Homework #6
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2014 **Solutions**

March 12, 2014

Problem 1 (20 points)

Given a connected undirected graph $G = (V, E)$, call a vertex $v \in V$ *vulnerable* if removing v (and all edges that touch vertex v) from graph G would result in G being disconnected.

(i) **15 points** Suppose that we run DFS on graph G starting at node $v \in V$. The resulting DFS tree (it's a tree, not a forest, because G is connected) is rooted at v. Prove that v is vulnerable if and only if v has more than one child in the DFS tree.

(ii) **5 points** Explain how you would use this fact to determine in $O(|V| + |E|)$ time whether or not vertex v is vulnerable in G (Of course, we could also just remove v from G and run DFS on the remaining graph to check for connectivity, but that isn't the point of this problem).

Solution

(i) Suppose that we start DFS search at $v \in V$, where v is a a vulnerable node. By definition, removing v from graph G would create at least two disconnected components. Call these components $C_1, \ldots, C_k$, where $k \geq 2$. We have $\{v\} \cup C_1 \cup \cdots \cup C_k = V$, and $C_i \cap C_j = \emptyset$ for all $i \neq j$. Let $s \in C_i$ and $t \in C_j$ for some i, j with $i \neq j$. By construction of the sets $C_1, \ldots, C_k$, any paths in G connecting s to t must pass through v. Now, we wish to show that the DFS tree that results from running DFS starting at v results in a DFS tree

in which v has multiple children. This fact follows directly from the parenthesis theorem and the fact that any path from a node in C_i to a node in C_j must pass through v. To see this, suppose that v has only one child in the DFS tree. Call this child vertex u. u must belong to some component C_i, defined above. Now, consider some other vertex s, from some other component C_j, with $j \neq i$. By the parenthesis theorem, we must have $v.d < s.d < s.f < v.f$, since we visit v first and s is reachable from v by the connectivity of G. But the fact that G is connected and the assumption that u is the only child of v jointly imply that we must also have that s is a descendant of u and thus it must be the case that $v.d < u.d < s.d < s.f < u.f < v.f$. But this means that s was visited from u *before* DFS search finished processing u, which implies by the white path theorem that there was a path from u to s not passing through v, contradicting our assumption that $s \in C_j$ and $u \in C_i$. Thus, we have shown that if v is the root of the DFS tree, it must have exactly one child.

Conversely, suppose that v has more than one child in the DFS tree. We must show that removing v from graph G would cause G to become disconnected. Toward this end, consider two of the children of v in the DFS tree, say, s and t. We must have $[s.d, s.f]$ and $t.d, t.f]$ disjoint by the fact that both vertices are children of v (if s and t had non-disjoint intervals then one would be a child of the other). Suppose without loss of generality that $s.d < t.d$. Then it must be that $v.d < s.d < s.f < t.d < t.f < v.f$ by the parenthesis theorem. Consider some descendant of s in the DFS tree, call it s', and some descendant of t in the DFS tree, call it t'. Suppose that there is a path connecting s' and t' in G that does not pass through v. Then by the white path theorem we must have $s'.d < t'.d < t'.f < s'.f$. But by the assumption that s' is a descendant of s in the DFS tree and t' is a descendant of t, we must have $v.d < s.d < t'.d < t'.f < s.f$. But we must also have $t.d < t'.d < t'.f < t.f$ by our assumption that t' is a descendant of t. These two facts jointly contradict our assumption that $v.d < s.d < s.f < t.d < t.f < v.f$, from which we conclude that no such path from t' to s' exists. It follows that v is vulnerable, by the fact that removing v would eliminate the only path connecting vertex s' to t'.

(ii) We can determine whether or not v is vulnerable by running DFS starting at v and examining whether or not v has multiple children in the resulting DFS tree.

Problem 2 (20 points)

Problem 2.1 (5 points)

Prove that every directed acyclic graph (DAG) has at least one vertex with no entering edges. That is, for any DAG $G = (V, E)$, there exists a node $v \in V$ for which there **do not** exist any edges of the form $(u, v) \in E$.

Solution

Let $G = (V, E)$ be a DAG. Then by definition it contains no cycles. By way of contradiction, suppose that every vertex $v \in V$ has at least one entering edge. That is, for every $v \in V$, there exists a vertex $u \in V$ distinct from v such that $(u, v) \in E$. Now, consider some vertex $v_1 \in V$, and consider one of its predecessors $v_2 \in V$. Such a vertex v_2 exists by our assumption. Now, consider one of the predecessors of v_2, which again is guaranteed to exist by our assumption. Call this predecesor v_3. If this predecessor is equal to v_1, then we have found a cycle, violating the assumption that G is a DAG. Thus we must have $v_3 \neq v_2$, and we have $v_3 \neq v_2$ by the fact that v_3 is a predecessor of v_2. Continuing in this way, we can construct a sequence of vertices $v_1, v_2, v_3, \ldots, v_k$ where v_i is the predecessor of v_{i-1} for each $i = 2, 3, \ldots, k$, and all the v_i are distinct. Consider, however, what happens once we have constructed such a sequence with length $n = |V|$. That is, consider the situation when we have vertices $v_1, v_2, \ldots, v_n$, with all vertices distinct. By assumption, v_n has at least one predecessor, call it u. but since our sequence is all of the vertices in V, u must be one of the vertices in our sequence. By definition of this sequence, $v_n, v_{n-1}, \ldots, v_2, v_1$ form a directed path from v_n to v_1. Consider the subpath of this path, running from v_n to u. The fact that $(u, v_n) \in E$ means we can add this edge to the path to make a cycle, contradicting our assumption that G is a DAG. Thus, we have shown a contradiction, and it must NOT be the case that every vertex in V has a predecessor in G. That is, there must exist at least one vertex with no entering edges.

Problem 2.2 (5 points)

Find a necessary and sufficient set of conditions for a DAG to have a unique topological sort. That is, find a set of statements $S_1, S_2, \ldots, S_m$ such that directed acyclic graph G has a unique topological sort if and only if $S_1, S_2, \ldots, S_m$ are all true.

Solution

A graph $G = (V, E)$ has a unique topological sort when there exists exactly one ordering of its nodes $v_1, v_2, \ldots, v_n$ that satisfies $(v_i, v_j) \in E \Rightarrow i < j$.

First, consider a directed acyclic graph $G = (V, E)$, and consider some topological sorting of its nodes, $v_1, v_2, \ldots, v_n$. Consider vertices v_i and v_{i+1} for $1 \leq i < n$. If no edge exists connecting these two vertices (i.e., $(v_i, v_{i+1}) \notin E$ and $(v_{i+1}, v_i) \notin E$), then switching these two vertices in the ordering does not violate the sort. If no such pair of adjacent vertices satisfies this condition, that is, if for all $i = 1, 2, \ldots, n-1$ we have an edge $(v_i, v_{i+1}) \in E$, then the path $(v_1, v_2, \ldots, v_n)$ constitutes a Hamiltonian path in G. Thus, G containing a Hamiltonian path is a necessary condition for G to have a unique topological sort. Conversely, suppose that G contains a Hamiltonian path $(v_1, v_2, \ldots, v_n)$. Any other ordering of the nodes will introduce an edge that violates the topological sorting condition. Thus, if G contains a Hamiltonian path, it has a unique topological sort, and therefore G containing a Hamiltonian path is a sufficient condition for G to have a unique topological sort.

We conclude that G has a unique topological sort if and only if G contains a Hamiltonian path.

Problem 2.3 (10 points)

Write a non-recursive version of DFS. That is, write a new version of depth-first search that doesn't need to call itself. Prove that your algorithm is correct.

Solution

Use a stack, rather than recursing. That is, rather than calling DFS-VISIT recursively on a node u, simply push node u onto the stack. To retrieve the next vertex for processing, simply pop from the stack. Correctness follows from the fact that the original algorithm (e.g., the recursive one in CLRS) is correct, and that this change changes nothing except that instead of using recursion to determine the order that we process nodes, we use a stack to keep track of which nodes to process next. Indeed, the stack used in this way exactly mirrors the memory stack that we would be using in the recursive case. The one difference is that if our original recursive algorithm visited children u, v, w of vertex x in the order u, v, w, our algorithm would visit them in the opposite order unless we changed the order that we put them on the stack.

Optional exercises

Solve the following problems and exercises from CLRS: 23.2-7, 23.2-1, 23-1, 23-4.