G22.3250-001 Honors Analysis of Algorithms November 3, 2016

Solutions to Problem Set 3

Name: Daniel Wichs (TA) *Due: October 27, 2008*

Problem 1

Given a tree with (possibly negative) weights assigned to its vertices, give a polynomial time algorithm to Find a subtree with maximum weight. Note that a subtree is a connected subgraph of a tree.

Solution: Let T be a tree with root v and let $v_1, \ldots, v_t$ be the children of v. Let $T_1, \ldots, T_n$ denote the n trees rooted at $v_1, \ldots, v_n$.

The key realization is that the maximal-weight subtree of T – call it $maxWeight(T)$ – either includes the root v or excludes it. In the latter case, it must just be the maximal-weight subtree of one of the children:

$$maxWeightWithout(T) = \begin{cases} v & v \text{ is a leaf, } weight(v) \geq 0 \\ \emptyset & v \text{ is a leaf, } weight(v) < 0 \\ \max\{maxWeight(T_1), \ldots, maxWeight(T_n)\} & \text{otherwise} \end{cases}$$

In the former case, if the maximal-weight subtree of T does include v, then the question is what other "subtrees" to add under v. To do so, we define $maxWeightWith(T)$ to be the maximal-weight subtree of T which *includes* the root v. Then this is simply the union of all $maxWeightWith(T_i)$ which have weight greater than 0 together with v.

Then

$$maxWeightWith(T) = \begin{cases} v & \text{if } v \text{ is a leaf} \\ \bigcup_{\geq 0}\{maxWeightWith(T_i)\} \cup v & \text{otherwise} \end{cases}$$

Lastly, putting this together we get:

$$maxWeight(T) = \max\{maxWeightWith(T), maxWeightWithout(T)\}$$

We note that the number of distinct recursive calls is at most $3|T|$ (three recursive functions which can get applied to any node) so, by cashing the answers on each recursive call (i.e. using dynamic programming) the run time of the algorithm is $3|T|$.

$\square$

Problem 2

Let $G = (V, E)$ be a directed acyclic graph (i.e. it does not contain any directed cycle).

1. Prove that the graph must have a vertex t that has no outgoing edge.

2. Suppose $|V| = n$. A topological ordering of the acyclic graph is a labeling of its vertices by integers from 1 to n such that

    - Any two distinct vertices receive distinct labels.
    - Every (directed) edge goes from a vertex with a lower label to a vertex with a higher label.

    Give a polynomial time algorithm to Find a topological ordering of the graph.

3. Fix a node t that has no outgoing edge. For every node $v \in V$, let $P(v)$ be the number of distinct paths from v to t. Define $P(v) = 0$ if no such path exists and define $P(t) = 1$ for convenience. Give a polynomial time algorithm to compute $P(v)$ for every node v.

Solution:

1. Pick an arbitrary vertex v and follow an arbitrary path "away from" v until you reach a vertex t that has no outgoing edges. Since there are no cycles, the path will never visit any vertex twice and hence the above process must terminate after some finite number of steps proving the existence of t.

2. See http://www.cs.nyu.edu/courses/fall06/G22.3520-001/lec14.pdf. Slides 20-23. Or CLRS pages 549-551.

3. Run a topological sort on G. Let label(v) be the value assigned to v by the sort, and $N(v)$ be the neighbor-set of v. Then

$$\text{npaths}(v,t) = \begin{cases} 0 & \text{label}(v) > \text{label}(t) \\ 1 & v = t \\ \sum_{v' \in N(v)} \text{npaths}(v',t) & \text{otherise} \end{cases}$$

□

Problem 3

(KT, Ch6:p21)

Let $p(1), \ldots, p(n)$ be positive real numbers. A k-shot strategy S is a sequence of at most k ordered integer pairs $(b_1, s_1), \ldots, (b_m, s_m)$, with $1 \le b_1 < s_1 < b_2 < s_2 \ldots < b_m < s_m$. Let $\text{val}(S) = \sum_{i=1}^{m} (p(s_i) - p(b_i))$. For any k we want to find the k-shot strategy which maximizes val(S).

Solution:

Let

$$M_{(i,j)} := \max_{i \le b < j} (p(j) - p(b))$$

be the maximum amount of money you can make in one buy/sell transaction with sell date j and buy date $b \ : \ i \le b < j$. It is easy to compute $M_{(i,j)}$ for all $1 \le i \le j$ in time $\mathcal{O}(n^3)$.

Now the best k-shot strategy in the days $i, i+1, \ldots, n$ must consist of making the best possible transaction with a sell date prior to some date b and the following the best $k-1$-shot strategy in the days $b+1, \ldots, n$. Formally,

$$BEST(k,i) = \begin{cases} 0 & k \leq 0 \text{ or } i \geq n \\ \max_{i \leq b \leq n} \left(M_{(i,b)} + BEST(k-1, b+1) \right) & \text{otherwise} \end{cases}$$

There are at most nk distinct values of $BEST(k,i)$ that need to be computed and each runs in time n for a total run time of $\mathcal{O}(n^3 + kn^2)$. The above algorithm needs to be modified to return the actual strategy rather than just the profit, but this just requires some simple book-keeping.

□

Problem 4

(KT, Ch6:p22)
Given a graph G with some edge weights such that the all cycles in G have positive weight, together with vertices s, t find the number of shortest paths from s to t.

Solution:
Use Bellman-Ford to find the length $\text{best}(u,t,n)$ of the shortest path from any node u to the node t which uses fewer than n edges. Now we define $\text{npaths}(u,t,n)$ to be the number of shortest-paths from u to t using fewer than n edges. Then $\text{npaths}(u,t,n) = \sum_{w \in S(u)} (\text{npaths}(w,t,n-1))$ is the sum of the number of shortest paths from w to t using fewer than $n-1$ edges, for all neighbors w such that some shortest path from u to t goes through w. We call this set $S(u)$. But $w \in S(u) \Leftrightarrow c(u,w) + \text{best}(w,t,n-1) = \text{best}(w,t,n)$ (where $c(u,w)$ is the cost of the edge (u,w)).So it is easy to check if a vertex is in $S(u)$. Therefore we get

$$\text{npaths}(u,t,n) = \begin{cases} 1 & u = t \\ 0 & u \neq t, n = 0 \\ \sum_{w \in S(u)} \text{npaths}(w,t,n-1) & \text{otherwise} \end{cases}$$

There are $|V|^2$ distinct problems each of which takes at most $|V|$ steps to compute for a run-time of $\mathcal{O}(|V|^3)$.

□

Problem 5

(KT, Ch6:p28)
Given n jobs such that job i takes time t_i and must finish before deadline d_i find a schedule which runs the maximum number of jobs.

Solution:

1. First we show that there is an optimal schedule in which the jobs run in order of increasing deadlines. Imagine that S is an optimal schedule and that, in S, tasks do not run in order of increasing deadlines. Then there must be two tasks which run adjacent in S such that the later one has an earlier deadline. But we can always switch the order of these tasks and they finish within their deadlines (and the rest of the schedule is unchanged). By performing this

re-ordering operation many times, we get a schedule where jobs run in order of increasing deadlines.

2. Sort tasks in order of increasing deadlines. In sorted order, let the deadlines be $d_1, \dots, d_n$ and the run-times $t_1, \dots, t_n$. Let $\text{sched}(i, s)$ be the optimal (value) of the schedule for tasks $i, i+1, \dots, n$ starting at time s. Then the optimal schedule either runs the first task, and then runs the optimal schedule of the remaining $n-1$ tasks from time $s + t_i$, or it does not run the first task and just runs the optimal schedule of the remaining tasks from time s.

$$\text{sched}(i,s) = \begin{cases} 0 & s > d_n \text{ or } i > n \\ \max(1 + \text{sched}(i+1, s+t_i), \text{sched}(i+1, s)) & \text{otherwise} \end{cases}$$

Figuring out the actual schedule requires simple additional book-keeping which we skip. We see that there are at most $n \times d_n$ possible problems each of which takes $\mathcal{O}(1)$ time so, using dynamic programming the run-time of the above recursion is $\mathcal{O}(n \times d_n)$ together with sorting we then get a run time of $\mathcal{O}(n \log n + nd_n)$ (where d_n is the maximal deadline).

□

Problem 6

An independent set Iin a graph is called maximal if the graph does not contain an independent set I' such that $I \subseteq I'$, $|I| < |I'|$. Given a tree on n vertices, and an integer $0 \le k \le n$, give a polynomial time algorithm to determine whether the tree has a maximal independent set of size k. (Hint: Design an algorithm that solves the problem for all possible values of k.).

Solution:
As the main idea, each node v of the tree will store two sets:

maxWith(v) : the set of all k such that the subtree rooted at v contains some maximal independent subset of size k which *includes* v.

maxSans(v) : the set of all k such that the subtree rooted at v contains some maximal independent subset of size k which *excludes* v.

We also define $\mathbf{maxAny}(v) = \mathbf{maxSans}(v) \cup \mathbf{maxWith}(v)$.

It is clear that, if v is a leaf then $\mathbf{maxWith}(v) = \{1\}$ and $\mathbf{maxSans}(v) = \{0\}$.

If v has children $v_1, \dots, v_m$, and grandchildren $w_1, ..., w_k$ then

$$\mathbf{maxWith}(v) = \{1 + t_1 + t_2 + \ldots + t_k \ : \ t_1 \in \mathbf{maxAny}(w_1), \dots, t_k \in \mathbf{maxAny}(w_k)\}$$

since an independent subset containing v must not include its children. Note that a maximal independent set S rooted at v that contains v can potentially have none of v's children or grand-children, which means $S \setminus \{v\}$ is not necessarily a maximal independent set for the subtree rooted at any of the children of v.

Also

$$\mathbf{maxSans}(v) = \bigcup_i \{t_1 + t_2 + \ldots + t_m \ : \ t_1 \in \mathbf{maxAny}(v_1), \dots, \mathbf{maxWith}(v_i), \dots, t_n \in \mathbf{maxAny}(v_n)\}$$

Now we just need to recursively compute **maxAny**$(root)$ and check if k is included in the answer. By caching the values at the nodes (i.e. using dynamic programming) we see that we actually only solve two problems per node. Also, the amount of work done at the nodes is only the merging of the sets **maxWith**(v_i) **maxSans**(v_i) computed for the children v_i. Each such set is of size at most $|V|$, and **there is an additional condition that any merged set also has to be of size at most** $|V|$ and hence the merging as well as the total algorithm run in polynomial time.

□

Problem 7

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays. Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil lg(n + 1)\rceil$, and let the binary representation of n be $< n_{k-1}, n_{k-2}, ..., n_0 >$. We have k sorted arrays $A_0, A_1, ..., A_{k-1}$, where for each $i = 0, 1, ..., k - 1$, the length of array A_i is 2^i . Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

Solution: a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.
To search, we can search each individual array using binary search, and since there are $\log n$ such arrays, the worst case running time is $O(\log^2 n)$.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.
To perform INSERT,

- Create a new array A with just the element to be inserted.
- Repeat for $i = 0, 1, ..., k - 1$:
  - Check if A_i is empty, if it is, set A_i to A and return.
  - If A_i is not empty, merge A and A_i and assign the merged list to A. Set A_i to empty and continue.
- If you have reached the end of the loop, create a new array A_k and assign A to it and return.

The algorithm always terminates, and since we only do merges, A always remains sorted and contains the element to be inserted. This shows correctness.
The worst case running time for this algorithm is when all sets $A_0, ...A_k$ are non empty, and therefore the algorithm does k merges. The total cost can be calculated as

$$T(n) = 2(2^0 + 2^1 + 2^2 + ... + 2^{k-1} = O(n)$$

Now, let us analyse the amortized running time. Consider n insertions into the data structure.

Consider A_i, which is initially empty. A_i is filled only when A_0 to A_{i-1} are filled and then an item is inserted, and subsequently, every time A_0 to A_{i-1} are filled and another item is inserted, it gets merged. Therefore, A_i is assigned, or merged once every 2^i insertions. The invariant hold for every single list. Therefore, the total cost of merges after n insertions is (Cost of merging 2 lists of size m is $2m$) $\sum_{i=1}^{k-1}(2|A_i| * n/2^i) = O(nk) = O(n \log n)$. Therefore, the amortized cost of each insertion is $O(\log n)$.

c. Discuss how to implement DELETE.

- Find the smallest A_i such that A_i is full, let y be the last element of A_i.
- Search for the element x that you want to delete, say it is in A_j.
- Remove x from A_j and insert y into A_j and move it to it's correct position in A_j.
- Divide A_i, which now has $2^i - 1$ elements, as follows. The first element is moved to A_0, the next 2 go to A_1 and so on, till 2^{i-1} elements go to A_{i-1} and A_i is empty. The new arrays created are sorted because of construction.

The algorithm terminates and removes the item x if it is present in any of the arrays. The worst case running time is $O(n)$ to search for the element and for the split if n is a power of 2. □

Problem 8

Amortized weight balanced trees.

Solution: a. Rebuilding the sub-tree in time $x.size$ and space $O(x.size)$.
Perform in-order traversal of the tree rooted at x to get a sorted array of elements. Then build a balanced tree out of this array as follows: choose the median element and make it a root (Constant time as the array is sorted). Then recursively build a left subtree from the portion of the array thats lower than the median, and a right subtree from the portion of the array which is greater than the median. In each call, one element is fixed and it takes constant time. Therefore, in linear time we can rebuild the tree.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(lgn)$ worst-case time.
The claim can be made because α is a constant that is strictly less than 1. In each iteration, we compare the element to the current root and recursively search in the appropriate subtree based on the result of the comparison. The worst case, is when the element, in each iteration, happens to be in the larger subtree. Since the tree is α balanced, we can write the following recursion for the worst case:

$$T(n) = 1 + T(\alpha n)$$

which gives $T(n) = O(\log n)$.

c. Argue that any binary search tree has nonnegative potential and that a 1/2- balanced tree has potential 0.

Potential is always non-negative as c is positive and so is $\Delta(x)$ (absolute value). If the tree is 1/2-balanced, the summation has no terms as there cannot be a node where the difference in the number of nodes between left and right subtree can be 2(or more), as that would contradict the balance condition for that node.

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 $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

Say without loss of generality that the left subtree is larger. Therefore, $\Delta(x) = x.left.size - x.right.size > \alpha \times x.size - (1-\alpha)x.size = m(2\alpha - 1)$.

The amortized cost is $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$ and we want it to be $O(1)$. Since we rebuild it to be 1/2 balanced, the potential after rebuilding is 0. Therefore, we want $m - \Phi(D_{i-1}) \leq O(1)$ which means $\Phi(D_{i-1}) \leq m$.

$\Phi(D_{i-1}) \geq c.\Delta(x) = cm(2\alpha - 1)$. Therefore, we want $c > \frac{1}{2\alpha-1}$.

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

The amortized cost is given by $\hat{c_i} = c_i + \Delta\Phi$. The cost of insertion or deletion is basically the cost of search, which is $O(\log n)$.

If a rebuild is necessary after the insertion or deletion, we know from the previous part that the rebuild takes amortized $O(1)$. Consider the potential difference when you insert or delete an element in the tree. For every node x on the path from the node inserted to the root, $\Delta(x)$ changes by 1, and for every other node, it doesn't change. Since the depth of the tree is $O(\log n)$, $\Delta\Phi = O(\log n)$. Therefore, the amortized cost is $O(\log n)$.

$\square$