International Journal of Foundations of Computer Science
Vol. 24, No. 8 (2013) 1281–1298

DOI: 10.1142/S0129054113500342



# BIT-PARALLEL ALGORITHMS FOR THE MERGED LONGEST COMMON SUBSEQUENCE PROBLEM

SEBASTIAN DEOROWICZ* and AGNIESZKA DANEK†
*Institute of Informatics*
*Silesian University of Technology*
*Akademicka 16, 44-100 Gliwice, Poland*
*sebastian.deorowicz@polsl.pl
†agnieszka.danek@polsl.pl

Received 13 December 2011
Accepted 29 April 2013
Communicated by Albert Zomaya

It is often a necessity to compare some sequences to find out how similar they are. There are many similarity measures that can be used, e.g., longest common subsequence, edit distance, sequence alignment. Recently a merged longest common subsequence (MergedLCS) problem was formulated with applications in bioinformatics. We propose the bit-parallel algorithms for the MergedLCS problem and evaluate them in practice showing that they are usually tens times faster than the already published methods.

*Keywords*: Sequence comparison; longest common subsequence; merged longest common subsequence; bit-parallelism.



## 1. Introduction

In many areas, there is a need or even a must to measure similarity of sequences. The sequences can store textual data, DNA or protein strings, musical transcriptions, and many more. An efficient measure of similarity can detect plagiarism in textual or music data, determine the relationship between species, etc. Some of the most popular similarity measures are: edit distance, longest common subsequence length, sequence alignment, and their variants [2, 12].

A formulation of the *longest common subsequence* (LCS) problem is simple: for two sequences $A$ and $B$ a longest sequence that is a subsequence[a] of both $A$ and $B$ is requested. Sometimes only its length is sufficient. Often in sequence comparison, the algorithms are based on dynamic programming technique and, e.g., such an algorithm of the worst-case time complexity $O(mn)$, where $m$ and $n$ are sequence lengths, can be used to compute an LCS or only its length.

[a]A subsequence can be obtained from a sequence by removing zero or more symbols.

In recent years, many variants of the basic LCS problem were proposed. A *longest common transposition invariant* problem (LCTS) [6, 8, 18, 19] was introduced to compare music sequences, and it allows that all the symbols from one input sequence are shifted by some amount. In a *constrained longest common subsequence* (CLCS) problem [5, 9, 21] a third sequence is added, which enforces some properties of the result (the output sequence must contain the third sequence as its subsequence). This problem is especially useful in bioinformatics. Some other examples of variants of the LCS problem are: *mosaic LCS* problem [15], *longest common increasing subsequence* problem [23].

Recently a *merged LCS* (MergedLCS) problem and its block variant were introduced [14]. Their inputs are three sequences $T$, $A$, $B$, and a requested output is a longest sequence $P$ that is a subsequence of $T$ and can be split into two subsequences $P'$ and $P''$ such that $P'$ is a subsequence of $A$ and $P''$ is a subsequence of $B$. The origins of the MergedLCS problem lay in bioinformatics, in which finding an *interleaving relationship* between sequences may be necessary to verify some biological hypotheses. One of them is a *whole-genome duplication* (WGD) followed by massive gene loss. In [17] an evidence of WGD in yeast was given. Kellis *et al.* show that *Saccharomyces cerevisiae* arose by duplication of eight ancestral chromosomes after which massive loss of genes (nearly 90%) took place. The deletions of genes were in paired regions, so at least one copy of each gene of the ancestral organism was preserved. Their proof is based on the comparison of DNA of two yeast species, *Saccharomyces cerevisiae* and *Kluyveromyces waltii* that descend directly from a common ancestor and diverged before WGD. These two species are related by 1:2 mapping satisfying several properties (see [17] for details), e.g., each of the two sister regions in *S. cerevisiae* contains an ordered subsequence of the genes in the corresponding region of *K. waltii* and the two sister subsequences interleaving contain almost all of *K. waltii* genes. Solving the MergedLCS problem for the three sequences (two regions in one species and one region in the other species) we can check whether such a situation (WGD) happened.

Some other situations in which the MergedLCS problem may be useful are given in [20]. The authors suggest to apply the MergedLCS problem in the signal comparison, where we have three sequences: one complete and two distorted (e.g., by noise) and we want to verify whether the distorted sequences were obtained by adding some noise to the complete sequence.

To date only two papers with algorithms for the MergedLCS problem were published. In [14], the authors proposed a dynamic programming approach, while in [20] the algorithms specialised for the case of large alphabets, when the dynamic programming matrix is sparse, were given.

Our proposal is based on a bit-parallelism (BP) technique that is often used in string and sequence processing algorithm. The BP approach was first introduced in [11] and reinvented several times. Majority of its modern applications is motivated by the works [3, 4]. The main concept was rather simple: modern computers work on words of sizes 32 or 64 bits, while results of some operations (like comparisons) can

often be stored in one bit only! Therefore, many operations can be made in parallel on a single word. An initial application of this idea was speeding up naive pattern matching algorithm [4], but then many other applications of BP were proposed. Some examples from the LCS-related field are BP algorithms for: the LCS and LCTS problems [1, 7, 16], the CLCS problem [10]. An advantage, in terms of speed, over the classical algorithms of the BP methods is often huge, e.g., about 50-fold for the classical DP algorithm solving the LCS problem [8].

The rest of the paper is organised as follows. In Sec. 2, some definitions are introduced. Section 3 describes the problem background, i.e., two existing MergedLCS solving algorithms. Then, in Sec. 4, bit-parallel algorithms for the MergedLCS problem are introduced and discussed in detail. The algorithms are compared in practice to the existing methods in Sec. 5. The last section concludes the paper.

## 2. Definitions

The sequences $T = t_1 t_2 \dots t_r$, $A = a_1 a_2 \dots a_n$, $B = b_1 b_2 \dots b_m$ are over $\Sigma$, where $\Sigma \subset \mathbb{Z}$, called an *alphabet*, is a finite subset of integers. Without loss of generality, we assume that $m \leq n$. A *length* of any sequence $S$, denoted by $|S|$, is the number of *elements* (*symbols*) it contains. The *size* of the alphabet is denoted by $\sigma$. $S_i$ is a prefix $s_1 s_2 \dots s_i$ of $S$.

For any sequence $S$, $S'$ is a *subsequence* of $S$ if it can be obtained from it by removing zero or more symbols, i.e., $s'_1 s'_2 \dots s'_k = s_{i_1} s_{i_2} \dots s_{i_k}$ and $1 \leq i_1 < i_2 < \dots < i_k \leq |S|$. A *longest common subsequence* of two sequences is a sequence that is a subsequence of both sequences and has the largest length. A *merged longest common subsequence* of $T$, $A$, and $B$ is a longest sequence $P = p_1 p_2 \dots p_z$, being a subsequence of $T$, such that its subsequence $P' = p_{i_1} p_{i_2} \dots p_{i_k}$, where $1 \leq i_1 < i_2 < \dots < i_k \leq z$, is a subsequence of $A$ and sequence $P''$ obtained from $P$ by removing symbols at indices $i_1, i_2, \dots, i_k$ is a subsequence of $B$. This means that MergedLCS of $T$, $A$, $B$ is a longest common subsequence of $T$ and any sequence that can be obtained by merging $A$ and $B$.

Bitwise operations used in the paper are: & (bitwise and), | (bitwise or), ~ (negation of each bit), ^ (bitwise xor), << (shift to the left by given number of bits). The notation $0^i$ and $1^i$ means $i$ 0 bits and $i$ 1 bits, respectively. The computer word size is denoted by $w$. For any bit vector $W$, $W^{[i,j]}$ means a sequence of bits of $W$ from $i$th to $j$th. This notation is usually used to specify a single (or its part) computer word of array emulating long bit vector in real implementations. $W^{[i]}$ means $i$th bit of $W$.

## 3. Background

The MergedLCS problem was introduced in [14]. The authors proposed an algorithm working in time $O(mnr)$ and consuming $O(mnr)$ or $O(mn)$ space, depending on the implementation. They also discussed a variant of the problem, a Block-Merged LCS, but it is out of our interest so we will not describe it in detail. An algorithm solving the MergedLCS problem was based on dynamic programming. The proposed

formula was [14]:

$$L(i,j,k) = \max \begin{cases} L(i-1,j-1,k)+1 & \text{if } t_i = a_j, \\ L(i-1,j,k-1)+1 & \text{if } t_i = b_k, \\ \max \begin{cases} L(i-1,j,k) \\ L(i,j-1,k) \\ L(i,j,k-1) \end{cases} & \text{if } (t_i \neq a_j) \wedge (t_i \neq b_k), \end{cases} \tag{1}$$

with the boundary conditions:

$$\begin{aligned} L(0,j,k) &= 0, \\ L(i,0,0) &= 0, \\ L(i,j,0) &= \mathrm{LLCS}(T_i, A_j), \\ L(i,0,k) &= \mathrm{LLCS}(T_i, B_k), \end{aligned} \tag{2}$$

for any valid $i$, $j$, $k$, where LLCS means the longest common subsequence length of the arguments. The length of the result is in $L(r,n,m)$ while the subsequence can be obtained by backtracking the 3-dimensional matrix.

This formula is, however, incorrect, which can be observed by computing the DP matrix for some sample sequences shown in Fig. 1(a). The value of $L(3,3,3)$ is 2, but the correct result is 3 since ABA is a subsequence of $T = \mathsf{ABA}$ and can be split into A (a subsequence of $A = \mathsf{DDA}$) and BA (a subsequence of $B = \mathsf{BAC}$). Fixing formula (1) is relatively easy (Fig. 1(b)):

$$L(i,j,k) = \max \begin{cases} L(i-1,j-1,k)+1 & \text{if } t_i = a_j, \\ L(i-1,j,k-1)+1 & \text{if } t_i = b_k, \\ L(i-1,j,k), \\ L(i,j-1,k), \\ L(i,j,k-1). \end{cases} \tag{3}$$

The boundary conditions are unchanged.

**Lemma 1.** *Formula* (3) *with boundary conditions* (2) *properly computes a MergedLCS for* $T$, $A$, $B$.

**Proof.** The proof is identical to the proof of Theorem 1 in [14] with one exception. For a match (situation for which $t_i = a_j$ or $t_i = b_k$) we cannot forget about the possibility that truncating any sequence can be better than following the match. □

Very recently algorithms for Merged LCS and Block-Merged LCS problems, specialised for the case that the matches are rare, were published [20]. Their main idea is to use the sparsity of the dynamic programming matrix and restrict the computations to matches only. A calculation cost of a single cell may be larger than constant, but in total the time is significantly better. The algorithm for the MergedLCS problem needs only $O(\ell mr)$ time, where $\ell$ is the length of the result. Typically, for large alphabets, $\ell$ is much smaller than $n$, so this method can be advantageous.

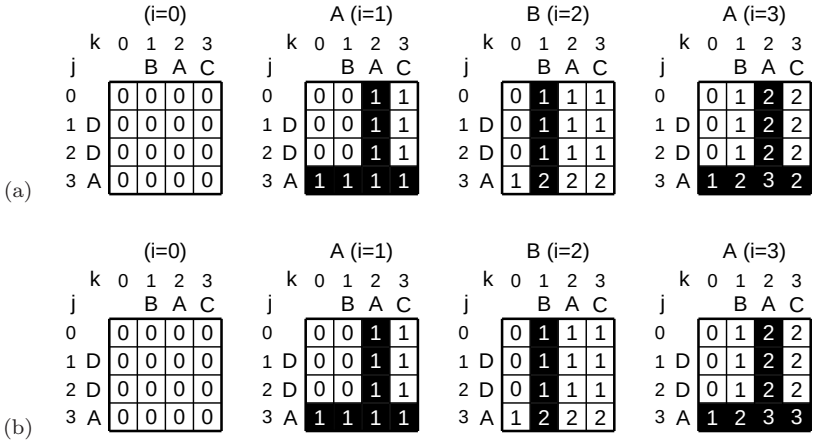



Fig. 1. Example of computation of the MergedLCS for $T = \mathsf{ABA}$, $A = \mathsf{DDA}$, $B = \mathsf{BAC}$ according to (a) original formula (1), (b) fixed formula (3). Black cells denote matches.



## 4. Bit-Parallel Algorithm

### 4.1. *Preliminaries*

To introduce the bit-parallel algorithm, it is necessary to prove some lemmas.

**Lemma 2.** *$L(i, j, k)$ is equal or larger by 1 than any of the neighbours: $L(i-1, j, k)$, $L(i, j-1, k)$, and $L(i, j, k-1)$.*

**Proof.** $L(i, j, k)$ stores the MergedLCS length for $T_i$, $A_j$, $B_k$. The mentioned neighbours store the MergedLCS lengths for the same sequences, but one of them is truncated by the last symbol. It is impossible that the truncation by one symbol of only one sequence decreases the MergedLCS length by more than 1, since only the truncated symbol may fall out from the output sequence. Similarly, it is impossible that if any of the sequences is truncated, the length of the result will be larger. □

Based on Lemma 2 it is possible to represent a 3-dimensional matrix $L$ as 2-dimensional matrix $M$ containing "change vectors" of integers from $[1, r]$ in each cell defined as:

$$i \in M(j, k) \text{ if and only if } L(i, j, k) - L(i-1, j, k) = 1 \text{ for } 1 \leq i \leq r. \tag{4}$$

An equivalence of $L$ and $M$ is due to the equality:

$$L(i, j, k) = |\{x | x \in M(j, k) \wedge x \leq i\}|. \tag{5}$$

Before we formulate a direct computation rule for $M(j,k)$, it is convenient to reformulate (3) as follows:

$$L'(i,j-1,k) = \max \begin{cases} L'(i-1,j-1,k), \\ L(i-1,j-1,k)+1, & \text{if } t_i = a_j, \\ L(i,j-1,k), \end{cases} \tag{6}$$

$$L''(i,j,k-1) = \max \begin{cases} L''(i-1,j,k-1), \\ L(i-1,j,k-1)+1, & \text{if } t_i = b_k, \\ L(i,j,k-1), \end{cases} \tag{7}$$

$$L(i,j,k) = \max \begin{cases} L'(i,j-1,k), \\ L''(i,j,k-1). \end{cases} \tag{8}$$

Let us note that the first components of the maximum functions in (6)–(7) are only to assure that the values of $L'$, $L''$, and $L$ do not decrease while $i$ is increasing (c.f., the third term of (3)). This requirement is inherent in the formulation of $M$. The change vector $M'(j-1,k)$ for $L'(i,j-1,k)$ can be computed according to $M(j-1,k)$ as follows:

(1) $M'(j-1,k) \leftarrow M(j-1,k)$.
(2) For each $i$ from $r$ to 1 such that $t_i = a_j$ proceed:
   (a) if $i \in M'(j-1,k)$ do nothing,
   (b) otherwise insert $i$ to $M'(j-1,k)$ and remove from $M'(j-1,k)$ the smallest (if any) integer larger than $i$.

In a similar way $M''(j,k-1)$ can be computed.

**Lemma 3.** *The above procedure properly computes $M'(j-1,k)$ and $M''(j,k-1)$ change vectors that are equivalent to $L'(i,j-1,k)$ and $L''(i,j,k-1)$, respectively, for all valid $i$.*

**Proof.** A *rank* of element in a change vector is the number of elements not larger than this element. Let us note that for each $i$ in $M'(j-1,k)$ of rank $r(i)$:

- $L(i,j-1,k) = r(i)$ (so $i \in M(j-1,k)$), or
- $L(i-1,j-1,k) = r(i)-1$ and $t_i = a_j$.

Thus, $i$ belongs to $M'(j-1,k)$ if and only if:

- $i \in M(j-1,k)$ and there is no $i' < i$ such that $L(i'-1,j-1,k) = r(i')-1$ and $t_{i'} = a_j$, or
- $i$ is the smallest index such that $L(i-1,j-1,k) = r(i)-1$ and $t_i = a_j$.

The proof for $M''(j,k-1)$ is similar. □

Note that the above is essentially the same what Observation 4 in [16] says for the BP algorithm solving the LCS problem.

Having change vectors $M'(j-1,k)$, $M''(j,k-1)$, we are ready to compute $M(j,k)$. According to (5) and (8), the value of $L(i,j,k)$ can be obtained as:

$$L(i,j,k) = \max \begin{cases} |\{x \mid x \in M'(j-1,k) \wedge x \leq i\}|, \\ |\{x \mid x \in M''(j,k-1) \wedge x \leq i\}|. \end{cases} \tag{9}$$

Therefore, to compute $M(j,k)$ one should proceed as follows:

(1) For each pair of integers from $M'(j-1,k)$ and $M''(j,k-1)$ of equal rank, take the smaller one and insert it to $M(j,k)$.
(2) If there is an integer of unique rank in $M'(j-1,k)$ or $M''(j,k-1)$ also insert it to $M(j,k)$.



### 4.2. *The length computation algorithm*

Use of bit vectors is an efficient method of representation of change vectors, especially if the change vectors are relatively dense. Since all the change vectors can contain integers from range $[1,r]$, the bit vectors are of size $r$. To simplify the presentation, bits will be numbered from 1, but in a real implementation they start from 0. For each change vector $M(j,k)$, $M'(j-1,k)$, and $M''(j,k-1)$, a related bit vector: $W(j,k)$, $W'(j-1,k)$, and $W''(j,k-1)$ is defined as follows: all bits are set to 1 except for the ones that indexes appear in the related change vector.

We also define an array of bit vectors $Y_c$ for each alphabet symbol $c$ that are defined in the following way: all bits of $Y_c$ for each valid $c$ are set to 0 except for the ones that reflect the positions of symbol $c$ in $T$. These bit vectors represent the positions at which we have matches when comparing sequence $T$ to some symbol of $A$ or $B$.

To compute $W'(j-1,k)$ based on $W(j-1,k)$ we use $Y_{a_j}$, as this bit-vector represents the positions at which we have matches in $T$ with $a_j$. Since what is needed here is exactly the same what is done in the bit-parallel LCS computing algorithms, a sequence of bit operations from [16] is used here (the additions and subtractions on bit-vectors are made be treating them as unsigned integers):

$$W'(j-1,k) \leftarrow W(j-1,k) \;\&\; Y_{a_j},$$
$$W'(j-1,k) \leftarrow (W(j-1,k) + W'(j-1,k)) \mid (W(j-1,k) - W'(j-1,k)).$$

An analogical rule can be formulated to compute $W''(j,k-1)$.

Computing in a bit-parallel way $W(j,k)$ on $W'(j-1,k)$ and $W''(j,k-1)$ is more complicated. It will be helpful to use the following lemma.

**Lemma 4.** *For any $x \in M'(j-1,k)$ of rank $r(x)$ and $y \in M''(j,k-1)$ of rank $r(y)$ holds:*

(a) *if $x > y$ then $r(x) \geq r(y)$,*
(b) *if $x < y$ then $r(x) \leq r(y)$,*
(c) *if $x = y$ then $|r(x) - r(y)| \leq 1$.*

**Proof.** We will prove by contradiction.

*Case a*: Let us assume that $x > y$ and $r(x) < r(y)$. From (6) and (7) we have:

$$r(x) = L'(x, j-1, k) = L(x-1, j-1, k) + 1, \tag{10}$$

$$r(y) = L''(y, j, k-1) = L(y-1, j, k-1) + 1. \tag{11}$$

By assumption $r(x) < r(y)$:

$$L'(x, j-1, k) < L''(y, j, k-1). \tag{12}$$

According to (8) we have:

$$L(y, j, k) \geq L''(y, j, k-1), \tag{13}$$

and from $x > y$:

$$L(x-1, j-1, k) \geq L(y, j-1, k). \tag{14}$$

By combining (10–14), we obtain:

$$L(y, j, k) \geq L''(y, j, k-1) > L'(x, j-1, k) > L(x-1, j-1, k) \geq L(y, j-1, k). \tag{15}$$

From the above:

$$L(y, j, k) - L(y, j-1, k) > 1, \tag{16}$$

which is impossible due to Lemma 2.

*Case b*: Proof is similar as for Case a.

*Case c*: Let us assume that $|r(x) - r(y)| > 1$. From (10) and (11) we have:

$$|L(x-1, j-1, k) - L(x-1, j, k-1)| > 1, \tag{17}$$

which is impossible since both $L(x-1, j-1, k)$ and $L(x-1, j-1, k)$ are neighbours of $L(x-1, j, k)$ and the difference between them must be not larger than 1 (Lemma 2). □

**Lemma 5.** *To compute $M(j, k)$ based on $M'(j-1, k)$ and $M''(j, k-1)$ it suffices to:*

(1) *Join sets $M'(j-1, k)$ and $M''(j, k-1)$ to a multiset (some integers may be in two copies) $M^*(j, k)$.*
(2) *Remove from $M^*(j, k)$ integers of even ranks obtaining set $M(j, k)$.*

**Proof.** By recurrence on rank $x$ in $M(j, k)$. For $x = 1$ the smallest integer in multiset $M^*(j, k)$ goes to $M(j, k)$ (it is an element of rank 1 in $M'(j-1, k)$ or $M''(j, k-1)$). The next smallest element in $M^*(j, k)$ (of rank 2) must be of rank 1 in $M''(j, k-1)$ or $M'(j-1, k)$, respectively (Lemma 4), so in $M(j, k)$ we have the smaller element from the input sets of rank 1.

Let $x > 1$ and for elements of ranks $2x-3$ and $2x-2$ in $M^*(j, k)$ one of them is of rank $x-1$ in $M'(j-1, k)$, and the second one is of rank $x-1$ in $M''(j, k-1)$. For elements of ranks $2x-1$ and $2x$ in $M^*(j, k)$, one of them must be of rank $x$

in $M'(j-1,k)$, and the second one of rank $x$ in $M''(j,k-1)$ (Lemma 4). We copy the smaller of them to $M(j,k)$. Of course, if there are two equal integers in $M^*(j,k)$ their ranks are of different parity, so exactly one of them goes to $M(j,k)$, and $M(j,k)$ is a set. □

Since the integers appearing in both $M'(j-1,k)$ and $M''(j,k-1)$ must belong to $M(j,k)$, we can remove them from $M^*(j,k)$ after the first step, and finally after removing even-rank integers add them (a single copy of each) to $M(j,k)$. In terms of bit-vector representation of change vectors, this can be done as follows:

$$W(j,k) \leftarrow (W'(j-1,k) \mid W''(j,k-1)) - (W'(j-1,k) \;\&\; W''(j,k-1)).$$

The or-term means joining sets while and-term is to select the integers belonging to both input bit-vectors that are removed by subtraction. A crucial part of the algorithm is to remove 1 bits of even ranks. In [22] the algorithm computing "parity" in a 32-bit computer word $x$ is given (adopting it to any $w$ size is straightforward):

$$\begin{aligned} y &\leftarrow x \;\hat{}\; (x << 1),\\ y &\leftarrow y \;\hat{}\; (y << 2),\\ y &\leftarrow y \;\hat{}\; (y << 4),\\ y &\leftarrow y \;\hat{}\; (y << 8),\\ y &\leftarrow y \;\hat{}\; (y << 16). \end{aligned}$$

The $i$th bit of $y$ is 1 if and only if the number of 1s in positions from 0 to $i$ in $x$ is odd. Therefore, to remove even-rank 1s in $W(j,k)$ we need to compute the "parity" of this bit-vector and mask even-rank 1 bits in $W(j,k)$.

A pseudocode of the complete bit-parallel algorithm solving the MergedLCS problem is given in Fig. 2. In its first part (lines 1–7) the LCS problem for the boundaries is solved, so we have the boundary conditions (2) satisfied. In lines 8–19 the main computations are performed: the bit vectors for all valid pairs $(j,k)$ are calculated. Finally, in lines 20–24 the number of 0 bits in bit vector $W(n,m)$ is determined and returned. Figure 3 illustrates how the algorithm processes bit vectors for the same sample sequences that were used in Fig. 1(b) for DP-based algorithm.

### 4.3. *Details*

The algorithm presented above assumes that $r \leq w$, i.e., each bit vector fits into a single computer word, which is rarely the case. Fortunately, it is rather easy to emulate bit vectors as arrays of computer words of size $\lceil r/w \rceil$. Some care is, however, necessary when implementing arithmetic and binary operations. All subtractions, bitwise | , & , ^ in lines 3–4, 6–7, and 10–15 can be easily implemented on arrays of computer words since no carry can occur. An exception is addition operation in lines 4, 7, 11, and 13, in which a carry may occur and must be handled, but this is

BitPar-MergedLCS

```
   {Initialisation}
 1  W(0,0) ← 1^r
   {Calculating boundaries}
 2  for k ← 1 to m do
 3      U ← W(0,k−1) & Y_{b_k}
 4      W(0,k) ← (W(0,k−1) + U) | (W(0,k−1) − U)
 5  for j ← 1 to n do
 6      U ← W(j−1,0) & Y_{a_j}
 7      W(j,0) ← (W(j−1,0) + U) | (W(j−1,0) − U)
   {Main calculations}
 8  for j ← 1 to n do
 9      for k ← 1 to m do
10          U' ← W(j−1,k) & Y_{a_j}
11          W' ← (W(j−1,k) + U') | (W(j−1,k) − U')
12          U'' ← W(j,k−1) & Y_{b_k}
13          W'' ← (W(j,k−1) + U'') | (W(j,k−1) − U'')
14          U ← W' | W''
15          W ← W' ^ W''
16          V ← W
17          for i ← 0 to ⌈log_2 r⌉ − 1 do
18              V ← V ^ (V << (1 << i))
19          W(j,k) ← ~(W & V) & U
   {Determining the result}
20  z ← 0;        V ← ~W(n,m)
21  while V ≠ 0^r do
22      V ← V & (V − 1)
23      z ← z + 1
24  return z
```

Fig. 2. Bit-parallel algorithm computing the MergedLCS length.



easy and have no asymptotic impact on the total time complexity. The loop in lines 17–18 should be implemented with special care. In fact, the loop is executed on each single computer word ($\lceil \log_2 w \rceil$ times) and the information about the parity of the most significant bit is stored as a carry. When processing the same on next computer word, after the loop all the bits are flipped if necessary (according to the carry from the previous word). Figure 4 shows the implementation of lines 17–19 of Fig. 2 on an array of computer words.

The total time complexity of the algorithm is determined by computation of $W(j,k)$. This is done in $\Theta(\lceil r/w \rceil \log w)$ steps. Therefore, the total time complexity is $\Theta(\lceil r/w \rceil mn \log w)$, which is $\Theta(w/\log w)$ times faster than the DP-based algorithm solving the MergedLCS problem [14] (see Table 1 for details). The space complexity is $\Theta(\lceil r/w \rceil m)$, as it is enough to store two 2-dimensional planes of the 3-dimensional matrix, since no backtracking to collect a MergedLCS is necessary.

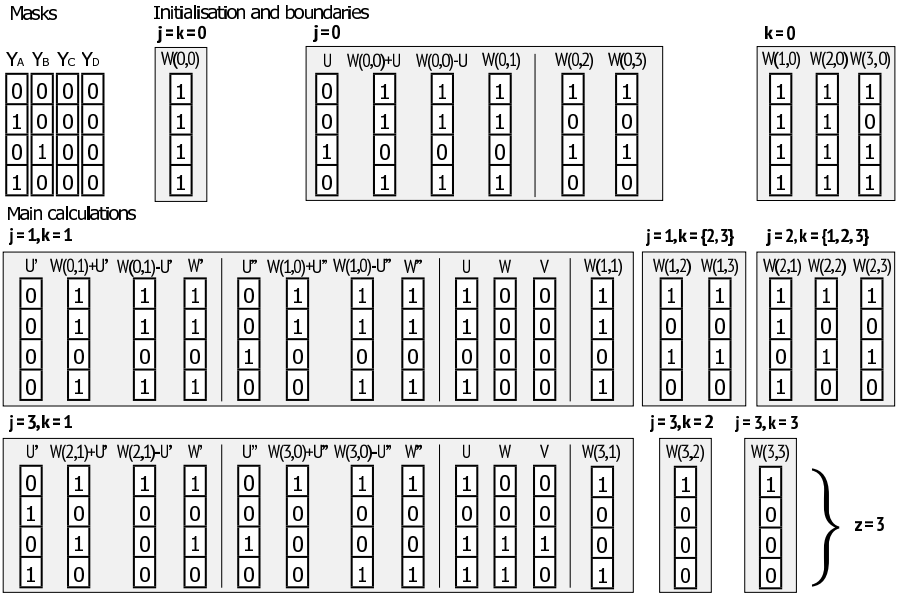



Fig. 3. Example of computation of the MergedLCS for $T = \mathsf{ABA}$, $A = \mathsf{DDA}$, $B = \mathsf{BAC}$ according to the bit-parallel algorithm presented in Fig. 2.

Lines 17–19 of BitPar-MergedLCS on array of computer words

```
1  f ← false
2  for i' ← 0 to ⌈r/w⌉ − 1 do
3      V ← W^[i'w,i'w+w−1]
4      for i ← 0 to ⌈log2 w⌉ − 1 do
5          V ← V ^ (V << (1 << i))
6      if f then V ← ~V
7      if V^[w−1] = 1 then f ← true
8      else f ← false
9      W(j,k)^[i'w,i'w+w−1] ← ~(W^[i'w,i'w+w−1] & V) & U^[i'w,i'w+w−1]
```

Fig. 4. Algorithm simulating removal of even rank bits (lines 17–19 in Fig. 2) in array of computer words.

### 4.4. *The sequence finding algorithm*

Obtaining one of the possible sequences being a solution of the MergedLCS problem is possible by backtracking changes in calculated bit vectors (Fig. 5). It is done till an entire output sequence is collected. The backtracking starts from $r$th bit of $W(n, m)$ vector (line 11). For any $i$th bit of vector $W(j, k)$ first it is verified if it is set to 1. If so, the symbol $T[i]$ does not belong to the result, so the algorithm immediately moves to $(i - 1)$th bit of $W(j, k)$ (line 14). Otherwise it is checked if there is a difference between number of 0s in vector $W(j, k)^{[0,i]}$ and one of neighbour vectors

Table 1. Time and space complexities of the evaluated algorithms.

| Algorithm | Time complexity | Space complexity |
|---|---|---|
| H08-len | $\Theta(nmr)$ | $\Theta(nm)$ |
| P10-len | $\Theta(\ell mr)$ | $\Theta(nm)$ |
| Our-len | $\Theta(nm\lceil r/w \rceil \log w)$ | $\Theta(m\lceil r/w \rceil)$ |
| H08-seq | $\Theta(nmr)$ | $\Theta(nmr)$ |
| P10-seq | $\Theta(\ell mr)$ | $\Theta(min\{\ell mr, \ell mn\})$ |
| Our-seq | $\Theta(nm\lceil r/w \rceil \log w)$ | $\Theta(nm\lceil r/w \rceil)$ |



BitPar-MergedLCS-seq

```
   {Function calculating parity of vector W(a,b)^[0,t]}
 1  parity(t,a,b)
 2      f ← false
 3      for i' ← 0 to ⌈t/w⌉ − 1 do
 4          V ← ~W(a,b)^[i'w,i'w+w−1]
 5          for i ← 0 to ⌈log2 w⌉ − 1 do
 6              V ← V ^ (V << (1 << i))
 7          if f then V ← ~V
 8          if V^[w−1] = 1 then f ← true
 9          else f ← false
10      return V^[t mod w]
   {Main calculations—finding sequence}
11  i ← r;      j ← n;      k ← m
12  while z ≠ 0 do
13      if W(j,k)^[i] = 1 then i ← i − 1
14      else if j > 0 and pairity(i,j,k) = pairity(i,j−1,k) then j ← j − 1
15      else if k > 0 and pairity(i,j,k) = pairity(i,j,k−1) then k ← k − 1
16      else if j > 0 and T[i] = A[j] and W(j−1,k)^[i] = 1 then
17          R[z] ← T[i]
18          i ← i − 1;      j ← j − 1;      z ← z − 1
19      else
20          R[z] ← T[i]
21          i ← i − 1;      k ← k − 1;      z ← z − 1
22  return R
```

Fig. 5. Bit-parallel algorithm computing a MergedLCS.

$W(j-1,k)^{[0,i]}$ or $W(j,k-1)^{[0,i]}$. As the only possible difference is 1 (representing one change), comparison of parities of vectors complements (parity calculation is done according to [22] , lines 1–10) is enough to find out whether the vectors have equal number of 0s. If they do, it is known that $A[j]$ or $B[k]$ does not belong to the result. Thus the algorithm goes to $i$th bit of $W(j-1,k)$ or $W(j,k-1)$ vector (lines 14–15). The last possible case is that a change related to $T[i]$ occurred. If $T[i]$ is equal to $A[j]$ and $i$th bit $W(j-1,k)$ is 1, which means that number of 0s

in $W(j-1,k)^{[0,i]}$ and $W(j-1,k)^{[0,i-1]}$ is the same and therefore different that in $W(j,k)^{[0,i]}$ (checked in line 14), the change related to $A[j]$ symbol is assumed. Otherwise the change related to $B[k]$ is the only choice. The algorithm adds $T[i]$ to the result, decreases the length of the searched sequence and goes to $(i-1)$th bit of $W(j-1,k)$ (in case of change related to $A[j]$, lines 16–18) or to $(i-1)$th bit of $W(j,k-1)$ (in case of change related to $B[k]$, lines 19–21).

Here the entire matrix of bit vectors is needed, so the space complexity is $\Theta(mn\lceil r/w\rceil)$. Collecting of MergedLSC is done in $\Theta((m+n)\lceil r/w\rceil \log w)$ steps. Therefore, the total time complexity is $\Theta(mn\lceil r/w\rceil \log w)$.



## 5. Experimental Results

In practical experiments, we compared the algorithms known from the literature with our proposal. The algorithms are denoted in figures and tables as:

- H08-len — algorithm introduced in [14] calculating sequence length only,
- P10-len — algorithm introduced in [20] calculating sequence length only,
- Our-len — algorithm proposed in this paper calculating sequence length only,
- H08-seq — algorithm introduced in [14] with sequence finding,
- P10-seq — algorithm introduced in [20] with sequence finding,
- Our-seq — algorithm proposed in this paper with sequence finding.

In the implementation of H08-len and P10-len algorithms only two 2-dimensional planes of the 3-dimensional matrix are stored. For H08-seq it is necessary to store the whole 3-dimensional matrix for backtracking. In P10-seq, two 2-dimensional planes are used to calculate the length of the MergedLSC and an additional 3-dimensional matrix with pointers to previous cells is calculated (1-dimensional array of indicators in our implementation).

There can be more than one sequence being a solution of the MergedLCS problem. In all *-seq implementations only one of them is reported. In the implementation of H08-seq, the backtracking is performed in a similar way as in Our-seq, that is all possible cases are checked and rejected or accepted in the same order. Therefore the output sequences are the same for these two algorithms. The backtracking for P10-seq is made in a different way (moving back using pointers to the previous cells), thus the output sequence is usually different than for the other two algorithms.

The implementations were made in C++ and compiled using gcc version 4.6.1 compiler with optimisation option –O3. The programs were run on a server equipped with four AMD Opteron(tm) Processor 6136 processors (2.4 GHz CPU clock) and 128 GB of RAM. Each experiment was repeated 201 times to get median values.

In the first experiment, we used the data sets proposed in [13]. A short summary of them is given in Table 2. The sequence $T$ in dodA set is a whole intron-exon interleaving DNA sequence of one gene, while $A$ and $B$ are concatenation of exon and intron parts. In case of p&d set, $T$ is a part of *D. melanogaster* DNA sequence, while $A$ and $B$ are DNA sequences of two nested genes located within that part.

Table 2. Data sets used in real data experiments.

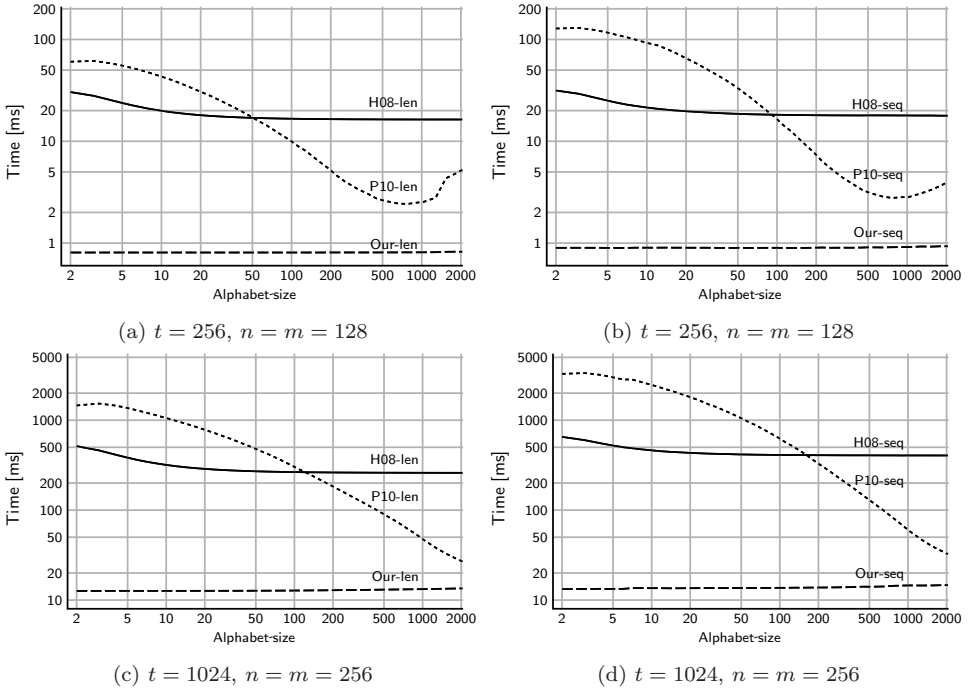
| Data set | $\|T\|$ | $\|A\|$ | $\|B\|$ | Description |
|---|---|---|---|---|
| dodA | 1629 | 687 | 942 | intron-exon interleaving DNA sequence of *Amanita muscaria dodA* gene (gi:2072623)<br>$T$ — whole sequence<br>$A$ — concatenation of exon parts<br>$B$ — concatenation of intron parts |
| p&d | 6000 | 2480 | 1756 | DNA sequences of *Drosophila melanogaster*<br>$T$ — part of chromosome 2R (gi:113194556) from 19436335 to 19443334<br>$A$ — reverse complement of pita (gi:24762318)<br>$B$ — death caspase-1 (gi:24762322) |



Table 3. Experimental results in real data experiments.

| Data set | H08 [MB] | P10 [MB] | Our [MB] | H08 [ms] | P10 [ms] | Our [ms] | Speedup over H08 | Speedup over P10 |
|---|---|---|---|---|---|---|---|---|
| Length computation only | | | | | | | | |
| dodA | 17 | 65 | 12 | 6,703.2 | 18,668.9 | 197.5 | 33.9 | 94.5 |
| p&d | 51 | 352 | 15 | 173,721.0 | 646,743.3 | 4 726.6 | 36.8 | 136.8 |
| Sequence computation | | | | | | | | |
| dodA | 4,100 | 12,000 | 150 | 9,109.2 | 25,914.5 | 275.0 | 33.1 | 94.2 |
| p&d | 97,600 | — | 3,200 | 273,308.4 | — | 6,507.3 | 42.0 | — |

Hence, for both data sets the length of MergedLCS is equal to total lengths of $A$ and $B$, with MergedLCS being a combination of these sequences. The experimental results for the real data are presented in Table 3 (due to memory requirements, it was impossible to run P10-seq for p&d data set). Here the alphabet size is small ($\sigma = 4$), so the number of matches is relatively large. Therefore, it is not surprising that sparse-DP-friendly P10-len algorithm is slower than H08-len, since the cost of calculation of a single cell in P10-len is much larger than in H08-len. Our algorithm shows the power of bit-parallelism. Theory says that it should be $\Theta(w/\log w)$ times faster than H08-len, but practice shows that its advantage is about 30-fold. This is because we use fast bitwise operations and store DP matrix in very compact way — only 1 bit per each cell of DP matrix, so cache memory is much better used. Moreover, the $(\log w)$-term in the complexity is because of the loop computing the parity of bits, but the total number of bitwise operations, per computer word, in this loop for typical $w$ is less than the number of other operations. The memory consumption is smallest for Our-len algorithm. The situation is similar for sequence finding algorithms: P10-seq, H08-seq and Our-seq. Here the advantage in memory consumption of Our-seq algorithm over others is even larger, as the entire structures for sequence finding are stored.



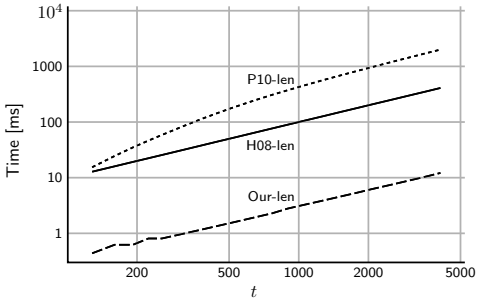

(a) $t = 256$, $n = m = 128$ (b) $t = 256$, $n = m = 128$

(c) $t = 1024$, $n = m = 256$ (d) $t = 1024$, $n = m = 256$

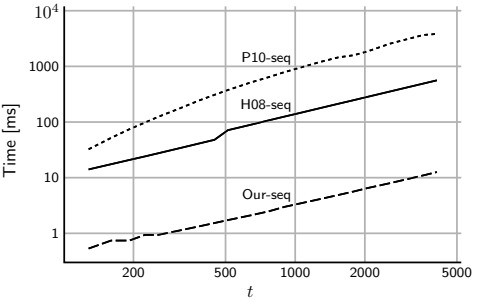
Fig. 6. Experimental comparison of the MergedLCS computing algorithms for various alphabet sizes; (a) and (c) are for algorithms calculating sequence length only, (b) and (d) are for algorithms finding also an output sequence.

An intention of the next experiments was to check the efficiency of the algorithms for various sequence lengths and alphabet sizes. Therefore, the input data were prepared by uniform random number generator. In these experiments, the data contents in each execution were different. We fixed the sequence lengths and determined the computation times of the algorithms for various alphabet sizes. The results (Fig. 6) show that P10 overtakes H08 (for both *-len and *-seq variants) for alphabet sizes around 50–170. The actual value depends on the sequence lengths. The speedup of Our-len algorithm over H08-len is in range $[19, 40]$ for both cases ((a) and (c)). The overtake over P10-len depends strongly on the alphabet size and is in range $[2, 121]$. For sequence finding algorithms (cases (b) and (d)) the speedup of Our-seq over H08-seq is in range $[19, 49]$ and over P10-len is in range $[2, 251]$.

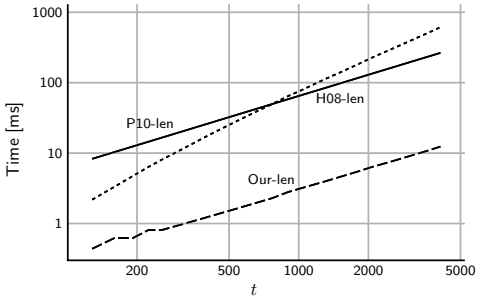
In the third experiment, the alphabet size was fixed to values: $\sigma = 4$ and $\sigma = 128$, and the sequence lengths were varied. The results are shown in Fig. 7. For relatively short sequences (cases (a) and (c) for *-len algorithms and cases (b) and (d) for *-seq algorithms) the speedup of Our-len approach over H08-len is from 16-fold to 33-fold and over P10-len from 5-fold to 164-fold, while speedup of Our-seq approach over H08-seq is from 15-fold to 44-fold and over P10-seq from 5-fold to 333-fold. The results are similar for longer sequences (cases (e)–(h)).



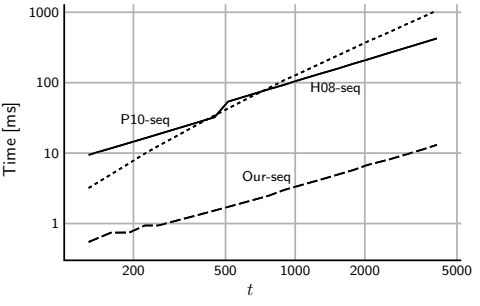

(a) $n = m = 128$, $\sigma = 4$



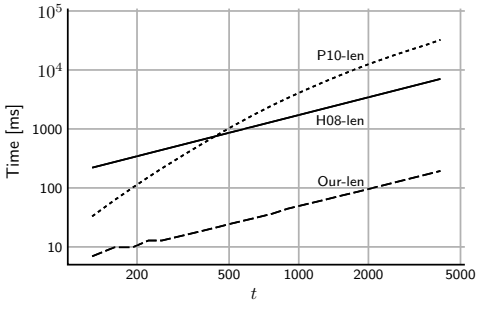

(b) $n = m = 128$, $\sigma = 4$



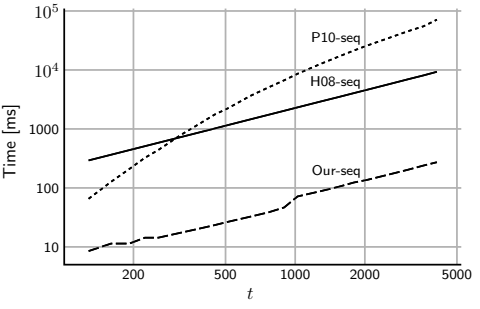

(c) $n = m = 128$, $\sigma = 128$



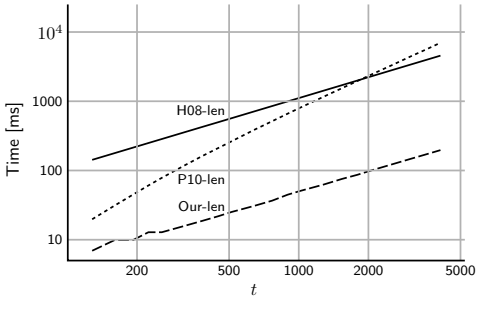

(d) $n = m = 128$, $\sigma = 128$



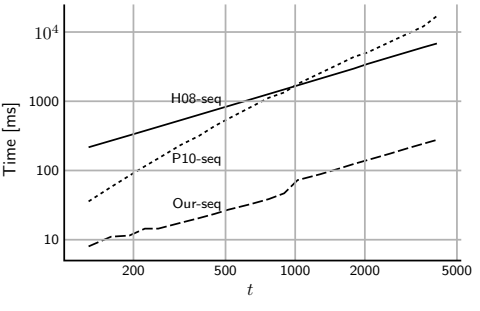

(e) $n = m = 512$, $\sigma = 4$




(f) $n = m = 512$, $\sigma = 4$




(g) $n = m = 512$, $\sigma = 128$




(h) $n = m = 512$, $\sigma = 128$

Fig. 7. Experimental comparison of the MergedLCS computing algorithms for various sequence lengths; (a), (c), (e), and (g) are for algorithms calculating sequence length only, (b), (d), (f), and (h) are for algorithms finding also an output sequence.

Summarising the experimental results we can note that the speedup of the bit-parallel algorithm over the classical method (H08-len or H08-seq), which was parallelised in a bit-parallel way is large and fits in the range $[15, 49]$. This is more than one could expect while considering the theoretical speedup of $\Theta(w/\log w)$. The advantage of our approach over P10-len or P10-seq depends strongly on the number of matches, which is dependent on the alphabet size. In all the experiments, our method was at least 2 times faster, but typically the advantage was 20-fold and larger.



## 6. Conclusions

We proposed two algorithms for the MergedLCS problem. First of them computes only the MergedLCS length, while the second produces also the resulting sequence. Their worst- and average-case time complexities are the same: $O(\lceil r/w \rceil mn \log w)$. The experiments show that they are about 15–49 times faster than the algorithms from [14] of the worst- and average-case time complexities $O(mnr)$, and much faster than the algorithms from [20] of the worst-case time complexities $O(\ell mr)$. These results show how powerful bit-parallelism is. Unfortunately, this kind of parallelism not always can be used, since often it is not easy to invent a sequence of bit-parallel operations that simulate what is going on in classical methods, e.g., it is an open question, whether similar bit-parallel algorithms can be proposed for the Block MergedLCS problem.

It is interesting what is the lower bound of time complexity for the MergedLCS problem. All the known algorithms work in $O(mnr)$ time in the worst case (for a constant word size), but it is an open question whether faster methods can be proposed.

## Acknowledgments

The author thanks Zbigniew J. Czech and Szymon Grabowski for reading preliminary versions of the paper and suggesting improvements. The work was partially supported by the Polish National Science Center upon decision DEC-2011/03/B/ST6/01588 (first author) and by the European Union from the European Social Fund (grant agreement number: UDA-POKL.04.01.01-00-106/09) (second author).

## References

[1] L. Allison and T.L. Dix. A bit-string longest common subsequence algorithm. *Information Processing Letters*, 23:305–310, 1986.

[2] A. Apostolico. General pattern matchings. In M.J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*, chapter 13. 1998.

[3] R.A. Baeza-Yates. *Efficient text searching*. PhD thesis, Department of Computer Science, University of Waterloo, Ontario, Canada, 1989.

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

[5] F.Y.L. Chin, A. De Santis, A.L. Ferrara, N.L. Ho, and S.K. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90(4): 175–179, 2004.
[6] T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.
[7] 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. *Information Processing Letters*, 80:279–285, 2001.
[8] S. Deorowicz. Speeding up transposition-invariant string matching. *Information Processing Letters*, 100:14–20, 2006.
[9] S. Deorowicz. Fast algorithm for the constrained longest common subsequence problem. *Theoretical and Applied Informatics*, 19(2):91–102, 2007.
[10] S. Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fundamenta Informaticae*, 99(4):409–433, 2010.
[11] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3: 29–46, 1964.
[12] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
[13] K.-S. Huang. *Some Common Subsequence Problems of Multiple Sequences and Their Applications*. PhD thesis, National Sun Yat-sen University, Taiwan, 2006.
[14] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng. Efficient algorithm for finding interleaving relationship between sequences. *Information Processing Letters*, 105(5):188–193, 2008.
[15] K.-S. Huang, C.-B. Yang, K.-T. Tseng, Y.-H. Peng, and H.-Y. Ann. Dynamic programming algorithms for the mosaic longest common subsequence problem. *Information Processing Letters*, 102(2–3):99–103, 2007.
[16] H. Hyyrö. Bit-parallel lcs-length computation revisited. In *Proceedings of the* 15*th Australasian Workshop on Combinatorial Algorithms*, pages 16–27, 2004.
[17] M. Kellis, B.W. Birren, and E.S. Lander. Proof and evolutionary analysis of ancient genome duplication in the yeast saccharomyces cerevisiae. *Nature*, 428(6983):617–624, 2004.
[18] K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proceedings of Content-Based Multimedia Information Access Conference* (*RIAO*), pages 1261–1279, 2000.
[19] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proceedings of AISB'*2000 *Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, pages 53–60, 2000.
[20] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor. Efficient sparse dynamic programming for the merged lcs problem with block constraints. *International Journal of Innovative Computing, Information and Control*, 6(4):1935–1947, 2010.
[21] Y-T. Tsai. The constrained common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.
[22] H.S. Warren, Jr. *Hacker's Delight*. Addison-Wesley Professional, 2002.
[23] I-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93:249–253, 2005.