Theoretical Computer Science 92 (1992) 3–17
Elsevier

Fast linear-space computations of longest common subsequences

A. Apostolico*

Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA, and Dipartimento di Matematica Pura ed Applicata, U. of L'Aquila, L'Aquila, Italy

S. Browne

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

C. Guerra**

Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA, and Dipartimento di Matematica, University of Rome I, Rome, Italy

Abstract

Apostolico, A., S. Browne and C. Guerra, Fast linear-space computations of longest common subsequences, Theoretical Computer Science 92 (1992) 3–17.

Space saving techniques in computations of a longest common subsequence (LCS) of two strings are crucial in many applications, notably, in molecular sequence comparisons. For about ten years, however, the only linear-space LCS algorithm known required time quadratic in the length of the input, for all inputs. This paper reviews linear-space LCS computations in connection with two classical paradigms originally designed to take less than quadratic time in favorable circumstances. The objective is to achieve the space reduction without alteration of the asymptotic time complexity of the original algorithm. The first one of the resulting constructions takes time $O(n(m-l))$, and is thus suitable for cases where the LCS is expected to be close to the shortest input string. The second takes time $O(ml\log(\min[s, m, 2n/l]))$ and suits cases where one of the inputs is much shorter than the other. Here m and n $(m \leqslant n)$ are the lengths of the two input strings, l is the length of the longest common subsequences and s is the size of the alphabet. Along the way, a very simple $O(m(m-l))$ time algorithm is also derived for the case of strings of equal length.

1. Introduction

Given a *string* α over an *alphabet* $\Sigma=(\sigma_1, \sigma_2, \ldots, \sigma_s)$, a *subsequence* of α is any string γ that can be obtained from α by deleting zero or more (not necessarily consecutive)

*Work by this author was supported in part by the French Ministry of Education, by the NSF under Grant CCR-89-00305, by NIH Library of Medicine under Grant R01 LM05118, by AFOSR under Grant 90-0107, by the National Research Council of Italy, and by NATO under Grant CRG900293.

**Work by this author was supported in part by the Italian Ministry of Education.

symbols. The *longest common subsequence* (LCS) *problem* for input strings $\alpha = a_1 a_2 \ldots a_m$ and $\beta = b_1 b_2 \ldots b_n$ $(m \leqslant n)$ consists of finding a third string $\gamma = c_1 c_2 \ldots c_l$ such that γ is a subsequence of α and also a subsequence of β, and γ is of maximum possible length. In general, string γ is not unique.

The LCS problem arises in a number of applications spanning from text editing to molecular sequence comparisons, and it has been studied extensively over the past. General lower bounds for the problem are time $\Omega(n \log n)$ or linear time, according to whether the size s of Σ is unbounded or bounded. For unbounded alphabets, any algorithm using only "equal-unequal" comparisons takes $\Omega(nm)$ time in the worst case [1]. The asymptotically fastest general solution takes time $O(n^2 \log\log n/\log n)$ [12]. Time $\Theta(mn)$ is achieved by the following dynamic programming algorithm [7, 17]. Let $L[0 \ldots m, 0 \ldots n]$ be an integer matrix initially filled with zeros. Now execute:

for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do** **if** $a_i = b_j$ **then** $L[i,j] = L[i-1, j-1] + 1$
  else $L[i,j] = \text{Max}\{L[i, j-1], L[i-1, j]\}$.

The above code transforms L in such a way that $L[i,j]$ $(1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n)$ contains the length of an LCS between $\alpha_i = a_1 a_2 \ldots a_i$ and $\beta_j = b_1 b_2 \ldots b_j$. If only the length of γ is desired, then this code is easily adapted to run in linear space. If an LCS is wanted, it becomes necessary to keep a record of the decision made at every step, so that γ can be retrieved at the end through backtracking. The early $\Theta(nm)$ time algorithm in [7] achieves both a linear space bound and the production of an LCS at the outset, through a combination of dynamic programming and divide-and-conquer. Subsequent linear-space algorithms such as in [14, 3] follow the same basic divide-and-conquer scheme as in [7] but require less time than $\Theta(nm)$ for favorable inputs.

Efficient algorithmic design for the LCS problem has experienced a new wave of interest in recent years, especially due to the need to process increasingly numerous and long inputs that arise in molecular sequence comparisons (see, e.g. [11, 16]). The resulting constructions improve on the time performance in cases of special interest, or use only linear space, or do both. For instance, the algorithms in [4] improve on an early algorithm in [8] for the case of strings that differ in length considerably, and improve on the worst-case performance of the strategy in [9]. Another line of research has focused on the efficient handling of the cases where the length of an LCS is expected to be close to the length of the shorter input string. One of the early constructions in [8] achieves time $O((m-l)l \log n)$ for this case. (An additional $\Theta(n \log s)$ term is to be added to all time bounds reported here. Usually, this term is charged by a preprocessing phase.) More recently, an alternate construction requiring $O((m-l)n)$ was proposed in [15], along with another $O((m-l)l \log n)$ algorithm (it is relatively easy to check that the second bound can be reduced to $O(m(m-l)\min\{\log s, \log m, \log 2n/l\})$ by the techniques developed in [3]). Linear-space implementation of the $O((m-l)n)$ algorithm in [15] was subsequently achieved in [10], through a divide-and-conquer scheme that is reminiscent of, but

not identical to that of [7]. An algorithm taking time $O(ne)$ in terms of the quantity $e=m+n-2l$ was proposed in [14]. This algorithm has expected time $O(n+e^2)$ and a nice, though admittedly impractical, $O(n\log n+e^2)$ variation. Also these strategies can be implemented in linear space. However, since $l\leqslant m$, then $e^2=\Theta(n^2)$ for $n\geqslant 2m$. In other words, the bound in [14] is comparable to those in [8, 15, 10] only in the case of strings of nearly equal length.

In this paper, we study additional linear-space algorithms suitable for the case where l is close to m, or m is much smaller than n, or both conditions are met. We start by showing that, for $m=n$, an $O(n(n-l))$ algorithm of great conceptual simplicity results from introducing some kind of dualization in the classic strategy of [9]. Equally simple extensions enable us to handle the case $m\leqslant n$, in time $O(n(m-l))$ and linear space. Finally, we discuss a linear-space implementation of one of the algorithms in [3], preserving the $O(ml\log(min[s, m, 2n/m]))$ time bound of that algorithm.

2. Preliminaries

The ordered pair of *positions* i and j of L, denoted $[i, j]$, is a *match* iff $a_i=b_j=\sigma_t$ for some t, $1\leqslant t\leqslant s$. If $[i, j]$ is a match, and an LCS $\gamma_{i,j}$ of α_i and β_j has length k, then k is the *rank* of $[i, j]$. The match $[i, j]$ is *k-dominant* if it has rank k and for any other pair $[i', j']$ of rank k either $i'>i$ and $j'\leqslant j$ or $i'\leqslant i$ and $j'>j$. Computing the k-dominant matches ($k=1, 2, \ldots, l$) is all that is needed to solve the LCS problem (see, e.g. [3, 8]).

It is useful to define, on the set of matches in L, the following partial order relation: match $[i, j]$ *precedes* match $[i', j']$ if $i\leqslant i'$ and $j\leqslant j'$. Then, the LCS problem translates into the problem of finding a longest chain in the poset of matches. Most known approaches to the LCS problem compute a *minimal antichain decomposition* (refer, e.g. to [5]) for this poset. A set of matches having equal rank is an *antichain* in this decomposition. For general posets, a minimal antichain decomposition is computed by flow techniques [5], although not in time linear in the number of elements of the poset. The main algorithms discussed in this paper have their natural predecessors in [9, 8]. In terms of antichain decompositions, the approach of [8] consists of computing the antichains one at a time, while that of [9] extends partial antichains relative to all ranks already discovered, one step at a time. The interested reader shall find that also the approach in [15], which yields bounds of $O(n(m-l))$ or $O(m(m-l)\log n)$ may fall into this second category.

Our algorithms achieve linear space through a divide-and-conquer scheme similar to that of [10]. The recurrent step of this scheme takes as input: (1) two strings ε and δ such that ε is always a substring, say, of β and δ is always a substring of the other string; (2) the length l of an LCS of ε and δ. The task of the step is to produce an LCS of ε and δ. This is achieved by first computing a suitable *cut* for an LCS of ε and δ and then by applying the same scheme on the two subdomains of the problem induced by the cut. A cut is any pair $[u, v]$ such that an LCS of ε and δ can be formed by

concatenating an LCS of the prefixes ε_u and δ_v with an LCS of the corresponding suffixes of the two strings. A more detailed description of the scheme is as follows.

Procedure *lcs* $(\varepsilon, \delta, i_1, i_2, j_1, j_2, l, LCS)$
begin
  if $l = c$ *or* $\min[|\varepsilon|, |\delta|] - l = c$ *for some constant* c **then**
  determine an LCS in time $O(|\varepsilon||\delta|)$ *and space* $O(\min[|\varepsilon|, |\delta|])$
  else
  begin (split the problem into subproblems)
    choose a cut $[u, v]$, $1 \leqslant u \leqslant |\varepsilon|$, $1 \leqslant v \leqslant |\delta|$
    $lcs(\varepsilon, \delta, i_1, i_1 + u - 1, j_1, j_1 + v - 1, l_1, LCS1)$;
    $lcs(\varepsilon, \delta, i_1 + u, i_2, j_1 + v, j_2, l_2, LCS2)$;
    $LCS = LCS1 \parallel LCS2$;
  end
end.

The major difference between the above scheme and that in [7] is in the fact that here l has to be computed prior to running *lcs*. In the following sections, we present various ways of computing l and correspondingly choose and compute a suitable cut inside *lcs*. Obviously, the overall time performance of the scheme depends crucially on the way that cuts are chosen and computed. As in the algorithm of [10], we want to choose the cuts so as to achieve the best balance, in the sense that the total time required to solve both induced subproblems is about one half of the time required to solve the original problem.

3. The $O(n(n-l))$ procedure length1 for the case $n = m$

In this section, we assume $n = m$ and present a simple $O(n(n-l))$ time strategy which is complementary to that used in [9] (to keep our presentation short, some familiarity with [9] is assumed). The case $n = m$ arises in the row-wise comparison of digitized pictures and thus has special interest. The Hunt–Szymanski approach consists of detecting the dominant matches of all available ranks by processing the matches in the L matrix row by row. For this purpose, a list of thresholds we will call *row-THRESH* is used. After the processing of a row, the kth entry in *row-THRESH* contains the column of the leftmost k-dominant match found so far. For example, for $\alpha = abcdbb$ and $\beta = cbacba$, the L-matrix would be as shown in Fig. 1. After processing the sixth row, the final set of row thresholds would be $\{1, 2, 5\}$. The approach of [9] consists of updating *row-THRESH* row after row, based on the new matches introduced by each row. Note that $m - l = 3$ positions are missing from the final set of thresholds, namely positions 3, 4, and 6. We call each such missing position a gap, and we call the sorted list of gaps *row-COTHRESH*.

Similarly, we can define the list *colu-THRESH* such that the kth entry contains the row number of the rightmost k-dominant match found so far. For the example in

		1	2	3	4	5	6
		c	b	a	c	b	a
1	a	0	0	1	1	1	1
2	b	0	1	1	1	2	2
3	c	1	1	1	2	2	2
4	d	1	1	1	2	2	2
5	b	1	2	2	2	3	3
6	b	1	2	2	2	3	3

Fig. 1. The trace of *row-THRESH* on an L-matrix.

Fig. 1, the final set of column thresholds would be $\{1, 2, 5\}$. The corresponding set *colu-COTHRESH* of gaps would be $\{3, 4, 6\}$. Clearly, the *COTHRESH* lists can be deduced from the *THRESH* lists, and vice versa. If $m-l<l$, then *COTHRESH* lists give a more compact encoding of the final set of thresholds. Unfortunately, this is not always true at any stage of the row-by-row computation, since *THRESH* can be initially more sparse and *COTHRESH* correspondingly denser. However, if we consider only the upper-left square submatrices of the *L*-matrix, then we can obtain a suitable bound on the size of the *COTHRESH* lists.

Lemma 3.1. *The total number of gaps falling within the first i positions of either the ith row or the ith column of the L-matrix cannot be larger than* $m-l$.

Proof. Observe that there must be an equal number of gaps in the ith row and in the ith column. Let q be this number. Then the number of matches contributed to any LCS by the upper left $i \times i$ submatrix of the *L*-matrix cannot exceed $i-q$. Since the remaining portion of the *L*-matrix cannot contribute more than $m-i$ matches, it must be $l \leqslant (m-i)+(i-q)=m-q$. But then $m-l \geqslant q$. □

Lemma 3.1 suggests that the length of an LCS of α and β with $|\alpha|=|\beta|$ can be found by extending, one row and one column at a time, submatrices of the *L*-matrix. This is done by the procedure *length*1 which we now describe. At the ith iteration, the procedure scans from left to right the $O(m-l)$ cells of the two *COTHRESH* lists. If in the *row-COTHRESH* list we find a cell containing position $p<i$ such that $a_i=b_p$, then $[i, p]$ is a dominant match. Continuing the scan, the first cell (if any) is located with an entry larger than $1+p'$, where p' is the value stored in the immediately preceding cell. This jump in the list of gaps represents a threshold, namely, the first threshold to the right of p. If such a cell is found, then for some $i'<i$, $[i', p'+1]$ is a dominant match having the same rank as $[i, p]$. Hence, gap $p'+1$ is inserted into *row-COTHRESH*. If no such cell is found, then $[i, p]$ is the first dominant match found of its rank, and the cell containing i is removed from *colu-COTHRESH*. The processing of the *colu-COTHRESH* list is similar.

Note that we can easily determine the rank of any newly detected dominant match, as follows. Call a position in which a gap does not occur a *line*. Upon beginning the scan of a *COTHRESH* list, initialize r to 1. During the scan, increment r by the number (zero or greater) of lines that are skipped over at each step. Then, when a dominant match is found, it will be of rank r. The highest rank detected is the length of an LCS for the two input strings. Some extra bookkeeping can be added to the process to support the retrieval of an LCS γ at the end. This would, however, havoc the linearity of space. At this stage, we are interested mainly in the computation of $|\gamma|$, and the tedious details involved in such a bookkeeping are omitted. We summarize the preceding discussion in the following claim.

Theorem 3.2. *Given two strings* α *and* β *with* $|\alpha|=|\beta|=n$, *the procedure length*1 *computes the length of an* LCS *of* α *and* β *in time* $O(n(n-l))$ *and linear space.*

4. Computing the length when $n>m$

When $n>m$ the argument supporting Lemma 3.1 does no longer hold. We shall see, however, that the basic technique of the preceding section can still be applied, with small changes. The main tool needed is a procedure that tests, for any integer p in the range $[0, m]$, whether α and β have an LCS of length $m-p$. We describe first such a procedure, which we call *length*2. Later, we show that a procedure *length*3 for computing the length of an LCS of α and β in $O(n(m-l))$ time descends naturally from *length*2.

Procedure *length*2 uses the following simple observation. Suppose strings α and β have LCS length of l. Then there is at least one such LCS, say, γ, that uses only dominant matches. Let $[i,j]$ be one such match. Then, $[i,j]$ appears in the jth *colu-THRESH* list and, implicitly, in the jth *colu-COTHRESH* list. Let f be the number of gaps preceding $[i,j]$ in column j of the L-matrix. Then the prefix of γ that is an LCS for α_i and β_j uses precisely $i-f$ rows among the first i rows of the L-matrix. By an argument similar to that of Lemma 3.1, it must be that $f\leqslant m-l$ since the remaining $m-i$ rows cannot contribute more than $m-i$ matches to γ. In other words, no dominant match in an LCS can be preceded by more than $m-l$ gaps in the cothresh list relative to the column where that match occurs.

In conclusion, to test whether there is a solution of length $m-p$, it is sufficient to produce the n successive updates of the first p entries of *colu-COTHRESH*. By our preceding discussion, this takes time $O(np)$ and linear space. At the end, either we will obtain a match of rank $m-p$ or higher in this list, or we will know that no LCS of length at least $m-p$ exists. We are now ready to present procedure *length*3, which simply consists of running the $O(pn)$ procedure *length*2 with $p=0,1,2,4,8,\ldots$ until it succeeds. Procedure *length*2 will succeed when p is at most $2(m-l)$. Thus the total time spent by *length*3 is proportional to $2n(m-l)+n(m-l)+1/2n(m-l)+\cdots+2n+n+n=4n(m-l)+n$, which is $O(n(m-l))$. This establishes the following claim.

Theorem 4.1. *Procedure length3 computes the length l of γ in $O(n(m-l))$ time and linear space.*

5. The linear-space, $O(n(m-l))$ time algorithm *LCS1*

In this section, we show that *length2* and *length3* (*length1* if $m=n$) can be easily combined with *lcs* to produce an LCS of the two input strings α and β. We call the resulting algorithm *lcs1*. In what follows we describe the structure of *lcs1* and maintain the following bounds.

Theorem 5.1. *Algorithm lcs1 computes an LCS of α and β in time $O(n(m-l))$ and linear space.*

The two issues to be addressed are the computation of l that has to precede the execution of *lcs* and the choice and computation of a cut inside the body of *lcs*. We use *length1* or *length3*, depending on whether $m=n$ or $m<n$, to compute l. From this, we know $p=m-l$. This takes time $O(np)$ and linear space. We now call *lcs* on $\varepsilon=\beta$ and $\delta=\alpha$. Inside *lcs*, we will maintain that the value $w=|\delta|-l$ (i.e. the value of p relative to the current subdomain of the problem) is always known. More precisely, we maintain that at the kth level of recursion, $w\leqslant\lceil p/2^k\rceil$. This is achieved by computing cuts that always divide w in two halves. We call these cuts *balanced cuts*. We will show how the computation of all balanced cuts needed at the kth level of recursion can be carried out in time $O(np/2^k)$ and linear space. Before describing how this is done, we observe that this condition establishes, for the time bound $T(n, p)$ of *lcs*, a recurrence of the form: $T(n, p)=cnp+T(n_1, np/2)+T(n_2, np/2)$, with $n_1+n_2=n$ and c a constant. With initial conditions of the type $T(h, 0)\leqslant bph$, where b is another constant, this recurrence has solution $O(np)$.

Let n and $m\leqslant n$ be the lengths of ε and δ, respectively, and let $l=m-p$ be the length of an LCS for the two strings. The following lemma will be used to find a balanced cut for ε and δ (see Fig. 2).

Lemma 5.2. *Assume $m>p\geqslant 2$ and let $p=p_1+p_2+p_3$ with $p_1\neq 0$, $p_2=0$, and $p_3\neq 0$. Then, there is an LCS $\gamma=\gamma^1\gamma^2\gamma^3$ of ε and δ for which it is possible to write $\varepsilon=\varepsilon^1\varepsilon^2\varepsilon^3$ and $\delta=\delta^1 d\delta^2 d'\delta^3$ with d and d' symbols of Σ, in such a way that: (1) γ consists only of dominant matches; (2) for $i=1, 2, 3$, γ^i is an LCS of ε^i and δ^i and $|\delta^i|-|\gamma^i|=p_i$; (3) let e and e' be, respectively, the last symbol of ε^1 and the first symbol of ε^3, then e and d do not form a dominant match in L and $e'\neq d'$.*

Proof. In the L-matrix, consider in succession the columns relative to the positions of δ. We start with a counter initialized to zero and update it according to the following. Consider column 1. As is easy to check, if there is any match in column 1, then the one such match occupying the row of lowest index is also the unique dominant match in

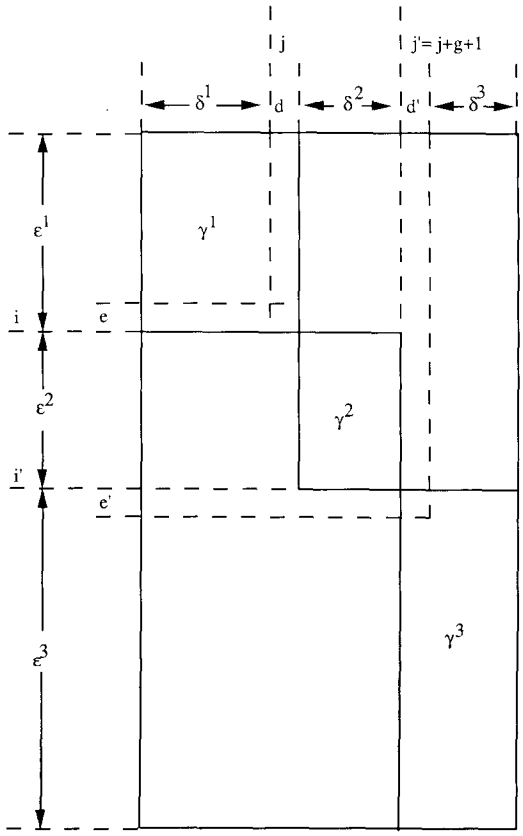

Fig. 2. Illustrating Lemma 5.2.

column 1. If there is a solution γ that uses a match in this column, then we pick the only dominant match in this column and initialize with it a string γ'. If this is not the case, we increment the counter by one. Assume we have handled all columns up to $h-1$ updating the counter or extending the prefix γ' of an optimal solution γ, according to the cases met. Considering column h, we increment the counter if and only if no match in that column could be used to extend the length of γ' by one unit in such a way that the extended string would still be the prefix of an optimal solution. If some such matches exist, we append to γ' the one such match contained in the row of smallest possible index (observe that the match thus selected is a dominant match). In conclusion, each column at which the counter is not incremented extends the subsequence γ' by one new dominant match, while the fact that the counter is incremented at some column h signals that γ' could not have been continued into an optimal solution γ had we picked a match in column h.

Let now j be the leftmost column at which the counter reaches the value p_1, and let i be the row containing the last one among the matches appended to γ'. We claim that

entry $[i,j]$ cannot be a dominant match. In fact, if $[i,j]$ is a match, then clearly its rank is at least $|\gamma'|$. Assuming the rank of $[i,j]$ higher than $|\gamma'|$ leads to a contradiction. In fact, in this case we can find a string η such that $\eta\gamma''$ is an LCS of ε and δ, $\gamma=\gamma'\gamma''$ is also an LCS of ε and δ and yet $|\eta\gamma|>|\gamma|$. Thus, either $[i,j]$ is not a match or it is a nondominant match of rank equal to the last match of γ' used so far. We set δ^1 equal to the prefix of δ of length $j-1$, ε^1 equal to the prefix of ε of length i, $\gamma^1=\gamma'$, $e=\varepsilon[i]$ and $d=\delta[j]$. These choices are consistent with the properties listed in the lemma for the objects involved.

To continue with the columns of L that fall past column j, we distinguish two cases, according to whether or not γ' can be extended with a match in column $j+1$. If γ' can be extended with a match in column $j+1$, let $j+1, j+2, \ldots, j+g$ be the longest run of consecutive columns such that each column contributes a new match to γ'. By the hypothesis $p_1<p$, we have $j+g<m$ (i.e. we must be forced to skip at least one more column). Let i' be the row such that $[i', j+g]$ is a match of γ'. Then, by our choice of g the entry $[i'+1, j+g+1]$ cannot be a match. We set ε^2 equal to the substring of ε that starts at position $i+1$ and ends at position i', δ^2 equal to the substring of δ that starts at $j+1$ and ends at $j+g$, and $e'=\varepsilon[i'+1]$ and $d'=\delta[j+g+1]$. Finally, we take the suffix of length g of γ' as γ^2. Clearly, these assignments satisfy the conditions in the claim. The choices performed so far induce a unique choice of ε^3, δ^3, and γ^3. By our construction of γ', there is an optimal solution γ which has $\gamma'=\gamma^1\gamma^2$ as a prefix. In any such solution, γ' must be followed by an LCS of ε^3 and δ^3 of length $|\delta^3|-(p-p_1-p_2)$, i.e. of length $|\delta^3|-p_3$. Thus the remaining conditions of the claim are also met. If γ' cannot be extended with a match in column $j+1$, then the claim still holds by simply taking δ_2 and γ_2 both empty. □

With $p_1=\lceil p/2 \rceil$ Lemma 5.2 can be used in the computation of a balanced cut for ε and δ, as follows. We treat the case where p is even, the case of odd p being quite similar. Let j and $j'=j+g+1$ be the positions in δ of d and d', respectively, and let i be the position in ε of the last symbol of ε^1. Clearly, $[i', j'-1]$ is a balanced cut. Observe that this cut coincides with $[i,j]$ if γ^2 is empty.

We now run *length2* on the ordered pair (δ,ε) and with parameter $p/2+1$. We use this run to prepare an array *REACH* with the property that *REACH* $[i]$ contains the column index relative to the $(p/2+1)$th gap in the *COTHRESH* list at row i. Observe that, by condition 3 of the lemma, if $i'+1$ is the position in ε of the first symbol of ε_3, then *REACH* $[i'+1]$ equals precisely the position j' of d' in δ.

Next, we run a copy of *length2* on the ordered pair $(\delta^R, \varepsilon^R)$ of the *reverse* strings of the two input strings, this time with parameter $p/2$. An array *REVREACH* similar to *REACH* is built in this way. Since $[i'+1, j]$ is not a match and we know that $|\delta_3|-|\gamma_3|=p/2$, then $REVREACH[i'+1]=j'$.

Clearly, any index i^* for which $REACH[i^*]=REVREACH[i^*]$ yields a corresponding balanced cut $[i^*-1, REACH[i^*]-1]$. By Lemma 5.2 and the above discussion, at least one such index is guaranteed to exist. In conclusion, we only need to scan the two arrays *REACH* and *REVREACH* looking for the first index k such

that $REACH[k]=REVREACH[k]$. Having found such an index, we can set, for our balanced cut $[u,v]$, $u=k-1$ and $v=REACH[k]-1=REVREACH[k]-1$.

As already mentioned, the case of odd p is dealt with similarly. At the top level of the recursion, this process takes $O(np)$ time and linear space. Since the parameter p is halved at each level, the overall time taken by the computation of cuts is still $O(np)$. The recursion can stop whenever the current partition of L has an associated value of either the l or p not larger than some preassigned constant. For any such partition, an LCS can be found by known methods in linear space.

6. The procedure *length*4

In this section, we study a procedure *length*4 that computes the length of an LCS of α and β in time $O(lm\log(\min[s,m,2n/m]))$. Since symbols not appearing in α cannot contribute to an LCS, we can eliminate such symbols from β and assume henceforth $s\leqslant m$, which eliminates the $\log m$ from the bound. The procedure *length*4 is a direct derivation of an algorithm in [3], which in turn follows a paradigm in [8]. For the subsequent developments, we need to describe *length*4 in some detail. The procedure consists of *lsub* stages which identify the *lsub* antichains of L in succession. It exploits the same criterion as in [8] to trace an antichain: if $[i,j]$ is a k-dominant match then $[i',j']$ with $i'>i$ is a k-dominant match iff $j'<j$. At stage k only the leftmost k-dominant match is recorded in the array $RANK$. The procedure uses the following auxiliary structures:

- For each symbol of the alphabet σ, a list σ-OCC of all the occurrences of σ in β;
- An array $PEBBLE$ such that $PEBBLE[i]$ $(i=1,\ldots,m)$ contains a pointer to an entry of a_i-OCC. At the beginning, $PEBBLE[i]$ $(i=i_1,\ldots,i_2)$ points to the entry j of a_i-OCC, which corresponds to the leftmost occurrence of a_i in the interval $[j_1,\ldots,j_2]$, if any. $PEBBLE[i]$ is then said to be *active*. The procedure advances an active pebble until it becomes *inactive*, i.e. reaches an entry larger than j_2, or the last entry of a_i-OCC. By the end of the execution of *length*4 each pebble is set to point to the rightmost position that it can occupy in the interval $[j_1,\ldots,j_2]$.

The algorithm uses also the function $closest(\sigma,t)$ which for any given character σ returns the pointer to the entry in the σ-OCC list corresponding to the leftmost occurrence of σ in β which falls past b_t.

Procedure *length*4 $(i_1,i_2,j_1,j_2,\ RANK,\ lsub)$

```
0 RANK[k]=0, k=1, 2,...,(i2−i1);
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] = closest[a_i, t];
9       if PEBBLE[i] is active and a_i-OCC[PEBBLE[i]] > j_2 then
10        begin PEBBLE[i] = PEBBLE[i] - 1; make PEBBLE[i] inactive end;
      end;
  end (lsub = k).
```

The procedure *length*4 detects all dominant matches [3]. Unlike the algorithm presented in [3], however, it records only the leftmost dominant match incurred for each k. This achieves the linear space bound.

All the elementary steps of *length*4, 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 *lsub* stages. Thus the total time spent by *length*4 is $O(mlsub + \text{total time required by } closest)$. The second term is obviously implementation dependent. One efficient implementation of *closest* is discussed in [3]. It rests on two auxiliary structures which we now proceed to describe. First, we prepare, in time $\Theta(n)$, the table $CLOSE[1 \ldots n+1]$ which is subdivided into consecutive blocks of size s and defined as follows. Letting $p = j \bmod s$ $(j = 1, \ldots, n)$, $CLOSE[j]$ contains the leftmost position not smaller than j where σ_p occurs in β. Such a table enables us to implement *closest* in time $O(\log s)$. By definition, if $p = j \bmod s$ then $CLOSE[j] = closest[\sigma_p, j]$ and thus constant time suffices in this case. Otherwise, let $j' = (j \operatorname{div} s)s + p$, where *div* stands for the integer division operation. Two cases are possible: $j' < j$ or $j' > j$. Assume $j' < j$. If $CLOSE[j'] > j$, then clearly $CLOSE[j'] = closest[\sigma_p, j]$. Otherwise, $closest[\sigma_p, j]$ is not smaller than $CLOSE[j']$ but not larger than $CLOSE[j'+s]$. Now, there cannot be more than s entries in σ_p-OCC list between the two entries $CLOSE[j']$ and $CLOSE[j'+s]$. Thus $closest[\sigma_p, j]$ can be retrieved in $\log s$ steps by performing a binary search in this segment of the σ_p-OCC list. Similar considerations apply to the case $j' > j$.

Next, we assume that each σ-OCC list is assigned a *finger tree* [3, 2, 6, 13]. 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 by finger-trees is that, in such a tree, the search for an item 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 $\sum_{k=1}^{m} d_k \leqslant 2n$. This sum is maximum when all intervals are equal, which yields the overall time bound of $O(m \log(2n/m))$.

In order to keep track of the fingers we institute a new global variable, namely, the array of integers $FINGER[1 \ldots m]$. At its inception, the procedure *length*4 moves all the fingers $FINGER[i_1], FINGER[i_1+1], \ldots, FINGER[i_2]$, originally coincident with the pebbles, onto the rightmost position in the interval $[j_1 \ldots j_2]$ that they can occupy on their corresponding σ-*OCC* lists. This positioning of each finger is accomplished in $O(\min[\log s, \log(j_2 - j_1)])$ time through an application of *closest*. Fingers set from different rows on the same σ-*OCC* list merge into one single *representative* finger.

During the execution of each stage of *length*4, the (representative) finger associated with each symbol in $[i_1 \ldots i_2]$ is reconsidered immediately following a *closest* query and the possible consequent update of the pebble (cf. lines 8–10 of *length*4). 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. Each of the manipulations just described takes constant time. Finally, both fingers and pebbles are taken back to their initial (leftmost) position immediately after the last stage of *length*4 has been completed. Overall, this takes time $O(i_2 - i_1)$. We summarize some results in [3] in the form of the following theorem.

Theorem 6.1. *By the combined use of* FINGER and CLOSE, *the procedure length*4 *computes the length lsub of an LCS of* $\alpha_{i_1} \ldots \alpha_{i_2}$ *and* $\beta_{j_1} \ldots \beta_{j_2}$ *in time* $O(lsub \cdot (i_2 - i_1) \cdot \min[\log s, \log(2n/(i_2 - i_1))])$ *and linear space.*

7. The linear-space algorithm *LCS*2

We now show that the procedure *length*4 can be cast in the divide-and-conquer scheme of Section 2 to produce an algorithm *lcs*2 that has time bound $O(ml \log(\min[s, 2n/l]))$ and space $\Theta(n)$. For $l = \Theta(m)$ (i.e. in applications that use this algorithm fruitfully), this time bound is equal to that of the algorithm in [3].

We remove the previous assumption according to which, upon calling *length*4 with j-parameters j_1, j_2, the procedure always finds pebbles and fingers pointing to the leftmost positions in the interval $[j_1 \ldots j_2]$. We replace it with the new assumption that either all pebbles and fingers occupy the rightmost positions in the interval $[j_1 \ldots j_2]$, or else they all occupy the leftmost one. Procedure *length*4 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. Algorithm *lcs*1 uses *length*4 both to compute l prior to executing *lcs* and to compute cuts inside the body of *lcs*. For this latter task we use a scheme similar to that of *lcs*1. We outline the method for the case of even l, the case of odd l being handled similarly. We run two copies of *length*4, on the two mirror images of the problem, with the proviso that computation in each row is stopped as soon as a dominant match of rank $l/2$ is detected. All matches of rank $l/2$ so detected by each version of the procedure are stored in one of two associated lists. Observe that the number of such matches cannot

exceed the total number of dominant matches detected, and this latter number cannot be larger than ml, the number of matches handled at most by the procedure. At the end, we scan the two lists looking for the first pair of matches, one from one list and one from the other, that form a chain. From the positions in L of these two matches, we can infer a balanced cut. In the present context, a cut is balanced if it identifies two submatrices L' and L'' of L with the property that an optimal solution γ can be formed by concatenating two optimal solutions γ' and γ'' entirely contained, respectively, in L' and L'' and both of length $l/2$. Leaving the details for an exercise, we concentrate on the following claim.

Theorem 7.1. *The procedure lcs2 finds an LCS in time* $O(ml\log(\min[s, 2n/l]))$ *and space* $\Theta(n)$.

Proof. Each execution of *length*4 at the kth level of the recursion can be bounded in terms of $m_f \cdot l/2^k \log(\min[s, 2n/m_f])$, where m_f denotes the number of rows assigned to the fth subproblem. By the preceding discussion, the time needed to scan each pair of antichains of maximum rank in order to find a balanced cut for that pair can be absorbed in this bound. There are 2^k calls at level k, yielding a total time:

$$\sum_{f=1}^{2^k} m_f \frac{1}{2^k} \log\left(\min\left[s, \frac{2n}{m_f}\right]\right),$$

up to a multiplicative constant. Now it is

$$\sum_{f=1}^{2^k} m_f = m.$$

Since $m_f \geqslant l/2^k$, we have that the total work at this level of recursion can be bounded in terms of the quantity:

$$m \cdot \frac{l}{2^k} \cdot \log\left(\min\left[s, \frac{2n}{l} 2^k\right]\right) \leqslant m \cdot \frac{l}{2^k} \cdot \log\left(\min\left[s2^k, \frac{2n}{l} 2^k\right]\right).$$

The right term can be rewritten as:

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

Adding up through $k = 1, 2, \ldots, \log l$ yields:

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

from which we obtain the $O(ml\log(\min[s, 2n/l])$ time bound. □

8. Conclusion

We have considered linear-space implementations of LCS algorithms that are faster than quadratic in favorable cases. Our focus was kept on implementations that would preserve the time complexity of the original algorithms. As noted, *lcs*2 is based on a classical paradigm established in [8] and preserves the time performance of the corresponding upgrade in [4]. Algorithm *lcs*2 is essentially similar to an earlier algorithm of [3], which appears to be the first linear-space algorithm produced with a time bound better than $\Theta(nm)$. While certainly an offspring of the paradigm of [9], the final algorithm *lcs*1 bears comparatively a smaller resemblance to it. However, algorithm *lcs*1 shows that the $O(n(m-l))$ performance in [15, 10] may descend somewhat naturally also from the paradigm of [9]. Also, our initial steps towards the design of *lcs*1 have exposed a very simple algorithm the asymptotic worst-case performance of which matches that of the algorithm in [14] for strings of equal lengths. The time complexity of the original algorithm in [9] is $O(r \log n)$, where r is the total number of matching pairs of symbols between α and β. We leave it as an exercise to derive a linear-space implementation of the algorithm in [9] based on the template procedure *lcs* of Section 2. A known upgrade of the Hunt–Szymanski algorithm [4] takes time $O(m \log n + d \log(2mn/d))$, where d is the total number of dominant matches. It is an interesting question whether this upgrade can be implemented in linear space without substantial denaturation of the formula expressing its time complexity.

References

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

[2] A. Apostolico, Improving the worst-case performance of the Hunt–Szymanski strategy for the longest common subsequence of two strings, *Inform. Process. Lett.* **23** (1986) 63–69.

[3] A. Apostolico and C. Guerra, A fast linear-space algorithm for computing longest common subsequences, in: *Proc. 23rd Allerton Conf. Monticello*, IL (1985).

[4] A. Apostolico and C. Guerra, The longest common subsequence problem revisited, *Algorithmica* **2** (1987) 315–336.

[5] K.P. Bogart, *Introductory Combinatorics* (Pitman, London, 1983).

[6] M.R. Brown and R.E. Tarjan, A representation of linear lists with movable fingers, in: *Proc. 10th STOC*, San Diego, CA (1978) 19–29.

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

[8] D.S. Hirschberg, Algorithms for the longest common subsequence problem, *J. ACM* **24**(4) (1977) 664–675.

[9] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM* **20** (5) (1977) 350–353.

[10] S.K. Kumar and C.P. Rangan, A linear-space algorithm for the LCS problem, *Acta Inform.* **24** (1987) 353–362.

[11] H.M. Martinez, ed., Mathematical and computational problems in the analysis of molecular sequences, *Bulletin of Mathematical Biology* (Special Issue honoring M.O. Dayhoff) **46**(4) (1984).

[12] W.J. Masek and M.S. Paterson, A faster algorithm for computing string editing distances, *J. Comput. System Sci.* **20** (1980) 18–31.

[13] K. Mehlhorn, *Data Structures and Algorithms* 1: *Sorting and Searching*, EATCS Monographs on TCS (Springer, Berlin, 1984).

[14] E.W. Myers, An O(ND) difference algorithm and its variations, *Algorithmica* **1** (1986) 251–266.

[15] N. Nakatsu, Y. Kambayashi and S. Yajima, A longest common subsequence algorithm suitable for similar text strings, *Acta Informatica* **18** (1982) 171–179.

[16] D. Sankoff and J.B. Kruskal (eds.), *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparisons* (Addison-Wesley, Reading, MA, 1983).

[17] R.A. Wagner and M.J. Fischer, The string to string correction problem, *J. ACM* **21** (1) (1974) 168–173.