CSE 202: Design and Analysis of Algorithms Winter 2015

Problem Set 2

Instructor: Kamalika Chaudhuri **Due on:** Thu Jan 31, 2013

Instructions

- For your proofs, you may use any lower bound, algorithm or data structure from the text or in class, and their correctness and analysis, but please cite the result that you use.
- Each problem is worth 10 points.
- If you do not prove that your algorithm is correct, we will assume that it is incorrect. If you do not provide an analysis of the running time, we will assume you do not know what the running time is.

Problem 1

A company is looking to open gas stations along Highway 5 from San Diego all the way to San Francisco. There are N possible locations where gas stations may be opened; the locations are all along a straight line from San Diego to San Francisco and location i is m_i miles from San Diego (where $m_1 \leq m_2 \leq \ldots \leq m_N$). If a gas station is opened at location i, then the profit from it is $p_i \geq 0$. However, no two gas stations may be closer than k miles.

Design a dynamic programming algorithm that outputs the set of locations where the company should open gas stations for maximum profit.

Solution

Subtasks. For any $1 \leq i \leq N$, let $M(i)$ denote the maximum profit that can be achieved by opening gas stations at a subset of locations $1, \ldots, i$.

Recurrence Relation. We calculate $M(i)$ based on $M(j)$ for $j < i$. The recurrence relation is:

$$M(i) = \max(M(i-1), p_i + M(prev(i)),$$

where $prev(i)$ is the largest index j for which $m_i - m_j \geq k$. Before solving the recurrence, the array $prev(i)$ can be pre-computed sequentially in linear time. We start with $prev(1) = 0$. To compute $prev(i+1)$, we do a linear search in increasing order of indices starting from $prev(i)$ until we find an index j such that $m_{i+1} - m_{j+1} < k$. We set $prev(i) = j$.

Base Case, Final Solution, Evaluation Order. For consistency we can set $m_0 = -\infty$; in this case, the base case will be $M(0) = 0$ and $prev(1) = 0$. The evaluation order is $i = 1, \ldots, N$.

To find the final set of locations, we maintain a second list L. If $M(i) = M(prev(i)) + p_i$, then $L(i) = i$; otherwise $L(i) = \perp$. We can reconstruct the final solution by following pointers back from $L(N)$. We start with an empty list and $i = N$. If $L(i) = i$, then we add i to our list of locations, and move to $prev(i)$. Otherwise, we move to $i - 1$.

Correctness. We will show by induction that the subtask $M(i)$ is correctly computed for all i. The base case is $M(1) = p_1$, which is correctly computed. For the inductive case, observe that either we include the location i or not; if not, then the value of the solution is $M(i-1)$. If we include location i, then the solution obtained consists of i plus the best solution obtained out of locations $1, \ldots, prev(i)$: we cannot include any location in $prev(i) + 1, \ldots, i - 1$ as these locations are closer than k.

Running Time. The running time is $O(N)$. It takes $O(N)$ time to construct $prev(i)$. Moreover, constructing $M(i)$ takes constant time per entry and thus $O(N)$ time in total.

Problem 2

Suppose you are given a labelled directed *acyclic* graph $G = (V, E, L)$, where L represents the labels on the edges. Each edge $e \in E$ has one of three possible labels – ((an open parenthesis),) (a closed parenthesis) and 0. A path in the G is said to be *balanced* if the sequence of labels of the edges along the path include the same number of open and closed parentheses. For example, paths with label sequences $(, 0,), (, 0,)$ and $), 0, (, 0, 0$ are balanced, while a path with a label sequence $(,),)$ is not balanced.

Given G and two specified nodes s and t in G, design a dynamic programming algorithm to find if there exists a balanced path between s and t in G, and output the path if it exists.

Solution

Subtasks. For any path P from u to v, we define the balance of P as the difference between the number of (labels and the number of) labels along P. Thus if P has balance 0 then P is balanced. We also define the balance of an edge $t(v, u)$ as follows: $t(v, u) = 1$ if the label on the edge (v, u) is (, $t(v, u) = -1$ if the label on the edge (v, u) is) and $t(v, u) = 0$, otherwise.

For each vertex $u \in V$, and $b \in \{-n+1, \ldots, -1, 0, 1, \ldots, n-1\}$, we define the subtask $B(u, b)$ as follows. $B(u, b) = 1$ if there exists a path between s and u with balance b and $B(u, b) = 0$ otherwise.

Recurrence Relation. Let P be a path between s to u of balance b; if (v, u) is the final edge on the path, then $B(u, b) = B(v, b - t(v, u))$. We can use this property to define a recurrence relation as follows:

$$B(u, b) = \max_{v:(v,u)\in E} B(v, b - t(v, u)) \tag{1}$$

Base Case, Final Solution, Evaluation Order. The base case is $B(s, 0) = 1$, $B(s, b) = 0$ for $b \neq 0$. The final solution is $B(t, 0)$. Now to run the dynamic programming algorithm, we sort all the nodes in the DAG in topological order, and then evaluate $B(u, b)$ for u in the topological order, and $b = -n + 1, \ldots, n - 1$.

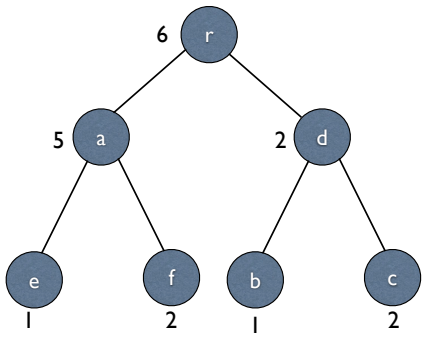
Correctness. We show a proof of correctness by induction. First, observe that the base case is correctly calculated for s and all b. Now we apply induction on nodes in the order in which they occur in a topological sort of the DAG, and argue that for a node u, $B(u, b)$ is correctly calculated for all b. This statement follows from the assumption that the nodes are topologically sorted – by the induction hypothesis, for all v such that $(v, u) \in E$, v occurs before u in the topological order and thus $B(v, b - t(v, u))$ is correctly evaluated. Thus from Equation 1, $B(u, b)$ is also correctly evaluated. Thus the inductive case holds, and the correctness proof follows.

Running Time. We maintain the subtasks $B(u, b)$ in a matrix of size $n \times n$; initializing this matrix takes $O(n^2)$ time. For each node u, updating $B(u, b)$ for all b takes $O(n \cdot indeg(u))$ time; thus the total running time is $O(mn)$. The total running time of the algorithm is thus $O(mn + n^2)$.

Problem 3

Suppose you are given a rooted binary tree T in which each node x has a score $s(x)$. A pruning of T is a set C of nodes in T with the property that C contains exactly one node on any root to leaf path. Thus, if a node x is in C, no ancestor or descendant of x can be in C.

The size of a pruning C is the number of nodes in C, and the score of a pruning is the sum of the scores of the nodes in it. For example, the following tree has 2 prunings of size 3: $C_1 = \{a, b, c\}$ with score 8 and $C_2 = \{d, e, f\}$ with score 5, and one pruning of size 2: $C_3 = \{a, d\}$, with score 7.



Given a binary tree T, with root r, and the scores $s(x)$ for each node x of T, design a dynamic programming algorithm to find the score of a pruning C of T such that (a) the size of C is exactly k (b) the score of C is as large as possible. To get any credit, the running time of your algorithm should be polynomial in n and k. (Hint: The exhaustive search algorithm which checks all possible prunings of size k has $n^{O(k)}$ running time.)

Solution

Subtasks. We give a dynamic programming algorithm to find the largest score of a pruning of size k. The first step is to divide the problem into subtasks. Let $BP(v, i)$ for a vertex $v \in T$ and a positive integer i be the score of the best pruning of size i of the subtree rooted at v.

Recurrence Relation. Next, we give a recursive definition of the subtasks. Suppose v is a non-leaf with two children ℓ and r. For the subtree rooted at v and $i > 1$, we have the recursive relation:

$$BP(v, i) = \max_{1 \le j < i} \left(BP(\ell, j) + BP(r, i - j) \right). \tag{2}$$

Base Case, Final Solution, Evaluation Order. The base case is as follows: for any vertex v (leaf or non-leaf), $BP(v, 1) = s(v)$. For a leaf u, $BP(u, i)$ is undefined for $i > 1$; the algorithm will work if we set this to $-\infty$.

Finally, we must give the correct order of solving the subtasks. We will solve them from bottom-up (a postorder traversal of T), and for each vertex, proceed from $i = 1$ to k. The complete algorithm is as follows. Compute a postorder traversal $v_1, \ldots, v_n$ of T. For $i = 1 \ldots n$ and $j = 1 \ldots k$, calculate $BP(v_i, j)$ using (2) and previously-computed values. Return $BP(r, k)$, where r is the root of T.

Correctness. Correctness proofs for dynamic programming algorithms are often quite simple, as they usually just involve using a recurrence. It should be easy to see that (2) is the correct definition by induction. For the base case, if v is a root, the only pruning is the single vertex v. For the induction step, assuming that $BP(u, i)$ is correct for descendants of v, the best pruning of size 1 for a subtree rooted at v is v itself; otherwise it is not a pruning. If v is not a leaf, then the best pruning of size > 1 cannot contain v (otherwise it is not a pruning), so it must be the best combination of the subtrees rooted at v's children, whose scores are accurately given by BP.

Running Time. The algorithm must compute $BP(i, j)$ for $i = 1 \ldots n$ and $j = 1 \ldots k$, or nk entries. Each computation uses only previously-computed entries, but it involves taking a max over up to k elements, which takes time linear in k. Therefore, the total running time is $O(nk^2)$.

A side note: if $n >> k$, we may not need to look at all nodes in the tree, because a pruning of size k cannot involve nodes of depth $> k$ in the tree. It is therefore sufficient to run the dynamic programming on nodes upto depth $\le k$. Since this is a binary tree, there are at most 2^k nodes at depth $\le k$, and hence a

running time of $\min(O(nk^2), O(2^k k^2))$ is sufficient. However, for the purpose of this class, you will get full credit for an otherwise correct and complete $O(nk^2)$ solution.