A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem

Maxime Crochemore[1⋆], Costas S. Iliopoulos[2⋆⋆]
, Yoan J. Pinzon[3⋆⋆⋆], and James F. Reid[4†]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, 5 Bd Descartes, Champs-sur-Marne, F-77454 Marne-la-Vallée CEDEX 2, France.
mac@univ-mlv.fr
www-igm.univ-mlv.fr/~mac

[2] Dept. Computer Science, King's College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.
csi@dcs.kcl.ac.uk,
www.dcs.kcl.ac.uk/staff/csi

[3] Dept. Computer Science, King's College London, London WC2R 2LS.
pinzon@dcs.kcl.ac.uk
www.dcs.kcl.ac.uk/pg/pinzon

[4] Dept. Computer Science, King's College London, London WC2R 2LS, and Dipartimento di Elettronica e Informatica, Università degli Studi di Padova, Italy.
jfr@dcs.kcl.ac.uk
www.dcs.kcl.ac.uk/pg/jfr

Abstract. This paper presents a new practical bit-vector algorithm for computing the well known Longest Common Subsequence (LCS) problem. Given two strings of length m and n, $n \geq m$, we present an algorithm which determines the length p of an LCS in $O(nm/w)$ time and $O(m/w)$ additional/working (other than input/output) space, where w is the number of bits in a machine word. This algorithm can be thought of as column-wise "parallelization" of the classical dynamic programming approach. Our algorithm works very efficiently in practice, where computing the length of an LCS of two strings can be done in linear time and constant (additional/working) space by assuming that $m \leq w$.

1 Introduction

Given a string x over an alphabet Σ ($s = |\Sigma|$), a *subsequence* of x is any string w that can be obtained from x by deleting zero or more (not necessarily consecutive) symbols. The *Longest Common Subsequence* (LCS) problem for strings $x = x_1 \cdots x_m$ and $y = y_1 \cdots y_n, (n \geq m)$ consists in finding a third string

⋆ Partially supported by the C.N.R.S. Program "Génomes".
⋆⋆ Partially supported by a Royal Society Grant.
⋆⋆⋆ Partially supported by an ORS studentship.
† Supported by a Marie Curie Fellowship of the European Commission Training and Mobility of Researchers (TMR) Programme.

$w = w_1 \cdots w_p$ such that w is a subsequence of both x and y of maximum possible length. The LCS problem is related to two well known metrics for measuring the similarity (distance) of two strings: the *Levenshtein distance* [7] and the *edit distance* [14].

The Levenshtein distance is defined as the minimum number of character insertions and/or deletions required to transform a string x into a string y. The edit distance of two strings x and y is a generalization of the Levenshtein distance where the operations allowed are symbol insertion, deletion and substitution with corresponding costs that are functions of the symbols involved. The string editing problem is that of finding the minimum cost of transforming x into y, usually called an optimal edit script. If we assign unit costs for the deletion and insertion operations and a cost of two for a substitution, then the LCS problem is directly related to the string editing problem. If d is the cost of such an optimal edit script and p is the length of an LCS then $p = 1/2(n + m - d)$.

Finding the longest common subsequence or the edit distance of two string are problems with numerous applications ranging from text editing to molecular sequence analysis, see e.g. [13]. A vast number of efficient algorithms have been designed over the last two decades to solve these problems. The lower bounds for the LCS problem are time $\Omega(n \log m)$ or linear time, according to whether the size of the alphabet Σ is unbounded or bounded ([6]). For unbounded alphabets, any algorithm using only "equal-unequal" comparisons must take $\Omega(nm)$ in the worst case ([1]). The fastest general solution for the LCS problem is the corresponding solution to the string editing problem by Masek and Paterson [9] taking $O(n^2 \log\log n / \log n)$ for unbounded alphabet size and $O(n^2/\log n)$ for bounded alphabet size . Due to the fact that the best worst-case algorithm is still sub-quadratic in the input size, many parameterized algorithms have been designed, i.e. input- or output-sensitive algorithms.

In this paper we present a new way of computing an LCS of two strings by using bit-vector operations which is really fast in practice. The idea of using the bits of the computer word has been used extensively in the last years. In [4], Baeza-Yates and Gonnet presented an $O(nm/w)$ algorithm for the exact matching case and an $O(nm \log k/w)$ algorithm for the k-mismatches problem, where w is the number of bits in a machine word, n the length of the text and m the length of the pattern. Wu and Manber in [16] showed an $O(nkm/w)$ algorithm for the k-differences problem. Furthermore, Wright ([15]) presented an $O(n \log |\Sigma| m/w)$ bit-vector style algorithm where $|\Sigma|$ is the size of the alphabet for the pattern. Recently, Myers ([10]) developed a particularly practical method to compute the edit distance in $O(nm/w)$. Related work to the computation of the length of an LCS can be found in [2], where Allison and Dix present an $O(nm/w)$ algorithm using a bit-vector formula with 6 bit-wise operations. In this paper, we present a similar approach to the one presented in [10] and [2] for the computing the length of an LCS. The theoretical runing time of the algorithm is still $O(nm/w)$ and $O(m/w)$ space but experiments show a practical speed-up over the Allison and Dix algorithm due to the fact that our bit-vector formula only uses 4 bit-wise operations. Furthermore, adapting the Myers algorithm for the edit distance for solving the LCS problem would only add extra overheads.

The paper is organised as follows. In the next section we present the basic dynamic programming procedure for computing the length of an LCS, some of the properties of the resulting matrix and few definitions concerning the dominant matches. In Section 3 we describe the preprocessing and the main bit-vector algorithm for computing the length of an LCS of two strings, followed by a way to recover efficiently an actual LCS.

2 Preliminaries

Let the two input strings be $x = x_1 \cdots x_m$ and $y_1 \cdots y_n$ and let $L[i,j]$ denote the length of an LCS (LLCS) of $x_1 \cdots x_i$ and $y_1 \cdots y_j$. The following simple recurrence formula by Hirschberg ([5]) computes $p = L[m,n]$.

$$L[i,j] = \begin{cases} 0, & \text{if either } i = 0 \text{ or } j = 0 \\ 1 + L[i-1, j-1], & \text{if } x_i = y_j \\ \max\{L[i-1,j], L[i,j-1]\}, & \text{if } x_i \neq y_j \end{cases} \tag{1}$$

This dynamic programming algorithm requireses $O(mn)$ time and space. But in fact, only linear space is needed to find the LLCS since the computation of each row only needs the preceding row. An LCS can be retrieved by backtracking through the matrix, which would imply the computation of the whole matrix L, requiring $O(mn)$ space.

As an example, Fig. 1 shows the computation of the matrix L for x="`survey`" and y="`surgery`".

y

			0	1	2	3	4	5	6	7
			ϵ	s	u	r	g	e	r	y
	0	ϵ	0	0	0	0	0	0	0	0
	1	s	0	1	1	1	1	1	1	1
	2	u	0	1	2	2	2	2	2	2
x	3	r	0	1	2	3	3	3	3	3
	4	v	0	1	2	3	3	3	3	3
	5	e	0	1	2	3	3	4	4	4
	6	y	0	1	2	3	3	4	4	**5**

Fig. 1. The LCS Matrix L for x="`survey`" and y="`surgery`".

The LLCS p for this example is $L[6,7] = 5$ and LCS w = "`surey`". In this case the computed LCS is unique but that is not always the case.

Various algorithms have been designed to exploit the inherent *sparsity* of the matrix L of lengths of the LCS's of two input strings in such a way as to relate the performance of the algorithm to different parameters than the lengths of the input. Many algorithms for the LCS have used the so-called k-dominant matches. The ordered pair of *positions* i and j of L, denoted $[i, j]$, is a *match* iff $x_i = y_j$. If $[i, j]$ is a match, and an LCS $w_{i,j}$ of $x_1x_2 \cdots x_i$ and $y_1y_2 \cdots y_j$ 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 $j' \le j$ or $i' \le i$ and $j' > j$.

Computing the k-dominant matches is all that is needed to solve the LCS problem because the LCS of x and y has length p iff the maximum rank attained by a dominant match is p. We define a partial order relation on the set of matches in L by considering matches that 'precede' each other.

Let r be the total number of matches, and d be the total number of dominant points (of all ranks). Then $0 \le p \le d \le r \le nm$.

A match $[i, j]$ *precedes* a match $[i', j']$ if $i < i'$ and $j < j'$. Let $\mathcal{R}$ denote this partial order relation on the set of matches in L. A set of matches such that in any pair one of the matches always precedes the other in $\mathcal{R}$ constitutes a *chain* relative to the partial order relation $\mathcal{R}$. A set of matches such that in any pair neither element of the pair precedes the other in $\mathcal{R}$ constitutes an *antichain*.

Sankoff and Sellers [12] observed that the LCS problem translates to finding a longest *chain* in the *poset* of matches induced by $\mathcal{R}$. A decomposition of a poset into antichains partitions the poset into the minimum possible number of antichains, see e.g. [3].

The computation of the LCS for x="tccagatg" and y="aaagtgacctagcccg" is depicted in Figure 2. The chain from $L[1, 5]$ to $L[8, 16]$ spells out the LCS w="tccagg".

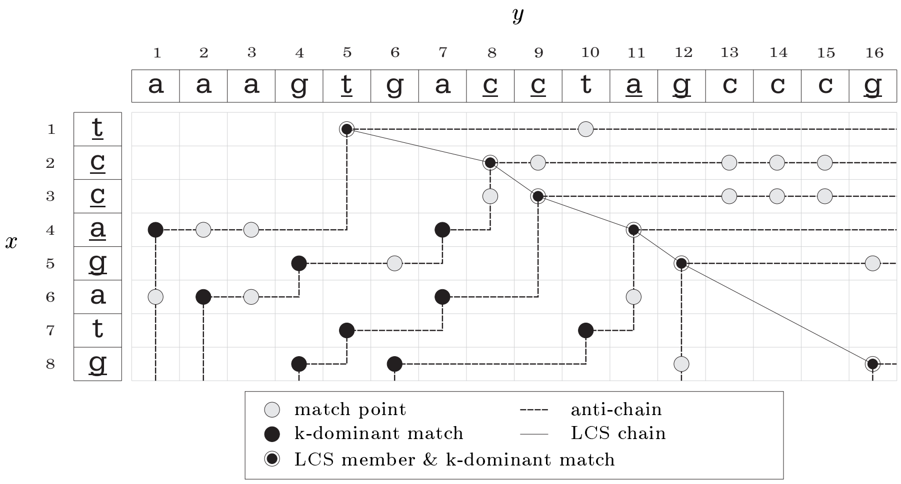

Fig. 2. LCS for x="tccagatg" and y="aaagtgacctagcccg".

3 A Simple Bit-Vector Algorithm

Here we will make use of word-level parallelism in order to compute the matrix L more efficiently, similar to the manner used by Myers in [10] for the edit distance problem. The algorithm is based on the $O(1)$-time computation of each column in L by using a bit-parallel formula under the assumption that $m \leq w$, where w is the number of bits in a machine word resulting in an $O(n)$ overall time complexity. In the general case, the time complexity is $O(nm/w)$ with $O(m/w)$ time per column. An interesting property of the LCS allows to represent each column in L by using $O(1)$-space. That is, the values in the columns (rows) of L increase by at most one. We will denote by ΔL the relative encoding of the table L, i.e. $\Delta L[i,j] = L[i,j] - L[i-1,j] \in \{0,1\}$ for any $(i,j) \in \{1..m\} \times \{1..n\}$. Table 1 shows the table L and the corresponding ΔL table for x=“`ttgatacat`” and y=“`gaataagacc`”. We also define $\Delta L'$ as the negation of ΔL ($\Delta L' = \texttt{NOT}\ \Delta L$).

What $\Delta L'_j$ holds is just the encoding of a *“cut”* of the k-dominant matches at position j and if we can compute the $\Delta L'_j$ column in $O(n/w)$, then we solve the LCS in $O(n/w)$ time. For instance, Figure 4 illustrates how to obtain $\Delta L'_6$ by drawing a vertical line (that we called “cut”) at position 6 and we define $\Delta L'_6[i] = 0$ iff the boundary line of k-dominant matches passes at this position and 1 otherwise.

The basic steps of the algorithm are as follows:

1. First we compute the array M of the vectors that result for each possible text character. If both the strings x and y range over the alphabet Σ then $M[\Sigma]$ is defined as $M[\alpha]_i = 1$ if $x_i = \alpha$ otherwise 0. This results in having a bit-vector associated to each symbol in the alphabet Σ. So for any given symbol, its vector contains 1's where a match of that symbol occurs in the string $x = x_1 \cdots x_m$. In fact from a practical point of view, we can restrict our attention to the alphabet over which the string x ranges, in case $|\Sigma| \geq m$. Here, we assume the alphabet Σ to be finite such as the ASCII machine character set. For infinite alphabets refer to [8] where hashing techniques are used to overcome this problem. We also define the array M' as the negation of M. Table 2 shows the preprocessing for our example.
2. Let y_j=α be the symbol being scanned in the string y at iteration j. Let $M[\alpha]_{1:m}$ and $M'[\alpha]_{1:m}$ be the vectors resulting from the previous step on the symbol $y_j = \alpha$. Let $\Delta L'_{j-1}$ be the bit-vector column of length m of the table $\Delta L'$ computed at the previous iteration $j-1$. We obtain the new column $\Delta L'_j$ using the following recursive formula:

$$\Delta L'_j = \begin{cases} 2^m - 1, & \text{for } j = 0 \\ (\Delta L'_{j-1} + (\Delta L'_{j-1}\ \texttt{AND}\ M[y_j]))\ \texttt{OR}\ (\Delta L'_{j-1}\ \texttt{AND}\ M'[y_j]), & \text{for } j \in \{1..n\} \end{cases}$$

3. Let LLCS be the number of times that a carry took place when computing $\Delta L'_j$ in the previous step.

The algorithm in Figure 3 computes the LLCS for two given strings x and y.

		0	1	2	3	4	5	6	7	8	9	10
		ϵ	g	a	a	t	a	a	g	a	c	c
0	ϵ	0	0	0	0	0	0	0	0	0	0	0
1	t	0	0	0	0	**1**	1	1	1	1	1	1
2	t	0	0	0	0	1	1	1	1	1	1	1
3	g	0	**1**	1	1	1	1	1	**2**	2	2	2
4	a	0	1	**2**	2	2	2	2	2	**3**	3	3
5	t	0	1	2	2	**3**	3	3	3	3	3	3
6	a	0	1	2	**3**	3	**4**	4	4	4	4	4
7	c	0	1	2	3	3	4	4	4	4	**5**	5
8	a	0	1	2	3	3	4	**5**	5	5	5	5
9	t	0	1	2	3	**4**	4	5	5	5	5	**5**

(a) Matrix L

		0	1	2	3	4	5	6	7	8	9	10
		ϵ	g	a	a	t	a	a	g	a	c	c
0	ϵ	0	0	0	0	0	0	0	0	0	0	0
1	t	0	0	0	0	1	1	1	1	1	1	1
2	t	0	0	0	0	0	0	0	0	0	0	0
3	g	0	1	1	1	0	0	0	1	1	1	1
4	a	0	0	1	1	1	1	1	0	1	1	1
5	t	0	0	0	0	1	1	1	1	0	0	0
6	a	0	0	0	1	0	1	1	1	1	1	1
7	c	0	0	0	0	0	0	0	0	0	1	1
8	a	0	0	0	0	0	0	1	1	1	0	0
9	t	0	0	0	0	1	0	0	0	0	0	0

(b) Matrix ΔL

		0	1	2	3	4	5	6	7	8	9	10
		ϵ	g	a	a	t	a	a	g	a	c	c
1	t	1	1	1	1	0	0	0	0	0	0	0
2	t	1	1	1	1	1	1	1	1	1	1	1
3	g	1	0	0	0	1	1	1	0	0	0	0
4	a	1	1	0	0	0	0	0	1	0	0	0
5	t	1	1	1	1	0	0	0	0	1	1	1
6	a	1	1	1	0	1	0	0	0	0	0	0
7	c	1	1	1	1	1	1	1	1	1	0	0
8	a	1	1	1	1	1	1	0	0	0	1	1
9	t	1	1	1	1	0	1	1	1	1	1	1

(c) Matrix $\Delta L'$

Table 1. (a) Standard dynamic programming matrix L with k-dominant matches in bold, (b) Bit-wise additive encoding ΔL of matrix L and (c) the complement of matrix ΔL ($\Delta L' = \texttt{NOT}\ \Delta L$) for x=“`ttgatacat`” and y=“`gaataagacc`”.

Building the array M and M' of bit-vectors can be done in $O(|\Sigma| + m)$ time and using $O(|\Sigma|)$ space, assuming that $m \leq w$ or $O(|\Sigma| m/w + m)$ for the unrestricted case. The formula in the second step can be performed in $O(m/w)$ since the 3 bit-wise operation in the formula (`AND`, `OR`, and `AND`) and the addition operation can be done in $O(m/w)$ time. Therefore, the computation of $\Delta L'_j$ for $j \in \{1..n\}$ can be done in $O(nm/w)$ time. The total overall complexity is dominated by step 2 and therefore we have a $O(nm/w)$-time complexity algorithm.

Our problem can also be seen as an automaton where the states are the columns of $\Delta L'$ matrix and the transition functions depends on the previous state and the preprocessing for the symbol being scanned, i.e. $\Delta L'_j = f(\Delta L'_{j-1}, M[y_j])$. Strictly speaking $\Delta L'_j = f(\Delta L'_{j-1}, M[y_j], M'[y_j])$.

Figure 5 illustrates the computation of $\Delta L'_4$ using the previous state $\Delta L'_3$ and the preprocessing array for the symbol y_4 =‘`t`’ found in M[‘`t`’]. Computing the

x		a	c	g	t
1	t	0	0	0	1
2	t	0	0	0	1
3	g	0	0	1	0
4	a	1	0	0	0
5	t	0	0	0	1
6	a	1	0	0	0
7	c	0	1	0	0
8	a	1	0	0	0
9	t	0	0	0	1

x		a	c	g	t
1	t	1	1	1	0
2	t	1	1	1	0
3	g	1	1	0	1
4	a	0	1	1	1
5	t	1	1	1	0
6	a	0	1	1	1
7	c	1	0	1	1
8	a	0	1	1	1
9	t	1	1	1	0

Table 2. The arrays M and M' for $x =$ "ttgatacat" and $\Sigma = \{$'a','c','g','t'$\}$

CIPR(x, y) $\quad\triangleright\ n = |y|,\ m = |x|,\ p = 0$

begin
 $\triangleright$ *Preprocessing*
 for $i = 1$ **to** m **do**
  $M[\alpha]_i \leftarrow 0\ \forall\alpha \in \Sigma$
  $M[x_i]_i \leftarrow 1$
 $M' \leftarrow$ NOT M
 $\triangleright$ *Initialization*
 $\Delta L'_0 = 2^m - 1$
 $\triangleright$ *The Main Step*
 for $j = 1$ **to** n **do**
  $\Delta L'_j \leftarrow (\Delta L'_{j-1} + (\Delta L'_{j-1}\ \texttt{AND}\ M[y_j]))\ \texttt{OR}\ (\Delta L'_{j-1}\ \texttt{AND}\ M'[y_j])$
  if $\Delta L'_j[m+1] = 1$ **then** $p \leftarrow p + 1$ $\quad\triangleright$ *there was a carry*
 return p
end

Fig. 3. The CIPR algorithm

transition function f consists of two steps. In the first step we want to select the right-most bit in $M[y_j]$ for each block of 1's in $\Delta L'_{j-1}$ (see the circled numbers in Figure 5). The second step involves swapping each of the selected bits with the closest 0 to the left in $\Delta L'_{j-1}$. In other words, the first step selects the matches that are candidates to update a current k-dominant match (0 bits in the previous state) and the second step updates them but just swapping the two. Notice that we will have a bit carry in the new state iff a new k-dominant is introduced, so if we count the number of carries (number of k-dominants introduced) we will also have the LLCS value. One could just count the number of 0's in $\Delta L'_n$. The number of bits set in a word q can be counted in $O(\log_2 q)$ (see [11]).

We now explain how the formula was obtained:

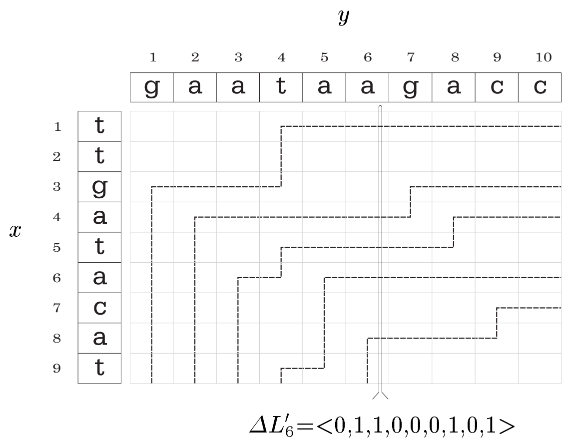

Fig. 4. The entries of the array $\Delta L'$ can be seen as "cuts" across k-dominant matches boundaries in the original array L for x="`ttgatacat`" and y="`gaataagacc`".

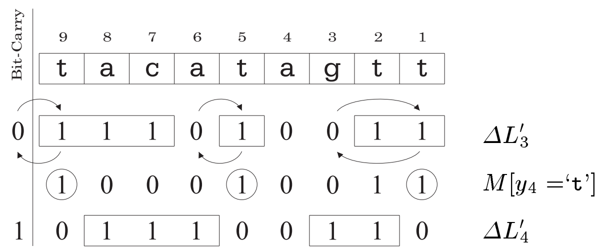

Fig. 5. Illustration of $\Delta L'_4$ for for x="`ttgatacat`" and y="`gaataagacc`".

– In the first place, those matches in positions where there is a k-dominant match currently must be set off. This is obtained by *and*-ing L_{j-1} with $M[y_j]$. From our example, 111010011 `AND` 100010011 = 100010011.
– Now we want to select only the right-most bit among each block of 1's and swap it with the closest 0 on its left. This is obtain by using the addition operation, which will change the left most bit for 0 and propagate this bit until a zero is found, i.e. a k-dominant match to be updated. This propagation occurs in each block simultaneously and we are ensured that the propagation is independent in each block since we have at least a zero between the blocks. This can be easily seen by showing the automata for the addition operation when assuming that the first number is a block of 1's (see Figure 6). Clearly, we can observe that if there is at least a 1 in the second number we can be sure that there will be a carry since there is no way back once you reach a state with a carry and we can also see that the first bit will be set off. So for instance, $111010011 + 100010011 = 011100110$ with a bit-carry.

  However, as you can see during the process there are cases where more than 1 bit is set off and we only want the right-most bit to be set off. This problem can be corrected by *or*-ing what we have so far with $(\Delta L'_{j-1}$ AND $M'[y_j])$. This has the effect of correcting the bits that were set off during the addition. For instance, for our example, 011000000 OR 011100110 = 011100110, which is exactly what we wanted to get. i.e. $\Delta L'_4$.

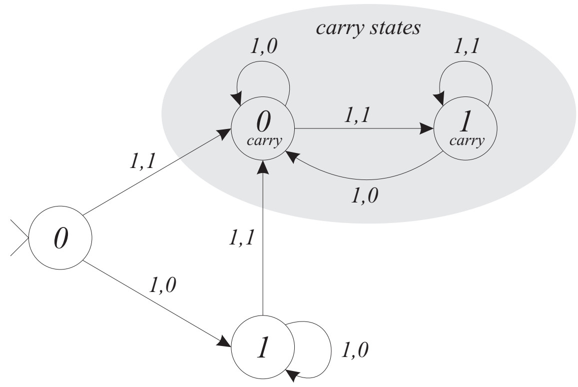

Fig. 6. Automata for the addition operation assuming that the first number is a block of 1's.

Figure 7 shows the timing[1] for the AllisonDix algorithm (see [2]) and our CPIR algorithm. These experiments were done assuming $|x| = |y|$ with alphabet size $s = 26$. Our algorithm CIPR was about twice faster for small lengths ($n = 5000$) and eight times faster for longer lengths ($n = 10000$). This is mainly due to the reduction of the amount of bit-wise operations from 6 to 4 and the use of the addition operator instead of the subtraction operator which is much more expensive in terms of time complexity.

4 Retrieving an LCS

In some applications it is required to produce an LCS instead of just its length. One option would be to use Hirschberg's paradigm [5] to recover an LCS by using divide-and-conquer together with our bit-vector algorithm for evaluating the length of an LCS. The divide-and-conquer determines the middle point of an LCS curve and then applies the procedure recursively. Each iteration uses linear space, and it can be shown that the total time used is still $O(nm/w)$.

References

1. A.V. Aho, D.S. Hirschberg and J.D. Ullman, Bounds on the complexity of the longest common subsequence problem, *J. Assoc. Comput. Mach.*, 23, 1–12, (1976).

[1] Using a SUN Ultra Enterprise 300MHz running Solaris Unix.

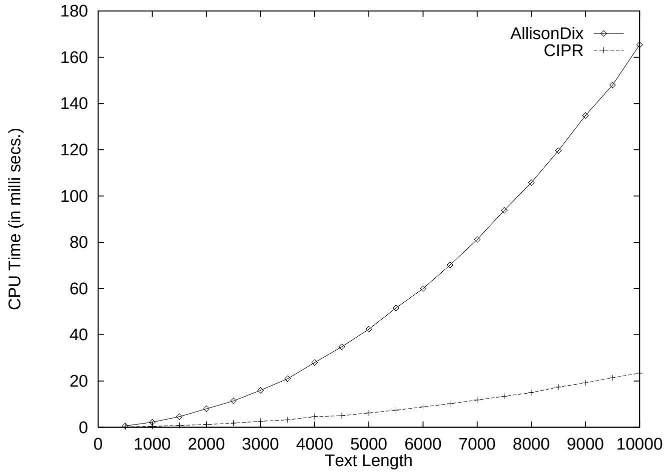

Fig. 7. Timing curves for the AllisonDix algorithm and our CIPR algorithm.

2. L. Allison and T.L. Dix, A bit-string longest common subsequence algorithm, *Inform. Process. Lett.*, 23, 305–310, (1986).
3. K.P. Bogart, *Introductory Combinatorics*, Pitman, N.Y., (1983).
4. R. A. Baeza-Yates and G. H. Gonnet, A new approach to text searching, *Comm. Assoc. Comput. Mach.*, 35, 74–82, (1992).
5. D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Comm. Assoc. Comput. Mach.*, 18:6, 341–343, (1975).
6. D.S. Hirschberg, An information theoretic lower bound for the longest common subsequence problem, *Inform. Process. Lett.*, 7:1, 40–41, (1978).
7. V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Sov. Phys. Dokl.*, 6, 707–710, (1966).
8. U. Manber, E. Myers and S. Wu, A subquadratic algorithm for approximate limited expression matching, *Algorithmica*, 15, 50–67, (1996).
9. W.J. Masek and M.S. Paterson, A faster algorithm computing string edit distances, *J. Comput. System Sci.*, 20, 18-31, (1980).
10. E. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming, *J. Assoc. Comput. Mach.*, 46, 3, 395–415, (1999).
11. E.M. Reigold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, (1977).
12. D. Sankoff and P.H. Sellers, Shortcuts, diversions and maximal chains in partially ordered sets, *Discrete Mathematics*, 4, 287–293, (1973).
13. D. Sankoff and J.B. Kruskal (eds), *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Addison-Wesley, Reading, MA., (1983).
14. R.A. Wagner and M.J. Fischer, The string-to-string correction problem, *J. Assoc. Comput. Mach.*, 21, 1, 168–173, (1974).
15. A.H. Wright, Approximate string matching using within-word parallelism, *Soft. Pract. Exper.*, 24, 337–362, (1994).
16. S. Wu and U. Manber, Fast text searching allowing errors, *Comm. Assoc. Comput. Mach.*, 35, 83–91, (1992).