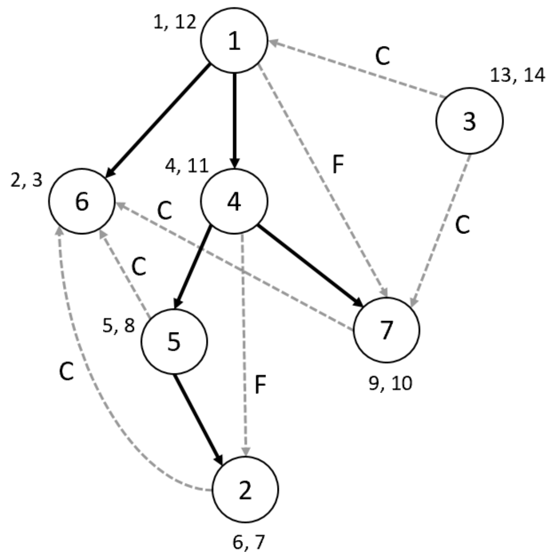
Computer Science CSC263H
St. George Campus

March 28, 2019
University of Toronto

Solutions for Homework Assignment #6

Answer to Question 1.

a.



b. The above DFS has 0 back edges, 2 forward edges, and 5 cross edges.

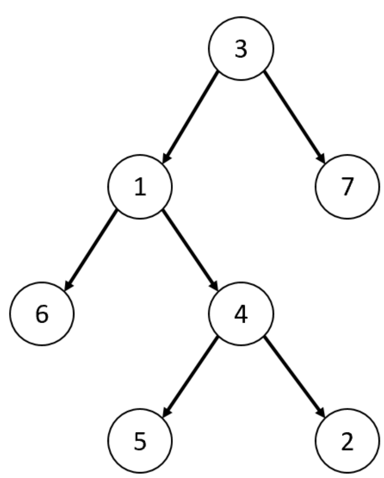
c. By part (b), some DFS of G has no back-edges. In class we proved that:

Theorem: For every directed graph G and every DFS of G, G has a cycle iff the DFS of G has a back edge.

Thus, G has no cycles. Therefore there is a topological sort of G, i.e., the courses can be taken in an order that satisfies all the prerequisites.

d. The topological sort algorithm outputs all the nodes of G in order of decreasing $f[]$ "finish" times. This gives the following list: 3, 1, 4, 7, 5, 2, 6.

e. Draw a Breadth-First Search tree of G that ***starts at node* 3** and explores the edges in the order of appearance in the above adjacency lists.



Answer to Question 2. Let L be a list of constraints.The high-level idea of the algorithm is similar to the one for Question 2 of Assignment 4:

1. Use the *equality* constraints in L to build the sets of variables that are *equal to each other*.
2. For each set, assign the same integer to all the variables in this set; different sets must get different integers.
3. For each *inequality* constraint $x_i \neq x_j$ in L, use the integers assigned to x_i and x_j to determine whether x_i and x_j are in the *same* set; if they are, then output NIL and stop.
4. Output the integer assignment computed in Step 2.

To build the sets of variables that are equal to each other, we use a undirected graph $G = (V, E)$, where the set of vertices $V = \{1, 2, \ldots, n\}$ represents the set of n variables $\{x_1, x_2, \ldots, x_n\}$, i.e., vertex i represents variable x_i; and the set of edges is $E = \{(i, j) \mid x_i = x_j \text{ is in } L\}$, i.e., there is an edge between vertex i and vertex j if and only if the list of constraints L contains $x_i = x_j$. Note that $|V| = n$ and $|E| \leq m$ (because L has m constraints).

Observe that, by transitivity, there is a path between two vertices u and v in G if and only if the variables x_u and x_v are equal to each other according to L. So the sets of variables that are *equal to each other* according to L are the *connected components* of G.

To do Step 1 and 2 above, we first use the list of constraints L to build the ***adjacency list*** of graph G. We then perform a slightly modified DFS of G to (a) find all the connected components of G, and (b) assign the same integer to all the vertices in each connected component that the DFS finds (with different components getting different integers). The pseudo-code is given below, where $i.num$ denotes the integer that the algorithm assigns to vertex i.

```
SatisfyingAssignment(n, L)
1   G ← BuildGraph(n, L)
2   for each vertex u ∈ G.V
3       u.color = WHITE
4   k = 0
5   for each vertex u ∈ G.V
6       if u.color = WHITE
7           k = k + 1
8           DFS-Visit(G, u, k)
9   for each inequality constraint x_i ≠ x_j in L
10      if i.num = j.num
11          return NIL
12  A[1..n] = []          // A is the output array. A[i] will store the integer assigned to x_i.
13  for each vertex u ∈ G.V
14      A[u] = u.num
15  return A
```

```
BuildGraph(n, L)
1   G.V ← {1, 2, ..., n}
2   for each vertex i ∈ G.V
3       G.Adj[i] ← EmptyList
4   for each equality constraint x_i = x_j in L
5       insert i into the list G.Adj[j]
6       insert j into the list G.Adj[i]
7   return G
```

```
DFS-Visit(G, u, k)
1   u.color = GREY
2   u.num = k
3   for each vertex v ∈ G.Adj[u]
4       if v.color = WHITE
5           DFS-Visit(G, v, k)
6   u.color = BLACK
```

The BuildGraph(n, L) of line 1 builds the adjacency list of graph $G = (V, E)$, and this takes $O(n + m)$ time in the worst-case. Lines 2-8 of SatisfyingAssignment(n, L) and the procedure DFS-Visit(G, u, k), is essentially a DFS of the graph G. *Since this DFS uses the adjacency list of* G, the worst-case time complexity of this part of the algorithm is just $O(|V|+|E|)$, i.e., $O(n+m)$. The loop of lines 9-11 goes over the m constraints of L, and it is clear that it takes $O(m)$ time in the worst-case. The loop of lines 12-14, goes over the n nodes of G, and it is clear that it takes $O(n)$ time in the worst-case. Thus, overall the algorithm's worst-case time complexity is $O(n+m)$.

Answer to Question 3. Suppose, for contradiction, that some MST T of G contains the edge e_{max}.

First remove e_{max} from T. This splits T into a spanning forest of G consisting of 2 trees, T_1 and T_2, disconnected from each other. So the edge e_{max} connects T_1 and T_2 into spanning tree T.

By hypothesis, G has a cycle C that contains the edge e_{max}. Since e_{max} connects T_1 and T_2, the cycle C that contains e_{max} must have *another* edge e that also connects T_1 and T_2.

Now add edge e. This reconnects the spanning forest T_1 and T_2 into a single spanning tree T' of G. Note that the weight of T' is $w(T') = w(T) - w(e_{max}) + w(e)$. Since $w(e_{max}) > w(e)$, we have $w(T') < w(T)$. So T is not a *minimum* spanning tree of G — a contradiction.