Introduction to Algorithms November 14, 2003
Massachusetts Institute of Technology 6.046J/18.410J
Professors Shafi Goldwasser and Silvio Micali Handout 24

Problem Set 6 Solutions

Problem 6-1. Cycles

(a) Give an $O(E^2 \lg V)$ algorithm to find a minimum weight cycle in a given weighted, undirected, connected graph in which all edge weights are non-negative. Prove your algorithm is correct and analyze its running time. (**Optional Extra Credit Problem:** Can you find a more efficient algorithm for this problem?)

Solution: For every edge $(i, j) \in E$, we run Dijkstra's algorithm to find the shortest path from i to j in $G \setminus (i, j)$. This yields the shortest cycle that uses edge (i, j). Thus, we find E cycles and we choose the one with minimum weight.

This algorithm runs in $O(E^2 \lg V)$ time since we run Dijkstra's algorithm E times.

(b) Give an $O(VE)$ algorithm to determine if a given directed, connected graph contains an edge that is in every cycle. Prove your algorithm is correct and analyze its running time. (**Optional Extra Credit Problem:** Can you find a more efficient algorithm for this problem?)

Solution: We run DFS to find a cycle in the given graph G. Suppose we call this cycle C. Note that C contains at most V edges. If G contains an edge in every cycle, then it must be one of the edges in C. Then we run DFS again on $G \setminus \{C\}$. If the resulting graph contains a cycle, then G contains two edge-disjoint cycles, and so there is no edge in every cycle. Otherwise, for each edge e_i in the cycle C, we can check if $G \setminus \{e_i\}$ is acyclic. If so, e_i is the edge we are seeking. If $G \setminus \{e_i\}$ is not acyclic for any edge e_i in the cycle C, then G does not contain a single edge in every cycle.

This algorithm runs DFS $O(V)$ times. DFS takes $O(E)$ time (since the graph is connected) so the running time is $O(EV)$.

(c) Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove your algorithm is correct and analyze its running time.

Solution: Run BELLMAN-FORD on the graph and determine whether or not a negative-weight cycle exists. If one exists, then we discover it by finding an edge (u, v) that is part of the cycle. Once we have this, we can use a simple dynamic programming algorithm to find a negative-weight cycle containing u by computing the shortest path from u to every other node containing at most k edges. That is, compute the $n \times n$ matrix S, where $S[k, v]$ is the length of the shortest path from u to v containing at most

k edges, and also remember the path along the way (using a separate predecessor matrix). Note that $n = |V|$. This can be done in $O(n^3)$ time. Notice that this is almost exactly what BELLMAN-FORD does, and so we can simply use the predecessor field that it computes. Specifically the following procedure should return a negative-weight cycle if one exists, and return NIL if none does.

```
BELLMAN-FORD-FIND-NEGATIVE-CYCLE(G, w, s)
 1  INITIALIZE-SINGLE-SOURCE(G, s)
 2  for i ← 1 to |V[G]| − 1
 3       do  for each edge (u, v) ∈ E[G]
 4                do  RELAX(u, v, w)
 5  for each edge (u, v) ∈ E[G]
 6       do  if d[v] > d[u] + w(u, v)
 7                then mark v
 8                     x ← v
 9                     while π[x] is not marked
10                          do  mark π[x]
11                              x ← π[x]
12                     return marked nodes
13  return NIL
```

Problem 6-2. Minimum Spanning Trees

(a) Consider the following algorithm, NEW-MST, for computing a minimum spanning tree. The algorithm takes as input an undirected, weighted, connected graph G. It sorts the edges in non-increasing order according to edge weight. It then goes through each edge in G, in the sorted order, and determines if removing that edge disconnects the graph. If not, then the edge is removed permanently, otherwise the edge remains. Below, we provide pseudocode for the NEW-MST algorithm.

```
NEW-MST(G)
1  Sort edges according to edge weight in non-increasing order: e_1, e_2, ..., e_m
2  i ← 1
3  While i ≤ m:
4     If G \ {e_i} is connected
5          G ← G \ {e_i}
6     i ← i + 1
7  Output G.
```

Prove that the NEW-MST algorithm outputs a minimum spanning tree of the input graph G.

Solution: To prove this, we need to show that the result of the algorithm is a tree, spans the vertices of G, and has minimum weight.

First we will observe that after every step (every removal of an edge), the graph is still connected. This is an easy inductive argument. It is given that the graph we start with is connected. At every step, we remove only an edge which does not disconnect the graph; so if it is connected after some number of steps, it will still be connected after the next. This proves the claim. This in turn shows that the graph resulting from the algorithm spans the vertices of G, since in specific the graph is connected after the final step.

Now, we will show that the output graph is a tree. Suppose that it contains some cycle. Then removing one of the edges in the cycle does not disconnect the graph. Therefore, at least one of these edges would have been removed, which is a contradiction.

It remains to show that the spanning tree has minimum weight. Suppose there were a spanning tree with lower total weight than ours. It includes some edges which are not part of our tree. (Both trees have the same number of edges, since they both have V vertices, but they can't have exactly the same ones.) Consider the one with smallest weight. If we add that edge back into our tree, it will create a cycle. Now consider the heaviest edge in that cycle; at the point when the NEW-MST algorithm decides whether or not to delete it, none of the other edges have yet been considered, since they are all lighter. So the cycle is present in the graph at that time, and this edge would have been deleted. Therefore the NEW-MST algorithm couldn't have produced that nonoptimal tree.

(b) Give an efficient algorithm to find a spanning tree for a connected, weighted, undirected graph G such that the weight of the maximum-weight edge in the spanning tree is minimized. Prove your algorithm is correct.

Solution: First we prove the following claim: If T is a minimum spanning tree of G then T minimizes the maximum-weight edge over all spanning trees of G.

Let (u, v) be the heaviest edge in T. Now let $(S,\ V - S)$ be the cut which respects $T - \{(u, v)\}$. The edge (u, v) must be a light edge crossing this cut, since otherwise we could replace it with a lighter edge, contradicting the fact that T is a minimum spanning tree. Any spanning tree of G includes at least one edge that crosses this cut, which must be at least as heavy as (u, v). Hence, T minimizes the maximum-weight edge.

Thus we can use any minimum spanning tree algorithm, and in particular, we can use Prim's algorithm with Fibonacci heaps to find a spanning tree which minimizes the maximum-weight edge in $O(E + V \lg V)$ time.[1]

Problem 6-3. Labeled Graphs and Longest Paths

You are given a directed graph $G = (V, E)$, in which each vertex has a unique label $\ell(v) : V \to Z^+$, a cost function $w : E \to \mathcal{R}$ assigning a weight to each edge, and a source $s \in V$. Additionally,

[1] It is possible to solve this in $O(E)$ time, but we won't do this here.

every edge (i, j) has the property that $\ell(i) < \ell(j)$. Design an algorithm to construct an output array D such that $D[i]$ is the length of the longest path from s to v_i in G.

Solution: Note that the input graph is a directed acyclic graph since $\ell(i) < \ell(j)$ for all edges (i, j). If all vertices are labeled with a positive integer, as specified in the problem statement, then every cycle must contain at least some edge (u, v) such that $\ell(u) > \ell(v)$. With this observation, we note that the problem is equivalent to finding the longest directed path in a DAG.

Our algorithm is as follows. First run topological sort on the given DAG, and relabel the vertices of G such that each edge (v_i, v_j) satisfies $i < j$. Then do the following for s: For all $i < s$ set $D[i] = -\infty$. (If a node is to the left of the source in the topological order, then there can be no path from the source to that node.) Now, we have that if $s \to v_1 \to v_2 \to \ldots \to v_k \to v$ is the longest path from s to v, then $s \to v_1 \to v_2 \to \ldots \to v_k$ is the longest path from s to v_k. This observation immediately presents a dynamic programming solution to the given problem. Now move right across the array D, filling in the remaining values according to the following rule:

$$D[v] = \max_{\substack{s<u<v \\ (u,v)\in E}} [D[u] + w(u, v)]$$

It is possible to implement this procedure so that it runs in time $O(V + E)$, since in building D we only need to look at each edge once. (The initial topological sort can also be performed in time $O(V + E)$.)

Problem 6-4. Amortized Weight-Balanced Trees

Consider an ordinary binary search tree augmented by adding to each node x the field $size[x]$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $\frac{1}{2} \leq \alpha < 1$. We say that a given node x is α-*balanced* if

$$size[left[x]] \leq \alpha \cdot size[x]$$

and

$$size[right[x]] \leq \alpha \cdot size[x].$$

The tree as a whole is α-*balanced* if every node in the tree is α-balanced. We will study the following amortized approach to maintaining weight-balanced trees.

(a) Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $\frac{1}{2}$-balanced. You algorithm should run in time $\Theta(size[x])$ time and can use $O(size[x])$ auxiliary storage.

Solution: Do an inorder walk of the subtree rooted at x to sort the elements. That will take $\Theta(size[x])$. Now build a $\frac{1}{2}$-balanced tree by making the median the new root and continuing recursively. Note that we are building an AVL tree.

(b) Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

Solution: It takes $O(\log_{1/\alpha} n)$ to search an α-balanced tree. For α constant, this is $O(\log n)$. We can show this by induction on n. If $n = 1$ the height of the tree is $0 = \log_{1/\alpha} 1$. Suppose the claim is true for $k < n$. The height of a tree of size n rooted at x is

$$1 + max(\text{ height}[left[x]], \text{height}[right[x]])$$

We know $size[left[x]] \leq \alpha n$ and $size[left[x]] \leq \alpha n$, since the tree is α-balanced, so by induction, height[$left[x]$] and height[$right[x]$] are both at most $\log_{1/\alpha}(\alpha n)$. So the total height is at most

$$1 + \log_{1/\alpha}(\alpha n) = \log_{1/\alpha}(1/\alpha) + \log_{1/\alpha}(\alpha n) = \log_{1/\alpha}((1/\alpha)(\alpha n)) = \log_{1/\alpha} n.$$

For the remainder of this problem, assume that the constant α is strictly greater than $\frac{1}{2}$. Suppose that INSERT and DELETE are implemented as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then the subtree rooted at the highest such node in the tree is "rebuilt" so that it becomes $\frac{1}{2}$-balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

and we define the potential of T as

$$\Phi(T) = \sum_{x \in T : \Delta(x) \geq 2} c \cdot \Delta(x),$$

where c is a sufficiently large constant that depends on α.

(c) Argue that any binary search tree has nonnegative potential and that a $\frac{1}{2}$-balanced tree has potential 0.

Solution: We know $\Delta(x) \geq 0$ because $\Delta(x)$ is an absolute value, so $\Phi(T) \geq 0$ for all T. If T is $\frac{1}{2}$-balanced then $\Delta(x) < 2 \; \forall x$, so $\Phi(T) = 0$.

(d) Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortize time to rebuild a subtree that is not α-balanced.

Solution: Suppose we have an m-node subtree rooted at x that is not α-balanced, and say the left size is the one that violates the balanced property, i.e. $size[left] > \alpha \cdot size[x]$. This means that $size[right] < (1-\alpha) \cdot size[x]$, and so $size[left] - size[right] > (\alpha - (1-\alpha))m = (2\alpha - 1)m$. Since $\Phi(T) \geq c\Delta(x)$, $\Phi(T) > c(2\alpha - 1)m$.

Rebuilding an m-node subtree to be $\frac{1}{2}$-balanced causes the potential of every node in the subtree to drop to 0, including x. So, the change in potential is at least $c(2\alpha - 1)m$. If we make $c \geq 1/(2\alpha - 1)$, then the potential will drop by at least m. Since rebuilding the tree costs m units of potential (as the problem states), the amortized time to rebuild is $O(1)$.

(e) Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\log n)$ amortized time.

Solution: Inserting a node into the tree consists of searching for a place for the node, attaching it to the tree, then rebuilding all subtrees that are not α-balanced. The searching takes real time $O(\log n)$, as argued in part (b). Each "rebuilding" takes $O(1)$ amortized time, as argued in part (c).

How many "rebuilds" must we perform? It turns out we only need one. The algorithm specified in the problem says that we rebuild the subtree rooted at the highest unbalanced node. All the nodes that could potentially become unbalanced when we insert

the new node are all on the path from the root to the point of insertion. So, if there are some unbalanced nodes, one such node is "higher" than all the others on this path, and this is the root of the subtree we rebuild. This rebuild makes all the other unbalanced nodes balanced, so it is the only rebuild we need to perform. Thus the total amortized time for insert is $O(\log n)$.