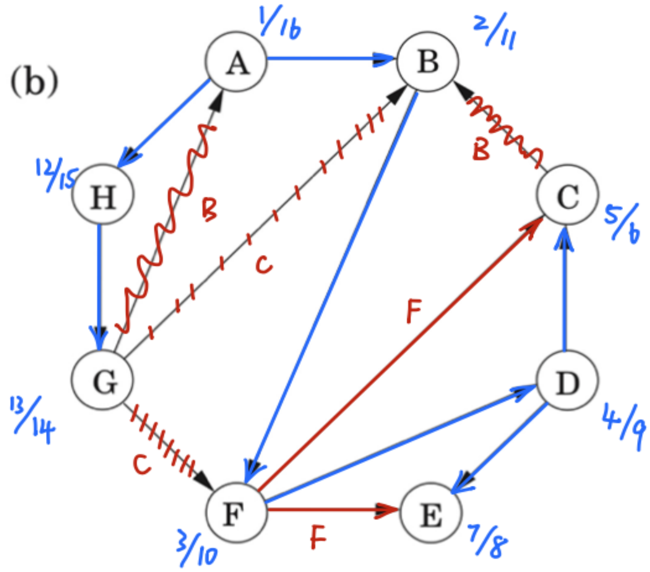
Introduction to Data Structures and Algorithms (CS 512)

Homework 2

---

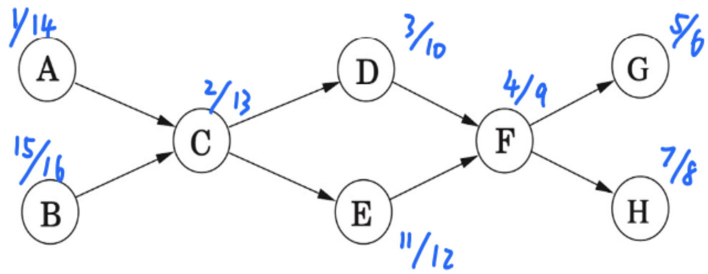
1. Perform depth-first search on each of the following graphs; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge, or cross edge, and give the pre and post number of each vertex.





2. Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.

(a) Indicate the **pre** and **post** numbers of the nodes.



(b) What are the sources and sinks of the graph?

Sources: A, B
Sinks: G, H

(c) What topological ordering is found by the algorithm?

$B \to A \to C \to E \to D \to F \to H \to G$

(d) How many topological orderings does this graph have?

The order between (A,B); (D,E); (G,H) can be switched
$\Rightarrow$ so the numbers of topological orderings $= 2^3 = 8$

3. What is the smallest possible depth of a leaf in a decision tree for comparison based sorting?

$$\begin{aligned} & \text{Let: the component of the list } = n \\ \Rightarrow \quad & \text{Number of leaves of a comparison based decision tree } = n! \\ \Rightarrow \quad & \text{The smallest possible depth of a leaf } = \log_2(n!) \end{aligned}$$

4. Rewrite the **Explore** algorithm so that it is non-recursive (that is, explicitly use a stack). The calls to **previsit** and **postvisit** should be positioned so that they have the same effect as in the recursive procedure.

Algorithm 1 Explore

```
S ← Stack()
Count ← 1
function EXPLORE(v)
    v.marked ← True
    v.pre ← Count
    S.push(v)
    while S is not empty do
        for u ∈ V do
            if (v, u) ∈ E, and u.marked = False then
                Count++
                u.marked ← True
                u.pre ← Count
                S.push(u)
                v ← u
                flag ← False
                break
            end if
        end for
        if flag = True then
            Count++
            v.post ← Count
            S.pop()
            v ← S.top()
        end if
    end while
end function
```

5. You are given a binary tree $T = (V, E)$ (in adjacency list format), along with a designated root node $r \in V$. A vertex u 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?

Firstly, we can Explore the tree from the root node r, and record each tree nodes' **previsit** and **postvisit** numbers. Then we can check "is u an ancestor of v?" by checking whether the orders of their **previsit** and **postvisit** numbers satisfy the following orders:

$$u.\mathbf{pre} < v.\mathbf{pre} < v.\mathbf{post} < u.\mathbf{post}$$

The preprocessing function **Explore(r)** takes linear time $O(|E|)$, and the checking function takes constant time by just comparing the **previsit** and **postvisit** numbers of u and v.

Algorithm 2 is u an ancestor of v?

```
Explore(r)
if u.pre < v.pre < v.post < u.post then
    return True
else
    return False
end if
```

6. Give an efficient algorithm that takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $x \in V$ from which all other vertices are reachable.

If there is a vertex $x \in V$ from which all other vertices are reachable, x must be in the **source of SCC**, otherwise x will not able to reach any vertex in the **source of SCC**. And it's easy to noted that if there is a x in the **source of SCC** satisfies the condition, then all the vertices in the **source of SCC** satisfy the condition.

So all we need to do is to find a x in the **source of SCC** and check whether it can reach all other vertices.

Algorithm 3 whether or not there is a vertex $x \in V$ from which all other vertices are reachable?

```
DFS(G)
x ← vertex with the biggest postvisit number
Explore(x)
for v ∈ V do
    if v.marked = False then
        return False
    end if
end for
return True
```

7. Give a linear-time algorithm that takes as input a **DAG** $G = (V, E)$ and determines whether or not G contains a directed path that touches every vertex exactly once.

Suppose G contains a directed path: $v_1, v_2, \ldots, v_n$ that touches every vertex exactly once, v_1 must be the source otherwise there will be a backedge to make the graph impossible to be a **DAG**. And if we delete v_1, v_2 will become the new source, which give us the information that $v_1, v_2, \ldots, v_n$ are in the order of descending order of their **postvisit** number. So what are we gonna do here is **DFS** the graph and order the vertices in descending order of **postvisit** numbers and check whether there is a path between each of them.

Algorithm 4 G contains a directed path that touches every vertex exactly once?

```
DFS(G)
V' ← Ordered V in descending order of postvisit numbers
for i from 2 to V.length do
    if (V'[i − 1], V'[i]) ∉ E then
        return False
    end if
end for
return True
```