ESO207: Data Structures and Algorithms	*Time : 90min.*
Quiz: 2	*Max. Marks: 100*

Instructions.

a. Please answer all parts of a question together. You may leave blank spaces if you wish to return later to complete parts of the question.

b. The exam is closed-book and closed-notes. Collaboration of any kind is ***not*** permitted. You may not have your cell phone on your person. Please use the mounted clock in the rear of the examination hall for time reference.

c. You may cite and use results done in the class.

d. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity of your arguments. Always argue correctness of your algorithms or conclusions.

Problem 1. DFS Basics.

1. Let $\{u, v\}$ be an edge in an undirected graph G and during DFS of G, $u.f < v.f$. Then, v is an ancestor of u in the DFS tree. Prove or give a counterexample. (12)

   Soln. . *Proof:* During the DFS search, either u is discovered first or v is discovered first. If u is discovered first, then, v is explored fully and v finishes before u finishes, that is, $u.f > v.f$. So this is not the case given. Thus, in the given case, v is discovered first, and so u is a descendant of v and finishes before v finishes.

2. Prove that in any connected undirected graph $G = (V, E)$, there is a vertex $v \in V$ whose removal leaves the graph connected. (*Hint*: Consider the DFS search tree for G). (12)

   Soln. . Consider a DFS tree of G. There are two cases: either there is a back-edge found during DFS or there is no back-edge found. If there is no back edge found, then G is a tree, and if the DFS tree for G is rooted at vertex s, then, any leaf vertex may be deleted without disconnecting G. Conversely, if there is a back edge, then there is a cycle say $v_1, v_2, \ldots, v_k, v_1$. Now, removing v_k leaves G connected. *In either case, removing one leaf vertex of the DFS tree leaves the graph connected.*

3. In the city of *Uniwaytown*, all streets are one-way, that is, a street from one intersection u to another intersection is one-way. (There may or may not be a reverse one-way street from that intersection to the previous one.) The mayor claims that there is a way to drive *legally* from any intersection to any other intersection. Model the problem as a graph problem and give a linear time solution to check the mayor's claim. Argue your modeling. (*Note*: You may use results of the class without re-proving them). (18)

   Soln. . Model the problem as a graph as follows. Each intersection is a vertex and let V be the set of intersections. If there is a one-way street from one intersection u to another intersection v, then, (u, v) is modeled as an edge in E. Now we obtain a directed graph $G = (V, E)$. The mayor's claim is that there exists a directed path from any intersection to any other intersection. This equivalently means that the entire graph is one strongly connected component. To test this, find the strongly connected components of G and check whether there is only one component.

Problem 2. DFS for Least Common Ancestor. In this problem, we consider a rooted tree $T = (r, V, E)$, where, V is the vertex set, $r \in V$ is the root of the tree, and E is a set of undirected edges. (T need not be a binary tree) It is guaranteed that E is an acyclic set of edges and T is connected. Given two nodes u and v, the least common ancestor $lca(u, v)$ is the node w that is an ancestor of both u and v and has the greatest depth in T (or, is the furthest away from the root r among all the common ancestors of u and v). Note that if u is an ancestor of v, then, $lca(u, v) = u$, since u is considered as an ancestor of itself. Figure 1 shows an example rooted tree.

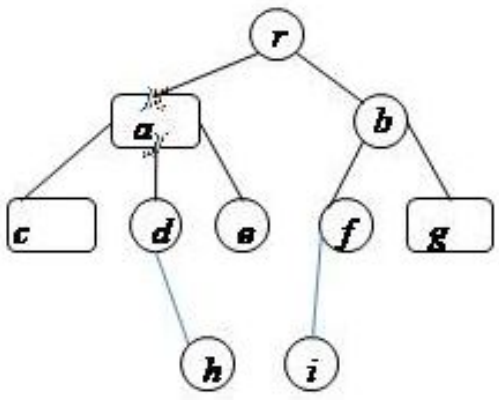

Figure 1: In this tree $lca(e, h) = a$, $lca((e, f) = r$, etc.

(*Note:* There are no *NIL* vertices, etc..

(a) What is $lca(c, h)$? $lca(c, b)$? $lca(b, i)$? (1+1+1=3)

Soln. . $lca(c, h) = a$, $lca(c, b) = r$ and $lca(b, i) = b$.

Suppose a DFS has been performed on T starting from r and associated with each node $v \in V$, the discovery and finish times are stored. Now, *given* two nodes u and v, design an $O(|V|)$ time algorithm to compute $lca(u, v)$.

1. *Case 1.* $lca(u, v) \in \{u, v\}$. Give a test for this condition and decide when $lca(u, v) = u$ and when $lca(u, v) = v$. (8)

   Soln. . $lca(u, v) \in \{u, v\}$ means that one of the vertices is an ancestor of the other. By DFS property if the interval $[v.d, v.f]$ is completely contained in $[u.d, u.f]$, then, $lca(u, v) = u$, that is, v is a descendant of u. Conversely, $lca(u, v) = v$ iff $[v.d, v.f]$ completely contains the interval $[u.d, u.f]$.

2. *Case 3.* $lca(u, v) \notin \{u, v\}$. Give an $O(|V|)$ time algorithm to compute $lca(u, v)$. (12)

   Soln. . In this case, neither u is a descendant of v nor v is a descendant of u. Start at a vertex, say u, and proceed upwards in the tree until the first vertex w is found such that $[w.d, w.f]$ completely contains both $[u.d, u.f]$ and $[v.d, v.f]$. Return this vertex w as the $lca(u, v)$.

Problem 3. A Die Hard puzzle. We are given three containers whose sizes are 10 liters, 7 litres, and 4 liters, respectively. Initially, the 7 litre and 4 litre containers are filled completely with water and the 10-liter container is empty. We are allowed one type of pouring operation: pouring the contents of one container into another, stopping only when either the source container is empty or the destination container is full. We want to know if there is a sequence of pouring operations that leaves exactly 2 litres in the 7-or 4-litre container.

(a) Model this as a graph problem: Give a precise definition of the graph, its vertices and edges and state the specific question about this graph that needs to be answered. (20)

(b) What algorithm should be used to solve this problem. Argue your answer. (15)

Soln. . For modeling purposes, let us generalize the problem to having three containers with sizes A, B and C liters respectively. A state of the containers is defined as a triple (a, b, c), where, the first container contains a liters of water out of a capacity of A liters, the second container contains b liters of water out of a capacity of B liters and the third container contains c liters out of C liters. For the given problem, the initial state is given as $(0, 7, 4)$. Since water is neither lost nor gained in the pourings, the possible states are

$$V = \{(a, b, c) \mid a + b + c = 7 + 4 = 11\} \ .$$

Each of the valid states may be thought of as a vertex and V is the set of vertices (states).

What is an edge? There is an edge from the state (a, b, c) to the state (a', b', c') provided, in **one pouring**, we can transform from the state (a, b, c) to (a', b', c'). Note that a pouring only involves two containers, so by definition, the amount of water in the other (third) container remains invariant. There are 3×2 cases in general, the first container from which the water is poured can be chosen in 3 ways and the second container to which water is poured can now be chosen in 2 ways, for a total of $3 \times 2 = 6$ ways.

1. *Pouring container 1 to container 2.*
   if $a \leq B - b$, add edge $((a, b, c), (0, a + b, c)$ else add edge $((a, b, c), (a + b - B, B, c))$.
2. *Pouring container 1 to container 3.*
   if $a \leq C - c$, add edge $((a, b, c), (0, b, a + c))$ else add edge $((a, b, c), (a + c - C, b, c))$.
3. *Pouring container 2 to container 3.*
   if $b \leq C - c$, add edge $((a, b, c), (a, 0, b + c)$ else add edge $((a, b, c), (a, b + c - C, C))$.
4. *Pouring container 2 to container 1.*
   if $b \leq A - a$, add edge $((a, b, c), (a + b, 0, c)$ else add edge $((a, b, c), (A, a + b - A, c)$.
5. *Pouring container 3 to container 1.*
   if $c \leq A - a$, add edge $((a, b, c), (a + c, b, 0))$ else add edge $((a, b, c), (A, b, a + c - A)$.
6. *Pouring container 3 to container 2.*
   if $c \leq B - b$, add edge $((a, b, c), (a, b + c, 0))$ else add edge $((a, b, c), (a, B, b + c - B)$.

This can be more succintly expressed as follows. Let A_1, A_2, A_3 be the capacities of the three containers respectively. If the state is (a_1, a_2, a_3), then, for every $i \in \{1, 2, 3\}$ and every $j \in \{1, 2, 3\} \setminus \{i\}$,
add edge from $(a_1, \ldots, a_3)$ to $(b_1, \ldots, b_3)$, where,

- $b_k = a_k$, for $k \in \{1, 2, 3\} \setminus \{i, j\}$.
- If $a_i \leq A_j - a_j$, then, $b_i = 0$ and $b_j := a_j + b_j$.
- else if $b_j = B$ and $b_i = a_i + a_j - B$.

The specific question that we want answered is: from the initial state $(0, 7, 4)$, is a state $(a, 2, c)$ is reachable, where, a, c are arbitrary, or is a state $(a, b, 2)$ is reachable, for a, b arbitrary ?

Note that the graph is conceptual, it is not created on the computer explicitly offline, it is discovered **incrementally**, see below.

Outline of the algorithm:

1. Run BFS on this *implicit* graph from the starting vertex $(0, 7, 4)$.
2. Every time a vertex or state (a_1, a_2, a_3) is first visited, we check whether our termination criterion is satisfied, that is, whether $a_2 = 2$ or $a_3 = 2$. If so, we terminate BFS. The *pred* attributes gives a backwards path which can be reversed to obtain the sequence of pourings.
3. When a state (a_1, a_2, a_3) is dequeued, its neighbors are *generated*, and checked if they were visited earlier, and if not, they are enqueued.
4. If BFS terminates and the termination criterion was never satisfied, this means that the destination state(s) are not reachable from the initial state.