Problem Set 5

Assigned: June 24
Due: July 1

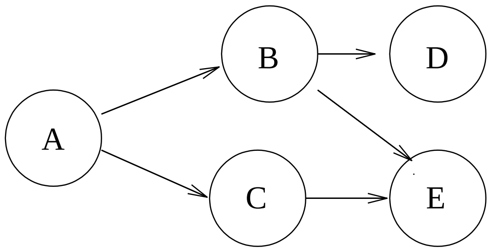
Problem 1.

Let G be a DAG. A vertex in G is a sink if it has no outarcs. A forward path from vertex U is a path that ends in a sink. Vertex V is a terminus of vertex U if V is a sink and there is a path from U to V.

A. Construct an algorithm NumForPaths(G) that computes the number of forward paths from every node in DAG G in linear time. If U is itself a sink, then NumForPath(G)[U] should be 1.

B. Construct an algorithm NumTerminus(G,U) that computes the number of terminuses for vertex U in DAG G in linear time.

For instance in the following graph G



NumForPaths(G)[A] = 3: A→B→D; A→B→E; and A→C→E
NumTerminus(G,A) = 2: D and E.

Ans:

A: Let T be a hash table of size equal to the number of nodes in the graph, with key as the node and value as the number of paths from that node. Initialize the values initialized to zero.

```
NumForPaths(G){
        For (each u(vertex) in G with no inarc){
                NumPaths(u);
        }
        Return T;
}
NumPaths(u){
        For (each u -> v) {
                NumPaths(v);
                T(u).value = T(u).value + T(v).value;
        }
        If(T(u).value == 0){
                T(u).value = 1;
        }
}
```

```
B: Let NumPath and Numt be two global variables initialized to zero.
NumTerminus(G, u){
      If ((NumPath == 0) && (u.NumOutArc == 0) {
      Numt = 0;
     }
     If ((NumPath == 0) && (u.NumOutArc > 0)) {
      NumPath = u.NumOutArc
     }
     If ((NumPath > 0) && (u.NumOutArc == 0)) {
      Numt = Numt + 1;
     }
     If ((NumPath > 0) && (u.NumOutArc > 0)) {
      NumPath = u.NumOutArc -1;
     }
     For (each u -> v) {
      NumTerminus(G,v);
     }
     Return Numt;
}
```

Problem 2

(Siegel). Write an algorithm that takes a DAG G as input and prints out all the possible topological sorts of G. For instance, given the graph in problem 1, the algorithm would output

```
A,B,C,D,E
A,B,C,E,D
A,B,D,C,E
A,C,B,D,E
A,C,B,E,D
```

It should print out each sort only once. Your algorithm does not have to produce the sorts in this order.

Ans:

The idea here is to identify the set of nodes at each level of the sorting and print the result by applying permutation to each set of nodes at their level and print all combinations obtained from the permutation..

```
TopoSort_All(G){
      Build the forest of tree from the post order DFS stack for the graph;.
      Perform the permutation of the nodes at each level and the cross edges and print;
}
```

Or,

Here we need to maintain a dequeue to track the nodes traversed and print each time when the graph is empty.

```
TopSort_Driver(G){
      Color all the vertices to white at G.
      Create a copy G' of G;
TopSort_All(G'){
        If (G' is empty) {
            For (k = 1 to size of queue) {
            Print (queue (k));
            }
            Return;
        }
        Repeat {
            Identify all the vertices which have no inarc and have color white or black;
                If (no inarc found) {
                    Pop_back_queue();
                    Return
                }
            Select vertex v out of them;
            Push_back_queue(v);
            Color the vertex (v) grey at G if it is white and all other vertices with no
            inarc as black at G.
            Delete the vertex from G' and all its outarcs.
            TopSort_All(G');
            For all vertex I > v set the color to white and the vertices with color black
to white at G.
        }
}
}
```

Problem 3

(Siegel). To see how essential marking is for graph traversal, consider the application of depth-first-traversal on a DAG G where node marking is not used. That is, the DFS code is rewritten in the form

```
Procedure DFS (v) {
   For (each outarc v --> w) DFS (w)
}
```

Consider the complete DAG on n vertices; that is, the vertices are numbers 1 . . . n and there is an arc from i to j for every pair $i < j$. What is the running time of this modified DFS on that graph?

Ans: T (1) = T (n) + T (n-1) + T (n-2) + T (n-3) + … + T (2)

Here as 'n' is the largest number the vertex n will not have any out arcs. So the running time of DFS (n) would be O (1).

Now, the vertex n-1 would have only one outarc to vertex n, DFS (n-1) = DFS (n) = O (1).

Similarly, the vertex n-2 would have two outarcs to vertex n-1 and n, DFS (n-2) = DFS (n-1) + DFS (n) = 1 + 1 = 2

Moving on, DFS (n-3) = DFS (n-2) + DFS (n-1) + DFS (n) = 2 + 1+ 1 = 4

DFS (n-4) = DFS (n-3) + DFS (n-2) + DFS (n-1) + DFS (n) = 4+ 2+1+1 = 8

.

.

.

DFS (1) = O ($2^{(n-2)}$).

Therefore, the running time of this method would be O ($2^{(n-2)}$).

Problem 4.

Let G be a DAG where the vertices are labelled with numerical values.

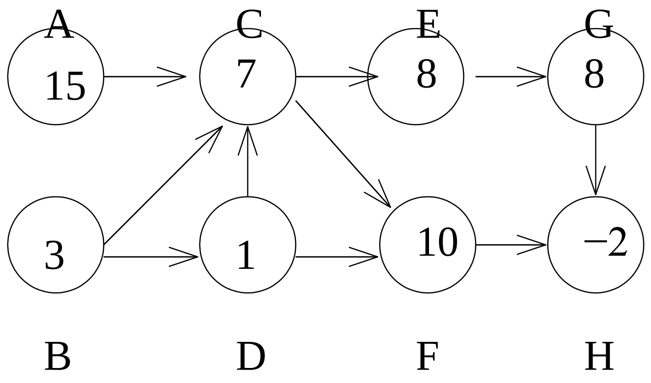
A. Write a function MaxReachable(u) which returns the maximum label on a vertex reachable from vertex u (including u itself.)

B. Write a function TotalReachable(u) which returns the sum of the labels on vertices reachable from vertex u.

C. Write a function MaxPathFrom(u) which returns the maximum sum of the labels on any path starting at u.

All these should run in time linear in the size of G.

For example, in the graph below:



MaxReachable(B) = 10 (corresponding to F).
TotalReachable(B) = 35 (B+C+D+E+F+G+H)
MaxPathFrom(B) = 27 (corresponding to B-D-C-E-G).

Ans:

A: Let Maxlabel be a global variable initialized to zero.

```
MaxReachable(u){
    If (Maxlabel == 0)
        Maxlabel = u.label;
    Endif.
    For (each arc u -> v){
        If (v.label > Maxlabel)
            Maxlabel = v.label
        Endif
        MaxReachable(v);
```

```
            }
            Return Maxlabel;
        }
```

B: Let Total be a global variable initialized to zero.
And each node has an attribute named color with values either white, grey or black. All nodes initialized to white.

```
TotalReachable(u){
        If(u.color == white){
            Total = Total + u.label;
            u.color = grey;
        endif.
        For(each u->v){
            TotalReachable(v);
        }
        Return Total;
}
```

C: S: set of vertices for which we know the MaxPath
D[w]: cost of LargestPath to W found so far.

```
MaxPathFrom(u){
        S = empty;
        For (V in vertices) D [V] = -∞;
        D [u] = u.label;
        For (I = 1 to |vertices| -1) {
            W <- the vertex not in S, with largest value of D;
            If((I > 1) && (W has no inarc)){
                Exit;
            }
            Add W to S;
            For (each arc W->V) {
                If (D [W] + V.label > D [V])
                    Then set D [V] <- D [W] + V.label

                    P[V] <- W
            }
        }
        LargestPath = D[1];

        For (i = 1 to |D|) {
            If (D[i] > LargestPath){
                LargestPath = D[i];
            }
        }
        Return LargestPath;
}
```