Computing and Informatics, Vol. .., ..., 1–18, V 2009-Nov-9

CONSTRAINED LONGEST COMMON SUBSEQUENCE COMPUTING ALGORITHMS IN PRACTICE

Sebastian Deorowicz

Institute of Informatics
Silesian University of Technology
Akademicka 16
44-100 Gliwice, Poland
e-mail: `sebastian.deorowicz@polsl.pl`

Joanna Obstój

Abstract. The problem of finding a constrained longest common subsequence (CLCS) for the sequences A and B with respect to the sequence P was introduced recently. Its goal is to find a longest subsequence C of A and B such that P is a subsequence of C. There are several algorithms solving the CLCS problem, but there is no real experimental comparison of them. The paper has two aims. Firstly, we propose an improvement to the algorithms by Chin *et al.* and Deorowicz based on an entry-exit points technique by He and Arslan. Secondly, we compare experimentally the existing algorithms for solving the CLCS problem.

Keywords: Longest common subsequence, constrained longest common subsequence, sparse dynamic programming, string matching, sequence alignment

Mathematics Subject Classification 2000: 68W05

1 INTRODUCTION

The knowledge of the similarity of two sequences is crucial in various applications. The sequence similarity can be defined in many ways and one of the most commonly used is the length of their longest common subsequence (LCS).[1] In this problem, we are interested in a longest sequence[2] which is a subsequence of both sequences (a subsequence is obtained from a sequence by deleting zero or more symbols). The problem is well studied [2, 3, 9, 15] and is used in many applications, like DNA and protein analysis, text information retrieval, file comparing, music information retrieval, or spelling correction.

There are also a lot of generalizations of this similarity measure. One of the recent is the length of a *constrained longest common subsequence* (CLCS) [19]. It generalizes the LCS measure by introduction of a third sequence, which allows to extort that the obtained CLCS has some special properties. We deal here with three sequences, A, B, and P, and look for a longest sequence being a subsequence of both A and B, and containing P as a subsequence.

The CLCS problem emerged in bioinformatics, where sequences containing DNA and proteins are compared. In the classical LCS problem, the obtained result is sometimes of little biological value. Therefore, the biologists wanted to have a possibility to use their prior, biological, knowledge of the sequences. Tang *et al.* [18] illustrates the problem giving the following example. In the alignment of RNase sequences, it is known that sequences contain three active-site residues, His(H), Lyn(K), His(H). They are essential for RNA degrading. Therefore, the only sequences interesting for the biologists are those, in which these three residues occur in the given order (the sequence HKH is the constraint).

In Section 2, the necessary terms and a definition of the CLCS problem are given. Section 3 contains a description of the existing algorithms for a CLCS computing. In Section 4, some improvements to these methods are proposed. Then, in Section 5, the implementation details are discussed, and a comprehensive experimental comparison of the described algorithms are presented. The last section concludes the paper.

2 DEFINITIONS

Let us have three sequences: $A = a_1a_2 \ldots a_n$, $B = b_1b_2 \ldots b_m$, and $P = p_1p_2 \ldots p_r$, where $r \leq \min(m, n)$. Each of them is composed of symbols over an alphabet Σ of size σ. The *length* (*size*) of the sequence is the number of elements it contains. A sequence X', for any X, is a subsequence of X if it can be obtained from X by removing zero or more symbols. The LCS problem for A and B is to find a longest possible sequence C being a subsequence of both A and B. The CLCS problem, being a generalization of the LCS problem, for A, B, P is to find a longest possible sequence C being a subsequence of A and B and containing P as a subsequence.

The problem is symmetric, so it can be assumed without a loss of generality that $m \leq n$. The pair (i, j) is called a *match* iff $a_i = b_j$. A triple (k, i, j) is called a *strong match* iff $a_i = b_j = p_k$. For simplicity, the notation $X_{i \ldots j}$ for $x_ix_{i+1} \ldots x_j$, X_s

[1] Some other possibilities, like *edit distance*, *best alignment* are discussed, e.g., in [2, 9]

[2] There can be a number of such sequences of equal length

for $X_{1\dots s}$, and X_{-s} for $X_{s+1\dots|X|}$ is used.

3 ALGORITHMS FOR THE CLCS PROBLEM

There are several algorithms dealing with the CLCS problem. Their worst-case time and space complexities are shown in Table 1. Since, often the knowledge of the CLCS length only is sufficient, the space complexities are given for the two cases.

Authors	Year	Time	Space (CLCS)	Space (CLCS length)
Tsai [19]	2003	$O(m^2n^2r)$	$O(m^2n^2)$	$O(m^2n^2)$
Peng [16]	2003	$O(mnr)$	$O(mnr)$	$O(mr)$
Chin *et al.* [6]	2004	$O(mnr)$	$O(mnr)$	$O(mr)$
Peng, Ting [17]	2004	$O(mnr)$	$O(nr)$	$O(nr)$
Arslan, Eğecioğlu [1]	2005	$O(mnr)$	$O(mnr)$	$O(mn)$
Wang [20]	2006	$O(mnr)$	$O(mnr)$	$O(mr)$
Deorowicz [7]	2007	$O(r(m\ell + d) + n)$	$O(dr + \max(n, \sigma))$	$O(d + \max(n, \sigma))$
Iliopoulos, Rahman [13]	2008	$O(rd \log\log n)$	$O(\max(n, rd))$	$O(\max(n, d))$

Table 1. Time and space complexities of the CLCS computing algorithms (d means the total number of matches between A and B, and ℓ means the length of an LCS for A and B)

3.1 Tsai Algorithm

The CLCS problem was introduced by Tsai, who also published the first algorithm solving it [19]. His method is based on dynamic programming. The idea is to compute a CLCS for all the components of sequences A and B (a component can be obtained from a sequence by removing zero or more symbols from the beginning and the end). The algorithm's time complexity, $O(m^2n^2r)$, makes it completely impractical, so we will not consider it any more.

3.2 Chin et al. Algorithm

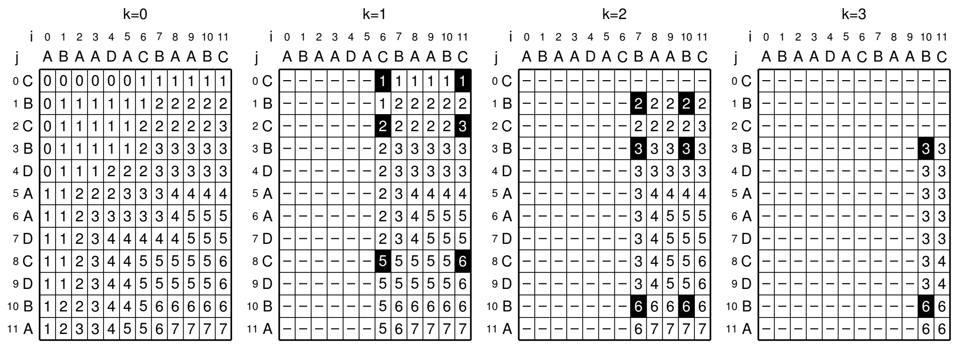
The method by Chin *et al.* [6] is also based on dynamic programming, but is much faster. The algorithm computes a three dimensional matrix M of sizes $(r+1) \times (n+1) \times (m+1)$ with the following simple recurrence:

$$M(k,i,j) = \begin{cases} M(k-1, i-1, j-1) + 1 & \text{if } i,j,k > 0 \wedge a_i = b_j = p_k, \\ M(k, i-1, j-1) + 1 & \text{if } i,j > 0, a_i = b_j \wedge \\ & \quad (k = 0 \vee a_i \neq p_k), \\ \max(M(k, i-1, j), M(k, i, j-1)) & \text{if } i,j > 0 \wedge a_i \neq b_j. \end{cases}$$

The boundary conditions are:

$$\begin{aligned}
M(0,i,0) &= 0, && \text{for } 0 \le i \le n,\\
M(0,0,j) &= 0, && \text{for } 0 \le j \le m,\\
M(k,i,0) &= -\infty, && \text{for } 0 \le i \le n, 1 \le k \le r,\\
M(k,0,j) &= -\infty, && \text{for } 0 \le j \le m, 1 \le k \le r.
\end{aligned}$$

An illustration of the algorithm in work is shown in Fig. 1, where the matrix is presented in a level-wise manner. The CLCS length is located in $M(r,n,m)$ cell. To obtain a CLCS itself, one needs to trace back the matrix according to the cells used to compute the actual cell starting from $M(r,n,m)$, which is easy and fast [6].



k=0

i		0	1	2	3	4	5	6	7	8	9	10	11
j		A	B	A	A	D	A	C	B	A	A	B	C
0	C	0	0	0	0	0	0	1	1	1	1	1	1
1	B	0	1	1	1	1	1	1	2	2	2	2	2
2	C	0	1	1	1	1	1	2	2	2	2	2	3
3	B	0	1	1	1	1	1	2	3	3	3	3	3
4	D	0	1	1	1	2	2	2	3	3	3	3	3
5	A	1	1	2	2	2	3	3	3	4	4	4	4
6	A	1	1	2	3	3	3	3	3	4	5	5	5
7	D	1	1	2	3	4	4	4	4	4	5	5	5
8	C	1	1	2	3	4	4	5	5	5	5	5	6
9	D	1	1	2	3	4	4	5	5	5	5	5	6
10	B	1	2	2	3	4	4	5	6	6	6	6	6
11	A	1	2	3	3	4	5	5	6	7	7	7	7

k=1

i		0	1	2	3	4	5	6	7	8	9	10	11
j		A	B	A	A	D	A	C	B	A	A	B	C
0	C	–	–	–	–	–	–	1	1	1	1	1	1
1	B	–	–	–	–	–	–	1	2	2	2	2	2
2	C	–	–	–	–	–	–	2	2	2	2	2	3
3	B	–	–	–	–	–	–	2	3	3	3	3	3
4	D	–	–	–	–	–	–	2	3	3	3	3	3
5	A	–	–	–	–	–	–	2	3	4	4	4	4
6	A	–	–	–	–	–	–	2	3	4	5	5	5
7	D	–	–	–	–	–	–	2	3	4	5	5	5
8	C	–	–	–	–	–	–	5	5	5	5	5	6
9	D	–	–	–	–	–	–	5	5	5	5	5	6
10	B	–	–	–	–	–	–	5	6	6	6	6	6
11	A	–	–	–	–	–	–	5	6	7	7	7	7

k=2

i		0	1	2	3	4	5	6	7	8	9	10	11
j		A	B	A	A	D	A	C	B	A	A	B	C
0	C	–	–	–	–	–	–	–	–	–	–	–	–
1	B	–	–	–	–	–	–	–	2	2	2	2	2
2	C	–	–	–	–	–	–	–	2	2	2	2	3
3	B	–	–	–	–	–	–	–	3	3	3	3	3
4	D	–	–	–	–	–	–	–	3	3	3	3	3
5	A	–	–	–	–	–	–	–	3	4	4	4	4
6	A	–	–	–	–	–	–	–	3	4	5	5	5
7	D	–	–	–	–	–	–	–	3	4	5	5	5
8	C	–	–	–	–	–	–	–	3	4	5	5	6
9	D	–	–	–	–	–	–	–	3	4	5	5	6
10	B	–	–	–	–	–	–	–	6	6	6	6	6
11	A	–	–	–	–	–	–	–	6	7	7	7	7

k=3

i		0	1	2	3	4	5	6	7	8	9	10	11
j		A	B	A	A	D	A	C	B	A	A	B	C
0	C	–	–	–	–	–	–	–	–	–	–	–	–
1	B	–	–	–	–	–	–	–	–	–	–	–	–
2	C	–	–	–	–	–	–	–	–	–	–	–	–
3	B	–	–	–	–	–	–	–	–	–	–	3	3
4	D	–	–	–	–	–	–	–	–	–	–	3	3
5	A	–	–	–	–	–	–	–	–	–	–	3	3
6	A	–	–	–	–	–	–	–	–	–	–	3	3
7	D	–	–	–	–	–	–	–	–	–	–	3	3
8	C	–	–	–	–	–	–	–	–	–	–	3	4
9	D	–	–	–	–	–	–	–	–	–	–	3	4
10	B	–	–	–	–	–	–	–	–	–	–	6	6
11	A	–	–	–	–	–	–	–	–	–	–	6	6

Fig. 1. Example of the algorithm by Chin *et al.* for A = ABAADACBAABC, B = CBCBDAADCDBA, P = CBB. (Grayed cells denote strong matches and '–' symbols denote $-\infty$ values.)

Matrix M of this algorithm is a concept shared by some other CLCS methods, so we will be referring to it in the rest of the paper.

3.3 Peng Algorithm

The algorithm by Peng [16] was invented independently, but actually is almost identical to the one by Chin *et al.* and can be seen as its variant, since the differences are mainly in the implementation field.

The main recurrence is split according to k into two equations. For $1 \le i \le n$, $1 \le j \le m$:

$$M(0,i,j) = \max \begin{cases} M(0,i-1,j), \\ M(0,i,j-1), \\ M(0,i-1,j-1)+1, & \text{if } a_i = b_j. \end{cases}$$

For $1 \le k \le r$, $1 \le i \le n$, $1 \le j \le m$:

$$M(k,i,j) = \max \begin{cases} M(k,i-1,j), & \\ M(k,i,j-1), & \\ M(k,i-1,j-1)+1, & \text{if } a_i = b_j, \\ M(k-1,i-1,j-1)+1, & \text{if } a_i = b_j \wedge a_i = p_k. \end{cases}$$

The boundary conditions are:

$$\begin{array}{lll} M(0,i,0) & = 0, & \text{for } 0 \le i \le n, \\ M(0,0,j) & = 0, & \text{for } 0 \le j \le m, \\ M(k,i,0) & = -\infty, & \text{for } 0 \le i \le n,\ 1 \le k \le r \\ M(k,0,j) & = -\infty, & \text{for } 0 \le j \le m,\ 1 \le k \le r. \end{array}$$

Finally, the CLCS length is located in $M(r,n,m)$.

3.4 Arslan–Eğecioğlu Algorithm

Another DP-based approach to solve the CLCS problem was proposed by Arslan and Eğecioğlu [1]. They started with the recurrence given by Tsai [19] and simplified it obtaining the following set of equations.

For $0 \le i \le n$, $0 \le j \le m$, $0 \le k \le r$:

$$M(k,i,0) = 0, \qquad M(k,0,j) = 0.$$

For $1 \le i \le n$, $1 \le j \le m$, $0 \le k \le r$:

$$\begin{array}{rcl} M(k,i,j) & = & \max\{M'(k,i,j), M(k,i-1,j), M(k,i,j-1)\}, \\ M'(k,i,j) & = & \max\{M''(k,i,j), M'''(k,i,j)\}, \\ M''(k,i,j) & = & \begin{cases} M(k-1,i-1,j-1)+1 & \begin{array}{l}\text{if } (k=0 \text{ or } (k>0 \\ \text{and } M(k-1,i-1,j-1)>0)) \\ \text{and } a_i = b_j = p_k,\end{array} \\ 0 & \text{otherwise },\end{cases} \\ M'''(k,i,j) & = & \begin{cases} M(k,i-1,j-1)+1 & \begin{array}{l}\text{if } (k=0 \text{ or} \\ \text{or } M(k,i-1,j-1)>0) \\ \text{and } a_i = b_j,\end{array} \\ 0 & \text{otherwise }.\end{cases} \end{array}$$

In this algorithm, there are more than one matrix, but the time and space complexity is still $O(mnr)$. The CLCS length is obtained in $M(r,n,m)$.

3.5 Peng–Ting Algorithm

The first algorithm for the CLCS problem, which does not fill one or more three dimensional matrices cell by cell, was presented by Peng and Ting [17]. Their method is an example of a divide and conquer technique.

A general scheme of the algorithm is:

1. Find the division point $(k, \lceil n/2 \rceil, j)$ for all $0 \leq j \leq m$ and $0 \leq k \leq r$.
2. Compute by recurrence C_1 for $A_{\lceil n/2 \rceil}, B_j, P_k$.
3. Compute by recurrence C_2 for $A_{-\lceil n/2 \rceil}, B_{-j}, P_{-k}$.
4. Return the concatenation of C_1 and C_2.

A key point is to find the j and k indexes in the first step. They are computed as:

$$\arg\max_{\substack{0 \leq j < m \\ 0 \leq k \leq r}} \{M(1, k, 1, \lceil n/2 \rceil, 1, j) + M(k+1, r, \lceil n/2 \rceil + 1, n, j+1, m)\}.$$

The necessary M values are calculated by the following recurrence ($1 \leq i \leq n$, $1 \leq j \leq m$ and $0 \leq k \leq r$):

$$M(1, k, 1, i, 1, j) = \begin{cases} M(1, k-1, 1, i-1, 1, j-1) + 1 & \text{if } a_i = b_j = p_k, \\ M(1, k, 1, i-1, 1, j-1) + 1 & \text{if } a_i = b_j \neq p_k, \\ \max(M(1, k, 1, i-1, 1, j), & \\ \quad\quad M(1, k, 1, i, 1, j-1)) & \text{if } a_i \neq b_j. \end{cases}$$

The boundary conditions are:

$$M(0, i, 0) = 0 \text{ and } M(k, i, 0) = -\infty, \qquad \text{for } 1 \leq k \leq r \text{ and } 0 \leq i \leq n,$$

$$M(0, 0, j) = 0 \text{ and } M(k, 0, j) = -\infty, \qquad \text{for } 1 \leq k \leq r \text{ and } 0 \leq j \leq m.$$

To compute the value of $M(k+1, r, \lceil n/2 \rceil + 1, n, j+1, m)$ the sequences A, B, P are reversed and then the same equations are used.

3.6 Deorowicz Algorithm

In matrix M of the algorithm by Chin *et al.*, in most cases, there are no differences between the neighbor cells, i.e., at least one of the: left or upper cell has the same value as the actual cell. Any important changes appear for matches. This property is often called a *sparsity* of the dynamic programming matrix and there are a number of algorithms for the LCS problem exploiting it. Most of them are based on the Hunt–Szymanski [12] or Hirschberg [11] proposals. There is, however, one reason that makes a direct application of such algorithms difficult for the CLCS problem. In the LCS problem, the neighbor cells differ by at most 1, while in the CLCS problem, the differences can be much larger. Therefore, a careful implementation of the idea is necessary. The first such an algorithm was proposed by Deorowicz [7]. The time complexity of the method is $O(r(ml + d) + n)$, where l is the LCS length for A and B and d is the number of matches between A and B.[3]

[3] This complexity can be alternatively expressed as $O(rn\ell)$, but we use the longer form to stress the dependence on the number of matches also

The main idea is to restrict the computation to only those cells that represent matches. Therefore, the following recurrence is used:

$$M(k,i,j) = \begin{cases} \max\limits_{\substack{0 \le i' < i \\ 0 \le j' < j \\ a_{i'} = b_{j'}}} M(k-1,i',j') + 1 & \text{if } i,j,k > 0 \land a_i = b_j = p_k, \\ \max\limits_{\substack{0 \le i' < i \\ 0 \le j' < j \\ a_{i'} = b_{j'}}} M(k,i',j') + 1 & \text{if } i,j > 0, a_i = b_j \land (k = 0 \lor a_i \neq p_k). \end{cases}$$

The boundary conditions are the same as in the Chin *et al.* algorithm. The key point is to effectively compute the maximum over the cells of M. The details are not difficult, but a detailed description would be to long, so we refer the interested reader to the original paper [7].

3.7 Wang Algorithm

Matrix M of the Chin *et al.* algorithm has several interesting properties that can be used to reduce the number of computations. Wang observed and proved [20] the two following:

- if $M(k-1,i,j) = M(k,i,j)$ then cell $M(k-1,i,j)$ never influences on the final result,
- if $M(k,i,j) = -\infty$ then the values of $M(g,i,j)$ for $k < g \le r$ are also equal $-\infty$.

Therefore, instead of storing the whole $(r+1) \times (n+1) \times (m+1)$ dynamic programming matrix, Wang postulates to maintain $(n+1) \times (m+1)$ matrix and in each cell (i,j) to store a list of only the necessary to compute cells (k,i,j), i.e., the cells that are known to contain no $-\infty$ values and influence on the total result. This strategy helps if for a large number of pairs (i,j) only a few levels are evaluated, which is often a case. The original work [20] contains the necessary pseudocodes, which are too long to be presented here.

3.8 Iliopoulos–Rahman Algorithm

The authors started from the recursive rule by Arslan–Eğecioğlu (see Section 3.4) and modified it in the following way:

$$\begin{aligned}
M(k,i,j) &= \max\{M'(k,i,j), M''(k,i,j), M(k,i-1,j), M(k,i,j-1)\},\\
V_1 &= \max_{1\le i'<i,\, 1\le j'<j,\, a_{i'}=b_{j'}} \{M(k-1,i',j')\}\\
V_2 &= \max_{1\le i'<i,\, 1\le j'<j,\, a_{i'}=b_{j'}} \{M(k,i',j')\}\\
M'(k,i,j) &= \begin{cases} V_1+1 & \text{if } (k=1 \text{ or } (k>1 \text{ and } V_1>0)) \text{ and } a_i=b_j=p_k,\\ 0 & \text{otherwise ,}\end{cases}\\
M''(k,i,j) &= \begin{cases} 1 & \text{if } (i=0 \text{ or } j=0) \text{ and } (a_i=b_j)\\ V_2+1 & \text{if } (k=0 \text{ or } \text{ or } V_2>0) \text{ and } a_i=b_j,\\ 0 & \text{otherwise .}\end{cases}
\end{aligned}$$

The most important problem in this algorithm is the calculation of V_1 and V_2 values. The authors show how the *bounded heap* introduced by Brodal *et al.* [4] can be used to calculate each V_1 and V_2 value in time $O(\log\log n)$. This data structure employs van Emde Boas trees [8] to achieve such complexities. Moreover, all the necessary updates can be made on bounded heaps in amortized time $O(\log\log n)$ per match. The time complexity of Iliopoulos–Rahman algorithm, $O(rd\log\log n)$, looks promising if the total number of matches, rd, is much smaller than mnr, since in the worst case the algorithm is $\Omega(mnr)$-time.

4 IMPROVEMENTS OF THE ALGORITHMS

Similar observations to those of Wang (see Section 3.7) ware made by He and Arslan [10]. They observed that it is unnecessary to compute the whole three dimensional dynamic programming matrix M, since there is no possibility that some its parts impact on the final results obtained in $M(r,n,m)$.[4] If the LCS length of A_i and P_k is less than $k>0$, then all the cells $M(k,i,j)$ for any j store $-\infty$. Moreover, if the LCS length of $A_{i\ldots n}$ and $P_{k\ldots r}$ for $0<k\le r$ is less than $r-k+1$, then all cells $M(k-1,i-1,j)$ for any j have no way to impact on $M(r,n,m)$. (It is clear when we realize that this is the same case as previous when all the sequences A, B, P are reversed.) A similar observations can be made for sequence B. Therefore, each matrix level can be trimmed to only that part, that stores positive values that can affect $M(r,n,m)$ (Fig. 2).

The strong matches are special, in such a sense that they represent the only cells in the matrix which are directly dependent on the matrix cells from a lower level.

[4] To make the description of the method clearer, we present the technique in terms of matrix M by Chin *et al.* even if He and Arslan solved in fact some other problem, namely pairwise sequence alignment

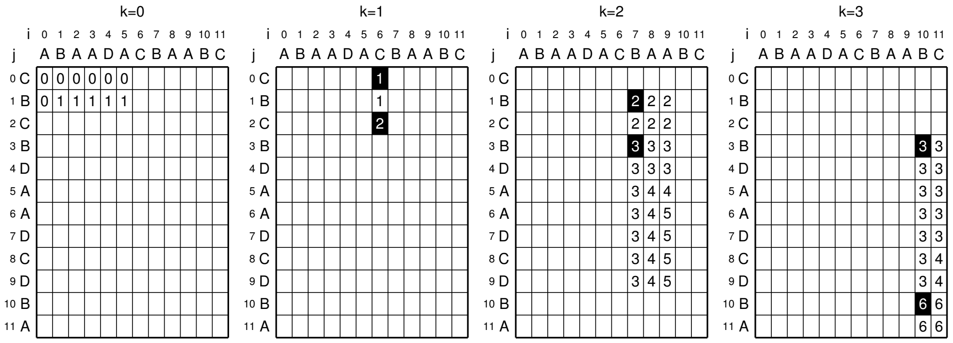


Fig. 2. Example of the algorithm by Chin *et al.* with trimmed levels for A = ABAADACBAABC, B = CBCBDAADCDBA, P = CBB. (Grayed cells denote strong matches and '–' symbols denote $-\infty$ values. The complete levels are shown, however, only the trimmed parts are filled with numbers.)

When we remove from the set of all the strong matches those that do not belong to the trimmed parts of the levels, we obtain the so-called entry-exit points (EEP). These are the only cells necessary to be computed to successfully process the matrix. He and Arslan show how the EEP can be easily found in a fast precomputing stage and that only these cells of previous level need to be stored during the computation of the actual level if we are interested in the CLCS length only.

As was said above, the idea of the EEP was proposed not for the CLCS problem but for a related pairwise sequence alignment (CPSA) problem, so it has not been used for the CLCS problem yet. Since this technique looks promising, we apply it in the algorithms by Chin *et al.* and Deorowicz. (Note, however, that this technique does not improve the worst-case time complexities of the algorithms.) We have not incorporated this technique into other algorithms as:

- Peng and Arslan–Eğecioğlu algorithms use a very similar recurrence to the one by Chin *et al.*,
- Peng and Ting algorithm is a divide and conquer method and the application of this techniques is not easy.
- Wang algorithm employs a similar idea,
- Iliopoulos–Rahman algorithm computes the dynamic programming matrix in some other way and the cells containing $-\infty$ values are efficiently handled.

The application of the EEP technique to the Chin *et al.* algorithm is straightforward. After precomputing the entry-exit points for each level, the computation of the cells of matrix M is restricted to only those cells that are necessary to obtain the values of the EEP, i.e., the trimmed part of each level.

The Deorowicz's algorithm processes M matrix level-wise and in each level—row-wise, so it is also easy to integrate the EEP technique with it. After precomputing the EEP, the processing of each level is reduced to only the trimmed part. The additional cost of the preprocessing is negligible, $O(n)$.

5 EXPERIMENTAL RESULTS

5.1 Implementation Details

The CLCS computing algorithms consume a lot of memory, so the details of the implementation, especially memory management, are important. All the mentioned CLCS computing algorithms were implemented by ourselves as presented in the original papers. We, however, took special care of speed and tried to achieve a similar code optimization level. For example, the formulation of the Chin *et al.* algorithm does not specify in which way to compute matrix M. It is convenient to describe the algorithm ideas when the matrix is computed level-wise and the memory is organized similarly, i.e., the levels are stored one after another. In practice, this approach leads to cache miss problems. If (k, i, j), cell at kth level, is a strong match, cell $M(k-1, i-1, j-1)$ is used in computation, but it is about $4nm$ bytes earlier[5], which even for moderate sizes of the sequences can mean megabytes. It is almost sure that this cell was flushed from the cache memory.

An alternative approach is to transpose (virtually) the matrix to have the mentioned cells at a distance approximately $4rm$ bytes, which in practice increases significantly the amount of cache hits, as r is typically small. In a preliminary experiment, this rearrangement of computations gave about 20% gain in speed, so in further experiments we consider only the faster alternative.

The used compiler was MS Visual C++ 2008 and the source codes were compiled with the maximal optimization for speed. In the experiments, we used a computer equipped with an AMD Phenom II X4 810 processor (2.6 GHz real CPU clock), MS Vista 64-bit, and 4 GB of RAM.

5.2 Data Sets and Testing Methodology

We made a number of experiments on random and real data. Random data were chosen to check the behavior of the algorithms for variable: main sequence lengths, constrained sequence length, size of the alphabet. In random tests, the sequences are produced by uniform random number generator. For tests on real data we built a corpus of sequences proposed for evaluation of algorithms for related problems, i.e., constrained pairwise sequence alignment (CPSA) and constrained multiple sequence alignment (CMSA). Chin et al. in their experiments on the CMSA problem [5] proposed four data sets that contain RNase sequences of lengths from range $[111, 327]$. All these data are included in our corpus. The last data set was taken from Lu and Huang [14]. They made experiments also for the CMSA problem and we picked the data set containing the aspartic acid protease family sequences (Lu and Huang

[5] It is assumed that a cell is stored in 4 bytes

made experiments also for some data from [5]). Main traits of the corpus are given in Table 2.[6] The 0-order entropy calculated for the sequences of each data set show that the frequencies of occurrences of symbols in each data set are relatively close to that of random sequences.

Data Set	Num. Seq.	Contents	Seq. Length (min, med, max)	Alph. Size	H_0	Origin
ds0	7	H-RNase3, H-RNase2, BP-RNaseA, BS-RNase, H-RNaseA, H-RNase4, RC-RNase	(111,124,134)	20	4.172	[5]
ds1	6	gi|119124|sp|P12724|ecp_human, gi|2500564|sp|P70709|ecp_rat, gi|13400006|pdb|ldyt|, gi|20930966|ref|xp_142859.1|, gi|20873960|ref|xp_127690.1|, gi|20930966|ref|xp_142859.1|	(124,149,185)	20	4.246	[5]
ds2	6	gi|20930966|ref|XP_142859.1|, gi|119124|sp|P12724|ECP_HUMAN, gi|2500564|sp|P70709|ECP_RAT, gi|13400006|pdb, gi|20930966|ref|XP_142859.1|, gi|20873960|ref|XP_127690.1|	(131,142,160)	20	4.189	[5]
ds3	5	gi|10068295|gb|AAE40716.1|, gi|17549935|ref|NP_510780.1|, gi|28509297|ref|XP_282983.1|, gi|28499937|ref|XP_204162.2|, gi|4902995|dbj|BAA77929.1	(189,277,327)	20	4.191	[5]
ds4	6	Protease: HTLV-1, RSV, HIV-1, SRV-1, CaMV, 17.6	(98,114,123)	20	4.111	[14]

Table 2. Data sets of real data used in the experiments. Column H_0 contains 0-order entropy of the sequences (in bits). Note that the entropy of uniform random sequence over alphabet of size 20 is about 4.322 bits

In tests on random data, in each experiment 201 triples of sequences of assumed lengths and alphabet size were prepared. Then, the median of the computing time was determined for graphs. In tests on real data, each algorithm is executed on each possible pair of different sequences within each data set and with some assumed constrained sequence. Then, the total time of computations per data set is calculated. These experiments were also run 201 times and medians of the computation time for each pair from a data set were summed up.

The tested algorithms are presented in figures and tables under the names:

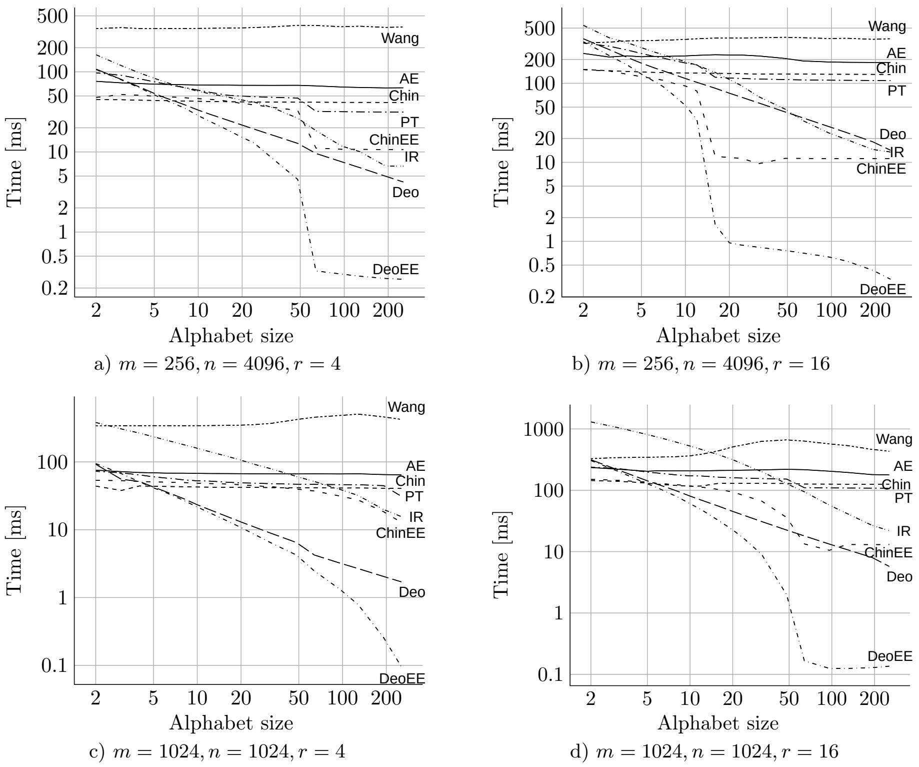
- AE — the algorithm by Arslan and Eğecioğlu [1],
- Chin — the algorithm by Chin *et al.* [6],
- ChinEE — the algorithm by Chin *et al.* with the EEP improvement,
- Deo — the algorithm by Deorowicz [7],

[6] Data sets can be obtained from `http://sun.aei.polsl.pl/~sdeor/pub/do09-ds.zip`

- DeoEE — the algorithm by Deorowicz with the EEP improvement,
- IR — the algorithm by Iliopoulos and Rahman [13],
- PT — the algorithm by Peng and Ting [17],
- Wang — the algorithm by Wang [20].

5.3 CLCS Computing Algorithms

In the first experiment on random data, we measured the influence of the alphabet size on the algorithms' speed. We performed four tests for various sizes of the three sequences (Fig. 3). The alphabet size does not appear explicitly in the time complexities, but for some algorithms it affects indirectly the speed. E.g., the algorithms Deo, DeoEE, and IR process the matches only and the number of matches is less if the alphabet size is large. Moreover, for the EEP-based algorithms, ChinEE and DeoEE, the influence of the alphabet size is also of other kind. The larger alphabet means, approximately, the shorter the resulting sequence and the smaller areas to compute at each level since the 'level trim' effect is bigger and there are less entry-exit points.



a) $m = 256, n = 4096, r = 4$

b) $m = 256, n = 4096, r = 16$

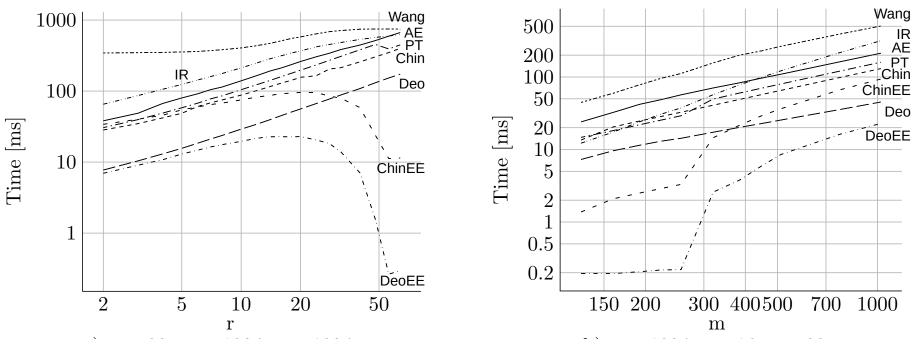
c) $m = 1024, n = 1024, r = 4$

d) $m = 1024, n = 1024, r = 16$

Fig. 3. Comparison of the CLCS computing time for changing alphabet size

For small alphabet sizes (less than 5) the fastest algorithm is the one by Chin *et al.* It computes always the whole three dimensional matrix, but does it extremely simply. The more sophisticated algorithms, like Deo and DeoEE, pay for their intricacy and due to large number of processed matches cannot beat Chin. When the alphabet size grows, the Deo and DeoEE algorithms clearly win. Also the IR and the ChinEE algorithms are quite fast for large alphabets, thanks to small number of matches and the 'level trim' effect, respectively.

The main field, in which the CLCS problem appears is bioinformatics, where, e.g., protein and RNase sequences are compared. The alphabet size for them is 20. The frequency of symbols is not exactly uniform, but for simplicity we used, in the next tests, the same uniform generator with the alphabet size fixed to 20. Figure 4 shows the influence of: a) the constrained sequence length, b) the shorter main sequence length, on the speed.



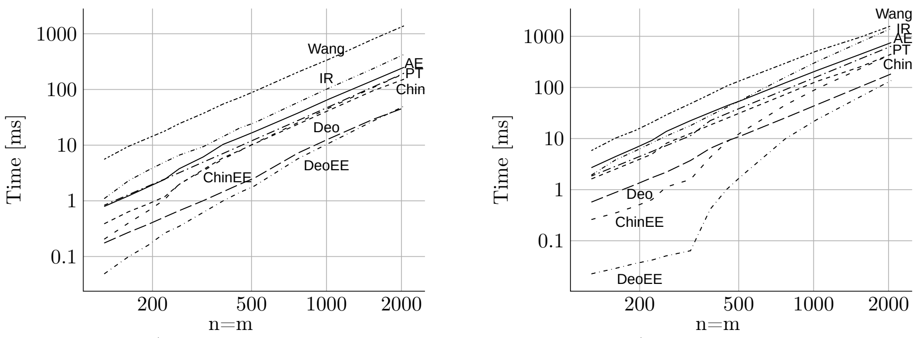
a) $\sigma = 20, m = 1024, n = 1024$ b) $n = 1024, r = 16, \sigma = 20$

Fig. 4. Comparison of the CLCS computing time for the: a) changing constrained sequence length, b) changing shorter main sequence length

As one would expect, the non-EEP-based algorithms slow down when the constrained sequence gets longer. It is because the computation time is proportional to the length of the constrained sequence for the algorithms computing the whole three dimensional matrix. For the EEP-based methods, the matrix size also grows when the constrained sequence comes longer and the algorithms primarily slow down, but the 'level trim' effect finally dominates and the speed up for long enough sequences can be observed.

In the last experiment of this series, we measured the impact of the main sequences length (Fig. 5). The Deo and DeoEE algorithms win, and the difference between them gets smaller when the sequences get longer. It is a result of the smaller 'level trim' effect for long main sequences.

The results for real data are given in Table 3. They confirm our conclusions drawn from simulating the bioinformatic sequences. The fastest algorithm, DeoEE, is 2.0–4.6 times faster than Deo and 3.2–5.4 times faster than ChinEE. The other algorithms are much slower. As can be observed for tests on data set ds1, the



a) $r = 4, \sigma = 20$ b) $r = 16, \sigma = 20$

Fig. 5. Comparison of the CLCS computing time for changing the length of the main sequences

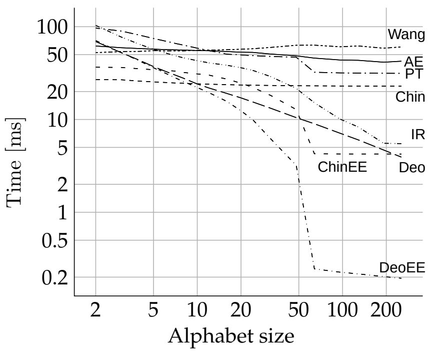
speedup of DeoEE over Chin grows when the constraint comes longer. Also the total time of computations made by EEP-based methods (ChinEE and DeoEE) is smaller for longer constraints due to 'level trim' effect.

Table 3. CLCS computation times (in ms) of algorithms on real data. In parenthesis: the number of times the given algorithm is faster than Chin

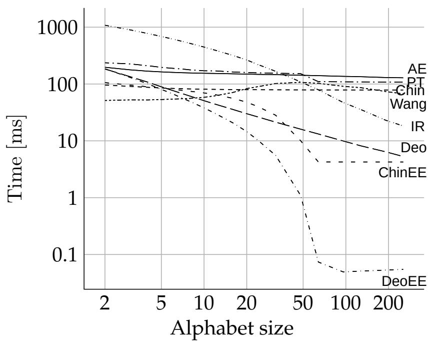
AE	Chin	ChinEE	Deo	DeoEE	IR	PT	Wang
			data set ds0, P = HKH				
10.827	5.969	3.871	3.012	**1.117**	22.084	12.504	116.018
(0.55)	(1.00)	(1.54)	(1.98)	(**5.34**)	(0.27)	(0.48)	(0.05)
			data set ds1, P = HKH				
11.964	6.662	5.442	3.227	**1.567**	23.247	13.557	137.817
(0.56)	(1.00)	(1.22)	(2.06)	(**4.25**)	(0.29)	(0.49)	(0.05)
			data set ds1, P = HKSH				
16.055	8.258	5.030	4.012	**1.423**	26.603	16.584	137.823
(0.51)	(1.00)	(1.64)	(2.06)	(**5.81**)	(0.31)	(0.50)	(0.06)
			data set ds1, P = HKSTH				
19.760	9.818	4.831	4.801	**1.393**	30.574	19.360	143.683
(0.50)	(1.00)	(2.03)	(2.04)	(**7.05**)	(0.32)	(0.51)	(0.07)
			data set ds2, P = HKSH				
13.631	7.047	3.745	3.534	**1.070**	24.701	14.447	110.691
(0.52)	(1.00)	(1.88)	(1.99)	(**6.59**)	(0.29)	(0.49)	(0.06)
			data set ds2, P = HKSTH				
16.770	8.386	3.310	4.207	**0.915**	27.752	16.964	113.112
(0.50)	(1.00)	(2.53)	(1.99)	(**9.16**)	(0.30)	(0.49)	(0.07)
			data set ds3, P = HKH				
27.313	17.220	15.676	5.868	**2.878**	43.931	24.778	280.309
(0.63)	(1.00)	(1.10)	(2.93)	(**5.98**)	(0.39)	(0.69)	(0.06)
			data set ds4, P = DGGG				
8.461	4.376	2.502	2.329	**0.791**	14.014	9.178	69.436
(0.52)	(1.00)	(1.75)	(1.88)	(**5.53**)	(0.31)	(0.48)	(0.06)

5.4 CLCS Length Computing Algorithms

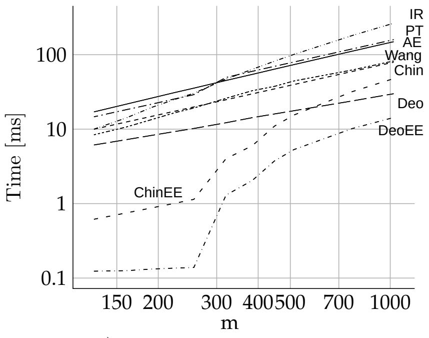
Sometimes the computation of a CLCS is superfluous, and only its length is needed. This length is a dual of so-called indel distance (the number of insertions and deletions of symbols necessary to transform the first sequence into the second). In such a case, the algorithms can work significantly faster. The time complexity does not change, but the memory consumption could be much lower. E.g., in the Chin *et al.* algorithm, it suffice to store only two levels: the current and the previous to compute $M(r, n, m)$ cell. The gain in memory consumption may lead to much faster computation since the memory (especially cache) is much better used. Moreover, the CLCS computing algorithms usually need to make some kind of trace back to obtain a CLCS, after computing its length, and now this is also unnecessary. Therefore, for the second set of experiments on random data (Fig. 6), we modified the original algorithms to make use of this simpler problem properties.



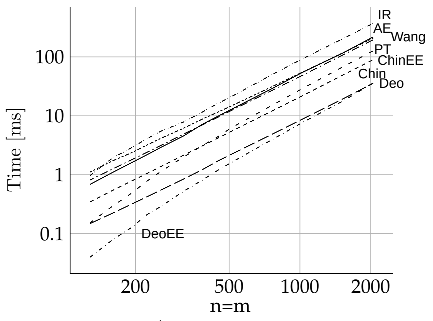
a) $m = 256, n = 4096, r = 4$



b) $m = 1024, n = 1024, r = 16$



c) $n = 1024, r = 16, \sigma = 20$



d) $r = 4, \sigma = 20$

Fig. 6. Comparison of the CLCS length computing time for various parameters

The algorithms for the simpler problem are faster by a factor from 1.5 (Deo) to 6.0 (Wang). Nevertheless, still the fastest for a typical ($\sigma = 20$) and large alphabets are the algorithms DeoEE, Deo, ChinEE.

The results for real data are given in Table 4. Similarly, like in the CLCS computing experiments, they confirm the results on random data. The fastest algorithm is DeoEE which is 2.1–5.1 times faster than Deo and 3.1–3.5 times faster than ChinEE. The other algorithms are much slower. It can be also noticed that most algorithms are about 20% faster when computing only the CLCS length rather than a CLCS (cf. Tables 3 and 4).

Table 4. CLCS computation times (in ms) of algorithms on real data. In parenthesis: the number of times the given algorithm is faster than Chin

AE	Chin	ChinEE	Deo	DeoEE	IR	PT	Wang
			data set ds0, P = HKH				
9.922	5.446	3.078	2.625	**0.922**	20.403	12.935	17.995
(0.55)	(1.00)	(1.77)	(2.07)	(**5.90**)	(0.27)	(0.42)	(0.30)
			data set ds1, P = HKH				
10.822	6.036	4.333	2.810	**1.310**	21.375	14.190	20.686
(0.56)	(1.00)	(1.39)	(2.15)	(**4.61**)	(0.28)	(0.43)	(0.29)
			data set ds1, P = HKSH				
16.407	7.565	3.917	3.499	**1.159**	24.405	17.278	20.620
(0.46)	(1.00)	(1.93)	(2.16)	(**6.53**)	(0.31)	(0.44)	(0.37)
			data set ds1, P = HKSTH				
19.270	9.051	3.734	4.193	**1.128**	28.003	20.488	21.539
(0.47)	(1.00)	(2.42)	(2.16)	(**8.03**)	(0.32)	(0.44)	(0.42)
			data set ds2, P = HKSH				
13.659	6.486	2.940	3.068	**0.869**	22.703	15.016	16.434
(0.47)	(1.00)	(2.21)	(2.11)	(**7.46**)	(0.29)	(0.43)	(0.39)
			data set ds2, P = HKSTH				
16.037	7.739	2.526	3.666	**0.720**	25.416	17.717	16.809
(0.48)	(1.00)	(3.06)	(2.11)	(**10.75**)	(0.30)	(0.44)	(0.46)
			data set ds3, P = HKH				
20.497	10.986	8.328	5.020	**2.389**	40.547	25.859	38.216
(0.54)	(1.00)	(1.32)	(2.19)	(**4.60**)	(0.27)	(0.42)	(0.29)
			data set ds4, P = DGGG				
8.858	4.040	1.986	2.019	**0.645**	12.860	9.447	10.614
(0.46)	(1.00)	(2.03)	(2.00)	(**6.26**)	(0.31)	(0.43)	(0.38)

6 CONCLUSIONS

We described the existing algorithms solving the CLCS problem. Then, we modified the algorithms by Chin *et al.* and Deorowicz by incorporation of the entry-exit points technique proposed by He and Arslan for the pairwise sequence alignment problem. All the algorithms were implemented and evaluated in practice for the CLCS problem and the simpler, CLCS length only problem, for both random and real data.

The results show that among the classical algorithms (without the EEP improvement) the fastest one is the algorithm by Chin *et al.* or by Deorowicz. In experiments on random data, which is the winner depends on the alphabet size. For small alphabets (of size less than 5) Chin *et al.* wins, but for larger, Deorowicz dominates the rest, and the difference becomes bigger when the alphabet size grows. In a typical case, $\sigma = 20$, the algorithm by Deorowicz is more than 5 times faster than the other methods. Experiments on real data (RNase and protein sequences) show that the algorithm introduced in this paper is about 2–3 times faster than the second best algorithm.

An application of the entry-exit points can give a significant speedup. The obtained gain depends on the data, since when the levels are trimmed only a bit, it can even make a little slow-down. Fortunately, the possible speedup is huge, and the cases in which the slow-downs were observed are rare, so the answer to the question whether the EEP technique should be used is definitely positive.

The most important open question related to the CLCS problem concerns the $O(mnr)$ worst-case time complexity. Today, no existing algorithm breaks this barrier and we also do not know if it is possible.

7 ACKNOWLEDGEMENTS

The research of this project was partially supported by the Minister of Science and Higher Education grant 3177/B/T02/2008/35. The authors thank Szymon Grabowski for reading preliminary versions of the paper and suggesting improvements.

REFERENCES

[1] Arslan A.N., Eğecioğlu O.: *Algorithms for the constrained longest common subsequence problems*, International Journal of Foundations of Computer Science 16(6): 1099–1109, 2005.

[2] Apostolico A. *General pattern matchings.* Chapter in *Handbook of Algorithms and Theory of Computation*, M.J. Atallah (Editor), Chapter 13, 1998.

[3] Bergroth L., Hakonen H., Raita T. *A survey of longest common subsequence algorithms.* In Proceedings of 7th International Symposium on String Processing Information Retrieval (SPIRE), Curuña, Spain, pp. 39–48, 2000.

[4] Brodal G.S., Kaligosi K., Katriel I., Kutz M. *Faster algorithms for computing longest common increasing subsequences.* In Lewenstein M. and Valiente G., editors, CPM, volume 4009 of Lecture Notes in Computer Science, pp. 330–341. Springer, 2006.

[5] Chin F.Y.L., Ho N.L., Lam T.W., Wong P.W.H. *Efficient Constrained Multiple Sequence Alignment with Performance Guarantee.* Journal of Bioinformatics and Computational Biology, 3(1): 1–8, 2005.

[6] Chin F.Y.L., De Santis A., Ferrara A.L., Ho N.L., Kim S.K.: *A simple algorithm for the constrained sequence problems*, Information Processing Letters, 90: 175–179, 2004.

[7] Deorowicz S.: *Fast Algorithm for Constrained Longest Common Subsequence Problem*, Theoretical and Applied Informatics, 19(2): 91–102, 2007.

[8] van Emde Boas P., Kaas R., Zijlstra E., *Preserving order in a forest in less than logarithmic time and linear space*, Information Processing Letters 6(3): 80–82, 1977.

[9] Gusfield D.: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[10] He D., Arslan A.N.: *A Space-Efficient Algorithm for the Constrained Pairwise Sequence Alignment Problem*, Genome Informatics 16(2): 237–246, 2005.

[11] Hirschberg D.S.: *Algorithms for the longest common subsequence problem*, Journal of the ACM 24: 664–675, 1977.

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

[13] Iliopoulos C.S., Rahman M.S.: *New Efficient Algorithms for LCS and Constrained LCS Problem*, Information Processing Letters 106(1): 13–18, 2008.

[14] Lu Ch.L, Huang Y.P. *A Memory-Efficient Algorithm for Multiple Sequence Alignment with Constraints.* Bioinformatics 21(1): 20–30, 2005.

[15] Navarro G. *A Guided Tour to Approximate String Matching.* ACM Computing Surveys, 33(1): 31–88, 2001.

[16] Peng Ch.-L.: *An Approach for Solving the Constrained Longest Common Subsequence Problem.* Master's Thesis, Department of Computer Science and Engineering, National Sun Yat-sen University, Taiwan, 2003. `http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/getfile?URN=etd-0828103-125439&filename=etd-0828103-125439.pdf`

[17] Peng Z.S., Ting H.F.: *Time and Space Efficient Algorithms for Constrained Sequence Alignment*, In Proceedings of the Implementation and Application of Automata, 9th International Conference (CIAA), pp. 237–246, 2004.

[18] Tang C.Y., Lu C.L., Chang M.D.-T., Tsai Y.-T., Sun Y.-J., Chao K.-M., Chang J.-M., Chiou Y.-H., Wu C.-M., Chang H.-T., Chou W.-I. *Constrained multiple sequence alignment tool development and its application to RNase family alignment.* In Proceedings of the first IEEE Computer Society Bioinformatics Conference (CSB 2002), pp. 127–137, 2002.

[19] Tsai Y.-T.: *The constrained common subsequence problem*, Information Processing Letters, 88: 173–176, 2003.

[20] Wang W.-L., *Longest Common Subsequence with Constraints*, Master's Thesis, Department of Computer Science and Information Engineering National Chi-NanUniversity, R. O. C., June, 2006. `http://alg.csie.ncnu.edu.tw/or/WLWang2006.pdf`