

# CMPS 102 Solutions to Homework 6

Solutions by Cormen and us

November 10, 2005

## Problem 1. 14.1-6 p.307

Whenever the *size* field of a node is referenced in either OS-SELECT or OS-RANK, it is used only to compute the rank of the node in the subtree rooted at that node. Suppose we store in each node its rank in the subtree which it is the root. Show how this information can be maintained during insertion and deletion.

When inserting node  $y$ , we search down the tree for the proper place for  $y$ . For each node  $x$  on this path, add 1 to  $rank[x]$  if  $y$  is inserted within  $x$ 's left subtree. Similarly when deleting, subtract 1 from  $rank[x]$  whenever the spliced-out node  $y$  had been in  $x$ 's left subtree.

We also need to handle the rotations that occur during the fix up procedures for insertion and deletion. Consider a left rotation on node  $x$ , where the pre-rotation right child of  $x$  is  $y$  (so that  $x$  becomes  $y$ 's left child after the left rotation). Suppose that  $x$  has a left subtree  $A$ , and  $y$  has left subtree  $B$  and right subtree  $C$ .

We leave  $rank[x] = |A| + 1$  unchanged. Before the left rotation  $rank[y] = |B| + 1$ . After the left rotation  $rank[y] = |A| + |B| + 2$ . Thus we need to execute the assignment

$$rank[y] := rank[x] + rank[y].$$

For a right rotation, the assignment  $rank[x]$  is again left unchanged and

$$rank[y] := rank[y] - rank[x].$$

## Problem 2. Augmented RB Tree

Augment RB trees so that the following operations can be implemented in  $O(\lg n)$  time.

Given a pointer to node  $x$  in tree  $T$  SMALLER( $x, T$ ) returns the sum of all elements in  $T$  that are less than  $x$ .

Assume all elements in  $T$  are distinct.

Define a new field  $sum[x]$  recursively:

$$\begin{aligned}sum[nil[T]] &= 0 \\sum[x] &= sum[left[x]] + sum[right[x]] + key[x]\end{aligned}$$

Store  $sum[x]$  in the node  $x$  along with the usual RB fields ( $right[x]$ ,  $left[x]$ ,  $p[x]$ ,  $color[x]$ ,  $key[x]$ ).

Let the sum of the elements in  $T$  that are less than  $x$  be  $S$ . Then  $SMALLER(T, x)$ , which computes  $S$ , can be defined as follows:

$SMALLER(T, x)$

1.  $S = sum[left[x]]$
2.  $y \leftarrow x$
3. **while**  $y \neq root[T]$  **do**
4.     **if**  $y = right[p[y]]$
5.          $S \leftarrow S + sum[left[y]] + key[y]$
6.      $y \leftarrow p[y]$
7. **return**  $S$

We now show that  $sum[x]$  can be implemented efficiently.

When inserting node  $z$ , we search down the tree for the proper place for  $z$ . While on this path if we are at node  $x$  we add to  $sum[x]$  the key of  $z$ .

When deleting node  $z$ , subtract  $key[z]$  from  $sum[x]$  whenever the spliced-out node  $z$  had been in  $x$ 's left or right subtree.

Now we show that rotations can be handled efficiently. Consider a left rotation on node  $x$  where the pre-rotation right child of  $x$  is  $y$  (so that  $x$  becomes  $y$ 's left child after the left rotation), see figure 13.2 on page 278 of Cormen.

We set

$$sum[x] \leftarrow sum[x] - sum[right[x]] + sum[left[y]],$$

then set

$$sum[y] \leftarrow sum[y] - sum[left[y]] + sum[x].$$

Right rotations are handled similarly. Consider a right rotation on node  $y$  where the pre-rotation left child of  $y$  is  $x$ . We set

$$sum[y] \leftarrow sum[y] - sum[left[y]] + sum[right[x]],$$

then set

$$sum[x] \leftarrow sum[x] - sum[right[x]] + sum[y].$$

These operations take  $\Theta(1)$  time. So the update can be maintained in time  $\Theta(\lg n)$ .

**Problem 3. 15-2 p.364 Printing neatly**

The input text is a sequence of  $n$  words of lengths  $l_1, l_2, \dots, l_n$  measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of  $M$  characters each. Note: We will assume that no word is longer than will fit into a line, i.e.,  $l_i \leq M$  for all  $i$ .

First, we'll make some definitions so that we can state the problem more uniformly. Special cases about the last line and worries about whether a sequence of words fit in a line will be handled in these definitions, so that we can forget about them when framing our overall strategy.

- Define  $extras[i, j] = M - \sum_{k=i}^j l_k - (j - i)$  to be the number of extra spaces at the end of a line containing words  $i$  through  $j$ . Note that  $extras$  may be negative.
- Now define the cost of including a line containing words  $i$  through  $j$  in the sum we want to minimize:

$$lc[i, j] = \begin{cases} \infty & \text{if } extras[i, j] < 0 \text{ (i.e. words } i, \dots, j \text{ don't fit),} \\ 0 & \text{if } j = n \text{ and } extras[i, j] \geq 0 \text{ (last line costs 0),} \\ (extras[i, j])^3 & \text{otherwise.} \end{cases}$$

By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimal sum, and by making the cost 0 for the last line (if the words fit), we prevent the arrangement of the last line from influencing the sum being minimized.

We want to minimize the sum of  $lc$  over all lines of the paragraph.

Our subproblems are how to optimally arrange words  $1, \dots, j$ , where  $j = 1, \dots, n$ .

Consider an optimal arrangement of words  $1, \dots, j$ . Suppose we know that the last line, which ends in word  $j$ , begins with word  $i$ . The preceding lines, therefore, contain the words  $1, \dots, i - 1$ . In fact, they must contain an optimal arrangement of words  $1, \dots, i - 1$ .

Let  $c[j]$  be the cost of an optimal arrangement of words  $1, \dots, j$ . If we know that the last line contains words  $i, \dots, j$ , then

$$c[j] = c[i - 1] + lc[i, j].$$

As a base case, when we're computing  $c[1]$ , we need  $c[0]$ . If we set  $c[0] = 0$ , then  $c[1] = lc[1, 1]$ , which is what we want.

But of course we have to figure out which word begins the last line for the subproblem of words  $1, \dots, j$ . So we try all possibilities for word  $i$ , and we pick the one that gives the lowest cost. Here,  $i$  ranges from 1 to  $j$ . Thus, we can define  $c[j]$  recursively by

$$c[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{1 \leq i \leq j} (c[i-1] + lc[i, j]) & \text{if } j > 0. \end{cases}$$

Note that the way we defined  $lc$  ensures that

- all choices made will fit on the line (since an arrangement with  $lc = \infty$  cannot be chosen as the minimum), and
- the cost of putting words  $i, \dots, j$  on the last line will not be 0 unless this really is the last line of the paragraph ( $j = n$ ) or words  $i, \dots, j$  fill the entire line.

We can compute a table of  $c$  values from left to right, since each value depends only on earlier values.

To keep track of what words go on what lines, we can keep a parallel  $p$  table that points to where each  $c$  value came from. When  $c[j]$  is computed, if  $c[j]$  is based on the value of  $c[k-1]$ , set  $p[j] = k$ . Then after  $c[n]$  is computed, we can trace the pointers to see where to break the lines. The last line starts at word  $p[n]$  and goes through word  $n$ . The previous line starts at word  $p[p[n]]$  and goes through word  $p[p[n]] - 1$ , etc.

In pseudo code we compute the following tables:

PRINT-NEATLY( $l, n, M$ )

1. Compute  $extras[i, j]$  for  $1 \leq i \leq j \leq n$ .
2. Compute  $lc[i, j]$  for  $1 \leq i \leq j \leq n$ .
3. Compute  $c[j]$  and  $p[j]$  for  $1 \leq j \leq n$ .
4. Return  $c$  and  $p$

Quite clearly, both the time and space are  $\Theta(n^2)$ .

In fact, we can do a bit better: we can get both the time and space down to  $\Theta(nM)$ . The key observation is that at most  $\lceil M/2 \rceil$  words can fit on a line. (Each word is at least one character long, and there's a space between words. ) Since a line with words  $i, \dots, j$  contains  $j - i + 1$  words, if  $j - i + 1 > \lceil M/2 \rceil$ , then we know that  $lc[i, j] = \infty$ . We need only compute and store  $extras[i, j]$  and  $lc[i, j]$  for  $j - i + 1 \leq \lceil M/2 \rceil$ .

**Problem 4. 15-6 p.368 Moving on a checkerboard**

Denote each square by the pair  $(i, j)$ , where  $i$  is the row number,  $j$  is the column number, and  $1 \leq i, j \leq n$ . Our goal is to find a most profitable way from any square in row 1 to any square in row  $n$ . Once we do so, we can look up all the most profitable ways to get to any square in row  $n$  and pick the best one.

A subproblem is the most profitable way to get from some square in row 1 to a particular square  $(i, j)$ . We have optimal substructure as follows. Consider a subproblem for  $(i, j)$ , where  $i > 1$ , and consider the most profitable way to  $(i, j)$ .

Because of how we define legal moves, it must be through square  $(i - 1, j')$ , where  $j' = j - 1, j, \text{ or } j + 1$ . Then the way that we got to  $(i - 1, j')$  within the most profitable way to  $(i, j)$  must itself be a most profitable way to  $(i - 1, j')$ .

Suppose that in our most profitable way to  $(i, j)$ , which goes through  $(i - 1, j')$ , we can earn a profit of  $d$  dollars to get to  $(i - 1, j')$ , and then  $p((i - 1, j'), (i, j))$  dollars getting from  $(i - 1, j')$  to  $(i, j)$ ; thus we earn

$$d + p((i - 1, j'), (i, j))$$

dollars getting to  $(i, j)$ .

Now suppose that there's a way to  $(i - 1, j')$  that earns  $d'$  dollars, where  $d' > d$ . Then we would use that to get to  $(i - 1, j')$  on our way to  $(i, j)$ , earning  $d' + p((i - 1, j'), (i, j)) > d + p((i - 1, j'), (i, j))$ , and thus contradicting the optimality of our way to  $(i, j)$ .

We also have overlapping subproblems. We need the most profitable way to  $(i, j)$  to find the most profitable way to  $(i + 1, j - 1)$ , to  $(i + 1, j)$ , and to  $(i + 1, j + 1)$ . So we'll need to directly refer to the most profitable way to  $(i, j)$  up to three times, and if we were to implement this algorithm recursively, we'd be solving each subproblem many times.

Let  $d[i, j]$  be the profit we earn in the most profitable way to  $(i, j)$ . Then we have that  $d[1, j] = 0$  for all  $j = 1, 2, \dots, n$ . For  $i = 2, 3, \dots, n$ , we have

$$d[i, j] = \max \begin{cases} d[i - 1, j - 1] + p((i - 1, j - 1), (i, j)) & \text{if } j > 1, \\ d[i - 1, j] + p((i - 1, j), (i, j)) & \text{always,} \\ d[i - 1, j + 1] + p((i - 1, j + 1), (i, j)) & \text{if } j < n. \end{cases}$$

To keep track of how we got to  $(i, j)$  most profitably, we let  $w[i, j]$  be the value of  $j$  used to achieve the maximum value of  $d[i, j]$ . These values are defined for  $2 \leq i \leq n$  and  $1 \leq j \leq n$ .

We can create the  $n \times n$  profit table ( $d$ ) and moves table ( $w$ ) in a bottom-up fashion. The columns are indexed by  $j$  and the rows  $i$ , we first compute row 1

$(1, 1) \dots (1, n)$ , then row 2  $(2, 1) \dots (2, n)$ , until row  $n$   $(n, 1) \dots (n, n)$ . The time to create the profit table is  $\Theta(n^2)$ . Once we have computed the  $d$  and  $w$  tables, we can determine the series of moves in  $\Theta(n)$  time.

**Extra Credit. 15-5 p.397 Viterbi algorithm**

We are given a directed graph  $G = (V, E)$ , where each edge  $(u, v) \in E$  is labeled with a sound  $\sigma(u, v)$  from a finite set of sounds from  $\Sigma$ , a start vertex  $v_0$  and a sequence  $s = (\sigma_1, \sigma_2, \dots, \sigma_k)$  of sounds from  $\Sigma$ .

1. Finding the path in the graph

There are several ways to do this problem. We'll do a dynamic programming style algorithm since it will be easy to adapt it in part b for the maximum probability path. The original problem statement is "Does there exist a path in the graph starting at  $v_0$  and labeled with  $\sigma_1 \dots \sigma_n$ ?". The subproblems would be "Does there exist a path in the graph starting at  $v_0$  and labeled  $\sigma_1 \dots \sigma_j$  and ending at  $v_i$ ?" (that is a subproblem asks is some prefix of the original string produced by starting a walk at this vertex). Thus a subproblem can be identified by 2 numbers -  $S(i, j)$ .  $i$  would be the vertex number and  $j$  the last sound in the prefix of the path. The values of the subproblems in this case are just boolean "yes/no" answers. The recurrence then looks like this:

$$\begin{aligned}
 S(i, j) &= \bigvee_{\{k : (v_k, v_i) \in E \wedge \sigma(v_k, v_i) = \sigma_j\}} S(k, j - 1) \\
 S(0, 0) &= 1 \\
 S(i, 0) &= 0 \text{ for } 1 \leq i \leq n - 1
 \end{aligned}$$

Next we need an order in which the table can be filled out. Our initial values are in the first column of the table. If we have solved all  $S(i, j)$  we can then solve all  $S(i, j + 1)$ . Thus we need to proceed by columns (e.g. top to bottom) starting on the left side of the table. Note that there is no need to keep the entire table around - we only need the column we are currently constructing and the one after it.

Another question is how to adapt this scheme so that some actual path can be extracted efficiently. Instead of storing yes/no answers in the table we can store the actual vertex numbers such that  $S(i, j) = k$  means "there is a path starting at  $v_0$  and ending at  $v_i$  and producing the prefix  $\sigma_1 \dots \sigma_j$ , the last step on that path is  $(v_k, v_i)$ ". We can set  $S(i, j) = -1$  to say that there is no such path. The recurrence would be the same, except instead of or we would need to use "select one from the set" operator. And now we would have to keep the entire table around since path reconstruction starting from the  $v_0$  vertex might involve any subproblems.

## 2. Finding the max probability path

Now that we have figured out in detail how to do part *a* with dynamic programming, finding the maximum probability path should be easy. The subproblems are same as before, except that instead of boolean values they will be real numbers indicating the maximum probability of path that produces that suffix and starts at that vertex. If a probability is 0 this means that there is no such path. To extract the path we will keep a separate table that will indicate which edge produces the maximum. Recurrences look like this ( $p$  is the mapping between edges and probabilities):

$$\begin{aligned}
 S(i, j) &= \max_{k : (v_k, v_i) \in E \wedge \sigma(v_k, v_i) = \sigma_j} \{p(v_k, v_i) \cdot S(k, j - 1)\} \\
 S(0, 0) &= 1 \\
 S(i, 0) &= 0 \text{ for } 1 \leq i \leq n - 1 \\
 R(i, j) &= \arg \max_{k : (v_k, v_i) \in E \wedge \sigma(v_k, v_i) = \sigma_j} \{p(v_k, v_i) \cdot S(k, j - 1)\}
 \end{aligned}$$

The time complexity of this algorithm is computed in the normal way for dynamic programming algorithms - table size times the time it takes to fill in one cell of the table. Table size is  $nk$ , where  $n$  is the number of vertices, and  $k$  is the length of  $s$ . But the time to fill in a cell depends on how many edges are there leaving that cell. Doing an upper bound by assuming that time per cell is as big as the total number of edges in the graph would not get us very good time bound. But we see that to fill in a column of the table we need to look at all vertices and at all edges leaving those vertices - which means we look at all the edges in the graph. Thus, even though time to fill in an individual cell could vary, the time to process a column is the same (a kind of amortized time argument is happening here). While filling in a single column we process all the vertices in the graph (height of the column) plus we process a number of edges per every vertex but the sum of that number over all vertices is just the entire number of edges in the graph. Therefore it takes  $O(n + e)$  time to fill in column and there are  $k$  columns ( $e$  is the number of edges in the graph). Therefore the running time of the algorithm is in  $O((n + e) \cdot k)$