CS 577: Introduction to Algorithms 11/2/2015

Homework 7 Solutions (Review Problems)

Instructor: Dieter van Melkebeek TA: Kevin Kowalski

Problem 1

We need to find the length of a shortest common supersequence of a and b. A trivial common supersequence is the concatenation of a and b. We may obtain a shorter common supersequence c by using a given position in c to cover both a position in a as well as a position in b. We develop a dynamic program to figure out an optimal way.

Consider the last position in c. We can use it to either cover the last position in a, or the last position in b, or both, where the last possibility is only an option when the last symbols in a and b agree. In each case, what remains is to find the length of a shortest common supersequence for the unmatched prefixes of a and b, and to take the minimum of the three possibilities.

Applying this reduction recursively leads to subproblems of the following form: For $0 \leq i \leq n \doteq |a|$ and $0 \leq j \leq m \doteq |b|$, we let $\mathrm{OPT}(i,j)$ denote the length of a shortest common supersequence of $a[1,\ldots,i]$ and $b[1,\ldots,j]$. In those terms the above recurrence reads as follows, for $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$\mathrm{OPT}(i,j) = \begin{cases} \min\left\{\mathrm{OPT}(i-1,j)+1, \mathrm{OPT}(i,j-1)+1, \mathrm{OPT}(i-1,j-1)+1\right\} & \text{if } a_i = b_j \\ \min\left\{\mathrm{OPT}(i-1,j)+1, \mathrm{OPT}(i,j-1)+1\right\} & \text{otherwise,} \end{cases}$$

where $\mathrm{OPT}(i,0) = i$, $\mathrm{OPT}(0,j) = j$, and $\mathrm{OPT}(0,0) = 0$. In fact, if $a_i = b_j$ one can argue that $\mathrm{OPT}(i,j) = \mathrm{OPT}(i-1,j-1)+1$. We leave this observation as an exercise as we do not need it for the solution of the problem. We use the recurrence to compute $\mathrm{OPT}(i,j)$ in the order of increasing values of $i+j$, from $i+j=0$ up to $i+j=n+m$, and we return $\mathrm{OPT}(n,m)$.

There are mn subproblems and each update takes $O(1)$ time, so the time complexity is $O(mn)$.

Alternate solution. The above solution looks very similar to the algorithm from class for sequence alignment. In fact, we can efficiently reduce the given problem to sequence alignment using the following penalty scheme: deleting a symbol has a cost of 1, as does matching positions that agree; the cost of matching positions that do not agree is $n+m+1$. The latter ensures that an optimal solution will never match positions that disagree.

We claim that a shortest sequence that contains a and b as subsequences is also an *optimal alignment* between a and b under our penalty scheme. Given an alignment with cost bounded by $n+m$, one can produce a common supersequence of length equal to the cost of the alignment. Conversely, given a common supersequence c of length ℓ we obtain an alignment of a and b of cost ℓ by identifying a, b in c, and producing a '-' symbol for unmatched positions. For example, for $a = 011$, $b = 101$ and $c = 0101$ we have

c	0	1	0	1
a	**0**	**1**	-	**1**
b	-	**1**	**0**	**1**

Therefore, we can invoke the dynamic programming algorithm for optimal alignment for this problem, and return the result. The time complexity is again $O(mn)$ since the sequence alignment problem from class runs in that amount of time.

Problem 2

Our plan is to write a dynamic program that recursively determines the last multiplication to perform in order to obtain the value a, if one exists at all. If the second operand is a, then the first operand must be c. We don't have to consider b as the second operand because there is no way to right multiply by b and obtain a. If the second operand is c, then there are two possibilities for the first operand: either a or b. We need to consider both possibilities. In general, for all multiplications, we can combine all possible values for both operands. This tells us which values are a possible result of every multiplication. Accordingly, we recursively compute all these possible values for every subexpression.

Each subexpression that we consider is a contiguous portion of the entire expression. We label the symbols as $s_1, s_2, \ldots, s_n$, so the input is $s_1 s_2 \cdots s_n$. We use $\mathrm{VAL}(i,k)$ for $1 \le i \le k \le n$ to denote the set of possible values achievable by the subexpression $s_i s_{i+1} \cdots s_k$. Furthermore, let

$$S_{i,k} = \bigcup_{i \le j < k} \Big\{ \ell r \;\Big|\; \ell \in \mathrm{VAL}(i,j) \text{ and } r \in \mathrm{VAL}(j+1,k) \Big\}.$$

Then we can recursively express $\mathrm{VAL}(i,k)$ as

$$\mathrm{VAL}(i,k) = \begin{cases} \{s_i\} & i = k \\ S_{i,k} & i \neq k. \end{cases}$$

Correctness follows from the above discussion. We can now describe the algorithm. We loop over m from 0 to $n-1$ and i from 1 to $n-m$ and compute the set $\mathrm{VAL}(i, i+m)$ using the above equations. We return the result of the test $a \in \mathrm{VAL}(1,n)$.

There are $O(n^2)$ possibilities for $1 \le i \le k \le n$ and we compute $\mathrm{VAL}(i,k)$ for each one. For different i and k fixed, we compute $S_{i,k}$ via $k-i-1 \le n = O(n)$ unions, each of which can be done in constant time since there are at most a constant (namely three) elements in any of these sets. Therefore, the running time of the algorithm is $O(n^3)$.