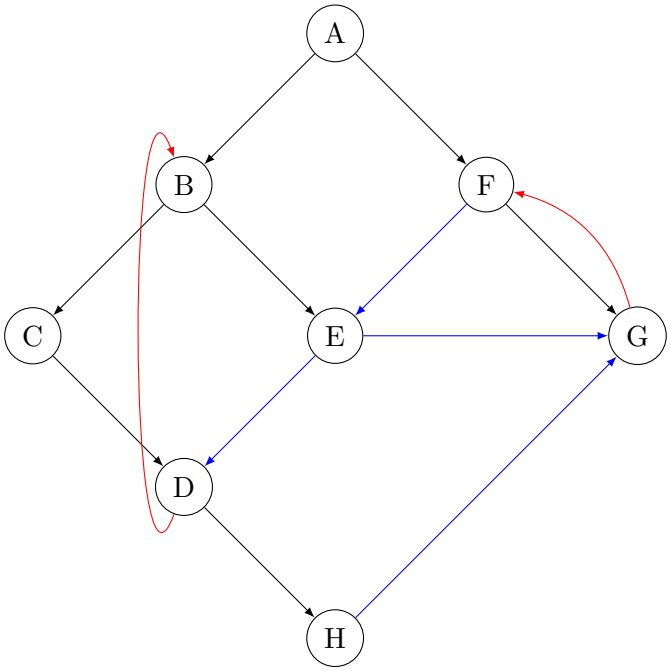
1.5em 0pt

# CS 5800 - Problem Set 3

**Name:** Ajeya Kempegowda

**1. (3 + 3 + 2 = 8 points) Graph Traversal**
Consider the following 2 directed graphs:

**(i)** Perform a breadth-first search on graph (a), pictured on the left. Whenever there's a choice of vertices to explore, pick the one that is alphabetically first. List the nodes in the order in which they are explored. For each node, indicate its distance from the root node, as well as its parent node. (You can specify the parent of the root node as NIL). Classify each edge as a tree edge, back edge, or cross edge.



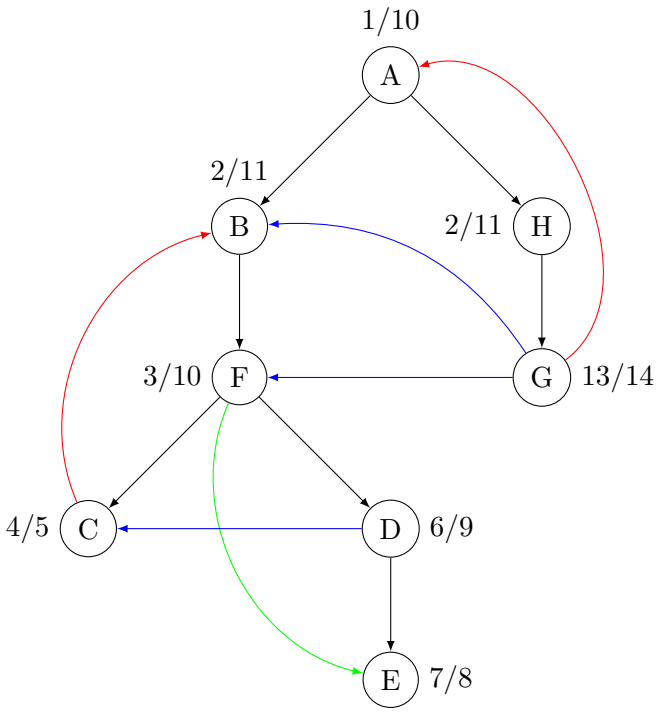

**Answer**:

Let vertex A be the sink. Exploring A, we've two children - F,B. Create an output list which stores the exploration traversal order. Currently, the output list, let's say ***output*** would hold **A, F, B**. Since, we're exploring alphabetically, we next choose B and explore the vertices of B, which gives us **C, E**. The output now holds, **A, B, F, C, E**. This is repeated by utilizing the Queue data structure and en-queuing the vertices starting from the source to the sink. Hence, following this procedure we finally end up with **output = A,B, F, C, E, G, D, H**

Here, Black edges represent tree edges, Blue edges represent cross edges and Red edges represent back edges.

| Vertices and their distances | | |
|---|---|---|
| Vertex | Dist.Root Node | Dist.Parent Node |
| A | Nil | Nil |
| B | 1 | 1 |
| C | 2 | 1 |
| D | 3 | 1 |
| E | 2 | 1 |
| F | 1 | 1 |
| G | 2 | 1 |
| H | 4 | 1 |

**(ii)** Perform a depth-first search on graph (b). Whenever there's a choice of vertices, pick the one that is alphabetically first. List the nodes in the order in which they are explored. For each node, indicate its discovery and finish times. Classify each edge as a tree edge, forward edge, back edge, or cross edge.




**Answer**:

Choosing as A as the Source, applying DFS in an alphabetical fashion, we start off with B vertex. Now, the **output** contains **A, B**. Exploring B, we get F and exploring F we get C. The output would now look like ***output***= A, B, F, C. The entire operation can be implemented using stack data structure. Since, C vertex doesn't have any more children, we return to F vertex and start exploring the children of F successively. Doing this recursively, we finally get the output sequence as **output=A, B, F, C, D, E, H, G**

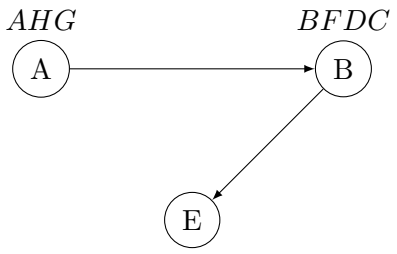
Here, Black edges represent tree edges, Blue edges represent cross edges and Red edges represent back edges and green edges represent forward edges.

The labels next to the vertices in the graph represent the Discovery and the finish time. The

Discovery time is assigned when the vertex is first explored by DFS algorithm and the finish time is assigned when the node has no where else to explore.

**(iii)** Identify the strongly-connected components of graph (b). Draw the component graph $G^{SCC}_{(b)}$ obtained by contracting all edges within each strongly connected component of $G_{(b)}$ so that only a single vertex remains in each component.

**Answer**: Vertices- B,C,D,F and A,H,G form strongly connected components. Hence the overall graph can be represented as follows.




**2. (2 + 4 + 2 = 8 points) 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.

**(a)** 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.

**Answer**:

To model this as a graph problem, we shall indicate each state of the three containers as an ordered pair (V,C), where V represents the movement of water between 7, 4 and 10 pints of water container C represents the amount of water in the containers. The original vertex, v0, would be (7,4,0), representing the initial condition for containers 7-pint, 4-pint and 10-oint containers, respectively. If any vertex exists such that v1 = (7,0,4) can be achieved through operation, it implies that there exists an edge between v0 and v1.

Finally we'd be interested to answer the question of the form: "Is there is a path between a defined vertex(sink or v0) and a vertex (2, b, c) or (a, 2, c)"

**(b)** Describe an algorithmic approach to solving the problem.

**Answer**: Since, we've a specific target to check, we can employ DFS traversal algorithm.

**(c)** Find the answer to the specific instance of the problem described above by applying the algorithm.

**Answer**:

$[(7,7),(4,4),(0,10)]$ →[(7,7), (0,4), (4,10)]→[(1,7), (0,4), (10,10)]→[(1,7), (4,4), (6,10)], [(5,7, (0,4), (6,10)]→ [(5,7), (4,4), (2,10)]→[(7,7), **(2,4)**, (2,10)]

**3. (7 points) Counting Paths**

Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices $s$ and $t$, and returns the number of paths from $s$ to $t$ in $G$. For example, in the directed acyclic graph given below, there are exactly four paths from vertex $p$ to vertex $v$ : $pov, poryv, posryv$, and $psryv$. (Your algorithm only needs to count the paths, not list them.)

**Answer**:

A linear time algorithm that takes up a DAG, G=(V,E) as input and two inputs s, t and return the various paths between them can be implemented using Topological sort and account for the overlapping/redundancy problem while counting the paths.

A Topological sort done between two vertices a and b would be $b[1], b[2], \cdots, b[n-1]$. Here $b[0]$ would be the first vertex a and $b[n]$ would be the target vertex b.

**Algorithm 1** FIND PATHS

```
1: procedure FIND PATHS DAG(G, s, t)
2:     direct paths = TOPOLOGICAL SORT(G)
3:     if b[1], b[2], ···, b[n − 1] are the vertices between s, t, then
4:     b[0] = s
5:     b[n] = t
6:     for k ∈ {0, . . . , n − 1} do
7:         direct paths[k] =0
8:     end for
9:     direct paths[n] =1
10:    return COUNT UNIQUE PATHS(G, direct paths, 0)
11: end procedure
```

**Algorithm 2** Count unique paths

```
1: procedure COUNT UNIQUE PATHS(G, directpaths, j)
2:     if j > k then
3:         return 0
4:     else if directpaths[j] ≠ 0 then
5:         return direct paths[j]
6:     else
7:         while 0 ≤ i ≤ n do
8:             for v[i] in Adjacency matrix of G do
9:                 direct paths[j] += COUNT UNIQUE PATHS(G, direct paths, i)
10:            end for
11:        end while
12:    end if
13:    return direct paths[j]
14: end procedure
```

**4. (7 points) Wrestling Rivalry**

There are two types of professional wrestlers: "good guys" and "bad guys." Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have $n$ professional wrestlers and we have a list of $r$ pairs of wrestlers between whom there are rivalries.

Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it.

**Answer**:

This is equivalent to finding if a graph is bipartite. We can say that a graph $G$ is bipartite if it can be partitioned into two sets $X, Y$ such that every edge has one end point in $X$ and the other end point in $Y$. Create a graph G where each vertex represents a wrestler. Let each edge represents a rivalry. The graph would represent n vertices and r edges. i.e $V = \{v_1, v_2, \dots v_n\} = n = \{n_1, n_2, \dots n\}$ and $E\{e_1, e_2, \dots e_r\} = \{r_1, r_2, \dots r\}$

Assign the first wrestler to be a good guy and then assign all its immediate neighbors to be bad guys, and so on. Further, trace multiple Breadth first searches to visit all vertices. Assign all wrestlers whose distance is even to be good guys and all wrestlers whose distance is odd to be bad guys. If a wrestler is assigned to be a good guy (or bad guy), but one of its neighbors has already been assigned to be a good guy (or bad guy), then this would be conflicting and would not be possible. Then check each edge to verify that it goes between a good guy and a bad guy. This solution would take O(n + r) time for the BFS, O(n) time to designate each wrestler as a good guy or bad, and O(r) time to check edges, which is O(n + r) time overall.