Suggested Solutions to HW #8

1. (7.1) Consider the problem of finding balance factors in binary trees discussed in class (see slides for "Design by Induction"). Solve this problem using DFS. You need only to define preWORK and postWORK.

Solution. (Yu-Chieh Tu)

preWORK:
$v.height := 0;$
$v.balance\_factor := 0;$
postWORK:
$v.height := MAX(v.height, w.height + 1);$
if $w = v.leftchild$ **then**
$v.balance\_factor := v.balance\_factor + (w.height + 1);$
else if $w = v.rightchild$ **then**
$v.balance\_factor := v.balance\_factor - (w.height + 1);$

Note that *preWORK* is performed at the time the vertex is marked, and *postWORK* is performed after we backtrack from an edge or find that the edge leads to a marked vertex. □

2. (7.3) Given as input a connected undirected graph G, a spanning tree T of G, and a vertex v, design an algorithm to determine whether T is a valid DFS tree of G rooted at v. In other words, determine whether T can be the output of DFS under some order of the edges starting with v. The running time of the algorithm should be $O(|E| + |V|)$.

Solution. (created by Chi-Jian Luo, modified by Yu-Chieh Tu)
To determine whether T, a subgraph of G, is a DFS tree of G, we have to consider two constraints: (1) T is a spanning tree of G and (2) there is no edge in G that is a cross edge for T. For the first constraint, we have to check if there is no cycle in T and all vertices in G can be reached by T. For the second constraint, we have to check if there is no edge in G that connects two subtrees in T.

Algorithm isDFSTree(G, T, v)
Input: $G = (V, E)$ (an undirected connected graph in the adjacency-list representation), $T = (V, E' \subseteq E)$ (a spanning tree of G in the adjacency-list representation), v (a vertex of G).
Output: result (a boolean variable, the default is **true**).

begin

mark v;
for *all edges* $(v, w) \in E'$ **do**
  if *w is marked and w.parent* $\neq v$ **then**
    result := **false**; {*T contains a cycle and is not a tree.*}
  else if *w is unmarked* **then**
    w.parent := *v*;
    isDFSTree(G, T, w);
for *all edges* $(v, w) \in E$ **do**
  if *w is unmarked* **then**
    result := **false**; {(v, w) *is a cross edge or w can't be reached from T.*}

end □

5. (7.38) Given a directed acyclic graph $G = (V, E)$, find a simple (directed) path in G that has the maximum number of edges among all simple paths in G. The algorithm should run in linear time.

Solution. (Jinn-Shu Chang)
We design an algorithm that is similar to the topological sorting algorithm in Fig 7.13 in Manber's book. We use *length* to denote the length of the path from a source vertex to the vertex which we are looking at. Initially, we set the *length* in all vertices to be *zero*. When we remove a vertex v from the Queue, we update the length of every successor of v, say w, if $v.length + 1$ is larger than $w.length$, to record the length of the longest path currently. When all the vertices are traversed, we can recover the longest path by backtracing from the latest vertex traversed.

Algorithm FindLongestPath(G)
Input: $G = (V, E)$ (a directed acyclic graph)
Output: *A Stack which records the longest path.*

begin

Initialize v.Indegree for every vertex v; {*e.g. by DFS*}
for *all vertices* $v \in V$ **do**
  if *v.Indegree* $= 0$ **then** *put v in Queue;* {*which are the source vertices*}
repeat
  remove vertex v from Queue;
  for *all edges* (v, w) **do**
    if *v.length* $+ 1 >$ *w.length* **then**
      w.length := *v.length* $+ 1$;
      w.pre := *v*;
    w.Indegree := *w.Indegree* $- 1$;
    if *w.Indegree* $= 0$ **then** *put w in Queue;*

if *Queue is empty* **then** $v_{path} := v$;

until *Queue is empty*

put v_{path} *on Stack;*

while v_{path} *has a predecessor* **do**

put $v_{path}.pre$ *on Stack;*

$v_{path} := v_{path}.pre$;

end

Since the main part of this algorithm is essentially the topological sorting algorithm, the time bound for traversing the vertices is $O(|V|+|E|)$. The last while loop for recording the longest path runs in $O(|V|)$ time. Hence, this algorithm runs in $O(|V| + |E|) + O(|V|) = O(|V| + |E|)$ time. $\square$