Research Contribution

A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabet Size

FRANCIS Y. L. CHIN*[1] and C. K. POON*

Given two strings of lengths m and $n \geq m$ on an alphabet of size s, the longest common subsequence (LCS) problem is to determine the longest subsequence that can be obtained by deleting zero or more symbols from either string. The first $O(mn)$ algorithm was given by Hirschberg in 1975. The algorithm was later revised to $O(ln)$, where l is the length of an LCS between the two strings. Another strategy given by Hunt and Szymanski takes $O(rlogn)$ time, where $r \leq mn$ is the total number of matches between the two strings. Apostolico and Guerra combined the two approaches and derived an $O(mlogn+dlog(mn/d))$ algorithm, where $d \leq r$ is the number of dominant matches (minimal candidates) between the two strings. Efficient algorithms for two similar strings were devised by Nakatsu et al. [7] and Myers [6] with time complexities of $O(n(m-l))$ and $O(n(n-l))$, respectively. This paper presents a new algorithm for this problem, which requires preprocessing that is nearly standard for the LCS problem and has time and space complexity of $O(ns+min\{ds, lm\})$ and $O(ns+d)$, respectively. This algorithm is particularly efficient when s (the alphabet size) is small. Different data structures are used to obtain variations of the basic algorithm that require different time and space complexities.

1. Introduction

Let $\alpha=a_1a_2 \ldots a_m$ and $\beta=b_1b_2 \ldots b_n$ be two strings on an alphabet $\Sigma=\{\sigma_1, \sigma_2, \ldots, \sigma_s\}$ of size s with $m \leq n$. A *subsequence* of α can be obtained by deleting zero or a number of (not necessarily consecutive) symbols from α. We say that γ is a *common subsequence* of α and β iff γ is a subsequence of both α and β. The *longest common subsequence (LCS) problem* is to find a common subsequence γ of α and β of maximal length.

Define the *L-matrix* for the two strings α and β as an integer array $L[0 \ldots m, 0 \ldots n]$ such that $L[i, j]$ is the length of an LCS for $\alpha[1 \ldots i]$ and $\beta[1 \ldots j]$. Since $\alpha[1 \ldots 0]$ and $\beta[1 \ldots 0]$ are two empty strings, $L[i, 0]=L[0, j]=L[0, 0]=0$. It was proved by Hirschberg [3] that for $1 \leq i \leq m$ and $1 \leq j \leq n$,

$$L[i,j]=\begin{cases} L[i-1,j-1]+1 & \text{if } a_i=b_j \\ \max\{L[i-1,j], L[i,j-1]\} & \text{otherwise.} \end{cases}$$

With this property, he applied a dynamic programming strategy to derive an $O(mn)$ algorithm that solves the problem by filling L row by row. Since L is nondecreasing in both arguments, we can draw contours [4] on L to separate regions of different values. Furthermore, the set of contours can be completely specified by their corner points, which are called *dominants*. As an illustration, Figure 1 shows an example of the L-matrix for the strings $\alpha=cbbdac$ and $\beta=abcddba$, where the circled entries are the dominants. It is easy to see that there are l contours, where l is the length of an LCS, and an LCS can be obtained by finding the set of dominants, instead of filling all the $m \times n$ entries in L.

	a	b	c	d	d	b	a
c	0	0	(1)	1	1	1	1
b	0	(1)	1	1	1	(2)	2
b	0	1	1	1	1	2	2
d	0	1	1	(2)	2	2	2
a	(1)	1	1	2	2	2	(3)
c	1	1	(2)	2	2	2	3

Fig. 1 The contents of an L-matrix when $\alpha=cbbdac$ (vertical) and $\beta=abcddba$ (horizontal). Row 0 and column 0 of L are omitted.

Although quadratic time complexity was shown to be necessary for this problem for sufficiently large s [1], various improvements to the simple $O(mn)$ algorithm concentrate on efficient generation of the dominants. One of the earliest variations was by Hirschberg [4],

*Department of Computer Science, University of Hong Kong, Hong Kong.

[1]This research is supported in part by ONR grant N00014-87-K-0833 while the author is visiting at the Department of Computer Science, University of Texas at Dallas, Richardson, Texas 75083.

whose algorithm repeatedly scans α and generates all the dominants of each contour after each scan. Since each scan takes n steps (in order to look for matching symbols in β) and there are l contours, the total time complexity of the algorithm is no more than $O(nlogs+ln)$ where *nlogs* is the preprocessing time. Hunt and Szymanski [5] attempted to find the dominants in L row by row. The matches associated with each row are considered one by one in succession from right to left, and the dominants are determined by binary search. As each match has to be considered once, the total time complexity is bounded by $O(nlogs+rlogn)$, where r is the total number of matches between the two strings. With the use of a proper data structure, Apostolico and Guerra [2] improved the time complexity to $O(nlogs+mlogn+dlog(mn/d))$ on the basis of the above ideas, where $d \le r$ is the total number of dominants between the two strings.

Our algorithm follows the same line of reasoning on the efficient generation of the dominants in L. The dominants associated with the contours are generated in order of increasing value. But instead of scanning α for the generation of the dominants in each contour, as in [4], the dominants in each contour are generated from those with a lower value. Since each dominant can generate at most s dominants of the next contour, no more than $O(ds)$ time will be required for this stage. With some careful analysis, it can be shown that our algorithm takes no more than $O(ns+min\{ds, lm\})$ time and $O(ns+d)$ space, where $O(ns)$ is the preprocessing time.

The next section gives the properties of L and the necessary concepts for our algorithm. Section 3 presents the algorithm, and its analysis follows in Section 4. Section 5 studies different variations of the basic idea and Section 6 concludes the paper by discussing some possible improvements.

2. Fundamental Concepts and Definitions

Denote the ordered pair of position i and j of L by $[i, j]$. $[i, j]$ is called a *match* iff $a_i = b_j$. Define a rectangular region in L, ${}^{ij}R_{kl}$, as the set of elements, $\{[p, q] \mid i < p \le k \text{ and } j < q \le l\}$. $[i, j]$ is a *dominant* or *k-dominant* iff $L[i, j] = k$ and $L[p, q] < k$ for all other elements $[p, q]$ in ${}^{00}R_{ij}$. Since $[i, j]$ is a dominant only if $[i, j]$ is a match, $r \ge d$.

Define D^k as the set of *k-dominants*, that is all the dominants on the contour with value k. Thus the set of dominants is composed of l disjoint subsets $D^1, D^2, \ldots, D^l$, where l is the length of an LCS.

Next we define an ordering for the elements in D^k. Given two k-dominants, $[i, j]$, $[u, v]$ in D^k, $[u, v]$ is said to be *lower* than $[i, j]$ or $[i, j]$ is *higher* than $[u, v]$ iff $u > i$ or, equivalently, iff $v < j$ [3]. $[u, v]$ is the *lower neighbor* of $[i, j]$ or $[i, j]$ is the *higher neighbor* of $[u, v]$ iff $[u, v]$ is the highest k-dominant lower than $[i, j]$. Informally, a dominant is higher than another one if it appears higher in the picture of L-matrix and is also to the right of the lower one. For the example in Fig. 1, the set of 2-dominants is $D^2 = \{[2, 6], [4, 4], [6, 3]\}$ with the order that $[2, 6]$ is higher than $[4, 4]$ and $[4, 4]$ is higher than $[6, 3]$.

Let $[i, j]$ be a k-dominant in D^k and $[u, v]$ be its *lower neighbor*; define $[u, v] = [m, 0]$ if $[i, j]$ is the lowest element in D^k. Define D_{ij} as the set of all the $(k+1)$-dominants in ${}^{i0}R_{un}$, that is, the set of $(k+1)$-dominants in the horizontal strip of L bounded (inclusively) by row $i+1$ and u. Consider the example in Fig. 1. The set of 1-dominants, $D^1 = \{[1, 3], [2, 2], [5, 1]\}$ has three elements. So D^2 is divided into three subsets, namely $D_{13} = \{[2, 6]\}$, $D_2 = \{[4, 4]\}$ and $D_{51} = \{[6, 3]\}$. Note that there should be no 2-dominant higher than row 2. In general, we have the following result[s].

Theorem 1: $D^{k+1} = \cup\{D_{ij} \mid [i, j] \in D^k\}$

Proof: If $[i, j]$ is the highest k-dominant, no $(k+1)$-dominant would lie on or above row i. The elements of D^k cut the rest of the rows into $|D^k|$ horizontal strips and any $(k+1)$-dominant must fall on one such strip. ⊹

Thus to find D^{k+1}, we can find D_{ij} for all $[i, j] \in D^k$, and then form the union. Now let us consider some useful properties of D_{ij}.

Assume $[i, j]$ is a k-dominant and $[u, v]$ is its lower neighbor. Suppose $[p, q]$ is in D_{ij}. (For convenience, these assumptions will hold for the rest of this section.)

Lemma 1: $[p, q]$ must lie on the right of $[i, j]$, that is $q > j$.

Proof: By contradiction, if $q \le j$, there would be a k-dominant $[w, x]$ on the left of $[i, j]$ and above $[u, v]$, that is, $x < j$ and $w < u$. If $w \le i$, $[i, j]$ would not be a k-dominant. If $w > i$, $[u, v]$ would not be the lower neighbor of $[i, j]$. ⊹

Theorem 2: If $[p, q]$ is a match of symbol σ, then $[p, q] = [p_\sigma, q_\sigma]$, where p_σ, q_σ are the positions of the first σ in $\alpha[i+1 \ldots m]$ and $\beta[j+1 \ldots n]$, respectively.

Proof: We shall prove the theorem by contradiction. From the definition of D_{ij} and Lemma 1, $p > i$ and $q > j$. So if $[p, q] \ne [p_\sigma, q_\sigma]$, then either $p_\sigma < p$ or $q_\sigma < q$. In other words, $[p_\sigma, q_\sigma]$ lies in the region ${}^{ij}R_{pq} \setminus \{[p, q]\}$, that is, the region ${}^{ij}R_{pq}$ except $[p, q]$. But we can form a common subsequence of length $k+1$ for $\alpha[1 \ldots p_\sigma]$ and $\beta[1 \ldots q_\sigma]$ by appending σ to the LCS (of length k) of $\alpha[1 \ldots i]$ and $\beta[1 \ldots j]$. Hence $L[p_\sigma, q_\sigma] \ge k+1$, contradicting the supposition that $[p, q]$ is a $(k+1)$-dominant. ⊹

A corollary of Theorem 2 is that D_{ij} is a subset of $\{[p_\sigma, q_\sigma] \mid \sigma \in \Sigma\}$, to be denoted as S_{ij}. Hence, D_{ij} has at most s elements. As an example (Fig. 2), let $[i, j] = [2, 2]$ be a 1-dominant in D^1. Then we have $[p_a, q_a] = [5, 7]$, $[p_b, q_b] = [3, 6]$, $[p_c, q_c] = [6, 3]$ and $[p_d, q_d] = [4, 4]$. Of these, only $[4, 4]$ is a 2-dominant in D_2. Moreover, D_{22} contains no other elements.

The following theorem gives the necessary and sufficient conditions for an element, $[p_\sigma, q_\sigma]$, of S_{ij} to be

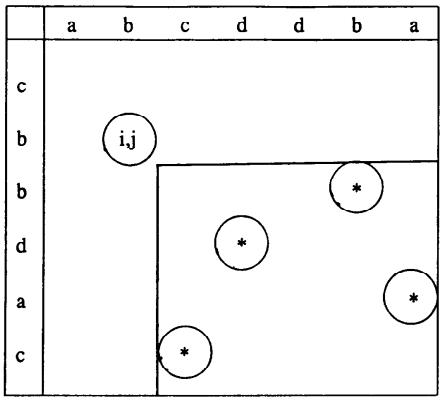

Fig. 2 The L-matrix for the example in Fig. 1. $[i, j]$ is a 1-dominant and the entries with '*' are elements of S_{ij}. $[p_\sigma, q_\sigma] = [4, 4]$ is the only element in D_{ij}.

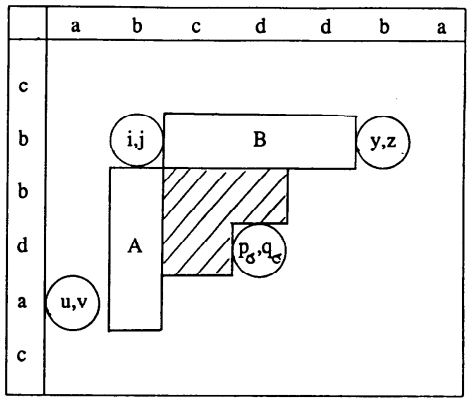

Fig. 3 A diagram to illustrate Theorem 3. $[i, j]$, $[u, v]$, $[p_\sigma, q_\sigma]$ are defined as above. $[y, z]$ is the lowest $(k+1)$-dominant above row p_σ. The L-matrix has the value k in the vertical and horizontal strips (regions A and B respectively). The shaped region is ${}^{ij}R_{p_\sigma, q_\sigma} \setminus \{[p_\sigma, q_\sigma]\}$.

in D_{ij} (Fig. 3). Readers are encouraged to verify that in S_{22}, only [4, 4] satisfies these conditions.

Theorem 3: A match $[p_\sigma, q_\sigma]$ in S_{ij} is a $(k+1)$-dominant in D_{ij} iff $p_\sigma \le u$ and $q_\sigma < z$, where $[y, z]$ is the lowest $(k+1)$-dominant above row p_σ and $z = n+1$ if there is no $(k+1)$-dominant above row q_σ.

Proof:

(Necessity) it should be easy to see that both $p_\sigma \le u$ and $q_\sigma < z$ are necessary for $[p_\sigma, q_\sigma]$ to be in D_{ij}.

(Sufficiency) Firstly, we note that $L[r, s] = k$ for all $[r, s]$ in the 1-column vertical strip: row $i+1$ to u in column j (region A in Fig. 3), and the 1-row horizontal strip: column $j+1$ to $z-1$ in row i (region B in Fig. 3). Secondly, if $q_\sigma < z$, there will be no match in ${}^{ij}R_{p_\sigma - 1, q_\sigma}$. Obviously, the 1-row horizontal strip: column $j+1$ to $q_\sigma - 1$ in row p_σ, would not contain any match, from the definition of p_σ and q_σ. Hence if $p_\sigma \le u$, $L[r, s] = k$ for all $[r, s]$ in ${}^{ij}R_{p_\sigma, q_\sigma}$ except that $L[p_\sigma, q_\sigma] = L[p_\sigma - 1, q_\sigma - 1] + 1 = k + 1$. ☼

3. The Basic Algorithm

Our algorithm will generate all the dominants in the order D^1 and then $D^2, \ldots, D^L$. To determine D^{k+1}, we find D_{ij} for all $[i, j]$ in D^k and then form the union. The procedure *EXPAND* shown below will be used to generate D_{ij}. It accepts four parameters, i, j, u and z, where $[u, v]$ is the lower neighbor of $[i, j]$ and $[y, z]$ is the lowest $(k+1)$-dominant in ${}^{00}R_{in}$, that is, on or above row i. In fact, $[y, z]$ is also the lowest $(k+1)$-dominant above the highest element in S_{ij}. After execution, *EXPAND* returns a list, *OUTPUT-LIST* of all the elements in D_{ij} sorted in row number.

Procedure *EXPAND* (i, j, u, z)
1. **while** *there exists* $[p_\sigma, q_\sigma]$ *in* S_{ij} *with* $p_\sigma \le u$ **do**
2. *get the highest* $[p_\sigma, q_\sigma]$
3. **if** $q_\sigma < z$ **then**
4. *append* $[p_\sigma, q_\sigma]$ *to OUTPUT-LIST*
5. $z := q_\sigma$
end-if
6. *remove* $[p_\sigma, q_\sigma]$ *from* S_{ij}
end-while
7. **return** (*OUTPUT-LIST*)

Observe that the algorithm enumerates $[p_\sigma, q_\sigma]$ in increasing order of row number. Then z will always be the column number of the lowest $(k+1)$-dominant above the next possible candidate in S_{ij}. Hence the correctness of algorithm *EXPAND* follows directly from Theorem 3.

However, we have implicitly assumed the knowledge of S_{ij} for any position $[i, j]$, and moreover, a data structure to enumerate S_{ij} in increasing order of row number efficiently. To realize this, we provide two tables: α-*CLOSEST*$'[1 \ldots s, 0 \ldots m]$ and β-*CLOSEST*$[\sigma_1 \ldots \sigma_s, 0 \ldots n]$. β-*CLOSEST*$[\sigma, j]$ specifies the position of the first σ in $\beta[j+1 \ldots n+1]$. (For convenience, we append to the ends of α and β a joker symbol, '\$', that matches all symbols. Therefore, if σ does not exist in $\beta[j+1 \ldots n]$, β-*CLOSEST*$[\sigma, j] = n+1$.) α-*CLOSEST*$'$ is slightly different from β-*CLOSEST* and α-*CLOSEST*$'[x, i]$ is set to the value p_x s.t. a_{p_x} is the x^{th} different symbol when $\alpha[i+1 \ldots m+1]$ is scanned from front to end and p_x is the position of its first occurrence. Note that $1 \le x \le s$ and $p_1 < p_2 < \ldots < p_s$. So α-*CLOSEST*$'[1 \ldots s, i]$ and β-*CLOSEST*$[\sigma_1 \ldots \sigma_s, j]$ store the row and column numbers of the $[p_\sigma, q_\sigma]$'s in S_{ij}, respectively. Moreover, the p_σ's are arranged in ascending order. With these two tables, we can easily get from α-*CLOSEST*$'$ the smallest p_σ of an unconsidered symbol σ in $\alpha[i+1 \ldots m+1]$ and look up q_σ in β-*CLOSEST*.

As an example, referring to Fig. 2, we have

α-*CLOSEST*$'[1, 2] = 3 \quad \alpha[3] = b$

β-*CLOSEST*[b, 2]=6

α-*CLOSEST'*[2, 2]=4 α[4]=d

β-*CLOSEST*[d, 2]=4

α-*CLOSEST'*[3, 2]=5 α[5]=a

β-*CLOSEST*[a, 2]=7

α-*CLOSEST'*[4, 2]=6 α[6]=c

β-*CLOSEST*[c, 2]=3

Hence, the elements of S_{22} (arranged in increasing row number) are [3, 6], [4, 4], [5, 7], and [6, 3].

To call *EXPAND*, we must supply u and z besides [i, j]. u can be retrieved easily if the elements of D^k are stored in a linked list sorted in increasing row number. To know z we have to 'expand' the higher neighbor of [i, j] before 'expanding' [i, j]. This order of expansion is also favored by the linked list data structure for D^k. Note that every element of D_{wx} is higher than every element of D_{ij} if [w, x] is the higher neighbor of [i, j]. Thus, after the call *EXPAND*(w, x, i, z'), the updated value of z' is the correct value of z for the next call *EXPAND*(i, j, u, z). Hence, *EXPAND* should also return the updated z.

Finally, to facilitate the retrieval of an LCS after finding all the dominants, we have to keep a *PARENT* pointer for every record of the dominant [p, q]. It should point to the record of [i, j] if [p, q] is in D_{ij}.

The modified procedure *EXPAND.MOD* and the main algorithm *FIND_LCS* are shown below.

Procedure *EXPAND.MOD* (i, j, p, u, z)
Remark: EXPAND.MOD returns D_{ij} in OUTPUT-LIST L, which is a linked list of dominant records, and the updated z. A dominant record has the format: ([i, j], p) where [i, j] is a dominant and p is a PARENT pointer. For the input parameters, pointer p points to the record of [i, j], [u, v] is the lower neighbor of [i, j], and [y, z] is the lowest (k+1)-dominant in $^{00}R_{in}$.

```
1.  x:=1
2.  while α-CLOSEST'[x, i]≤u do
3.    r:=α-CLOSEST'[x, i]; c:=β-CLOSEST[a_r, j]
4.    if c<z then
5.      append ([r, c], p) to L
6.      z:=c
        end-if
7.    x:=x+1
      end-while
8.  return(L, z)
```

Algorithm *FIND_LCS*(α, β)
Remark: FIND_LCS outputs the LCS of α abd β. α-CLOSEST' and β-CLOSEST are defined above. D[0 . . m] is an array of lists and each list D[k] is a list of dominant records, each storing an element of D^k. The records of each list will be stored in increasing row number order.

```
1.1  for 1≤x≤s do β-CLOSEST[σ_x, n]:=n+1
1.2  for j:=n-1 downto 0 do begin
1.3    for 1≤x≤s do β-CLOSEST[σ_x, j]:=β-
       CLOSEST[σ_x, j+1]
1.4    β-CLOSEST[b_{j+1}, j]:=j+1
       end-for

2.1  for 1≤x≤s do α-CLOSEST'[x, m]:=m+1
2.2  for i:=m-1 downto 0 do begin
2.3    α-CLOSEST[1, i]:=i+1
2.4    find the min. y s.t. a_{i+1}=a_i, where α-
       CLOSEST'[y, i+1]=i'
2.5    for 2≤x≤y do α-CLOSEST'[x, i]:=α-
       CLOSEST'[x-1, i+1]
2.6    for y≤x≤s do α-CLOSEST'[x, i]:=α-
       CLOSEST'[x, i+1]
       end-for

3.1  put [0, 0] into D[0]
3.2  k:=0
3.3  while D[k] not empty do
3.4    COL_max:=n+1; D[k+1]:=empty
3.5    repeat
3.6      get the highest not expanded [i, j] in D[k],
           with p pointing to the record of [i, j]
3.7      if [i, j] has lower neighbor [u, v] then
3.8        ROW_max:=u
3.9      else ROW_max:=m
3.10     (L, COL_max):=EXPAND.MOD(i, j, p,
         ROW_max, COL_max)
3.11     append L to D[k+1]
3.12     until all elements of D[k] expanded
3.13   k:=k+1
       end-while

4.1  Pick an element [p, q] in D[k-1]
       {k-1=l=length of LCS}
4.2  while k-1>0 do
4.3    output([p, q])
4.4    set [p, q] to parent of [p, q] by following the
       PARENT pointer
4.5    k:=k-1
       end-while
```

Steps 1 and 2 are the preprocessing steps that build the β-*CLOSEST* and α-*CLOSEST'* tables. Step 3 is the main body that computes $D^1, D^2, \ldots, D^l$. Since higher dominants of D^k will be expanded before the lower ones by the algorithm *FIND_LCS* and the procedure *EXPAND.MOD* generates the elements of D_{ij} also in increasing row number, the elements of D^{k+1} will be generated and hence stored in $D[k+1]$ in the same order. Step 4 recovers an LCS in reverse order. This step is relatively simple, as each dominant in D^k has a parent pointer pointing to a dominant in D^{k-1}.

4. Analysis of Algorithm

Steps 1 and 2 constitute the preprocessing, which takes time $O((n+m)s)$. In Step 3, the outer while-loop

(line 3.3) repeats for l times and the inner repeat-loop (line 3.5) loops for $|D^k|$ times. Therefore, there should be $|D^1|+|D^2|+\ldots+|D^l|=d$ calls to *EXPAND.MOD*. The procedure *EXPAND.MOD* requires at most $O(s)$ time to find D_{ij}. Thus Step 3 requires at most $O(ds)$ time. On the other hand, $D^{k+1}=\cup\{D_{ij}|[i,j]\in D^k\}$. In constructing D_{ij} from S_{ij}, *EXPAND.MOD* considers only $|S_{ij}\cap^{i0}R_{un}|$ elements. Thus determining D^{k+1} requires $\Sigma|S_{ij}\cap^{i0}R_{un}|=O(m)$ time. Consequently, Step 3 takes no more than $O(lm)$ time. Thus the total work for Steps 1, 2, and 3 has a time complexity of $O((n+m)s+min\{ds, lm\})$.

The required space includes tables α-*CLOSEST'* and β-*CLOSEST*, together with the linked lists $D[k]$ of the dominants. They require a total of $O((m+n)s+d)$ storage.

Let us consider the following example with $\alpha=0^n1^n0^n$, $\beta=1^n0^n1^n$ and $\Sigma=\{0, 1\}$. Obviously, γ, the LCS of α and β, is either 0^n1^n or 1^n0^n. Hirschberg's approach [3] takes $O(n^2)$ time, whereas that of Hunt and Szymanski [5] takes $O(n^2 \log n)$, that of Apostolico and Guerra [2] $O(n \log n)$, and those of Myers [6] and Nakatsu, Kambayashi and Yajima [7] $O(n^2)$, as α and β are not similar strings. However, our approach takes no more than $O(n)$. Note that our approach will perform best when s is small, but in some cases our approach may not be as good as others.

5. Variations of the Basic Algorithm

As shown in Section 3, we provide α-*CLOSEST'* and β-*CLOSEST* for easy enumeration of S_{ij} in the main program. This preprocessing requires $O(n+m)s)$ time and storage. With the help of these tables, expanding a dominant requires $O(\min\{s, u-i\})$ time. (For convenience, we let $[i, j]$ be a k-dominant, $[u, v]$ its lower neighbor, $[w, x]$ its higher neighbor, and $[y, z]$ the lowest $(k+1)$-dominant on/above row i. These assumptions will hold throughout this section.) In this section, we study two variations of the basic algorithm. In the first, we reduce the preprocessing time to $O((m+n)$ *logs*) and storage to $O(m+n)$ by using a persistent data structure [8]. This is a significant reduction, and the penalty is just a slight increase of the dominant expansion time to $O(min\{s, u-i+x-j\})$. In the second variation, we reduce the time of dominant expansion to $O(logs)$. This requires the construction of a special table, 1-*DOM*, using $O((s!)^2s)$ time and $O(s^{2s})$ space. In addition, we need α-, β-*CLOSEST* and α-, β-*INDEX*, which requires $O((m+n)s)$ time and space. However, the table 1-*DOM* is independent of the input strings. Therefore the method is suitable for applications in which the LCS of many pairs of strings is required and the alphabet size is small. In this situation, 1-*DOM* is calculated just once.

Variation I

A persistent tree allows *insertion* and *deletion*, as in many ordinary dynamic data structures such as AVL trees and red-black trees [9]. However, a persistent tree differs from these other trees in its ability to access information from the past. More precisely, only the following operations of the persistent tree are considered for our application:

access(x, t): Find the node in the tree at time t with the smallest key greater than or equal to x.
insert(x, t): At time t, insert the node with key x into the tree.
delete(x, t): At time t, delete the node with key x from the tree.

Moreover, the time specified in any update (*insert*/*delete*) operation should not be earlier than that of any operation already performed. For example, after the two operations *insert*$(x, 1)$ and *delete*$(x, 3)$, *insert*$(y, 2)$ is not allowed; *access*$(x, 2)$ will return the node with key x; *access*$(x, 4)$ will return '*key not found*'. Sarnak and Tarjan [8] have described several techniques for converting some ordinary dynamic data structures into persistent ones. Thus there are persistent AVL trees, persistent red-black trees, and so on.

Now consider α-*CLOSEST'*. Suppose we use a red-black tree to store the s elements in α-*CLOSEST'*$[1 \ldots s, i+1]$. We can still enumerate them in ascending order. If we then insert $i+1$ and delete i' (s.t. $a_{i+1}=a_{i'}$ and i' is the position of the first occurrence of the symbol a_{i+1} in $\alpha[i+2 \ldots m]$), the new tree will contain the elements in α-*CLOSEST'*$[1 \ldots s, i]$. However, we must not forget the old tree because α-*CLOSEST'*$[1 \ldots s, i+1]$ may be needed to expand a dominant in the main program. Therefore we can use a persistent red-black tree, α-*TREE*, to replace α-*CLOSEST'*. Although the usage of β-*CLOSEST* is slightly different from that of α-*CLOSEST'*, we also replace it by a persistent tree, β-*TREE*.

To construct α-*TREE*, we start with an empty tree at time $t=-m$. At time $t=-i$ $(0\le i\le m-1)$, we perform *insert*$(i+1, -i)$ and *delete*$(i', -i)$ on α-*TREE* so that it contains the same elements as in α-*CLOSEST'*$[1 \ldots s, i]$. Note that we need an auxillary table $P[\sigma_1 \ldots \sigma_s]$ so that at time just before $-i$, $P[\sigma_i]$=position of the first σ_i in $\alpha[i+2 \ldots m]$. In fact it contains the same elements as in α-*CLOSEST'*$[1 \ldots s, i+1]$ but in a different order. This table enables us to find i' by looking at $P[a_{i+1}]$. After the two updates on α-*TREE*, we set $P[a_{i+1}]$ to $i+1$ so that $P[\sigma_i]$=position of the first σ_i in $\alpha[i+1 \ldots m]$. The construction of β-*TREE* is similar. From Sarnak and Tarjan [8], updating a persistent red-black tree requires $O(logs)$ time, where s is the size of the tree at that moment. The space required is $O(m)$ when there are m updates. Therefore the constructions of α- and β-*TREE* require $O((m+n)$ *logs*$)$ time and $O(m+n)$ space, since there are $2(m+n)$ updates in total.

To expand a k-dominant $[i, j]$, we first retrieve all the q_σ's from β-*TREE* at time $t=-j$ s.t. $q_\sigma\le x$, and write them into an auxillary array $Q[\sigma_1 \ldots \sigma_s]$. ($Q$ is initializ-

ed to $n+1$ at all entries before expanding the highest k-dominant for every k. Thus $Q[\sigma_1 \ldots \sigma_s] = \beta$-*CLOSEST*$[\sigma_1 \ldots \sigma_s, n]$.) From Sarnak and Tarjan [8], if the updates are taken at regular time intervals (as in our situation), accessing a persistent red-black tree takes the same time as the non-persistent counterpart. Therefore retrieving the smallest r elements from a tree with s nodes requires $O(max\{logs, r\})$ time if an in-order traversal is done. In our situation, $r = min\{s, x-j\}$. If $x-j < O(logs)$, searching β-*TREE* will take $O(logs)$ time, but searching $\beta[j+1 \ldots x]$ will just take $O(x-j)$ time. We will therefore search β instead of β-*TREE*. Consequently, retrieving the q_σ's requires $O(min\{s, x-j\})$ time. After that, $Q[\sigma_1 \ldots \sigma_s] = \beta$-*CLOSEST*$[\sigma_1 \ldots \sigma_s, j]$. Then we retrieve, in ascending order, the p_σ's from α-*TREE* at time $t = -i$ s.t. $p_\sigma \le u$ (or from $\alpha(i+1 \ldots u]$ directly if $u-i < O(logs)$). This requires $O(min\{s, u-i\})$ time. For each p_σ retrieved, we immediately check $Q[\sigma]$ to see if $Q[\sigma] < z'$, where $[y', z']$ is the lowest $(k+1)$-dominant higher than row p_σ. If the test succeeds, $[p_\sigma, Q[\sigma]]$ is a $(k+1)$-dominant, else it is not. Note that next time we expand $[u, v]$, we retrieve the q_σ's between $v+1$ and j and overwrite Q. The resulting array will be equal to β-*CLOSEST*$'[\sigma_1 \ldots \sigma_s, v]$. Hence the time to expand a dominant is $O(min\{s, u-i+x-j\})$. By an analysis similar to that for the basic algorithm, the total time and space for this variation are $O((n+m)\,logs + min\{ds, l(n+m)\})$ and $O(n+m+d)$, respectively. The procedure *EXPAND*.1 that expands a dominant is shown below.

Procedure *EXPAND*.1 (i, j, p, u, z, x)

Remark: EXPAND.1 has the same input and output parameters as EXPAND.MOD, except x where [w, x] is the higher neighbor of [i, j]. However, it uses α- and β-TREE instead of α-CLOSEST$'$ and β-CLOSEST. Moreover it requires an auxillary array $Q[\sigma_1 \ldots \sigma_s]$ that is initialized to $n+1$ at all entries before expanding the highest k-dominant for any k.

1. **while** *there exists q_σ at time $-j$ with $q_\sigma \le x$* **do**
2. *get q_σ from β-TREE*
   (or from β directly if $x-j < O(logs)$)
3. $Q[\sigma] := q_\sigma$
   end-while
4. **while** *there exists p_σ at time $-i$ with $p_\sigma \le u$* **do**
5. *get the smallest p_σ from α-TREE*
   (or from α directly if $u-i < O(logs)$)
6. **if** $Q[\sigma] < z$ **then**
7. *append* $([p_\sigma, Q[\sigma]], p)$ *to* L
8. $z := Q[\sigma]$
   end-if
   end-while
9. **return** (L, z)

Variation II

In Section 3, Theorem 3 gives the condition for an element, $[p_\sigma, q_\sigma]$, of S_{ij} to be in D_{ij}: $p_\sigma \le u$ and $q_\sigma < z'$ where $[y', z']$ is the lowest $(k+1)$-dominant higher than row p_σ. We can however, break down the conditions into two parts:

($R1$) $p_\sigma \le u$ and $q_\sigma < z$; and

($R2$) no other element, $[p'_\sigma, q'_\sigma]$, in S_{ij} s.t. $p'_\sigma \le p_\sigma$ and $q'_\sigma \le q_\sigma$.

Observe that the relative positions between the elements in S_{ij} are sufficient to determine the subset that satisfies ($R2$). Thus S_{ij} has just $(s!)^2$ different cases with respect to ($R2$). Hence we can calculate the subset satisfying ($R2$) for all possible cases of S_{ij} and store them in a table, 1-*DOM*. When expanding a dominant, we get back the subset from 1-*DOM* and eliminate those elements violating ($R1$). The remaining subset will then be the required D_{ij}. If s is very small with respect to n, there will be many S_{ij}'s that have the same relative ordering of elements, in which case the construction of 1-*DOM* is worthwhile. Thus from now on, we will assume that $slogs < logn$.

To simplify the indexing, we define 1-*DOM*$[u \ldots v, u \ldots v]$ (where $u = \sigma_1 \ldots \sigma_1$, $v = \sigma_\sigma \ldots \sigma_\sigma$) as an $s^s \times s^s$ table although only $(s!)^2$ entries will be used. for any two permutations α', β' of the s symbols in the alphabet, 1-*DOM*$[\alpha', \beta']$ is defined as the set $\{\sigma \mid \sigma = \alpha'[i] = \beta'[j]$ and there are no other matches in ${}^{00}R_{ij}\}$. Note that this is just the set of 1-dominants for α' and β' (hence the name 1-*DOM*). We store the set as a list of s symbols arranged in increasing order of row number and pack it into one memory word (assumed to have $logn$ bits). The construction of an entry f 1-*DOM* takes $O(s)$ time, and the work is similar to the dominant expansion in the basic algorithm. Hence constructing 1-*DOM* requires $O((s!)^2 s)$ time and $O(s^{2s})$ space. Both are less than $O(n^2)$ as $slogs < logn$. Note that unused entries such as 1-*DOM*$[u, u]$ need not be calculated.

Besides 1-*DOM*, we need α-*INDEX*$[0 \ldots m]$ and β-*INDEX*$[0 \ldots n]$ for indexing 1-*DOM* when expanding a dominant. α-*INDEX*$[i]$ is defined as a string of length s such that its k^{th} symbol is the k^{th} different symbol met when scanning $\alpha[i+1 \ldots m]$ from front to end. β-*INDEX* is similarly defined. The construction is similar to that of α-*CLOSEST*$'$ in the basic algorithm. Hence the two tables require $O((m+n)s)$ time and $O(m+n)$ space (since we can pack a string of length s into one word). We also need α-*CLOSEST*$[\sigma_1 \ldots \sigma_s, 0 \ldots m]$ and β-*CLOSEST*$[\sigma_1 \ldots \sigma_s, 0 \ldots n]$ to eliminate those elements violating ($R1$). β-*CLOSEST* is as defined in the basic algorithm and α-*CLOSEST* is similar to β-*CLOSEST*. Hence they require $O((m+n)s)$ time and space.

When expanding a dominant $[i, j]$, we retrieve from 1-*DOM*$[\alpha$-*INDEX*$[i]$, β-*INDEX*$[j]]$ the subset of S_{ij} that satisfy {$R2$). Since the elements of the subset are ordered in increasing row number (and thus also in decreasing column number), we can do binary searches on the subset to select those elements satisfying ($R1$). We first look at the $(s/2)^{th}$ symbol, σ (say), of the subset and check whether β-*CLOSEST*$[\sigma, j] < z'$. If the test

succeeds, we check the $(3s/4)^{th}$; otherwise, we check the $(s/4)^{th}$, and so on. Thus in $O(logs)$ time, we can determine the border line that separates those symbols that are on the left of column z' from those that are not. Another $O(logs)$ time will determine those on/above row u.

Finally, we append each element in the remaining subset, D_{ij}, to the list $D[k]$. This requires a constant time for each element. We can therefore charge the work onto the elements of D_{ij}, rather than on the expansion of $[i, j]$. Hence the expansion of a dominant requires $O(logs)$ time. Consequently, the total time and space for this variation are $O((s!)^2 s+(m+n)s+dlogs)$ and $O(s^{2s}+(m+n)s+d)$, respectively.

6. Conclusion

The LCS problem has been studied by a number of researchers and its complexity has been improved in different respects. We have a solution that is efficient when the alphabet size for the strings is small. Using the same idea, we derive two variations of our algorithm with different time and space complexities for different data structures. However, it is still not certain whether better-than-quadratic-time uniform algorithms for this problem exist when s is fixed, even though a nonuniform linear algorithm exists for this problem when $s=2$.

Acknowledgment

The authors wish to thank Dr. M. Y. Chan for reading the early drafts of this paper.

References

1. AHO, A. V., HIRSCHBERG, D. S. and ULLMAN, J. D. Bounds on the complexity of the maximal common subsequence problem, *J. ACM*,. **23** (1976), 1-12.
2. APOSTOLICO, A. and GUERRA, C. The longest common subsequence problem revisited, *Algorithmica* (1987) 2: 315-336.
3. HIRSCHBERG, D. S. A linear space algorithm for computing maximal common subsequences, *Comm. ACM*, **18** (1975), 341-343.
4. HIRSCHBERG, D. S. Algorithms for the longest common subsequence problem, *J. ACM*, **24** (1977), 664-675.
5. HUNT, J. W. and SZYMANSKI, T. G. A fast algorithm for computing longest common subsequences, *Comm. ACM*, **20** (1977), 350-353.
6. MYERS, E. W. An O(ND) difference algorithm and its variations, *Algorithmica* (1986) 1:251-266.
7. NAKATSU, N., KAMBAYASHI, Y. and YAJIMA, S. A longest common subsequence algorithm suitable for similar text strings, *Acta Informatica*, **18** (1982), 171-179.
8. SARNAK, N. and TARJAN, R. E. Planar point location using persistent search trees, *Comm. ACM*, **29** (1986), 669-679.
9. TARJAN, R. E. *Data structures and network algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.

(Received March 5, 1990; revised July 2, 1990)