Homework 2

Kevin Martin
CIS675 - Syracuse University

January 28, 2020

1. Question 1

   (a) Pseudocode for an efficient algorithm to find and return majority element:

   ```
   def majorityFind(c [], size)
   candidate = 0, count = 1
   for i = 1 to (size - 1) do
       if c[candidate] == c[i] then
           count++
       else
           count- -
       if count == 0 then
           candidate = i
           count = 1
   return c[candidate]
   ```

   (b) Claim: the running time of the proposed algorithm is $O(n)$

   Proof. Because the above algorithm only needs one pass (and not recursion), we must look to the length of the input n. The for loop requires a look at all elements in the list, which is $O(n)$ running time. Setting up the variables is constant time, and the equality check (A[i]==A[j]) has already be stated to run in constant time $O(1)$. Thus the running time of the algorithm depends entirely on the for loop, and the total running time is $O(n)$, where $n > 1$.

   ■

2. Question 2

   (a) Claim: a directed acyclic graph (DAG) always contains a vertex with indegree 0.

   Proof. For a graph to be a DAG, it must have at least one source node. If we assume that every node in the graph has an indegree of at least one, then let us start at a given node v_1. We will follow the path to its parent. The parent node will also have an indegree of at least one. We continue this process, until each node has been visited, and we will arrive back at v_1, which would mean the graph is a cycle. This is a contradiction.

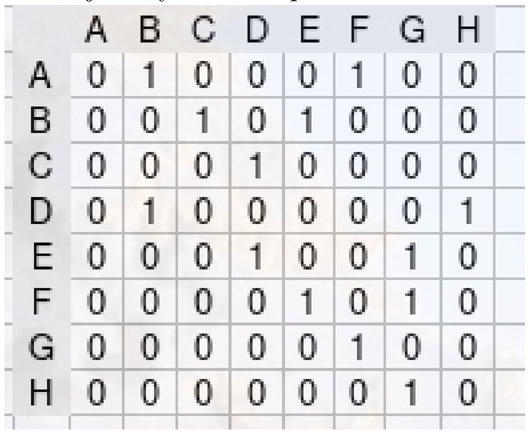
■

(b) Claim: any connected undirected graph has $|E| \geq (|V| - 1)$

Proof. For a graph to be connected, each vertex must have least one edge. However two vertices can share an edge. So between two vertices, there could be one edge. If every vertex only had one edge, the graph would not be connected. So there must be at least one pair of vertices that are connected by a common edge $(|V|-1)$. Therefore, there must be at least as many edges as there are vertices, less the (minimum) one shared edge. ■

3. Question 3

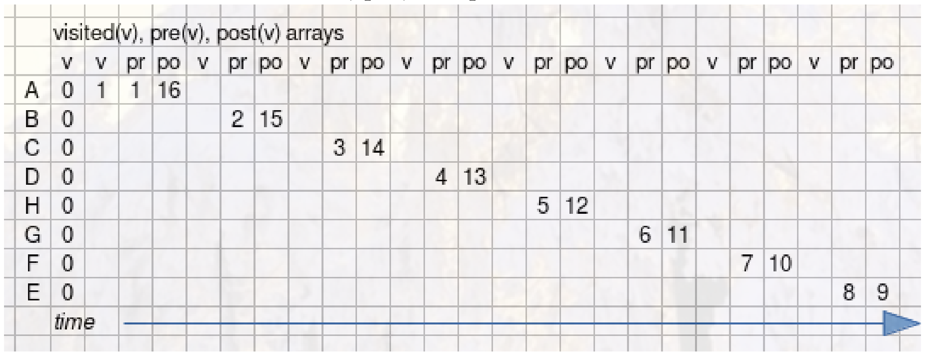
(a) The adjacency matrix representation:



	A	B	C	D	E	F	G	H
A	0	1	0	0	0	1	0	0
B	0	0	1	0	1	0	0	0
C	0	0	0	1	0	0	0	0
D	0	1	0	0	0	0	0	1
E	0	0	0	1	0	0	1	0
F	0	0	0	0	1	0	1	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	0	0	1	0

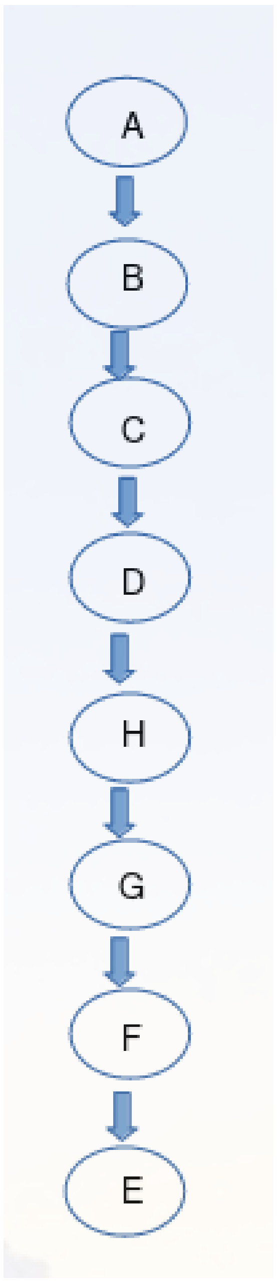
(b) The adjacency list representation:
$A : B \rightarrow F$
$B : C \rightarrow E$
$C : D$
$D : B \rightarrow H$
$E : D \rightarrow G$
$F : E \rightarrow G$
$G : F$
$H : G$

(c) Table for intermediate visited, pre, and post values of all nodes:



	visited(v), pre(v), post(v) arrays																								
	v	v	pr	po	v	pr	po	v	pr	po	v	pr	po	v	pr	po	v	pr	po	v	pr	po	v	pr	po
A	0	1	1	16																					
B	0					2	15																		
C	0								3	14															
D	0											4	13												
H	0														5	12									
G	0																	6	11						
F	0																				7	10			
E	0																							8	9
	time →																								

(d) Final DFS tree:



4. Question 4

   Pseudocode of an efficient algorithm to take in graph G(V, E), represetned by an adjacency list, and two vertices $x, y \in V$, where the output is the different paths from x to y in G:

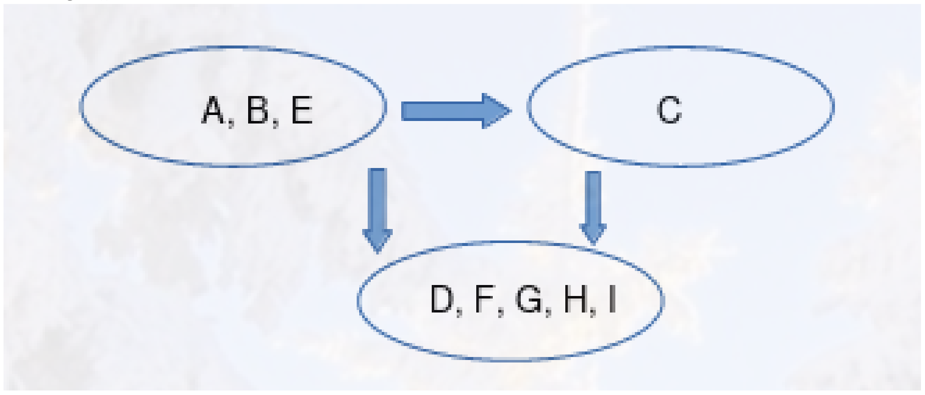
```
//wrapper function to set empty path
def diffPathsWrap(x, y, G)
for i = 1 to (size-1) do
    visited[i] = False
    path = [] //empty array for each path
diffPaths(x, y, G, path)



def diffPaths(x,y, visited, path)
visited[x]=True
if x == y
    return path
else
    for j in G[x] do
        if visited[j] == False
            return diffPaths(x, y, visited, path)
        path[j].remove //remove current vertex from path
        visited[j] = False
```

5. Question 5

   (a) The order of the strongly connected componets is:
   $A \to B \to E$
   C
   $D \to H \to F \to I \to H \to D$

   (b) The source SCC is $A \to B \to E$ and the sink SCC is $D \to H \to F \to I \to H \to D$

(c) Metagraph:



(d) The minimum number of edges one must add to make the graph strongly connected is just one: from $D \to G$. If that edge were added, then we could get to any vertex from any other.