1985

A Fast Linear Space Algorithm for Computing Longest Common Subsequences

A. Apostolio

C. Guerra

Report Number:
85-546

A FAST LINEAR SPACE ALGORITHM FOR COMPUTING LONGEST COMMON SBUSEQUENCES

A. Apostolico
C. Guerra

CSD-TR-546
October 1985

A FAST LINEAR SPACE ALGORITHM FOR COMPUTING LONGEST COMMON SUBSEQUENCES

A. APOSTOLICO and C. GUERRA

Department of Computer Science
Purdue University
West Lafayette, IN. 47907

ABSTRACT

Computing only the length of the longest common subsequence(s) (LCS) of two strings of lengths m and $n \geq m$, respectively, can be achieved in space $\Theta(n)$ by most of the algorithms proposed in the past. However, the only known algorithm that also computes an LCS in linear space is one by Hirschberg which never takes less than $\Theta(nm)$ time, i.e., the worst case time lower bound established for the LCS problem when the alphabet is unrestricted and the model of computation consists of the (fairly general) decision tree with $[=, \neq]$ comparisons. The algorithm proposed in this paper requires linear space to compute the LCS, and yet it can be expected to take considerably less than $\Theta(nm)$ time in most practical cases.

Key words and Phrases: Design and analysis of algorithms, Longest common subsequence, Space complexity.

1. PRELIMINARIES

We consider *strings* $\alpha,\beta,\gamma,$ ... of *symbols* on an *alphabet* $\Sigma = \{ \sigma_1,\sigma_2,...,\sigma_s \}$. A string γ is a *subsequence* of α if γ can be obtained from α by deleting a certain number of (not necessarily consecutive) symbols. Let $\alpha = a_1 a_2 ... a_m$ and $\beta = b_1 b_2 ... b_n$, with $m \le n$. The *longest common subsequence* (LCS) *problem* for input strings α and β consists of finding a common subsequence γ of α and β of maximum length.

We denote by $L[i,j]$ $(1 \le i \le m, 1 \le j \le n)$ the length of an LCS between $\alpha_i = a_1 a_2 ... a_i$ and $\beta_j = b_1 b_2 ... b_j$. Fig. 1 below displays the L-matrix for the strings α=*abcdbb* and β=*cbacbaaba*. We use this figure to recall basic concepts and notation. An ordered pair of positions of α and β is a *match* iff $a_i = b_j$. The r entries that correspond to *matches* are encircled in figure 1. Each encircled number is the *rank* of the underlying match. Emboldened circles circumscribe the d *k–dominant* matches for the various values of the rank k. A march $[i,j]$ is k–dominant if it has rank k, and for any other pair $[i',j']$ of rank k, either $i' > i$ and $j' \le j$, or $i' \le i$ and $j' > j$. The boundaries in the figure separate regions with constant L-entry.

		c	b	a	c	b	a	a	b	a
		1	2	3	4	5	6	7	8	9
a	1	0	0	①	1	1	①	①	1	①
b	2	0	①	1	1	②	2	2	②	2
c	3	①	1	1	②	2	2	2	2	2
d	4	1	1	1	2	2	2	2	2	2
b	5	1	②	2	2	③	3	3	③	3
b	6	1	②	2	2	③	3	3	④	4

Figure 1

The L-matrix for the strings α = *abcdbb* and β = *cbacbaaba*.

Let l be the length of an LCS of α and β. It is seen [HI] that computing the k-dominant matches (k=1,2,...,l) is all is needed to solve the LCS problem. In fact, once all k-dominant matches are available in suitable form, then $O(m)$ time suffices to retrieve γ. Most known approaches require $\Theta(n + d)$ space to compute an LCS, although linear space would suffice if one wanted to

compute only the length l of the LCS. By contrast, the dynamic programming implementation presented in [HC] takes never more than $\Theta(n)$ space, though never less than $\Theta(nm)$ time, the worst case lower bound for the LCS problem within the unrestricted-alphabet, $[=,\neq]$-comparison decision tree model of computation [AH]. The remainder of this paper is organized as follows. In the next section, we present the procedure *length*, which computes the length of an LCS on any pair of substrings of α and β. This procedure is adapted from an algorithm for computing the LCS recently introduced in [AG], which has a time bound of $O(ml \cdot \min[\log s, \log m, \log(2n/m)])$. We then show that *length* and its companion procedure *lengthrev* can be cast in a recursive construct, similar to that in [HC], and based on the two recursive procedures *lcs* and *lcsrev*. The overall strategy yields an LCS using only linear space, yet it is subject to a time bound that equals that of *length* up to an additive term $O(m \log m)$. Deleting from Σ all symbols which do not occur both in α and β, does not change the solution(s) to an instance of the LCS problem. Thus we can assume henceforth without loss of generality that $s \leq m$.

2. THE PROCEDURE 'LENGTH'

The procedure *length* is a direct derivation of the algorithm in [AG], except that it is devised to work on an arbitrary substrings $a_{i1},..,a_{i2}$, $b_{j1},...,b_{j2}$ of α and β, and that it does not keep a record of all dominant matches, so that it only yields the length *lsub* of the LCS for that subproblem. The procedure consists of *lsub* stages which identify the *lsub* boundaries of L in succession. It exploits the same criterion as in [HI] to trace a boundary: if $[i,j]$ is a k–dominant match then $[i',j']$ with $i'>i$ is a k–dominant match iff $j'<j$. The procedure is called by passing four parameters to it, namely, $i1, i2, j1$ and $j2$. It returns *lsub* and the the array *RANK* which contains the leftmost k-dominant match, for each $k=1,..,l$. It uses the following auxiliary structures:

– For each symbol of the alphabet σ, a list σ–*OCC* of all the occurrences of σ in β;

– A table *SYMB* defined as follows. $SYMB[j]=k$, if $b_j=\sigma_p$ and j is the k–*th* entry in σ_p–*OCC*. Thus the table *SYMB* enables constant time access to the entry in a σ–*OCC* list which

corresponds to any given position of β.

– An array *PEBBLE* such that *PEBBLE* [*i*] (*i*=1,..,*m*) contains a pointer to an entry of a_i–*OCC*. At the beginning, the procedure expects to find *PEBBLE* [*i*] pointing to the the entry *j* of a_i–*OCC*, which corresponds to the leftmost occurrence of a_i in the interval $[j_1,...,j_2]$. *PEBBLE* [*i*] is then said to be *active*. If the procedure finds that *PEBBLE* [*i*] falls outside the interval [*j*1...*j*2], then it marks this pebble *dead*, if it were not already such. The procedure advances the active pebbles of each row until all of them become *inactive*. A pebble becomes inactive as soon as either the procedure advances it onto an entry of the associated a_i–*OCC* list which is larger than *j*2, or it attempts at advancing the pebble past the last entry of a_i–*OCC*. When the first case applies, the pebble is retracted by one position on the list: thus by the end of the execution of *length* each non-dead pebble points to the rightmost position that it can occupy in the interval [*j*1...*j*2].

The algorithm uses also the function *closest*(σ,b_t) which for any given character σ returns the leftmost occurrence of σ in β which falls past b_t, (∞ , if there is no such occurrence).

```
Procedure length (i1, i2, j1, j2, RANK, lsub)
0 RANK [k] = 0, k=1,2,...,(i2–i1); mark dead pebbles outside [j1...j2];
1  k = 0
2  while there are active pebbles do (start stage k÷1)
3  begin T=j2÷1; k=k÷1;
4     for i = i1–1+k to i2 do (advance pebbles)
      begin
5        t = T;
6        if PEBBLE [i] is active and a_i–OCC [PEBBLE [i]] < T then
         (update threshold, update leftmost k-dominant match )
7              begin  T = a_i–OCC [PEBBLE [i]]; RANK [k]=T end;
         (advance pebble, or make it inactive)
8        PEBBLE [i]=SYMB [closest [a_i,t]];
9        if PEBBLE [i] is active and a_i–OCC [PEBBLE [i]] > j2 then
10           begin PEBBLE [i]=PEBBLE [i]–1; make PEBBLE [i] inactive end;
      end;

  end (lsub = k).
```

The procedure *length* detects all k–dominant matches, as is readily checked. Unlike the algorithm presented in [AG], however, it records only the leftmost k–dominant match incurred for each k. This obtains the linear space bound.

All the elementary steps of *length*, with the exception of the executions of *closest*, take constant time. On an input of size $n+m$ the procedure handles at most m pebbles during each of the l stages. Thus the total time spent by *length* is $O(ml$ + total time required by *closest*). The second term is obviously implementation dependent. One efficient implementation of *closest* is discussed in [AG]. It rests on two auxiliary structures which we now proceed to describe. First, we prepare, in time $\Theta(n)$, the table $CLOSE[1...n+1]$ which is subdivided into consecutive blocks of size s and defined as follows. Letting $p = j \bmod s$ $(j=1,...,n)$, $CLOSE[j]$ contains the leftmost position not smaller than j where σ_p occurs in $\beta\$$, with $\$$ a symbol not in Σ. $CLOSE[n+1]$ is set to $n+1$. Next, we assume that each σ–*OCC* list is assigned a *finger tree* [AG, BT, ME]. Roughly, a finger-tree is a balanced search tree which can be traversed in any direction. The finger is a pointer to any leaf in the tree. The main advantage conveyed to our discussion by the use of finger-trees is in that, with such trees, a search for an item which is displaced d positions (leaves) away from the current position of the finger can be carried out in $O(\log d)$ time. If the finger is updated to point to the last searched item at all times, then searching for m consecutive items in a tree which stores n keys is afforded in $O(\sum_{k=1}^{m} \log d_k)$, where the *intervals* d_k's are subject to the constraint that $\sum_{k=1}^{m} d_k \le 2n$. As is well known, this yields the overall time bound of $O(m \log(2n/m))$. A finger-tree incarnation well fit to the manipulation of integers is the *characteristic tree* described in [AG]. However, more general versions such as originally introduced in [BT] would do just as well. In the following, we will use σ–*OCC* to refer to the list as well as to the finger tree associated with it. Thus we can always speak of the current position of the finger on a given list. We summarize some results in [AG] in form of the following:

Lemma 1 [AG]. For any given p and j ($1 \leq p \leq s, j\,1 \leq j \leq j\,2$), if the finger falls d positions to the right of j, then $closest[\sigma_p, j]$ can be retrieved from $CLOSE[j]$ and from the σ_p–OCC list in time $O(min[\log d, \log s])$.

In the present context, taking full advantage of Lemma 1 requires some simple additions to the procedure *length* which we now describe informally. The procedure resulting from imposing these new specifications on the old procedure will be called still *length*. The discussion of such upgrade leads to the statement of Theorem 1 below, which expresses the time complexity of our new *length*.

First, in order to keep track of the fingers we introduce a new global variable, namely, the array of integers $FINGER[1...m]$. This array is similar to the array *PEBBLE*, and it plays a somewhat dual role. At its inception, the procedure *length* expects to find the fingers pointing to the same locations as the pebbles. We add to this the following specifications.

- As soon as it is invoked, *length* spots the dead pebbles and kills the corresponding fingers. It then moves the non-dead fingers among $FINGER[i\,1], FINGER[i\,1{+}1], ..., FINGER[i\,2]$: each such finger is brought onto the rightmost position in the interval $[j\,1...j\,2]$ that it can occupy on its corresponding σ–*OCC* lists. We will refer to this process as to the *finger positioning*. The positioning of each finger is clearly accomplished in $O(min[\log s, \log(j\,2 - j\,1)])$ time through an application of *closest*. While the pebbles retain their identity on the σ–*OCC* lists, fingers set from different rows on the same σ–*OCC* list merge instead into one single *representative* finger. The initial position occupied by each representative finger (i.e., the largest compatible entry in the associated σ–*OCC* list) is called its *home*. The introduction of representative fingers obtains that, if, during a stage of *length*, the finger associated with some symbol is moved, then this move is implicitly imposed on all other elements of $FINGER[i\,1...i\,2]$ which correspond to the same symbol. We omit the details. This latter process of instituting representative fingers takes time proportional to $(i\,2 - i\,1)$.

- During the execution of each stage of *length*, the (representative) finger associated with each

symbol in $[i1...i2]$ is reconsidered immediately following a *closest* query and the possible consequent update of the pebble (cfr. lines 8-10 of *length*). At that point, we simply set: $FINGER[i] = PEBBLE[i]$. Thus through each individual stage, the finger associated with each symbol moves from right to left. At the end of the stage, the finger is taken back to its home position. Each of the manipulations just described takes constant time.

- Finally, both fingers and pebbles are taken back to their initial (leftmost) position soon after the last stage of *length* has been completed. Overall, this takes time $O(i2-i1)$.

Theorem 1. By the combined use of *FINGER* and *CLOSE*, the procedure *length* computes the length *lsub* of an LCS of $\alpha_{i1}\cdots\alpha_{i2}$ and $\beta_{j1}....\beta_{j2}$ in time $O((i_2-i_1)+lsub\cdot(i2-i1)\cdot min[\log s, \log(2n/(i2-i1))])$ and linear space.

Proof. The linear space bound is trivially achieved. The total time spent by *closest* during the *lsub* stages, can be subjected to an upper bound tighter than the above, namely, $O(lsub\cdot(i2-i1)\cdot\min[\log s, \log(2(j2-j1)/(i2-i1))])$, by an argument in [AG].

We have already established that the calls to *closest* performed at the beginning for finger positioning charge $O((i2-i1)\cdot min[\log s, \log((j2-j1)/(i2-i1))])$. This term is absorbed in the previous one when $lsub \neq 0$. Otherwise the procedure only spends $O((i_2-i_1))$ to find that all pebbles are dead. This concludes our proof. □

3.THE LINEAR SPACE ALGORITHM 'LCS'

Our original goal was to design an algorithm having the same time performance as *length*, yet taking linear space to retrieve also an LCS. The peculiar structure of *length* enables to adopt here profitably a divide and conquer scheme such as in [HC]. Basically, this will consist of applying the procedure *length* recursively to smaller subproblems until we obtain a trivial one.

We need to make a few additional assumptions, namely:

– We stipulate that m is a power of 2.

– We remove the previous assumption according to which, upon calling *length* with j-parameters $j1, j2$, the procedure always finds pebbles and fingers pointing to the leftmost positions in the interval $[j1...j2]$. We replace it with the new assumption that either all pebbles and fingers occupy the rightmost positions in the interval $[j1...j2]$, or else they all occupy the leftmost one. Procedure *length* checks at its inception which case applies, and brings all pebbles to their leftmost positions, if necessary. This does not affect the time bound of the procedure.

Our final algorithm is actually based on the four procedures *length*, *lengthrev*, *lcs* and *lcsrev*. The companion procedure of *length*, *lengthrev*, is simply a replica of *length* just made suitable for processing the mirror image of any subproblem on the input strings. Thus, for instance, calling *lengthrev* with parameters: $1,m,1,n$, has the same effect as letting *length* run on the reverse of the input strings. The mirror procedure *lcsrev* is related to the procedure *lcs*, which is still to be described, in the same way. In conclusion, we only need to list *lcs*.

```
Procedure lcs (1,m,1,n,LCS)
begin
  1 if n=0 or m=1 then determine LCS in constant time
    else (split the problem into subproblems)
        begin
          2 length (1,m/2,1,n,RANK 1,lsub 1);
          3 lengthrev (m/2+1,m,1,n,RANK 2,lsub 2);
          4 j = findmax (RANK 1,RANK 2,lsub 1,lsub 2,lsub);
              (determine the length lsub of the LCS for this subproblem)
          5 lcs (m/2,m,1,j,LCS 1);
          6 lcsrev (m/2+1,m,n-j,n,LCS 2);
          7 combine the two outputs LCS 1 and LCS 2;
        end;
end.
```

The function *findmax* determines the value $j=RANK1[k]$ such that, if $j'=RANK2[k']$ is the smallest entry of $RANK2$ which is larger than j, then $lsub=k+k'$ is a maximum. Thus, the first time *findmax* is executed, it returns $lsub=l$, i.e., the length of any LCS of α and β. Moreover, the match $[i,j]$ with maximum $i \leq m/2$ belongs to an LCS of the two input strings.

The function *findmax* can be straightforwardly implemented in such a way as to require a number of steps proportional to $lsub_1+lsub_2 \leq 2lsub$.

The proof of correctness for *lcs* follows from arguments similar to those developed in [HC].

Theorem 2. The procedure *lcs* finds an LCS in time $O(m \log m + ml \log(min[s, 2n/m]))$ and space $\Theta(n)$.

Proof. We consider all the executions of *length* and *lengthrev* involved at the $k-th$ level of recursion of our strategy, at once. Such executions are relative to consecutive substrings of α of uniform length $m/(2^k)$, and consecutive substrings of β. Starting from the upper-left corner of the L-matrix, each such substring of β is paired up twice with a substring of α, as sketched in Fig. 2.

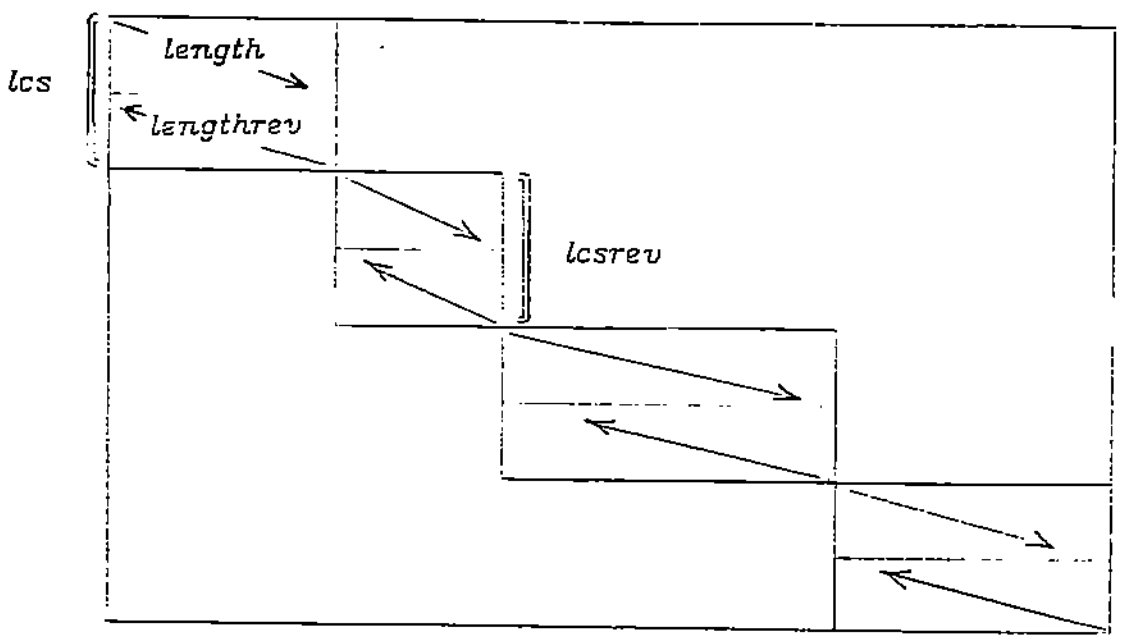

Figure 2

A possible partition of an L-matrix into rectangular subdomains whose associated subproblems have to be solved at the same depth of recursion.

The upper pairing involves an execution of *length*, the second one an execution of *lengthrev*. We define a *block* at level k as the submatrix which is the domain of two such consecutive subproblems.

All the executions of *findmax* at this level charge $O(l)$ time. Adding up for all values 1,2,...,log$m$ of k yields a bound $O(l \cdot \log m)$ for the total work performed by *findmax*.

The execution of each *length* (*lengthrev*) can be bounded in terms of $m/(2^k)+m/(2^k)\cdot l_f \log(\min[s, 2n\cdot 2^k/m])$, where l_f denotes the length of the LCS associated with the generic subproblem. There are 2^k calls at level k, yielding a total time:

$$m+\sum_{f=1}^{2^k}\frac{m}{2^k}l_f \log(min[s,\frac{2n}{m}2^k]),$$

up to a multiplicative constant. Now it is

$$\sum_{f=1}^{2^k} l_f \le 2l.$$

In fact, each l_f cannot be larger than the length of the solution to the corresponding block, and the sum of the 2^{k-1} such lengths cannot exceed l,i.e., the length of the global solution.

Thus we have, in conclusion, that the total work at this level of recursion can be bounded in terms of the quantity:

$$m + l\cdot\frac{m}{2^k}\cdot\log(min[s,\frac{2n}{m}2^k]) \le m + l\cdot\frac{m}{2^k}\cdot\log(min[s\,2^k,\frac{2n}{m}2^k]).$$

The right term can be rewritten as:

$$m + l\cdot\frac{m}{2^k}\log(2^k\cdot\min[s,\frac{2n}{m}]) = m + l\cdot k\cdot\frac{2m}{2^k} + l\cdot\frac{m}{2^k}\log(min[s,\frac{2n}{m}]).$$

Adding up through $k = 1,2,...,\log m$ yields:

$$ml\log m + ml\sum_{k=1}^{\log m}\frac{k}{2^k} + ml\ \log(min[s,\frac{2n}{m}])\sum_{k=1}^{\log m}\frac{1}{2^k}.$$

Since:

$$\sum_{k=1}^{\log m}\frac{1}{2^k} = 2 - \frac{1}{2^{\log m}} < 2,$$

and:

$$\sum_{k=1}^{\log m} \frac{k}{2^k} = (\log m - 1)\frac{1}{(2m)} + \frac{1}{2},$$

then we obtain the claimed $O(m \log m + ml \log(min[s, 2n/m])$ time bound.

The space bound for our strategy follows from an argument similar to that developed in [HC]. □

4. CONCLUDING REMARKS

We have presented an algorithm to retrieve an *LCS* of two input strings in linear space. The worst case time complexity of the algorithm is $\Theta(nm)$. However, our approach is considerably faster in favorable situations , notably, when one or more of the parameters s, m, and l is small compared to the others and to n. The only previous linear space algorithm [HC] known to these authors takes instead time $\Theta(nm)$ in all situations. Our approach exploits search techniques with the use of fingers in two ways, namely, to set up a time efficient procedure for determining the length of the solution to an arbitrary subproblem of the problem, and to randomize, up to an amortizedly negligible logarithmic cost, the initial accesses to the rectangular domains relative to the various subproblems.

Finally, we mention that new nonlinear space algorithms for the LCS problem, which also make use of some of the techniques discussed in this paper, have been introduced quite recently [AG,HD]. In particular, a strategy based on a schedule of primitive operations similar to that in [HS], but with a time bound $O(mlogn + dlog(2nm/d))$ is discussed in [AG]. Interestingly enough, it does not seem that this latter strategy is amenable to implementations that guarantee to occupy always linear space at run time.

REFERENCES

[AG] Apostolico, A. and C. Guerra. The longest common subsequence problem revisited, Tech. rep., Purdue Univ. CS Dept., (1985, submitted for publication).

[AH] Aho, A.D., D.S. Hirschberg and J.D. Ullman. Bounds on the complexity of the maximal common subsequence problem, *JACM* 23, 1, 1-12 (1976).

[BT] Brown, M.R., and R.E. Tarjan. A representation of linear lists with movable fingers. *Proceedings of the* 10–*th STOC*, San Diego, Ca., 19-29 (1978).

[HD] Hsu, W.J., and M.W. Du. New algorithms for the LCS problem, *JCSS* 29, 133-152 (1984).

[HI] Hirschberg, D.S. Algorithms for the longest common subsequence problem, *JACM* 24, 4, 664-675 (1977).

[HC] Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences, *CACM* 18, 6, 341-343 (1975).

[HS] Hunt, J.W., and T.G. Szymanski. A fast algorithm for computing longest common subsequences, CACM 20, 5, 350-353 (1977).

[ME] Mehlhorn, K. Data Structures and Algorithms 1: Sorting and Searching, Springer-Verlag, EATCS Monographs on TCS (1984).