Speeding-up Hirschberg and Hunt-Szymanski LCS Algorithms

Maxime Crochemore
Institut Gaspard-Monge
Université de Marne-la-Vallée
F-77454 Marne-la-Vallée CEDEX 2, France
mac@univ-mlv.fr

Costas S. Iliopoulos *
Dept. of Computer Science
King's College London
London WC2R 2LS, England
csi@dcs.kcl.ac.uk

Yoan J. Pinzon †
Dept. of Computer Science
King's College London
London WC2R 2LS, England
pinzon@dcs.kcl.ac.uk

Abstract

Two algorithms are presented that solve the problem of recovering the longest common subsequence of two strings instead of just its length. The first algorithm is an improvement of Hirschberg's divide-and-conquer algorithm. The second algorithm is an improvement of Hunt-Szymanski algorithm based on an efficient computation of all dominant match points. These two algorithms use bit-vector operations and are shown to work very efficiently in practice.

Keywords: Longest Common Subsequence, Bit-parallelism.

1. Introduction

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 *Longest Common Subsequence* (LCS) problem for strings $x = x_1x_2\ldots x_m$ and $y = y_1y_2\ldots y_n, (n \geq m)$ consists of finding a third string $w = w_1w_2\ldots w_p$ 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* [10] and the *edit distance* [18].

The LCS problem can be solved in $O(nm)$ time and space by a dynamic programming approach [16, 18]. The asymptotically fastest algorithm is due to Masek and Paterson [11] 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 [7] 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 [8], presented an $O(pn)$ algorithm and, in the same year, Hunt and Szymanski [9] gave an $O(r \log n)$ algorithm. It is important to note that these two algorithms are efficient, particularly for cases when r and p are small. In the worst case $p = n$ 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.

As pointed out in [13] we have to select one of the algorithms *a priori* depending on the kind of sequences we wish to compare. This might prove to be difficult because of insufficient knowledge of the nature of sequences to be compared; or the class of sequences is so heterogeneous that both short and long longest common subsequences are likely to occur; furthermore, when dealing with sequences of approximately equal length consisting of symbols drawn from a small alphabet in a more or less uniform manner, the length of the LCS can be expected to lie in the range between $\frac{1}{3}n$ and $\frac{2}{3}n$, depending on the alphabet size [6, 15, 16]. The bit-vector algorithms presented in this paper has the advantage of not being input- or output-sensitive, *i.e.* it is independent on any parameter other than the length of the sequences. The idea of 'packing' bits in a computer word to speed up algorithms has been used extensively

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

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

in the last few years. One of the simplest and best known algorithm is the Shift-And algorithm, originally by Baeza-Yates and Gonnet [3] and subsequently modified by Wu and Manber [19], that solves the exact pattern matching problem in $O(\frac{nm}{w})$, where n and m are the length of the two input strings and w the number of bits in a machine word (normally 32 or 64). Navarro and Raffinot [14] obtained a fast exact matching algorithm combining bit-parallelism and suffix automata. Recently, Myers [12] developed a competitive algorithm that computes the edit distance of two strings in $O(\frac{nm}{w})$ time.

Crochemore *et al.* [4, 5] gave an $O(\frac{n^2}{w})$ algorithm to compute the length of the LCS using $O(\frac{n}{w})$ space. In this paper we extend this algorithm to obtain faster divide-and-conquer Hirschberg algorithm and Hunt-Szymanski algorithm. We show that these improvements are of practical interest and outperform the basic algorithms.

The paper is organised as follows. In the next section we present some definitions and the basic background needed to understand most of the paper. In Section 3 and 4 we describe how to speed up Hirschberg algorithm and Hunt-Szymanski algorithm, respectively. In Section 5 we show the experimental results and in Section 6 we present our conclusions.

2 Basic Background

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 $i_1 < i_2 < \ldots < i_p$, $x_j = y_{i_j}$. 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 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 $\mathcal{R}$ [17].

Let $L[0..m, 0..n]$ be the *dynamic programming* matrix, where $L_{i,j}$ represents the length of the LCS for $x1x2 \ldots x_i$ and $y_1y_2 \ldots y_j$. Matrix L can be computed with the following recursive formula [7]:

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

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 $j \in \{0..n\}$), where '$\sim$' refers to the bit-wise NOT operation.

The *monotonicity* property in L-matrix allows us to store each column in the L-matrix using bit-vectors. Let $V[0..n]_{0..m}$ be the *relative-encoding* bit-vector of the L-matrix defined as follows:

$$\begin{array}{c} V[j]_i = L[i,j] - L[i-1,j] \in \{0,1\} \\ \text{for } (i,j) \in \{1..m\} \times \{1..n\} \end{array} \tag{2}$$

Let $M[1..|\Sigma|]_{0..m}$ be the *preprocessing* bit-vector defined as follows:

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

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 is a set of $O(1)$ bit-wise operations that depends on the previous state $V[j-1]$ and a (preprocessed) bit-vector of the match points in the current state $M[y_j]$. $V'[0..n]$ can be obtained as follows [5]:

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

where '&' corresponds to the bit-wise AND operation and '|' to the bit-wise OR operation as in the C language.

Table 1 shows matrix L and the bit-vectors M, V, M' and V' for x=“`ttatccg`”, y=“`agcaact`” and Σ={ '`a`','`c`','`g`','`t`'}.

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

(a) Bit-Vector M

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

(b) Bit-Vector M'

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

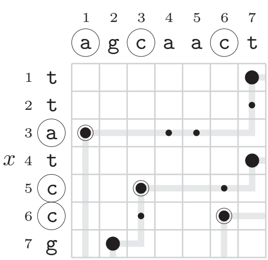
(c) The L-Matrix

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

(d) Bit-Vector V

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

(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".

3 Speeding-up the Hirschberg Algorithm

Here we will make use of word-level parallelism to speed up Hirschberg algorithm (H) [7]. We first explain how the basic H algorithm works and then present our contribution.

The main idea behind Hirschberg algorithm is to first determine the middle point of an LCS "curve" and then, recursively, determine the quartile points. See Fig. 1 (top). An efficient way to determine the middle point of an LCS curve is to use the linear space FINDROW algorithm (presented in Fig. 2) as follows:

- Use FINDROW to compute the middle row $midrow$ of the L-matrix.
- Use FINDROW to compute the middle row $midrow^R$ of the L^R-matrix, for the reverses of string x and y.
- Determine k in the range $0..n$ that maximizes $midrow[k] + midrow^R[n-k]$.

It can be shown that an optimal LCS curve intersects with the middle row at k. Once the middle point has been determined we apply this procedure recursively to quartiles $\{[0,0],[\lceil \frac{m}{2} \rceil,k]\}$ and $\{[\lceil \frac{m}{2} \rceil,k], [n,m]\}$. (see Fig. 1). Each iteration uses linear space but the total time used is about double what it was before. Fig. 2 shows the details of the algorithm.

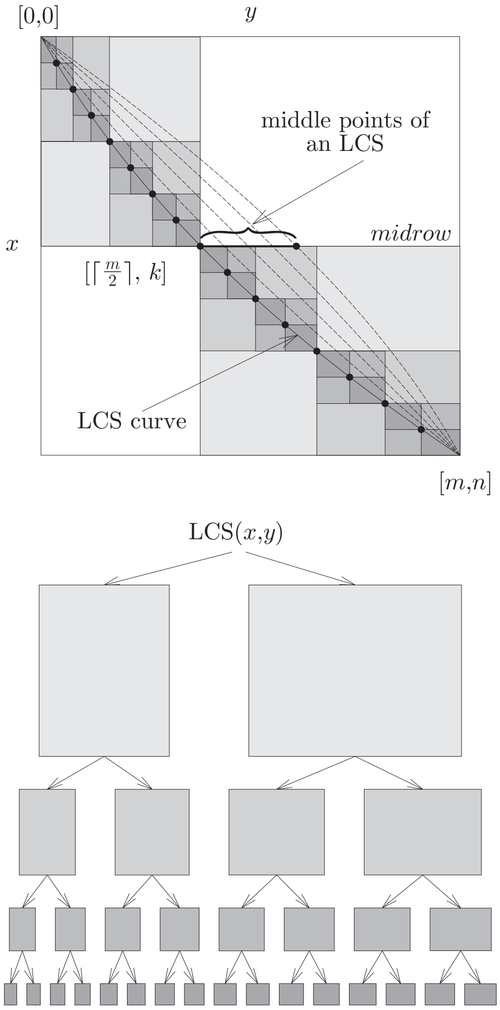


Figure 1. Illustration of the divide-and-conquer paradigm (top), and its recursive calls (bottom).

As an example, Table 2 shows the computation of the LCS for x="ttatccgggtgaga" and y="agcaactgtctaca". Fig. 3 shows all the recursive calls needed to recover the LCS for this example.

```
FINDROW(x, y, m, n, L)
L[0 : n] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then Lnew[j] ← L[j − 1] + 1
            else  Lnew[j] ← max(Lnew[j − 1], L[j])
    for j ← 1 to n
        do L[j] ← Lnew[j]
```

```
BV-FINDROW(x, y, m, n, L)
M[α] ← 0 for all α ∈ Σ
for i ← 1 to m
    do M[x_i]_i ← 1
M' ← ∼M
S ← 2^m − 1
for j ← 1 to n
    do S ← (S + (S & M[y_j])) | (S & M'[y_j])
       if S_{m+1} = 1
          then L[j] ← L[j − 1] + 1
          else  L[j] ← L[j − 1]
```

```
H(x, y, m, n, C, p)
 1  if n = 0
 2    then p ← 0
 3    else  if m = 1
 4             then if ∃j with x_i = y_j
 5                     then p ← 1
 6                          C[1] ← x_1
 7                     else  p ← 0
 8             else  i ← ⌈m/2⌉
 9                   FINDROW(x, y, i, n, L)
10                   FINDROW(x^R, y^R, m − i, n, L^R)
11                   k ← 0
12                   max ← 0
13                   for ℓ ← 0 to n
14                       do if L[ℓ] + L^R[n − ℓ] > max
15                             then max ← L[ℓ] + L^R[n − ℓ]
16                                  k ← ℓ
17                   H(x, y, i, k, C, q)
18                   H(x_{i+1..m}, y_{k+1..n}, m − i, n − k, D, r)
19                   p ← q + r
20                   for ℓ ← 1 to r
21                       do C[q + l] ← D[ℓ]
```

Figure 2. FINDROW, BV-FINDROW **and** H **algorithms.**

y

$x_{1..7}$		1 a	2 g	3 c	4 a	5 a	6 c	7 t	8 g	9 t	10 c	11 t	12 a	13 c	14 a	
1	t	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
2	t	0	0	0	0	0	0	1	1	2	2	2	2	2	2	
3	a	1	1	1	1	1	1	1	1	2	2	2	3	3	3	
4	t	1	1	1	1	1	1	2	2	2	2	3	3	3	3	
5	c	1	1	2	2	2	2	2	2	2	3	3	3	4	4	
6	c	1	1	2	2	2	3	3	3	3	3	3	3	4	4	
7	g	1	2	2	2	2	3	3	4	4	4	4	4	4	4	$midrow$ (L)

y^R

$x^R_{1..7}$		1 a	2 g	3 c	4 a	5 a	6 c	7 t	8 g	9 t	10 c	11 t	12 a	13 c	14 a	
1	a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
2	g	1	1	1	1	1	1	2	2	2	2	2	2	2	2	
3	a	1	1	2	2	2	2	2	2	2	3	3	3	3	3	
4	g	1	1	2	2	2	2	3	3	3	3	3	3	4	4	
5	t	1	1	2	3	3	3	3	4	4	4	4	4	4	4	
6	g	1	1	2	3	3	3	4	4	4	4	4	4	5	5	
7	g	1	1	2	3	3	3	4	4	4	4	4	4	5	5	$midrow^R$ (L^R)

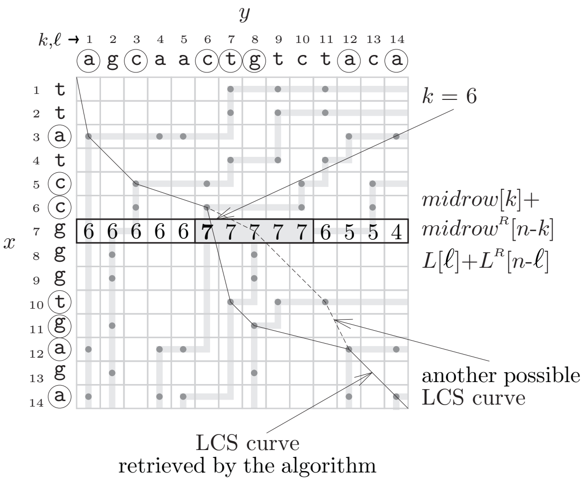


Table 2. Example for the computation of the middle point of an LCS curve for x**=**"ttatccgggtgaga" **and** y**=**"agcaactgtctaca" **using Hirschberg algorithm.**

As we said, each iteration uses linear space but the total time used is about double what it was before. This is due to the fact that we need to compute L-matrix and L^R-matrix at the same time. Thus, an improvement in FINDROW algorithm will automatically speed up H algorithm. Recently, a fast bit-vector algorithm that computes the length of an LCS was contrived [4, 5]. We use this algorithm to obtain a fast version of FINDROW algorithm, namely, the BV-FINDROW algorithm (BV stands for Bit-Vector). The space complexity is kept linear while its time complexity is improved by a factor of w (32 or 64, typically). The algorithm is based on the following observations: The bit $V'[j]_{m+1}$ in equation (4) is 1 if a new antichain is encountered, 0 otherwise. $L[j, m]$ is equal to $L[j-1, m]$ plus 1, if an antichain is encountered, 0 otherwise. See *e.g.* Table 1(c). So, we can say that $L[j, m]$ is equal to $L[j-1, m]$ plus 1, if there was a carry ($V'[j]_{m+1}$=1) computing $V'[j]$ in (4), 0 otherwise. To check if there is a carry can be done in many different ways in $O(1)$ time. Fig. 2 shows the new BV-FINDROW algorithm.

Having designed BV-FINDROW algorithm, just rest to produce the BV-H algorithm from H algorithm by exchanging FINDROW in lines 9 and 10 for our BV-FINDROW algorithm. Since the code for BV-H algorithm is basically

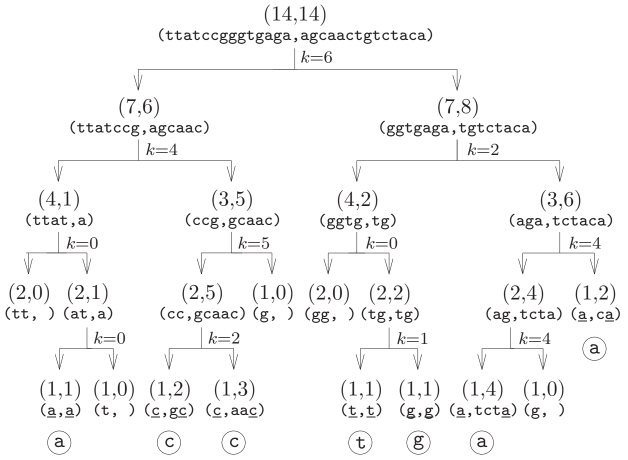


Figure 3. Example of all divide-and-conquer calls performed by Hirschberg algorithm for x=“ttatccgggtgaga” and y=“agcaactgtctaca”. The circled letters represent the recovered LCS w=“acctgaa”.

the same as H algorithm we do not show the algorithm in full detail.

4 Speeding-up the Hunt-Szymanski Algorithm

In this section we will also make use of word-level parallelism to speed up Hunt-Szymanski algorithm (HS) [9]. HS algorithm solves the problem of recovering an LCS 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. In particular, this algorithm has at least two applications. The main one is file comparison utilities such as the *diff* command in UNIX (UNIX is a trademark of Bell Laboratories). It can also be used for finding the longest ascending subsequence of a permutation of integers $\{1..n\}$ in which each letter occurs at most once. We first explain how the basic algorithm works and then we explain how to speed up this algorithm by using bit-wise operations.

The HS algorithm works by computing all dominant match points in a row-wise manner from top to bottom. For each i, the ordered list $MATCHLIST(i)$ is set to contain the descending sequence of positions $j_1 < j_2 \ldots < j_d$ for which $x_i = y_{j_\ell}$ for $\ell \in \{1..d\}$ (see Table 3(c)). This initialization process can be performed in $O(n \log n)$ by sorting y into $\hat{y}$ using the symbols in y as the first key (in ascending order) and the original position of each element as the second key (in descending order). $MATCHLIST(i)$ can now be implemented whit a count of the size and a pointer to the first element in $\hat{y}$ having symbol x_i. Then, iteratively for each row i, the algorithm evaluates the $threshold$ function $T(i,k)$ defined as follows:

$$T[i,k] = \begin{cases} \text{smallest } j \text{ such that } x_i = y_j \text{ and} \\ T[i-1,k-1] < j \le T[i-1,k] \\ T[i-1,k], \quad \text{if no such } j \text{ exists} \end{cases} \qquad (5)$$

In plain English, $T[i,k]$ corresponds to the position j where the k-antichain cuts row i (see Table 3(b)). By maintaining the T values in an one-dimensional array $THRESH$, for each j in $MATCHLIST(i)$, the k for which $T(i,k)$ differs from $T(i-1,k)$ can be determined in $O(\log n)$ time by using binary search on the $THRESH$ array. Fig. 4 shows the algorithm in full detail.

(a) L-matrix

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

(b) $THRESH$

	1	2	3	4	5	6	7
1	7	-	-	-	-	-	-
2	7	-	-	-	-	-	-
3	1	-	-	-	-	-	-
4	1	7	-	-	-	-	-
5	1	3	-	-	-	-	-
6	1	3	6	-	-	-	-
7	1	2	6	-	-	-	-

(c) $MATCHLIST$

	1	2	3
1	7		
2	7		
3	5	4	1
4	7		
5	6	3	
6	6	3	
7	2		

(d) $KMATCHLIST$

	1
1	7
2	
3	1
4	7
5	3
6	6
7	2

Table 3. Computation of $THRESH$ and $MATCHLIST$ for x=“ttatccg" and y=“agcaact".

Now we should give careful thought to the difference between the number of dominant match points (q) and the number of match points (r). The number of dominant match points is always a subset of the number of match points. See *e.g.* Fig. 5 for an experimental visualization of this fact.

We already know that there is not a general relationship between q and r. While for $x = y = a^n$, $r = n^2$ and $q = n$ (see Fig. 5(c)), for $x = (abc)^{\frac{n}{3}}$ and $y = (acb)^{\frac{n}{3}}$, $r = q = O(n^2)$ (see Fig. 5(d)). Despite that, in practice q can be about 40, 55, 70 and even 80 percent smaller than r

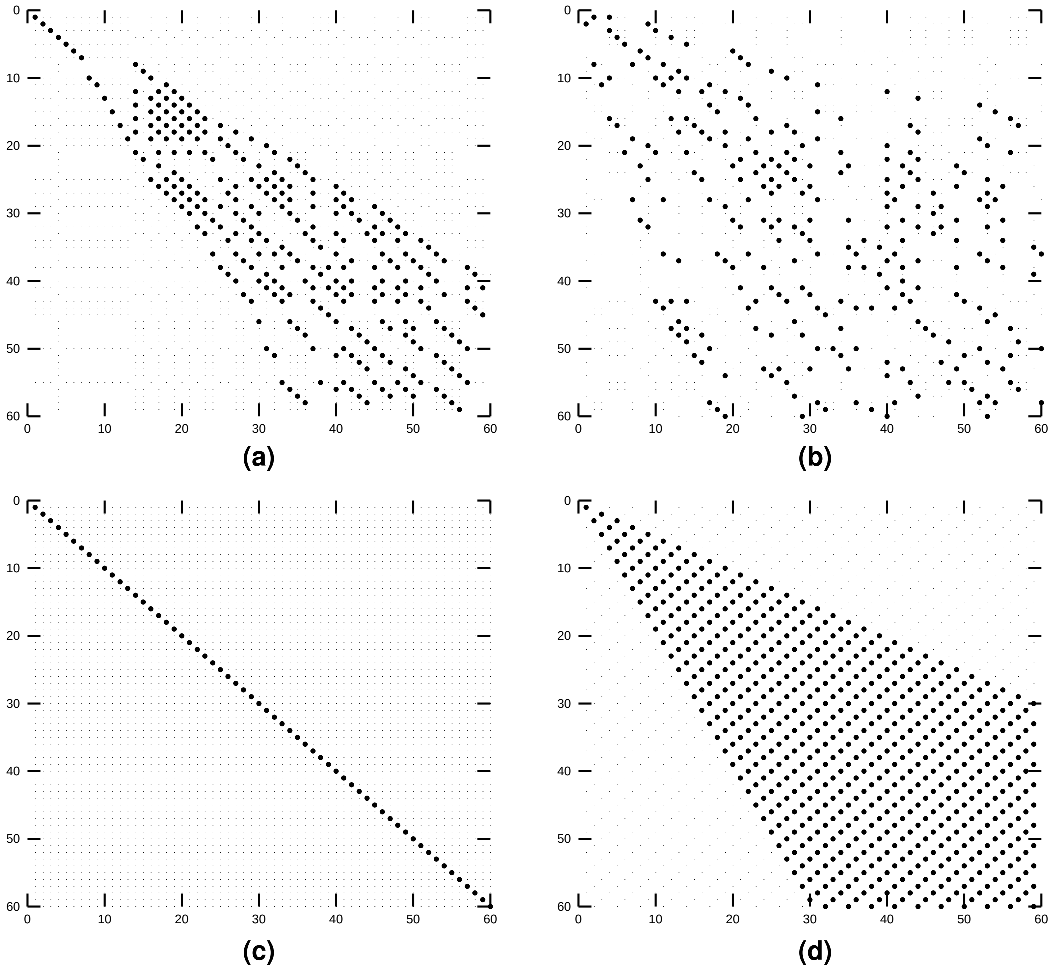


Figure 5. Dominant match points (black circles) versus match points (dots) for (a) random text with $|\Sigma|$=2, (b) random text with $|\Sigma|$=16, (c) special case for x=y=a^{60} and (d) special case for $x = abc^{20}$ and $y = acb^{20}$.

for random sequences drawn from an alphabet of size 2, 4, 8 and 16, respectively. See Fig. 6.

This means that if we can efficiently compute all the dominant match points, then it would be possible to speed up HS algorithm by substituting $MATCHLIST$ for the list of dominant match points $KMATCHLIST$. Apostolico [1, 2] exploits this notion and gave an $O((q + n)\log n)$ variant of the $O((r + n)\log n)$ HS algorithm. However, the gain is obtained at expense of complicated data structures such as balanced binary search trees and therefore imposing a large multiplicative constant that makes this improvement of theoretical interest only.

Here, we propose a new practical bit-vector algorithm to compute all the dominant match points in $O(q)$ and therefore speeding up HS algorithm. The algorithm is based on the following observations:

A match point $[i, j]$ in L-matrix is a k-dominant match point if and only if

$$k = L[i, j] = L[i-1, j-1]+1 = L[i-1, j]+1 = L[i-1, j] \tag{6}$$

or

$$V'[j-1]_i = 1 \text{ and } V'[j]_i = 0 \tag{7}$$

by using bit-vector V' (complement of the relative-encoding of L-matrix). Thus, we can identify all the dominant match points if we can find a bit-vector formula that detects all locations where there is a *horizontal* (from left to right) change of bits from 1 to 0. *e.g.* in Table 1(e) the dominant match points for the string x="ttatccg" and y="agcaact" are

$$[1, 7],\ [3, 1],\ [4, 7],\ [5, 3],\ [6, 6] \text{ and } [7, 2]$$

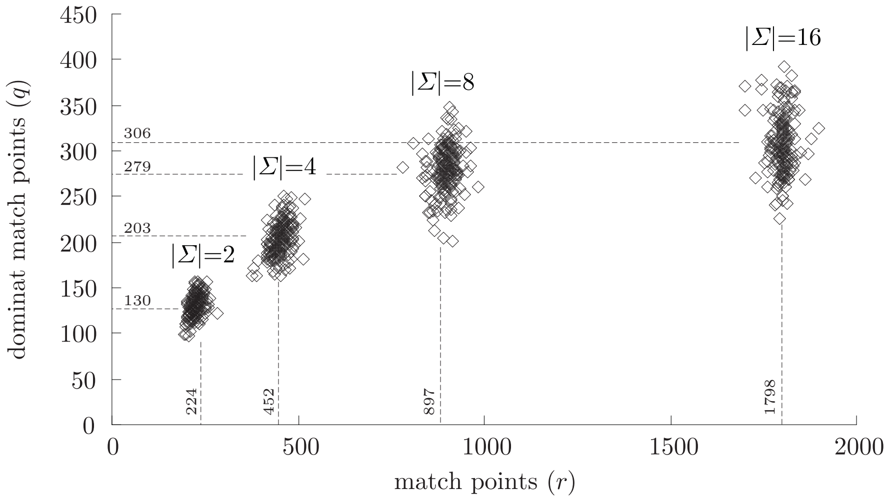


Figure 6. Dominant match points versus match points.

```
HS(x, y, m, n, MATCHLIST, C, p)
p ← 0
THRESH[0] ← 0
LINK[0] ← null
for i ← 1 to m
    do THRESH[i] ← n + 1
       LINK[i] ← null
for i ← 1 to m
    do for each j in MATCHLIST[i]
          do find k such that THRESH[k − 1] < j ≤ THRESH[k]
             if j < THRESH[k]
                then THRESH[k] ← j
                     LINK[k] ← newnode(i, LINK[k − 1])
t ← largest k such that THRESH[k] ≠ n + 1
temp ← LINK[t]
while temp.next ≠ null
    do p ← p + 1
       D[p] ← x_temp.i
       temp ← temp.next
C ← D^R
```

Figure 4. HS **LCS algorithm.**

To detect all the $1 \to 0$ changes, we can use the following bit-vector formula:

$$(V'[j-1] \hat{\ } V'[j]) \ \& \ V'[j-1] \qquad (8)$$

where '$\hat{\ }$' denotes the XOR bit-wise operation. The first part of the formula detects whether there is a change of bits (either $0 \to 1$ or $1 \to 0$). The second part makes sure that only $1 \to 0$ changes are considered. The code in Fig. 7 (BV-KMATCHLIST algorithm) uses formula (8) to compute all the dominant match points of two given sequences x and y.

In spite of this, BV-KMATCHLIST algorithm is not useful as it is. That is because HS algorithm requires all the match points to be sorted from top to bottom and from right to left if there is more than one match point in a given row. The $KMATCHLIST$ computed by BV-KMATCHLIST algorithm sort all the dominant math points from left to right and from bottom to top if there is more than one dominant match point within the same column. This is exactly all the opposite. We could sort $KMATCHLIST$ in $O(q \log q)$ time but there is another way around. HS algorithm works in a row-wise manner. *i.e.* it discovers all the dominant match points for each row from top to bottom. We can change HS algorithm to work column-wise instead. If that is the case, then $KMATCHLIST$ does not need to be sorted and even more important, does not need to be stored. This is possible if we include BV-KMATCHLIST algorithm inside HS algorithm and update the threshold array accordingly, immediately after a dominant match point is found. BV-HS algorithm (see Fig. 7) implements all these ideas.

5 Experimental Results

We implemented algorithms H, HS, BV-H and BV-HS in C++ and run on a SUN Ultra Enterprise 300MHz running Solaris Unix with a 32-bits word. We used random text with uniformly distributed alphabets of sizes from 2 to 100. Fig. 8 (top) shows the results for long strings ($1000 < n = m < 7000$). H and BV-H algorithms were not sensitive to changes in the alphabet size. Fig. 8 (bottom) shows the results for HS and BV-HS algorithm using different alphabet sizes and small strings ($100 < n = m < 1000$). We only consider the *unrestricted* model ($n = m > w$).

6 Conclusion

We present two new algorithms, BV-H algorithm and BV-HS algorithm, based on bit-parallelism. This idea has

BV-KMATCHLIST$(x, y, m, n, KMATCHLIST)$

$M[\alpha] \leftarrow 0$ for all $\alpha \in \Sigma$
for $i \leftarrow 1$ **to** m
    do $M[x_i]_i \leftarrow 1$
$M' \leftarrow \sim M$
$S \leftarrow 2^m - 1$
for $j \leftarrow 1$ **to** n
    do $SS \leftarrow (S + (S \ \& \ M[y_j])) \mid (S \ \& \ M'[y_j])$
        $K \leftarrow (S \hat{\ } SS) \ \& \ S$
        while $K > 0$
            do $lastbit \leftarrow \lfloor \log(K) \rfloor$
                ADD$(KMATCHLIST, (lastbit, j))$
                $K_{lastbit} \leftarrow 0$
        $S \leftarrow SS$

BV-HS(x, y, m, n, C, p)

$p \leftarrow 0$
$THRESH[0] \leftarrow 0$
$LINK[0] \leftarrow null$
$M[\alpha] \leftarrow 0$ for all $\alpha \in \Sigma$
for $i \leftarrow 1$ **to** m
    do $M[x_i]_i \leftarrow 1$
        $THRESH[i] \leftarrow m + 1$
        $LINK[i] \leftarrow null$
$M' \leftarrow \sim M$
$S \leftarrow 2^m - 1$
for $j \leftarrow 1$ **to** n
    do $SS \leftarrow (S + (S \ \& \ M[y_j])) \mid (S \ \& \ M'[y_j])$
        $K \leftarrow (S \hat{\ } SS) \ \& \ S$
        while $K > 0$
            do $i \leftarrow \lfloor \log(K) \rfloor$
                find k such that THRESH$[k-1] < i \leq$ THRESH$[k]$
                $THRESH[k] \leftarrow i$
                $LINK[k] \leftarrow newnode(j, LINK[k-1])$
                $K_i \leftarrow 0$
        $S \leftarrow SS$
$t \leftarrow$ largest k such that $THRESH[k] \neq m + 1$
$temp \leftarrow LINK[t]$
while $temp.next \neq null$
    do $p \leftarrow p + 1$
        $D[p] \leftarrow y_{temp.j}$
        $temp \leftarrow temp.next$
$C \leftarrow D^R$

Figure 7. BV-KMATCHLIST **and** BV-HS **algorithms.**

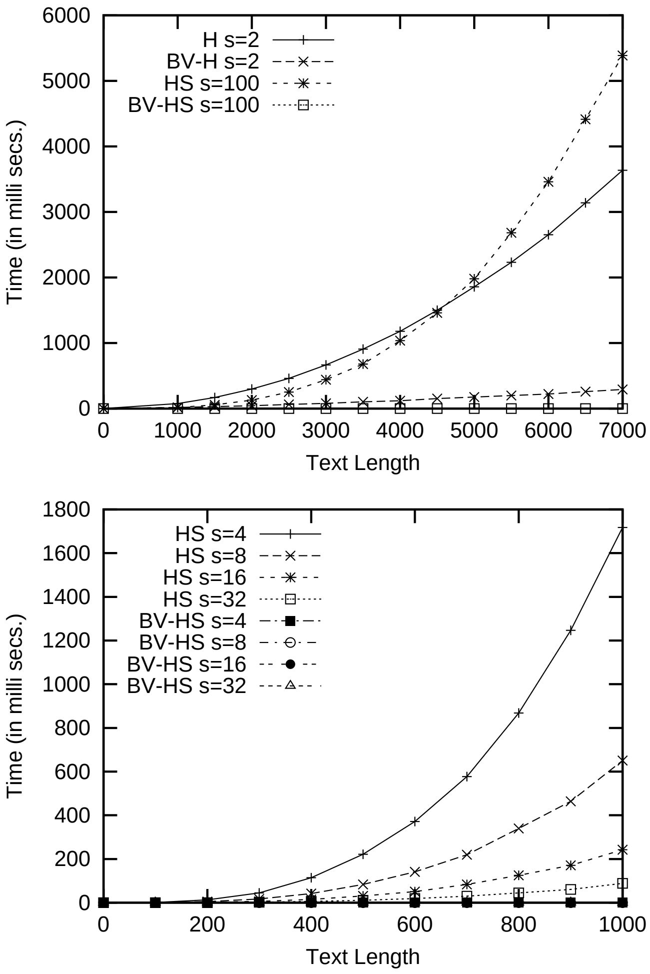


Figure 8. Timing curves for H, HS, BV-H **and** BV-HS **algorithms.**

been previously used to find the length of an LCS but not to recover the longest common subsequence as we do. Our algorithms showed to be very fast on average, especially the BV-HS algorithm which behaviour is optimal regardless the values of output parameters like r, the number of match points and p, the length of the LCS. This makes it a very good choice for the cases when we have insufficient knowledge of the nature of the sequences to be compared, before hand. We think that bit-parallel techniques would permit to improve the practical speed of pattern matching software and if that is the case, there is a lot of work that need to be done.

References

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

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

[3] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[4] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 75–86, Hunter Valley, Australia, 2000.

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

[6] V. Dančík. *Expected length of longest common subsequences*. Ph. D. Thesis, Computer Science Departement, University of Warwick, UK, 1994.

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

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

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

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

[11] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

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

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

[14] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 14–33, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[15] M. Paterson and V. Dančík. Longuest common subsequences. In I. Prívara, B. Rovan, and P. Ružička, editors, *Proceedings of the 19th Symposium on Mathematical Foundations of Computer Science*, number 841 in Lecture Notes in Computer Science, pages 127–142, Košice, Slovakia, 1994. Springer-Verlag, Berlin.

[16] D. Sankoff and J. B. Kruskal. *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*. Addison-Wesley, Reading, MA, 1983.

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

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

[19] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.