Two Algorithms for the Longest Common Subsequence of Three (or More) Strings

Robert W. Irving and Campbell B. Fraser*

Computing Science Department,
University of Glasgow,
Glasgow,
Scotland.

Abstract. Various algorithms have been proposed, over the years, for the longest common subsequence problem on 2 strings (2-LCS), many of these improving, at least for some cases, on the classical dynamic programming approach. However, relatively little attention has been paid in the literature to the k-LCS problem for $k > 2$, a problem that has interesting applications in areas such as the multiple alignment of sequences in molecular biology.
In this paper, we describe and analyse two algorithms with particular reference to the 3-LCS problem, though each algorithm can be extended to solve the k-LCS problem for general k. The first algorithm, which can be viewed as a "lazy" version of dynamic programming, has time and space complexity that is $O(n(n-l)^2)$ for 3 strings, and $O(kn(n-l)^{k-1})$ for k strings, where n is the common length of the strings and l is the length of an LCS. The second algorithm, which involves evaluating entries in a "threshold" table in diagonal order, has time and space complexity that is $O(l(n-l)^2 + sn)$ for 3 strings, and $O(kl(n-l)^{k-1} + ksn)$ for k strings, where s is the alphabet size. For simplicity, the algorithms are presented for equal-length strings, though extension to unequal-length strings is straightforward.
Empirical evidence is presented to show that both algorithms show significant improvement on the basic dynamic programming approach, and on an earlier algorithm proposed by Hsu and Du, particularly, as would be expected, in the case where l is relatively large, with the balance of evidence being heavily in favour of the threshold approach.

Key words: string algorithms, longest common subsequence.

1 Introduction

The longest common subsequence problem for 2 strings (2-LCS), and the related, more general, minimum edit-distance problem, have been extensively studied, and various algorithms for these problems have been proposed — see, for example, [14], [4], [5], [7], [9], [11], [13], [10], [1], [3], [15]. However relatively little attention has been paid to the general k-LCS problem for k (> 2) strings, although there are

* Supported by a postgraduate research studentship from the Science and Engineering Research Council

some important areas of application, particularly in multiple sequence alignment in molecular biology.

The well-known basic dynamic programming scheme for the 2-LCS problem ([4], [12], [14]) was explicitly extended to the k strings case by Itoga [8]. The time and space requirements of the resulting algorithm are $O(k|S_1|.|S_2|\ldots|S_k|)$, where $|S_i|$ is the length of the ith string, or $O(kn^k)$ if all strings have length n. (In what follows, we generally suppose, for simplicity, that all strings have length n, though all of the results can be extended in obvious ways to the case of unequal-length strings.)

In the case of the 2-LCS problem, many variations of and alternatives to the basic dynamic programming algorithm have been proposed — see the citations above. Masek and Paterson's "Four Russians" approach [9] achieves a worst-case complexity that is $O\left(n^2 \frac{\log\log n}{\log n}\right)$, but is of theoretical rather than practical interest. None of the other proposed algorithms gives an improvement on the $O(n^2)$ performance of dynamic programming in the *worst* case, but generally their complexities can be expressed in terms of other parameters, such as the length of an LCS, and this leads to significant improvements in practice in many circumstances.

Hsu and Du [6] addressed explicitly the k-LCS problem, and proposed a general algorithm that involves systematic enumeration in a so-called *CS-tree* — a tree containing an explicit representation of all the common subsequences of the set of strings. By suitably pruning the tree to avoid repeating earlier computations, Hsu and Du's algorithm requires $O(ksn+ksr)$ time and $O(ksn+r)$ space, where s is the alphabet size and r is the number of tuples $(i_1, i_2, \ldots, i_k)$ such that $S_1[i_1] = S_2[i_2] = \cdots = S_k[i_k]$. This algorithm will clearly be at its most effective when r is relatively small. However, when the alphabet size is small, as will be the case in at least some areas of application, we can expect r to be relatively large, with a detrimental effect on the efficiency of the algorithm. Hsu and Du presented no empirical evidence in support of their algorithm.

In this note, we describe and analyse, with particular reference to the 3-LCS problem, two new algorithms, which can be seen as building on the approaches to the 2-LCS problem of Ukkonen [13], Myers [10] and Wu et al [15] in the one case, and of Hunt and Szymanski [7] and Nakatsu et al [11] in the other case, in the context of the 2-LCS problem. Each of these algorithms can be extended in fairly obvious ways to deal with k strings for any fixed $k > 3$. The first algorithm, which we refer to as the "lazy" algorithm, uses $O(n(n-l)^2)$ time and space in the case $k = 3$, and $O(kn(n-l)^{k-1})$ time and space for general k. The second algorithm, which we refer to as the "diagonal threshold" approach, uses $O(l(n-l)^2 + sn)$ time and space in the case $k = 3$, and $O(kl(n-l)^{k-1} + ksn)$ time and space for general k. Our empirical evidence for the case $k = 3$ suggests that, as would be expected, the new algorithms improve substantially on basic dynamic programming and on the Hsu and Du algorithm when l is relatively large, and the dynamic threshold method is also very effective when l is very small. However, this empirical work also indicates strongly that, in most circumstances when $k \geq 3$, the major constraint determining which LCS instances can and which cannot be solved in practice, at least using current methods, is likely to be space rather than time.

The remainder of the paper is structured as follows. Section 2 contains a description and analysis of the lazy algorithm, section 3 a description and analysis of

the diagonal threshold algorithm, and section 4 reports the empirical findings and conclusions.

2 The "Lazy" Approach to Dynamic Programming

Now consider the problem of finding an LCS of 3 strings A, B and C, each of length n. We shall assume equal length strings throughout, for simplicity, but all of our results can be extended in a straightforward way to cases where the strings are of different lengths. Denote by $L(A, B, C)$ the length of an LCS of A, B and C, and by $L(i, j, k)$ the length of an LCS of the ith prefix $A^i = A[1 \ldots i]$ of A, the jth prefix $B^j = B[1 \ldots j]$ of B and the kth prefix $C^k = C[1 \ldots k]$ of C. As in the classical case of two strings [14], a basic dynamic programming scheme, using $\Theta(n^3)$ time and space in this case, can be set up, based on the recurrence

$$L(i,j,k) = \begin{cases} L(i-1,j-1,k-1)+1 & \text{if } A[i] = B[j] = C[k] \\ \max(L(i-1,j,k), L(i,j-1,k), L(i,j,k-1)) & \text{otherwise} \end{cases} \quad (1)$$

and the initial conditions

$$L(i,j,0) = L(i,0,k) = L(0,j,k) = 0 \quad \text{for all } i,\ j,\ k.$$

We now investigate how this dynamic programming approach can be speeded up by suppressing unnecessary evaluations. Our approach is similar to that employed in [10], [15] and [13] for the case of two strings, and we refer to our algorithm as the "lazy" approach to dynamic programming.

We first look at the problem from a slightly different point of view. Define the *difference factor* $D = D(A, B, C)$ of the three strings to be the smallest total number of elements that need be deleted from the strings in order to leave three identical strings. Clearly D is a multiple of 3 if the strings are of equal lengths, since the same number of elements must be deleted from each string.

Lemma 1. $D(A, B, C) = 3(n - L(A, B, C))$.

Proof. The identical strings that remain when D elements are deleted from A, B, C form an LCS of A, B and C. Hence the deleted symbols, together with the 3 copies of the LCS constitute the $3n$ symbols of the original strings.

It follows that evaluation of $D(A, B, C)$ gives the value of $L(A, B, C)$, and that identification of a minimum set of deletions reveals an LCS. This is the approach that we shall now follow.

Let $D(i, j, k)$ be the difference factor of the prefixes A^i of A, B^j of B and C^k of C. Then the recurrence corresponding to 1 above is

$$D(i,j,k) = \begin{cases} D(i-1,j-1,k-1) & \text{if } A[i] = B[j] = C[k] \\ 1 + \min(D(i-1,j,k), D(i,j-1,k), D(i,j,k-1)) & \text{otherwise} \end{cases} \quad (2)$$

subject to

$$D(i,j,0) = i+j,\ D(i,0,k) = i+k,\ D(0,j,k) = j+k \quad \text{for all } i,\ j,\ k.$$

Clearly, in the light of this equation, cells (i,j,k) such that $A[i] = B[j] = C[k]$ have a special significance — we refer to such a cell as *a match position.*

We now consider the task of evaluating $D(n,n,n)$ while calculating as few as possible of the other D values. We refer to the 3-dimensional table $D(i,j,k)$ $(1 \leq i,j,k \leq n)$ as the *D-table* and we say that the cell (i,j,k) has *D-value* $D(i,j,k)$. Note that we will include coordinate values of -1 and 0 when implementing the table to facilitate appropriate initialisation in our algorithm.

By a *diagonal* of the D-table we mean an ordered sequence of cells with coordinates $(i+l, j+l, k+l)$ where $\min(i,j,k) = -1$ and $l = 0, \ldots, n - \max(i,j,k)$. The diagonal containing cell (i,j,k) will be denoted by $\langle i,j,k \rangle$. We say that a given cell (i,j,k) occupies *position* $i+j+km$ on its diagonal. Note that the positions on diagonal $\langle i,j,k \rangle$ increase by 3 from cell to cell, and are all $\equiv i+j+k \pmod 3$.

We also define

$$\ll i,j,k \gg = \{\langle i,j,k \rangle, \langle i,k,j \rangle, \langle j,i,k \rangle, \langle j,k,i \rangle, \langle k,i,j \rangle, \langle k,j,i \rangle\} \ ,$$

noting that this set of diagonals is of size 6, 3 or 1 depending whether the number of distinct values in the set $\{i,j,k\}$ is 3, 2 or 1. Any such set of diagonals has a *canonical* representation as $\ll i,j,0 \gg$ with $i \geq j \geq 0$.

The following lemma is immediate:

Lemma 2. *If* $\langle x,y,z \rangle \in \ll i,j,0 \gg$, *with* $x,y,z \geq 0$ *and* $i \geq j \geq 0$, *then* $D(x,y,z) \geq i+j$.

The *neighbours* of diagonal $\langle i,j,k \rangle$ are the 3 diagonals $\langle i-1,j,k \rangle$, $\langle i,j-1,k \rangle$ and $\langle i,j,k-1 \rangle$ and the *neighbourhood* of the set $\ll i,j,k \gg$ consists of all the diagonals that neighbour at least one diagonal in that set. (Note that the neighbour relation is not a symmetric one — indeed it is antisymmetric, in that if diagonal $\langle i',j',k' \rangle$ is a neighbour of diagonal $\langle i,j,k \rangle$ then $\langle i,j,k \rangle$ is not a neighbour of $\langle i',j',k' \rangle$. However, the following lemma may easily be verified.

Lemma 3. *If diagonal* Δ' *is a neighbour of diagonal* Δ, *then there is a diagonal* Δ'' *such that* Δ'' *is a neighbour of* Δ' *and* Δ *is a neighbour of* Δ''.

The *neighbours* of a cell (i,j,k), likewise, are defined to be the cells $(i-1,j,k)$, $(i,j-1,k)$ and $(i,j,k-1)$. Obviously, the cell in position p in a given diagonal has, as its neighbours, the cells in position $p-1$ in each of the neighbouring diagonals.

For the cell in position p in a given diagonal Δ, let us denote the D-value of that cell by $D(p,\Delta)$. The following technical lemmas may be established easily from the definitions of difference factor, neighbour and position.

Lemma 4. *Let diagonal* Δ' *be a neighbour of diagonal* Δ. *Then*

$$D(p,\Delta) \leq 1 + D(p-1,\Delta') \ .$$

Lemma 5. *The values in any diagonal of the D-table form a non-decreasing sequence, with the difference between successive elements in that sequence being 0 or 3.*

```
max := 0; {length of LCS found so far}
d := −1 ; { d is the current diagonal }
repeat
    d := d + 1 ;
    i := d ; { i is the row number in the τ table }
    τ_{i,0} := {(0, 0)} ;
    m := 0 ; { m is the column number in the τ table }
    repeat
        i := i + 1 ;
        m := m + 1 ;
        if m > max then
            τ_{i−1,m} := ∅ ; { further initialisation }
        σ := τ_{i−1,m} ∪ ⋃_{(j,k)∈τ_{i−1,m−1}} (b_{i,j}, c_{i,k}) ;
        τ_{i,m} := σ̂ ;
    until (τ_{i,m} = ∅) or (i = n) ;
    if m − 1 > max then
        max := m − 1
until i = n ;
if τ_{n,m} = ∅ then
    l := m − 1
else
    l := m
```

Fig. 5. The threshold algorithm for the length of the LCS of 3 strings

The crux of the main part of the algorithm is the evaluation of $\tau_{i,m}$ from $\tau_{i-1,m-1}$ and $\tau_{i-1,m}$. This can be achieved in $O(|\tau_{i-1,m}| + |\tau_{i-1,m-1}|)$ time by maintaining each set as a linked list with (j,k) preceding (j',k') in the list if and only if $j < j'$. For then the lists may easily be merged in that time bound to form a list representing $\sigma_{i,m}$, and as observed earlier, the list representing $\tau_{i,m} = \hat{\sigma}_{i,m}$ may be generated by a further single traversal of that list, deleting zero or more elements in the process.

It follows that the total number of operations involved in the forward pass of the algorithm is bounded by a constant times the number of pairs summed over all the sets $\tau_{i,m}$ evaluated. The backward pass involves tracing a single path through the τ table, examining a subset of the entries generated during the forward pass, and hence the overall complexity is dominated by the forward pass.

A trivial bound on the number of pairs in $\tau_{i,m}$ is $n - m + 1$, since the first component of each pair must be unique and in the range $m, \ldots, n$. Hence, since the number of sets $\tau_{i,m}$ evaluated is bounded by $l(n - l + 1)$ — at most l sets in each of $n - l + 1$ diagonals — this leads to a trivial worst-case complexity bound of $O(l(n-l)(n-\frac{l}{2}))$. In fact, it seems quite unlikely that all, or even many, of the sets $\tau_{i,m}$ can be large simultaneously, and we might hope for a worst-case bound of, say, $O(l(n-l)^2)$. But it seems to be quite difficult to establish such a bound for the algorithm as it stands. To realise this bound, we adapt the algorithm to a rather

2.1 Description of the "Lazy" Algorithm

The objective of the pth iteration of our so-called "lazy" algorithm is the determination of the last (i.e., highest numbered) position in the main diagonal (diagonal $\langle 0,0,0\rangle$) of the D-table occupied by the value $3p$. But this will be preceded by the determination of the last position in each neighbouring diagonal occupied by $3p-1$, which in turn will be preceded by the determination of the last position in each neighbouring diagonal of these occupied by $3p-2$, and so on. As soon as we discover that the last position in the main diagonal occupied by $3p$ is position $3n$ (corresponding to cell (n,n,n)), we have established that $D(A,B,C)=3p$, and therefore, by Lemma 1, that $L(A,B,C)=n-D(A,B,C)/3=n-p$.

In general, when we come to find the last position in diagonal $\langle i,j,k\rangle$ occupied by the value r, we will already know the last position in each neighbouring diagonal occupied by $r-1$. If we denote by $t(\Delta,r)$ the last position in diagonal Δ occupied by the value r, then we can state the fundamental lemma that underpins the lazy algorithm.

Lemma 6. *If $x=\max t(\Delta',r-1)$, where the maximum is taken over the neighbours Δ' of diagonal Δ, then*

$$t(\Delta,r)=x+1+3s\ ,$$

where s is the largest integer such that positions $x+4,x+7,\ldots,x+3s+1$ on diagonal Δ are match positions. (These positions constitute a "snake" in the terminology of Myers [10])

Proof. It follows from Lemma 4 that $D(x+1,\Delta)\leq r$, and clearly if its value is $<r$ then it is $\leq r-3$. But in this latter case, by Lemmas 3 and 4, $D(x+3,\Delta')\leq r-1$ for Δ' a neighbour of Δ, contradicting the maximality of x. Hence $D(x+1,\Delta)=r$. Further, if $D(y,\Delta)=r$ for any $y>x+1$ then, since no neighbouring diagonal has a value less than r beyond position x, according to the basic dynamic programming formula (2) for D, it must follow that y is a match position in that diagonal.

We define the *level* of a diagonal Δ to be the smallest m such that there is a sequence $\Delta_0=\langle 0,0,0\rangle,\Delta_1,\ldots,\Delta_m=\Delta$ with Δ_s a neighbour of Δ_{s-1} for each s, $s=1,\ldots,m$. It may easily be verified that the level of $\Delta=\langle i,j,k\rangle$ is $f_{ijk}=3\max(i,j,k)-(i+j+k)$, and therefore that the level of diagonal $\Delta=\langle i,j,0\rangle$ $(i\geq j)$ is $2i-j$. This can be expressed differently, as in the following lemma.

Lemma 7. *The diagonals at level x are those in the sets $\ll x-y,x-2y,0\gg$ for $y=0,1,\ldots,\lfloor\frac{x}{2}\rfloor$.*

The next lemma is an easy consequence of Lemmas 2 and 7.

Lemma 8. *The smallest D-value in any cell of a diagonal at level x is $\frac{x}{2}$ if x is even, and $\frac{x+3}{2}$ if x is odd.*

During the pth iteration of our algorithm, we will determine, in decreasing order of x, and for each relevant diagonal Δ at level x, the value of $t(\Delta,3p-x)$. Since, by

Lemma 8, no diagonal at level $> 2p$ can contain a D-value $\leq p$, we need consider, during this pth iteration, only diagonals at levels $\leq 2p$. Furthermore, by Lemma 2, the only diagonals at level x that can include a D-value as small as $3p-x$ are those diagonals in the family $\ll x-y, x-2y, 0 \gg$ with $(x-y)+(x-2y) \leq 3p-x$, i.e., with $y \geq x-p$.

Hence, to summarise, the pth iteration involves the evaluation, for $x = 2p, 2p-1, \ldots, 0$, of $t(\Delta, 3p-x)$ for $\Delta \in \ll x-y, x-2y, 0 \gg$ with $\max(0, x-p) \leq y \leq \lfloor \frac{x}{2} \rfloor$.

Lemma 9. *If the t values are calculated in the order specified above, then, for arbitrary Δ, r, the values of $t(\Delta', r-1)$, for each neighbour Δ' of Δ, will be available when we come to evaluate $t(\Delta, r)$.*

Proof. Consider, without loss of generality, $\Delta = \langle i, j, k \rangle$ with $i \geq j \geq k$. If the t values are calculated according to the scheme described, then $t(\langle i, j, k, \rangle, r)$ is evaluated during iteration $p = \frac{r+2i-j-k}{3}$. Furthermore, $t(\langle i, j-1, k \rangle, r)$ and $t(\langle i, j, k-1 \rangle, r)$ are also evaluated during iteration p, and $t(\langle i-1, j, k \rangle, r)$ is evaluated during iteration $p-1$ or p according as $i > j$ or $i = j$. Since, during iteration p, any $t(., r-1)$ is evaluated before any $t(., r)$, it follows that $t(\Delta', r-1)$ is evaluated before $t(\langle i, j, k \rangle, r)$ for every neighbour Δ' of $\langle i, j, k \rangle$.

We can now express the algorithm as in Fig. 1.

```
p := -1 ;
repeat
    p := p + 1 ;
    Initialise t values for pth iteration ;
    for x := 2p downto 0 do
        for y := max(0, x - p) to ⌊x/2⌋ do
            for Δ ∈≪ x - y, x - 2y, 0 ≫ do
                evaluate t(Δ, 3p - x)
until t(⟨0, 0, 0⟩, 3p) = n
```

Fig. 1. The lazy algorithm for the length of the LCS of 3 strings

The evaluation of $t(\Delta, 3p-x)$ is carried out using Lemma 6, following a snake, as in Fig. 2 for diagonal $\langle i, j, k \rangle$.

The relevant initialisation for the pth iteration involves those diagonals not considered in the $(p-1)$th iteration, and these turn out to be the diagonals in the set $\ll p, r, 0 \gg$ for $p \geq r \geq 0$. We initialise t for such a diagonal so that the (imaginary) last occurrence of the value $p+r-3$ is in position $p+r-3$ — see Fig. 3.

Recovering an LCS Recovering an LCS involves a (conceptual) trace-back through the n-cube from cell (n, n, n) to cell $(0, 0, 0)$ under the control of the t values. At

$u := 1 + \max t(\Delta', 3p - x - 1)$ over neighbours Δ' of $\langle i, j, k \rangle$;
$v := (u - i - j - k)$ **div** 3 ;
while $(A[i + v + 1] = B[j + v + 1] = C[k + v + 1])$ **do** {Match position}
    $v := v + 1$; {Assuming sentinels}
$t(\langle i, j, k \rangle, 3p - x) := i + j + k + 3v$

Fig. 2. Evaluation of $t(\Delta, 3p - x)$ for $\Delta = \langle i, j, k \rangle$

for $r := 0$ **to** p **do**
    for $\langle i, j, k \rangle \in \ll p, r, 0 \gg$ **do**
        $t(\langle i, j, k \rangle, p + r - 3) := p + r - 3$

Fig. 3. Initialisation for the pth iteration of the lazy algorithm

each step, we are in a cell whose D-value became known during the execution of the algorithm. We step back one position in the current diagonal if that cell is in a snake, and otherwise step into a neighbouring cell whose D-value (which can be discovered from the t array) is exactly one smaller. In the former case, we will have found one more character in the LCS, and as this backward trace unfolds, the LCS will be revealed in reverse order.

The precise algorithm is shown in Fig. 4.

Time Complexity of the Algorithm We now consider the time and space requirements of the algorithm. As far as time is concerned, it is not hard to see that the worst-case complexity is dependent on the total number of elements of the t table that are evaluated. In iteration p, the diagonals for which a t value is calculated for the first time are precisely the diagonals in the sets $\ll p, r, 0 \gg$ for $r = 0, \ldots, p$. There are $6p$ such diagonals in total, since $| \ll p, r, 0 \gg | = 3$ for $r = 0$ or $r = p$, and $| \ll p, r, 0 \gg | = 6$ for $1 \leq r \leq p - 1$. The number of iterations is the value of p for which $D(A, B, C) = 3p$, and by Lemma 1, this is exactly equal to $n - l$, where $l = L(A, B, C)$.

So, the total number of diagonals involved during the execution of the algorithm is

$$\sum_{p=0}^{n-l} 6p = 3(n - l)(n - l + 1) \ .$$

Furthermore, the $6p$ diagonals that are started at the pth iteration each contains at most $n - p$ entries in the n-cube, so the total number of t elements evaluated cannot

```
{ Throughout, Δ represents diagonal ⟨i, j, k⟩ }
i := n ; j := n ; k := n ;
d := 3p ;
m := n − p ;
repeat
  if A[i] = B[j] = C[k] then
    begin
    LCS[m] := A[i] ; m := m − 1 ;
    i := i − 1 ; j := j − 1 ; k := k − 1 ;
    end
  else
    begin
    find neighbour Δ′ of Δ such that t(Δ′, d − 1) = i + j + k − 1 ;
    d := d − 1 ;
    decrement i, j or k according as Δ′ differs from Δ
      in its 1st, 2nd or 3rd coordinate
    end
until i = 0
```

Fig. 4. Recovering an LCS from the t table

exceed

$$\sum_{p=0}^{n-l} 6p(n-p) = (n-l)(n-l+1)(n+2l-1). \tag{3}$$

As a consequence, the worst-case complexity of the algorithm is $O(n(n-l)^2)$, showing that we can expect it to be much faster than the naive dynamic programming algorithm in cases where the LCS has length close to n.

Space Complexity Diagonal $\langle i, j, k\rangle$ may be uniquely identified by the ordered pair $(i-j, j-k)$ (say), and such a representation leads in an obvious way to an array based implementation of the algorithm using $O(n^3)$ space. But appropriate use of dynamic linked structures enables just one node to be created and used for each t value calculated, and so by Equation 3, this yields an implementation that uses both $O(n(n-l)^2)$ time and space in the worst case. A little more care is required, in that case, in recovering the LCS from the linked lists of t values.

As with standard algorithms for the LCS of two strings, the space requirement can be reduced to $O((n-l)^2)$ if only the length of the LCS is required — in that case, only the most recent t value in each diagonal need be retained.

Extension to $\geq$ 4 Strings The lazy algorithm described in the previous section may be extended in a natural way to find the LCS of a set of k strings for any fixed $k \geq 3$. Such an extended version can be implemented to use $O(kn(n-l)^{k-1})$ time and space in the worst case, the factor of k arising from the need to find the smallest of k values in the innermost loop.

3 A Threshold Based Algorithm

Hunt and Szymanski [7] and Nakatsu et al [11] introduced algorithms for the LCS of two strings based on the so-called "threshold" approach. In the case of 3 strings A, B and C of length n, we define the *threshold set* $\tau_{i,m}$ to be the set of ordered pairs (j,k) such that A^i, B^j and C^k have a common subsequence of length m, but neither A^i, B^j and C^{k-1} nor A^i, B^{j-1} and C^k have such a common subsequence.

For example, for the sequences

$$A = abacbcabbcac \quad B = bbcabcbaabcb \quad C = cabcacbbcaba$$

we have $\tau_{5,2} = \{(2,7),(3,4),(5,3)\}$, corresponding to the subsequences bb, bc, and ab or cb respectively.

If (x,y), (x',y') are distinct ordered pairs of non-negative integers, we say that (x,y) *dominates* (x',y') if $x \leq x'$ and $y \leq y'$. It is immediate from the definition of the set $\tau_{i,m}$ that it contains no two pairs one of which dominates the other — we say that it is *domination-free.*

It is also clear from the definition of $\tau_{i,m}$ that the length of a longest common subsequence of A, B and C is the largest value of m for which $\tau_{n,m}$ is non-empty. Hence, evaluation of the threshold sets $\tau_{i,m}$ in some appropriate order will enable us to determine the length of the LCS, and by suitable back tracing, to find an actual LCS of the 3 strings.

We shall now describe an algorithm for the LCS of 3 strings based on the evaluation of the threshold sets $\tau_{i,m}$ in diagonal order — i.e., in general we evaluate $\tau_{i,m}$ immediately after $\tau_{i-1,m-1}$. To explain the algorithm we need some additional terminology and notation.

Let S be a set of ordered pairs of non-negative integers. The *dominating reduction* of S, denoted $\hat{S}$ is the minimal subset of S such that

$$(x',y') \in S \Rightarrow \exists (x,y) \in \hat{S} \text{ such that } (x,y) \text{ dominates } (x',y').$$

Clearly $\hat{S}$ is domination-free for any set S, and can be found from S by successively removing from S any pair that is dominated by another pair in S. If the pairs in S are arranged in a list so that (x,y) precedes (x',y') in the list whenever $x < x'$, then a similarly ordered list representing $\hat{S}$ can be obtained by a single scan of the original list, comparing successive neighbours on the list and deleting any pair found to be dominated by its neighbour. So deriving $\hat{S}$ from S can be done in time linear in the size of S.

For any character α in the string alphabet, and any position i $(0 \leq i \leq n)$, we define the *next-occurrence* table N_B for string B by

$$N_B[\alpha,i] = \begin{cases} \min j : j > i \text{ and } B[j] = \alpha & \text{if such a } j \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

So, as a special case, $N_B[\alpha,0]$ is the position of the first occurrence in string B of character α. The next-occurrence table N_C for string C is defined analogously.

For any positions i and j in strings A and B respectively, we define $b_{i,j}$ to be the first position after j in B that is occupied by character $A[i]$, i.e.,

$$b_{i,j} = N_B[A[i],j] \ .$$

Similarly, we define

$$c_{i,j} = N_C[A[i], j] \ .$$

The basis of our algorithm is the following Lemma.

Lemma 10. *If* $\sigma_{i,m} = \tau_{i-1,m} \cup \bigcup_{(j,k)\in\tau_{i-1,m-1}} (b_{i,j}, c_{i,k})$ *then* $\tau_{i,m} = \hat{\sigma}_{i,m}$.

Proof. Let $(j,k) \in \tau_{i,m}$, so that A^i, B^j and C^k have a common subsequence of length m but neither A^i, B^{j-1}, C^k nor A^i, B^j, C^{k-1} have such a common subsequence. If A^{i-1}, B^j, C^k have a common subsequence of length m, then $(j,k) \in \tau_{i,m-1}$. Otherwise any common subsequence of A^i, B^j, C^k of length m contains $A[i] = B[j] = C[k]$ as its last element. So there is some $(j',k') \in \tau_{i-1,m-1}$ such that $b_{i,j'} = j$, $c_{i,k'} = k$. As a consequence, any element of $\tau_{i,m}$ is in the set $\sigma_{i,m}$ defined in the lemma, and since $\tau_{i,m}$ is domination-free, it is included in $\hat{\sigma}_{i,m}$.

On the other hand, any member of $\tau_{i-1,m}$ and any pair $(b_{i,j'}, c_{i,k'})$ for $(j',k') \in \tau_{i-1,m}$ is bound to be dominated by a member of $\tau_{i,m}$, so that there are no elements in $\hat{\sigma}_{i,m}$ that are not in $\tau_{i,m}$.

Our algorithm begins the evaluation of the sets $\tau_{i,m}$ on the main diagonal — i.e., it uses the above lemma to evaluate $\tau_{1,1}$, $\tau_{2,2}, \ldots$ until $\tau_{i,i} = \emptyset$, this last condition indicating that the ith prefix of A is not a common subsequence of A, B and C. Next comes the evaluation of $\tau_{2,1}$, $\tau_{3,2}, \ldots$, again continuing until the first empty set is reached.

The process continues until $\tau_{n,m}$ is evaluated for some m. At that point the algorithm terminates, for no $\tau_{n,m'}$ can be non-empty for any $m' > m$. If $\tau_{n,m} \neq \emptyset$ then m is the length of the LCS of A, B and C, otherwise its length is $m-1$, since $\tau_{n-1,m-1}$ must have been non-empty. To enable the recurrence scheme to work correctly, at the beginning of the ith iteration $\tau_{i,0}$ is initialised to $(0,0)$, and whenever an unevaluated set $\tau_{i,m}$ is required (in the diagonal above the current one) it is initialised to the empty set.

The algorithm to find the length l of the LCS by this method is summarised in Fig. 5.

Recovering an LCS If the algorithm terminates with $\tau_{n,m}$ non-empty then we start the trace-back at an arbitrary pair (j,k) in $\tau_{n,m}$, otherwise at an arbitrary pair (j,k) in $\tau_{n-1,m-1}$. In either case, $B[j] = C[k]$ is the last character of our LCS. At each stage, when we have reached $\tau_{i,m}$ say, we repeatedly decrement i until the current pair (j,k) is not a member of $\tau_{i-1,m}$. This indicates that, when constructing the τ table, we had $A[i] = B[j] = C[k]$, and (j,k) must have arisen from an element (j',k') in $\tau_{i-1,m-1}$ that dominates it — so we locate such a pair, and record $A[i]$ as an element in the LCS. We then decrement i and m, and the pair (j',k') becomes the new (j,k). This part of the algorithm is summarised in Fig. 6.

Time Complexity of the Algorithm As described, the algorithm requires the pre-computation of the N_B and N_C tables, each requiring $\Theta(sn)$ steps, where s is the alphabet size.

```
if τ_{n,m} = ∅ then
    begin
    choose (j,k) ∈ τ_{n-1,m-1} ;
    i := n − 1 ;   r := m − 1
    end
else
    begin
    choose (j,k) ∈ τ_{n,m} ;
    i := n ;   r := m
    end;
while m > 0 do
    begin
    while (j,k) ∈ τ_{i-1,r} do
        i := i − 1 ;
    {A[i] = B[j] = C[k] in LCS}
    i := i − 1 ;   r := r − 1 ;
    choose (j',k') ∈ τ_{i,r} dominating (j,k) ;
    (j,k) := (j',k') ;
    end
```

Fig. 6. Recovering an LCS from the threshold table

different form, with the aid of a trick also used by Apostolico et al [2] in a slightly different context.

The crucial observation is that, if a pair (j,k) in set $\tau_{i,m}$ is to be part of an LCS of length l, then we must have $0 \leq j - m \leq n - l$ and $0 \leq k - m \leq n - l$, since there cannot be more than $n - l$ elements in either string B or string C that are not part of the LCS. So, if we knew the value of l in advance, we could immediately discard from each set $\tau_{i,m}$ any pair (j,k) for which $j > n - l + m$ or $k > n - l + m$. Hence each set would have effective size $\leq n - l + 1$, and since the number of sets is $O(l(n-l))$, this would lead to an algorithm with $O(l(n-l)^2)$ worst-case complexity.

Of course, the problem with this scheme is that we do not know the value of l in advance — it is precisely this value that our algorithm is seeking to determine. However, suppose we parameterise the forward pass of our algorithm with a bound p, meaning that we are going to test the hypothesis that the length of the LCS is $\geq n - p$. In that case, we need evaluate only $p + 1$ diagonals (at most), each of length $\leq l$, and in each set $\tau_{i,m}$ we need retain at most $p+1$ pairs. So this algorithm requires $O(lp^2)$ time in the worst case.

Suppose that we now apply the parameterised algorithm successively with $p = 0, 1, 2, 4, \ldots$ until it returns a "yes" answer. At that point we will know the length of the LCS, and we will have enough of the threshold table to reconstruct a particular LCS. As far as time complexity is concerned, suppose that $2^{t-1} < n - l \leq 2^t$. Then the algorithm will be invoked $t + 1$ times, with final parameter 2^t. So the overall worst-case complexity of the resulting LCS algorithm is $O(l \sum_{i=0}^{t} (2^i)^2) = O(l 2^{2t+2}) = O(l(n-l)^2)$, as claimed.

As it turns out, the above worst-case analysis of the threshold algorithm does not depend on the fact that the sets $\tau_{i,m}$ are domination-free, merely on the fact that if $(j,k),(j',k') \in \tau_{i,m}$ then $j \neq j'$ and $k \neq k'$ (though the maintenance of the sets as domination-free, by the method described earlier, will undoubtedly speed up the algorithm in practice.)

As an aside, we observe that, when restricted to the case of two strings, this threshold approach differs from, say, the approach of Nakatsu et al [11], in the order of evaluation of the elements of the threshold table. The resulting algorithm has $(O(l(n-l)+sn)$ complexity, a bound that does not appear to have been achieved previously.

Space Complexity As described above, the space complexity of the algorithm is also $O(l(n-l)^2+sn)$, the first term arising from the bound on the number of elements summed over all the $\tau_{i,m}$ sets evaluated, and the second term from the requirements of the next-occurrence tables. The first term can be reduced somewhat by changing the implementation so that, for each pair (j,k) that belongs to some $\tau_{i,m}$, a node is created with a pointer to its "predecessor". In each column of the τ table, a linked list is maintained of the pairs in the most recent position in that column. By this means, each pair generated uses only a single unit of space, so the overall space complexity is big oh of the number of (different) pairs generated in each column, summed over the columns. An LCS can be re-constructed by following the predecessor pointers.

A Time-Space Tradeoff As described by Apostolico and Guerra [3], the next-occurrence tables can be reduced to size $1 \times n$ rather than of size $s \times n$, for an alphabet of size s, at the cost of an extra $\log s$ factor in the time complexity — each lookup in the one-dimensional next-occurrence table takes $\log s$ time rather than constant time. In fact, for most problem instances over small or medium sized alphabets, the size of the next-occurrence tables is likely to be less significant than the space needed by the threshold table.

Extension to ≥ 4 Strings The idea of the threshold algorithm can be extended to find an LCS of k strings for any fixed $k \geq 4$, using sets of $(k-1)$-tuples rather than sets of pairs. In the general case, it is less clear how the sets $\tau_{i,m}$ can be efficiently maintained as domination-free, but as observed above for the case of three strings, the worst-case complexity argument does not require this property. In general, the extended version of the algorithm has worst-case time and space complexity that is $O(kl(n-l)^{k-1}+ksn)$, with the same time-space trade-off option available as before.

4 Empirical Evidence and Conclusions

To obtain empirical evidence as to the relative merits of the various algorithms, we implemented, initially for 3 strings, the basic dynamic programming algorithm (DP), the lazy algorithm (Lazy), the diagonal threshold algorithm (Thresh), and the algorithm of Hsu and Du (HD) [6]. In the case of the lazy algorithm, we used a

Table 1. Cpu times when LCS is 90% of string length

			String lengths						
			100	200	400	800	1600	3200	6400
Algorithms	DP		2.5	19.2	-	-	-	-	-
	Lazy		0.05	0.18	1.5	11.9	-	-	-
	Thresh	$s = 4$	0.03	0.1	0.4	2.3	21.7	136.2	-
		$s = 8$	0.03	0.1	0.3	1.7	8.8	54.8	-
		$s = 16$	0.02	0.1	0.3	1.3	7.1	33.6	212.5
	HD	$s = 4$	0.9	10.5	-	-	-	-	-
		$s = 8$	0.4	5.0	73.4	-	-	-	-
		$s = 16$	0.2	2.8	36.7	-	-	-	-

Table 2. Cpu times when LCS is 50% of string length

			String lengths					
			100	200	400	800	1600	3200
Algorithms	DP		2.3	18.9	-	-	-	-
	Lazy		3.0	23.1	-	-	-	-
	Thresh	$s = 4$	0.2	1.5	11.7	107.8	-	-
		$s = 8$	0.1	0.7	5.5	48.3	-	-
		$s = 16$	0.05	0.4	2.5	18.8	184.0	-
	HD	$s = 4$	0.8	9.7	-	-	-	-
		$s = 8$	0.3	4.5	59.6	-	-	-
		$s = 16$	0.2	2.3	31.8	-	-	-

dynamic linked structure, as discussed earlier, to reduce the space requirement. We implemented both the straightforward threshold algorithm of Figs. 5 and 6, and the version with the guaranteed $O(l(n-l)^2+sn)$ complexity, and found that the former was consistently between 1.5 and 2 times faster in practice — so we have included the figures for the faster version. The Hsu-Du algorithm was implemented using both a 3-dimensional array and a dynamic structure to store match nodes. There was little difference between the two in terms of time, but the latter version allowed larger problem instances to be solved, so we quote the results for that version.

Table 3. Cpu times when LCS is 10% of string length

			String lengths					
			100	200	400	800	1600	3200
Algorithms	DP		2.4	18.0	-	-	-	-
	Lazy		17.0	-	-	-	-	-
	Thresh	$s = 4$	0.02	0.4	2.6	17.1	142.2	-
		$s = 8$	0.02	0.3	2.3	15.3	128.5	-
		$s = 16$	0.02	0.2	1.5	10.6	117.2	-
	HD	$s = 4$	0.06	0.5	-	-	-	-
		$s = 8$	0.06	0.4	9.7	-	-	-
		$s = 16$	0.06	0.3	5.0	-	-	-

The algorithms were coded in Pascal, compiled and run under the optimised Sun Pascal compiler on a Sun 4/25 with 8 megabytes of memory. We used alphabet sizes of 4, 8 and 16, string lengths of 100, 200, 400, ... (for simplicity, we took all 3 strings to be of the same length), and we generated sets of strings with an LCS of respectively 90%, 50% and 10% of the string length in each case. In all cases, times were averaged over 3 sets of strings.

The results of the experiments are shown in the tables, where the cpu times are given in seconds. Times for DP and Lazy were essentially independent of alphabet size, so only one set of figures is included for each of these algorithms. In every case, we ran the algorithms for strings of length 100, and repeatedly doubled the string length until the program failed for lack of memory, as indicated by "-" in the table.

Conclusions. As far as comparisons between the various algorithms are concerned, the diagonal threshold method appears to be the best across the whole range of problem instances generated. The lazy method is competitive when the LCS is close to the string length, and the Hsu-Du algorithm only when the LCS is very short.

However, the clearest conclusion that can be drawn from the empirical results is that, at least using the known algorithms, the size of 3-LCS problem instances that can be solved in practice is constrained by space requirements rather than by time requirements. Preliminary experiments with versions of the threshold and Hsu/Du algorithms for more than 3 strings confirm the marked superiority of the former, and show that the dominance of memory constraint is likely to be even more pronounced for larger numbers of strings. So there is clearly a need for algorithms that use less space. One obvious approach worthy of further investigation is the application of space-saving divide-and-conquer techniques [4] [2], and other possible time-space trade-offs, particulary, in view of the evidence, to the threshold algorithm.

References

1. A. Apostolico. Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Information Processing Letters*, 23:63–69, 1986.
2. A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92:3–17, 1992.
3. A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.
4. D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the A.C.M.*, 18:341–343, 1975.
5. D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the A.C.M.*, 24:664–675, 1977.
6. W.J. Hsu and M.W. Du. Computing a longest common subsequence for a set of strings. *BIT*, 24:45–59, 1984.
7. J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the A.C.M.*, 20:350–353, 1977.
8. S.Y. Itoga. The string merging problem. *BIT*, 21:20–30, 1981.
9. W.J. Masek and M.S. Paterson. A faster algorithm for computing string editing distances. *J. Comput. System Sci.*, 20:18–31, 1980.

10. E.W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.
11. N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.
12. D. Sankoff. Matching sequences under deletion insertion constraints. *Proc. Nat. Acad. Sci. U.S.A.*, 69:4–6, 1972.
13. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
14. R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the A.C.M.*, 21:168–173, 1974.
15. S. Wu, U. Manber, G. Myers, and W. Miller. An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35:317–323, 1990.