REVISTA COLOMBIANA DE COMPUTACIÓN
Volumen 3, número 1
Págs. 41-51

Recovering an LCS in $O(\frac{n^2}{w})$ Time and Space

Costas S. Iliopoulos *† Yoan J. Pinzón ‡§

Abstract

Here we make use of word-level parallelism to recover a longest common subsequence of two input strings both of length n in $O(\frac{n^2}{w})$ time and space, where w is the number of bits in a machine word. For the special case where one of the input strings is close to w its complexity is reduced to linear time and space

Keywords: *Longest Common Subsequence, Bit-parallelism.*

1 Introduction

The *Longest Common Subsequence* (LCS) of two strings is one of the main problems in combinatorial pattern matching. The LCS problem arises in a number of applications such as text editing, file comparison utilities (*e.g.* the *diff* command in UNIX[1]), file compression utilities, artificial intelligence (*e.g.* facial, handwriting, and speech recognition), molecular biology (*e.g.* DNA sequences - genes - or protein alignment, molecular sequence comparisons, study of the evolution of long molecules such as proteins), and several others. It has therefore been extensively studied in the literature on sequential and parallel algorithms.

Given a string x over an alphabet Σ, a *subsequence* of x is any string w that can be obtained from x by deleting zero or more (not necessarily consecutive) symbols. The LCS problem for strings $x = x_1x_2 \dots x_m$ and $y = y_1y_2 \dots y_n, (n \geq m)$ consists of finding a third string $w = w_1w_2 \dots w_p$ 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* [11] and the *edit distance* [20].

The LCS problem can be solved in $O(nm)$ time and space by a dynamic programming approach [18, 20]. The asymptotically fastest algorithm is due to Masek and Paterson [13] that uses the "four Russians" trick and takes $O(\frac{n^2}{\log n})$ time. Most other algorithms use either divide-and-conquer or dominant-match-point paradigms. The divide-and-conquer solutions is due to Hirschberg [8] who presented a variation of the dynamic programing algorithm using $O(n^2)$ time but only $O(n)$ space. The dominant-match-point algorithms have complexity that depends on output parameters such as r, the total number of matching pairs, and p, the length of the LCS. Hirschberg [9], presented an $O(pn)$ algorithm and, in the same year, Hunt and Szymanski [10] gave an $O(r \log n)$ algorithm. It is important to note that these two algorithms are efficient for cases where r and p are small. In the worst case

*Deptarment of Computer Science, King's College London, Strand, London, WC2R 2LS, England, csi@dcs.kcl.ac.uk, www.dcs.kcl.ac.uk/staff/csi

†Partially supported by a Marie Curie fellowship, NATO, Wellcome and Royal Society grants.

‡Laboratorio de Cómputo Especializado, Universidad Autónoma de Bucaramanga, Calle 48 # 39 - 234, Bucaramanga, Colombia, ypinzon@unab.edu.co, www.unab.edu.co/∼ypinzon

§Partially supported by an ORS studentship and EPSRC GR/L92150.

[1]UNIX is a trademark of Bell Laboratories.

$p = n$ and $r = n^2$, thus, $O(pn)$ becomes $O(n^2)$ and $O(r \log n)$ becomes $O(n^2 \log n)$ which is even worse than the dynamic programming algorithm.

The dynamic programing matrix can be parallelized to get better algorithms. The idea of 'packing' bits in a computer word to speed up algorithms has been used extensively in the last few years. The first bit-parallel algorithm was the exact string matching algorithm Shift-Or. The Shift-Or algorithm was originally presented by Baeza-Yates and Gonnet [4], where it was extended to handle multiple patterns, and mismatches. The Shift-Or algorithm was subsequently modified by Wu and Manber [21] to handle regular expressions and all these ideas were used to build a software called *agrep*. Myers [14] developed a competitive algorithm that computes the edit distance of two strings in $O(\frac{nm}{w})$ time.

Recently, Crochemore *et al.* [5, 6] and Allison and Dix [1] gave, separately, an $O(\frac{n^2}{w})$ bit-vector algorithm to compute the length of the LCS using $O(\frac{n}{w})$ space. In this paper we extend these algorithms to solve the problem of recovering the longest common subsequence instead of computing just its length. Our algorithm has $O(\frac{n^2}{w})$ time and space complexity. Thus, linear time and space for the case where the length of one of the strings is close to w.

The paper is organised as follows. In the next section we present some definitions and the basic background. In Section 3 we explain the CIPR algorithm and in Section 4 we explain how to adapt it to recover an LCS. In Section 5 we present some experimental results and in Section 6 we give our conclusions.

2 Preliminaries

Given an alphabet Σ, an element of Σ^* is called a *string* or *sequence* and is denoted by one of the letters x or y. For two sequences $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \ldots y_n$ the numbers m and n $(n \geq m)$ are called the *length* of x and y, respectively. We say that x is a *subsequence* of y and equivalently, y is a *supersequence* of x, if for some $i_1 < i_2 < \ldots < i_p$, $x_j = y_{i_j}$ where $1 \leq j \leq m$. Given a finite set of sequences, S, a *longest common subsequence* (LCS) of S is a longest possible sequence w such that each sequence in S is a supersequence of w. The LCS of two strings, x and y, is a subsequence of both x and of y of maximum possible length. The ordered pair of *positions* i and j, denoted $[i, j]$, is a *match* if and only if $x_i = y_j$. If $[i, j]$ is a match, and if an LCS w of $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$ 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 $j' \leq j$ or $i' \leq i$ and $j' > j$. A match $[i, j]$ *precedes* a match $[i', j']$ if $i < i'$ and $j < j'$. Let r be the total number of match points, and q be the total number of dominant points (all ranks). Then $0 \leq p \leq q \leq r \leq nm$. Computing the k-dominant match points is all that is needed to solve the LCS problem since the LCS of x and y has length p if and only if the maximum rank attained by a dominant match is p. Let $\mathcal{R}$ denote a partial order relation on the set of match points between x and y. A set of match points such that in any pair one of the match points always precedes the other in $\mathcal{R}$ constitutes a *chain* relative to the partial order relation $\mathcal{R}$. A set of match points such that in any pair neither element of the pair precedes the other in $\mathcal{R}$ constitutes an *antichain*. A decomposition of a partially ordered set (poset) into antichains partitions the poset into the minimum possible number of antichains. The LCS problem translates to finding a longest *chain* in the *poset* of match points induced by $\mathcal{R}$ [19].

Let $L[0..m, 0..n]$ be the *dynamic programming* matrix, where $L_{i,j}$ represents the length of the LCS (LLCS) for $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$. The following simple recurrence formula by Hirschberg [8] computes $p = L[m, n]$ in $O(nm)$ time and space.

$$L[i,j] \leftarrow \begin{cases} 0, & \text{if either } i=0 \text{ or } j=0 \\ L[i-1,j-1]+1, & \text{if } p_i = t_j \\ \max\{L[i-1,j], L[i,j-1]\}, & \text{if } p_i \neq t_j \end{cases} \quad (1)$$

In fact, only linear space is needed to find the LLCS because the computation of each row/column only requires its preceding one.

As an example, Fig. 1 shows the computation of the L-matrix 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
3	r	0	1	2	3	3	3	3	3
x 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**

Figure 1: The LCS L-matrix 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.

Throughout the paper we will make use of C-like notation for the following bit-wise operations:

bit-wise operation	C-like symbol	Example
OR	|	0101 | 1100 = 1101
AND	&	0101 & 1100 = 0100
XOR (exclusive OR)	^	0101 ^ 1100 = 1001
NOT (complement)	$\sim$	$\sim$1100 = 0011

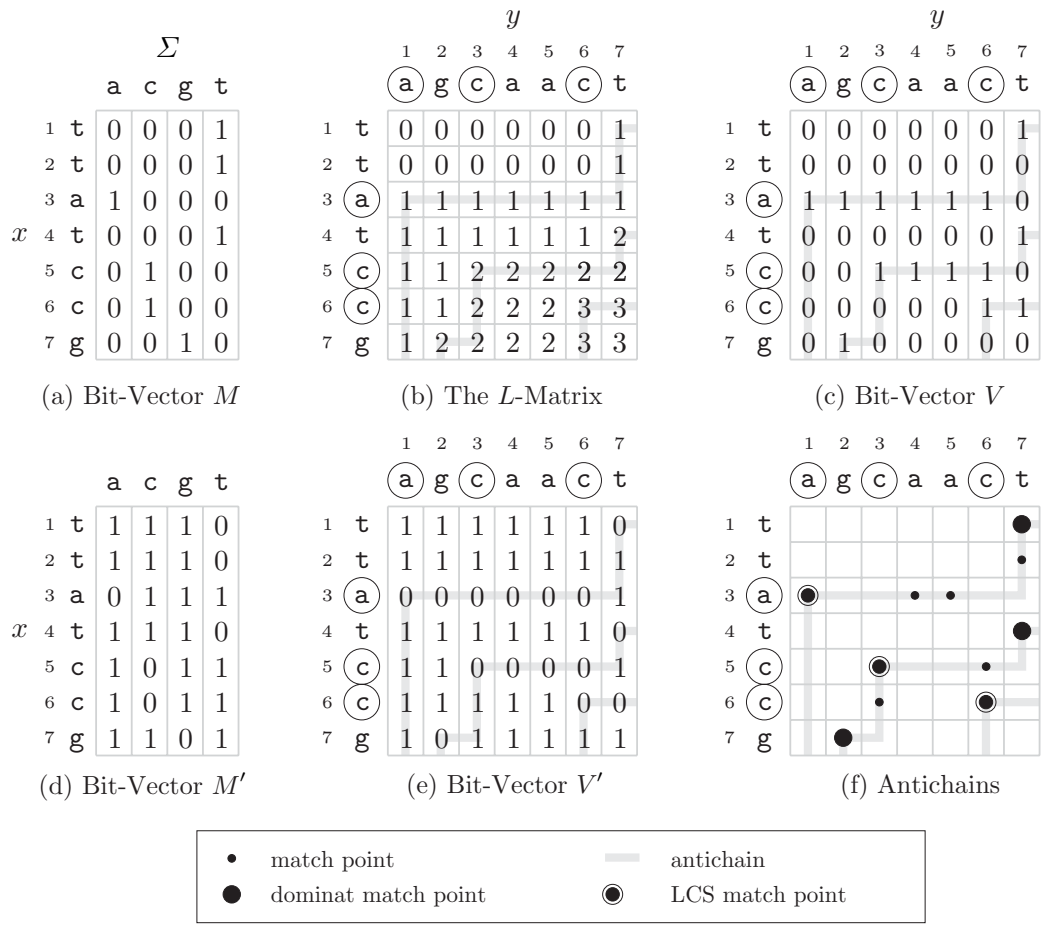
Let $A[0..n]_{0..m}$ be a *bit-vector* with n *binary words* of m bits each, where $A[j]_i \in \{1,0\}$ refers to the ith bit of the binary word $A[j]$. We also define its complement as $A'[0..n]_{0..m}$ (*i.e.* $A'[j] = \sim A[j]$ for $j \in \{0..n\}$).

3 The CIPR algorithm

In this section we show how the CIPR algorithm presented by Crochemore *et al.* [5, 6] works and in the next section we explain how to adapt it to recover an LCS.

The CIPR algorithm computes the length of the LCS of two input strings both of length n using $O(\frac{n^2}{w})$ time and only $O(\frac{n}{w})$ space. It makes use of the *monotonicity* property to store each column of the L-matrix into a bit-vector with n bit-words of m bits each, thus, using only $\Theta(n\lceil\frac{m}{w}\rceil)$[2] bits instead of $\Theta(8nm)$ bits. Let $V[0..n]_{0..m}$ be the *relative-encoding* bit-vector of the L-matrix defined as follows:

[2]Here, we assume two input strings of length n and m, $m \leq n$.



(a) Bit-Vector M (b) The L-Matrix (c) Bit-Vector V

(d) Bit-Vector M' (e) Bit-Vector V' (f) Antichains

Table 1: Matrix L and the bit-vectors M, V, M' and V' for x="ttatccg" and y="agcaact".

$$V[j]_i \leftarrow L[i,j] - L[i-1,j] \in \{0,1\} \quad \text{for } (i,j) \in \{1..m\} \times \{1..n\} \tag{2}$$

Table 1(b) and 1(c) show the L-matrix and the relative-encoding bit-vector V, respectively, for x="ttatccg" and y="agcaact".

We can think of the L-matrix as an automaton with n states where each state will be a bit-vector $V'[j]$ with m bits. The transition function to compute a new state $V'[j]$, is a set of $O(\frac{m}{w})$ bit-wise operations that depends only on the previous state $V[j-1]$ and a (preprocessed) bit-vector of the match points between the symbol y_j and x. This preprocessed bit-vector $M[1..|\Sigma|]_{0..m}$ can be computed in $O(m)$ time and is defined as follows:

$$M[\alpha]_i \leftarrow \begin{cases} 1, & \text{if } x_i = \alpha \\ 0, & \text{otherwise} \end{cases} \quad \text{for } \alpha \in \Sigma \text{ and } i \in \{1..m\} \tag{3}$$

Table 1(a) and 1(d) show the resulting bit-vectors M and M' for x="ttatccg", y="agcaact" and Σ={ 'a','c','g','t'}. Recall M' is the complement of M, *i.e.* $M'[\alpha] = \sim M[\alpha]$ for all $\alpha \in \Sigma$.

The transition function to compute the bit-vector $V'[0..n]$ is defined as follows:

$$V'[j] \leftarrow \begin{cases} 2^m - 1, & \text{for } j = 0 \\ (\hat{V} + (\hat{V} \;\&\; M[y_j])) \,|\, (\hat{V} \;\&\; M'[y_j]), & \text{for } j \in \{1..n\} \end{cases} \tag{4}$$

where $\hat{V} = V'[j-1]$. Table 1(e) shows the resulting bit-vector V' for our example. Fig. 2 depicts the automata for x="ttatccg" and y="agcaact". The numbers inside each state corresponds to the decimal value of $V'[j]$. For instance, $V'[0]$=$2^m - 1$=$2^7 - 1$=$128 - 1$=127 and $V'[2] = (0111011)_2 = 59$.

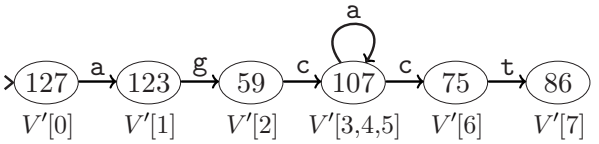


Figure 2: Automata-like representation of the L-matrix for x="ttatccg" and y="agcaact".

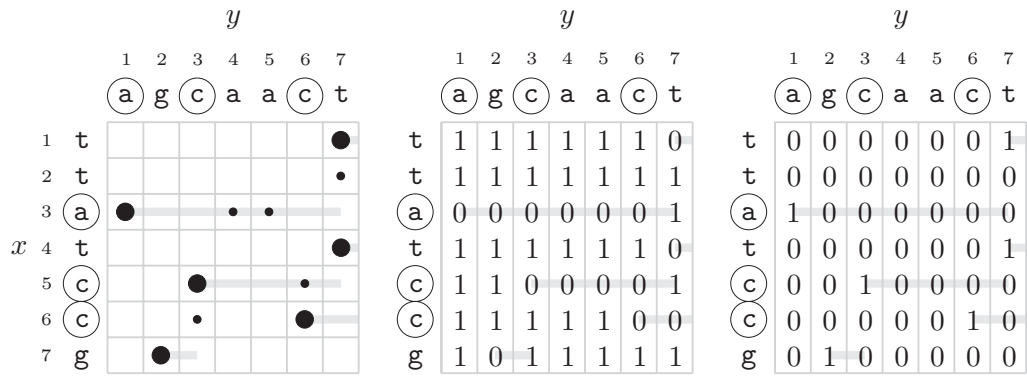
Now, let us trace the computation of $V'[3]$ (state 3) using equation (4).

$V'[2]$	0	1	1	1	0	1	1	&
$M[$'c'$]$	0	1	1	0	0	0	0	
(i)	0	1	1	0	0	0	0	
$V'[2]$	0	1	1	1	0	1	1	&
$M'[$'c'$]$	1	0	0	1	1	1	1	
(ii)	0	0	0	1	0	1	1	
$V'[2]$	0	1	1	1	0	1	1	+
(i)	0	1	1	0	0	0	0	
(iii)	1	1	0	1	0	1	1	
(iii)	1	1	0	1	0	1	1	|
(ii)	0	0	0	1	0	1	1	
$V'[3]$=(iv)	1	1	0	1	0	1	1	

After these four simple bit-wise operations (two bit-wise AND, one bit-wise OR and one bit-wise addition) we get $V'[3] = (1101011)_2 = 107$.

$V'[3]$ is the relative-encoding of the third column of the L-matrix (complement two), then, by looking at the bits in it, we know that the 1-anti-chain passes at position 3 (the first zero counting from left to right) and the 2-anti-chain passes at position 5 (the second zero). We also know that LLCS=2 by counting the number of zeros.

$$\overbrace{V'[3] \leftarrow (\, V'[2] + \overbrace{(V'[2] \;\&\; M[y_3 = \text{'c'}])}^{(i)} \,) \;|\; \overbrace{(V'[2] \;\&\; M'[y_3 = \text{'c'}])}^{(ii)}}^{(iii)}{}^{(iv)}$$



(a) Horizontal Antichains (b) Bit-Vector V' (c) Bit-Vector K

Figure 3: Computation of bit-vector K for x="ttatccg" and y="agcaact".

4 The New Bit-Vector Algorithm

In this section we extend the CIPR algorithm presented in the previous section so that it computes the length of the LCS and recovers it. Unfortunately, the bit-vector V' in not all that is needed, we also need the exact location of all the k-dominant match points. We now define and show how to compute the *k-dominant matches* bit-vector K

$$K[j]_i \leftarrow \begin{cases} 0, & \text{if } [i,j] \text{ is a } k\text{-dominant match} \\ 1, & \text{otherwise} \end{cases} \qquad \text{for } [i,j] \in \{1..m\} \times \{1..n\} \tag{5}$$

This bit-vector can be computed using the information in V'. V' contains the encoding of the horizontal part of all the anti-chains (refer to Fig. 3). To compute all the k-dominant match points (those matches that are positioned at the convex corners of each anti-chain if viewed from point $V'[n]_m$) we just need to find those locations where there is a horizontal change of bits from 1 to 0. *i.e.* All $[i,j]$ such that $V'[j-1]_i = 1$ and $V'[j]_i = 0$. For instance (in Fig. 3(b)), [1,6]→[1,7], [3,0]→[3,1], [4,6]→[4,7], [5,2]→[5,3], [6,5]→[6,6] and [7,1]→[7,2]. This can be accomplished by `XOR`-ing $V[j-1]$ with $V[j]$. However, we will get some false matches, namely, those locations where there is a change from 0 to 1. We can fix this problem by `AND`-ing it with $V'[j-1]$. Therefore, we can get K as follow:

$$K[j] \leftarrow (V'[j-1] \hat{} V'[j]) \;\&\; V'[j-1] \quad \text{for } j \in \{1..n\} \tag{6}$$

Now that we have bit-vectors V' and K, we can proceed with the design of the new algorithm.

The main idea to solve the problem is quite simple: if we make sure that each anti-chain is contributing with exactly one k-dominant match point then we can be sure that the resulting set of selected matches conform an LCS. We go ahead as follows: First, we would like to identify a p-dominant match point (recall p is the length of the LCS). Let $[i', j']$ be that match. So we can be sure that there is not any anti-chain crossing the area γ' defined by the square $\{[i', j'], [m, n]\}$ (refer to Fig. 4). The next step is to find a $(p-1)$-dominant match $[i'', j'']$ such that there are not anti-chains crossing the area γ'' defined by the square $\{[i'', j''], [i', j']\}$. If we repeat this process p times until we find the 1-dominant match point $[i^{(p)}, j^{(p)}]$, all the discovered matches correspond to an LCS. The pseudo-code in Figure 5 uses this idea to recover the LCS of two given strings x and y. We use masking to select the

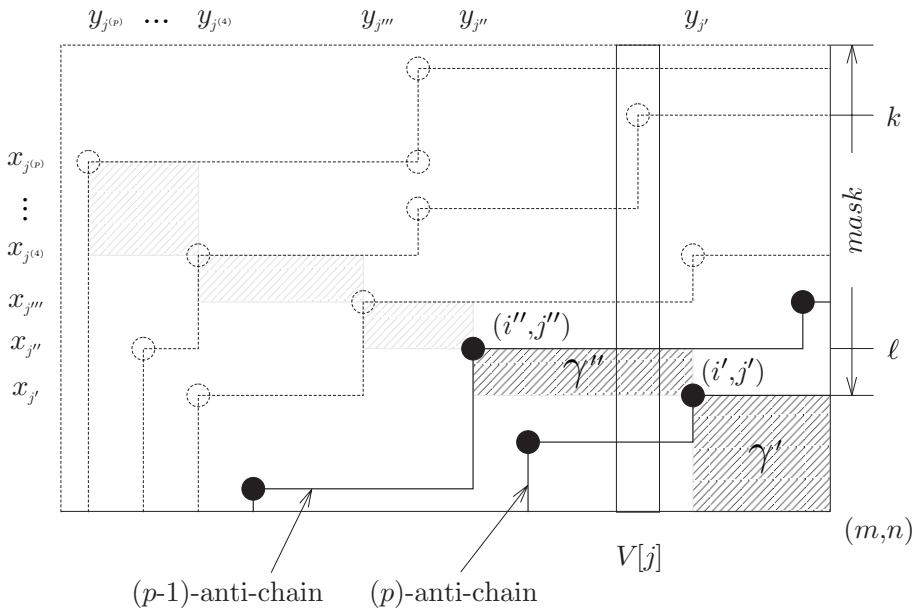


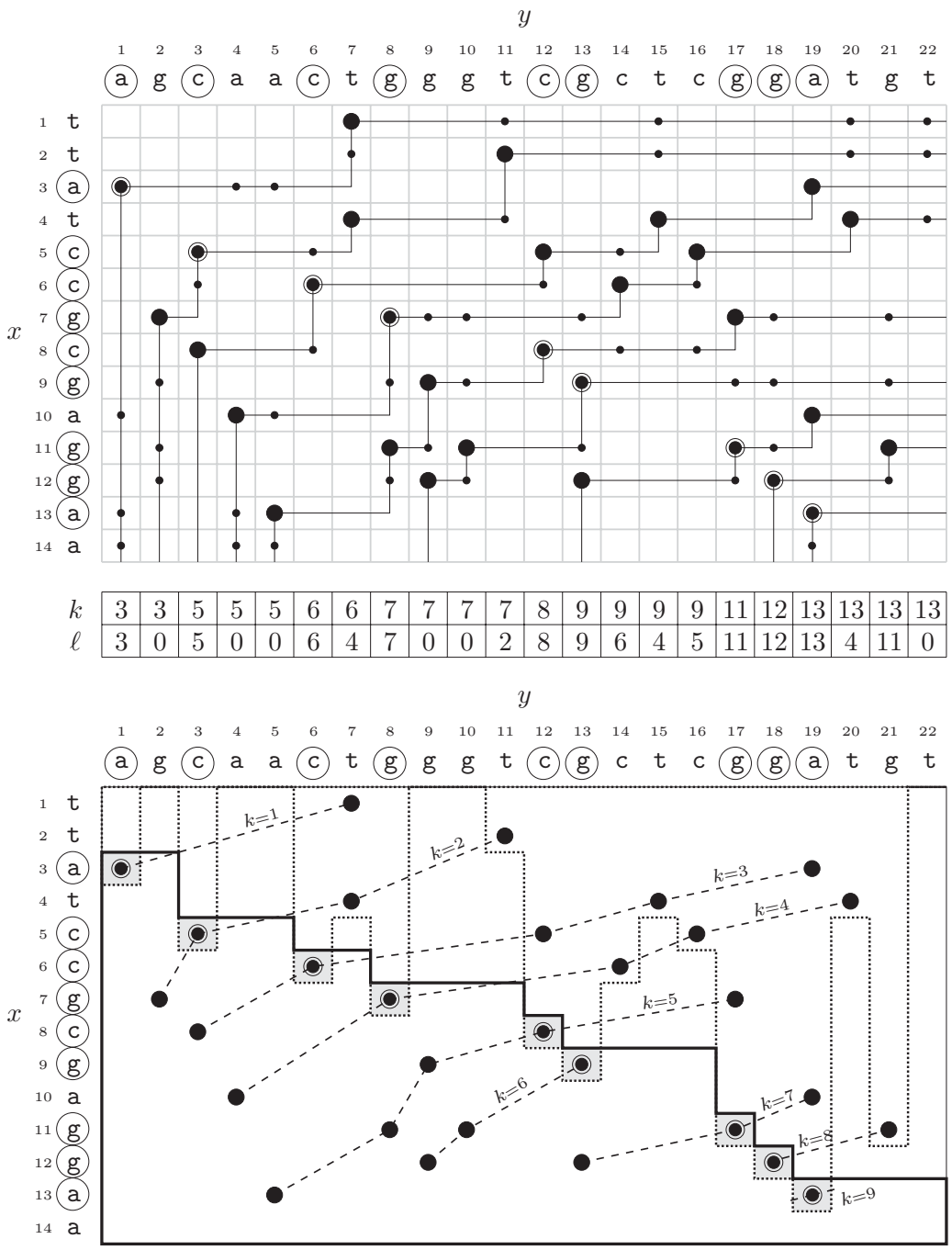
Figure 4: Illustration of BV-LCS algorithm computation.

bits of *interest* and to ignore the others. At the beginning all the bits are of interest and that is the reason way in line 1 we set $mask = 2^m - 1$. Let us assume that we want to consider $V[j]$ in Fig. 5. $mask$ will take care of ignoring all the bits from m to i'. $Vmasked$ contains the valid bits from i' up to 1. We want to know if the last bit of the bit-words $Vmasked$ and $Vmasked'$ ($Vmasked' = Vmasked$ & $K[j]$) are the same. If they are different we just move to the next state ($V[j-1]$), otherwise (they are the same) we report (k, j) as an LCS match point.

Fig. 6 shows a full example. An LCS match point is reported whenever k and ℓ are the same.

```
BV-LCS(x, y, m, n, V', K)
1  mask ← 2^m − 1
2  for j ← n downto 1
3       do Vmasked ← V'[j] & mask
4          Vmasked' ← Vmasked & K[j]
5          k ← lastbit(Vmasked)
6          ℓ ← lastbit(Vmasked')
7          if k = ℓ and Vmasked' > 0
8            then mask ← 2^k − 1
9                 REPORT(k, j)
```

Figure 5: BV-LCS algorithm.



k	3	3	5	5	5	6	6	7	7	7	7	8	9	9	9	9	11	12	13	13	13	13
ℓ	3	0	5	0	0	6	4	7	0	0	2	8	9	6	4	5	11	12	13	4	11	0

Figure 6: Illustration of the computation of the BV-LCS algorithm for Σ={'a','c','g','t'}, x="agcaactgggtcgctcggatgt" and y="ttatccgcgaggaa". The table on the top presents all the k-dominant match points and all the k-anti-chains. The middle table shows the values for k and ℓ computed by the BV-LCS algorithm. The bottom table shows these values graphically. The dotted line represents the values for ℓ and the bold line the values for k. Whenever they intersect (grey boxes) we have an LCS match point member. The final discovered LCS string for the input strings is w="accgcggga" of length 9.

5 Experimental Results

We implemented algorithm BV-LCS and we compared it with two classic algorithms. More precisely, we compared the following algorithms: (1) The unrestricted model of BV-LCS algorithm (unrestricted means that n can be of any size, *i.e.* $n > w$). (2) The basic Hirschberg algorithm (H) [8]: This algorithm is a clever modification of the dynamic programing algorithm using $O(n^2)$ time but only $O(n)$ space. (3) The basic Hunt-Szymanski algorithm (HS) [10]: This algorithm runs in $O((r+n)\log n)$ time and $O(r+n)$ space. Clearly, this algorithm is efficient for applications where r, the number of match points, is small. To have this fact into account, we used an alphabet size of 100 for the HS algorithm and size 4 for the others. All these algorithms were implemented in C++ and run on a SUN Ultra Enterprise 300MHz running Solaris Unix with a 32-bits word. Fig. 7 shows the results. BV-LCS algorithm is always superior for the values used in this experiment. However, there might be special cases where HS outperforms BV-LCS algorithm, namely, when the number of match points between the sequences is significantly small.

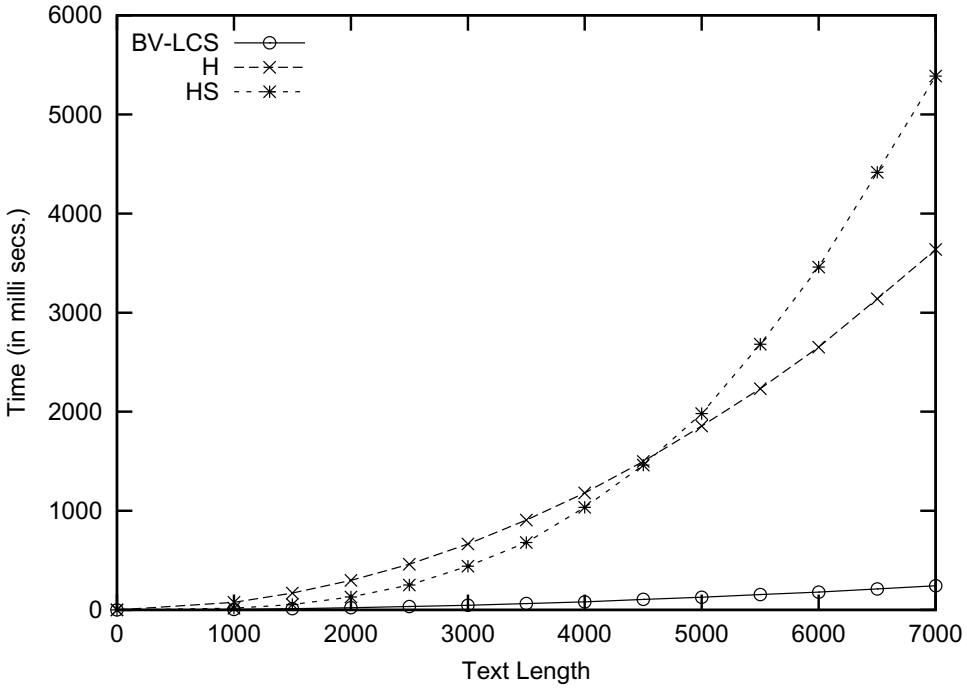


Figure 7: Timing curves for BV-LCS, H and HS algorithms.

6 Conclusion

We have presented a new algorithm based on bit-parallelism that recovers an LCS of two input strings both of length n in $O(\frac{n^2}{w})$ time and space. This algorithm behaves optimally regardless of other input/output parameters such as r, the number of match points and p, the length of the LCS. This makes it a very good choice for those cases where we have

insufficient knowledge of the nature of the sequences to be compared, before hand. More study needs to be carried out to, firstly, improve its space complexity, and secondly, find suitable applications where it can be used. For instance, it seems that we could use the BV-LCS algorithm to speed-up the basic Hirschberg's divide-and-conquer algorithm by splitting the text in block of size w and then run the BV-LCS algorithm on those blocks. We might get a significant improvement in its running time while using the same space complexity (*i.e.* linear space).

References

[1] L. Allison and T.L. Dix, A bit-string longest common subsequence algorithm, *Inform. Process. Lett.*, 23, 305–310, (1986).

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

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

[4] R. A. Baeza-Yates and G. H. Gonnet, A new approach to text searching, *Comm. Assoc. Comput. Mach.*, 35, 74–82, (1992).

[5] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon and J. F. Reid, A fast bit-vector algorithm for the longest common subsequence problem, *Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms AWOCA'00*, 74–82, (2000).

[6] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon and J. F. Reid, A fast bit-vector algorithm for the longest common subsequence problem, *Inform. Process. Lett.*, to appear, (2001).

[7] V. Dančík, Expected length of the longuest common subsequences, *PhD thesis*, University of Warwick, (1950).

[8] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Comm. Assoc. Comput. Mach.*, 18:6, 341–343, (1975).

[9] D.S. Hirschberg, Algorithms for the longest common subsequence problem, *J. Assoc. Comput. Mach.*, 24:4, 664–675, (1977).

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

[11] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Sov. Phys. Dokl.*, 6, 707–710, (1966).

[12] U. Manber, E. Myers and S. Wu, A subquadratic algorithm for approximate limited expression matching, *Algorithmica*, 15, 50–67, (1996).

[13] W.J. Masek and M.S. Paterson, A faster algorithm computing string edit distances, *J. Comput. System Sci.*, 20, 18-31, (1980).

[14] E. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming, *J. Assoc. Comput. Mach.*, 46:3, 395–415, (1999).

[15] G. Navarro and M. Raffinot, A bit-parallel approach to suffix automata: fast extended string matching, *Combinatorial Pattern Matching*, LNCS 1448, 14–33, (1998).

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

[17] M. Paterson, V. Dančík, Longest common subsequence, *Proceedings of the 19th Intern. Symp. on Mathematical Foundations of Computer Science*, LNCS 841, 127–142, (1994).

[18] 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).

[19] D. Sankoff and P.H. Sellers, Shortcuts, diversions and maximal chains in partially ordered sets, *Discrete Mathematics*, 4, 287–293, (1973).

[20] R.A. Wagner and M.J. Fisher, The string-to-string correction problem, *J. Assoc. Comput. Mach.*, 21:1, 168–173, (1974).

[21] S. Wu and U. Manber, Fast text searching allowing errors, *Comm. Assoc. Comput. Mach.*, 35, 83–91, (1992).