Quick-MLCS: A New Algorithm for the Multiple Longest Common Subsequence Problem

Majid Sazvar
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
sazvar@stu-mail.um.ac.ir

Mahmoud Naghibzadeh
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
naghibzadeh@um.ac.ir

Nayyereh Saadati
Ghaem Hospital
Mashhad University of Medical Sciences
Mashhad, Iran
saadatin@mums.ac.ir

ABSTRACT

Finding the longest common subsequence (LCS) of multiple strings is a well-known problem that has many applications in various fields, such as computational biology and computational genomics. This problem has been studied by a number of researchers and over the years, its complexity has been improved from various aspects. This paper presents a new algorithm for the general case of multiple LCS (MLCS) which is based on one of the fastest existing algorithms. The proposed algorithm is founded on the dominant point approach and uses a linear sorting technique to minimize the dominant points set. The main idea is that, after linearly sorting dominant points, a one-pass linear algorithm can minimize the dominant points set. The results of theoretical and experimental evaluations indicate that the efficiency of the newly proposed algorithm in different scenarios is better than the fastest existing algorithm.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *Computations on discrete structures, Sorting and searching.*

General Terms

Algorithms, Performance.

Keywords

Longest common subsequence, multiple longest common subsequence, dominant point approach, linear sorting, DNA sequence, protein sequence.

1. INTRODUCTION

The multiple longest common subsequence (MLCS) problem attempts to find the longest common subsequence between two or more input strings. This problem, in general, is NP-Hard [15] and is a common task in the sequence comparison of DNA sequences and amino acid sequences of proteins [3][5][18]. The application of the MLCS problem is not only limited to the field of computational biology but has many applications in file comparison [1] and text processing [19]. In all of these applications, the MLCS problems are identical and differ only in the alphabet, numbers and size of their sequences.

So far, significant efforts have been made to find an efficient algorithm for the general MLCS problem [9][6][13][14][22] and for special cases with a limited number of strings [10][16][17][21][9][20]. These works can be classified into two main categories; classical methods based on the dynamic programming technique and methods based on the dominant point. Generally, in most applications, the MLCS problem for many strings arises and so a more efficient algorithm for the problem is requested.

This paper presents a fast algorithm for the MLCS problem having any number of strings. This new algorithm is based on work done in [22], but with the difference that a linear sorting technique is utilized to minimize the dominant points set. The central concept is that, a one-pass linear algorithm can minimize the dominant points set after it has linearly sorted them. This method is more efficient than the divide and conquer technique used in [22] for the minimization of the dominant points set. The findings of theoretical and experimental evaluations show that the efficiency of the presented algorithm in different scenarios exceeds that of the algorithm presented in [22].

The rest of the paper is organized as follows. In the next section, the basic concepts of the MLCS problem are presented and related work is investigated. Section 3 presents the new algorithm which is based on the dominant point approach. In Section 4, the efficiency of the newly proposed algorithm is compared with that of the best work reported so far. Finally, in Section 5, the results of the evaluation are analyzed and possible future extensions are presented.

2. BASIC CONCEPTS AND RELATED WORK

2.1 Basic Definitions

Suppose that $\mathbf{a}$ is a finite sequence of elements from the Σ alphabet. The length of $\mathbf{a}$ is represented with $|\mathbf{a}|$. $\mathbf{a}[i]$ is the *i*th element of $\mathbf{a}$ and $\mathbf{a}[i{:}j]$, $1 \le i \le j \le |\mathbf{a}|$ denotes the sequence $\mathbf{a}[i]$, $\mathbf{a}[i+1]$, ..., $\mathbf{a}[j]$.

Definition 1. If $\mathbf{a}$ and $\mathbf{b}$ are finite sequences from the Σ alphabet, then $\mathbf{a}$ is said to be a *subsequence* of $\mathbf{b}$ if there exists a monotonically increasing sequence of integers $r_1, r_2, ..., r_{|a|}$ such that $\mathbf{a}[i] = \mathbf{b}[r_i]$, $1 \le i \le |\mathbf{a}|$.

Definition 2. If $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are finite sequences from the Σ alphabet, then $\mathbf{c}$ is said to be a *common subsequence* of $\mathbf{a}$ and $\mathbf{b}$ iff $\mathbf{c}$ is a subsequence of both $\mathbf{a}$ and $\mathbf{b}$. The *longest common subsequence* is the common subsequence with the maximal length.

For example, “ACA” is a common subsequence of “ACAGTAG” and “CTTAGCA”, while “CTAG” and “CAGA” are the LCS of the two strings.

The algorithms so far presented for the MLCS problem can be classified into two main categories: classical methods based on the dynamic programming technique and methods based on the dominant point. Dynamic programming algorithms are simpler and more straightforward than those of the dominant point. However, in situations in which there is no restriction on the number of sequences, these algorithms are very time-consuming and take up much space in the cases of very large sequences [9].

2.2 Dynamic Programming Approach

The dynamic programming approach to solving the MLCS problem will first be explained as it is the basis of most algorithms for this problem. The main strategy behind the dynamic programming method is to successively evaluate the distance between longer and longer prefixed strings until the final result is obtained.

More formally, given two sequences, **a** and **b,** of the length |**a**| and |**b**|, respectively, a dynamic programming algorithm iteratively builds an (|**a**|+1) × (|**b**|+1) score matrix L in which $L[i,j]$, $0 \le i \le |\mathbf{a}|$ and $0 \le j \le |\mathbf{b}|$ denote the length of an LCS between the **a**[1:i] and **b**[1:j] prefixes. $L[i,0] = L[0,j] = 0$ for $0 \le i \le |\mathbf{a}|$, $0 \le j \le |\mathbf{b}|$ and

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & if\ a[i]=b[j] \\ \max\{L[i-1,j], L[i,j-1]\} & otherwise \end{cases}$$

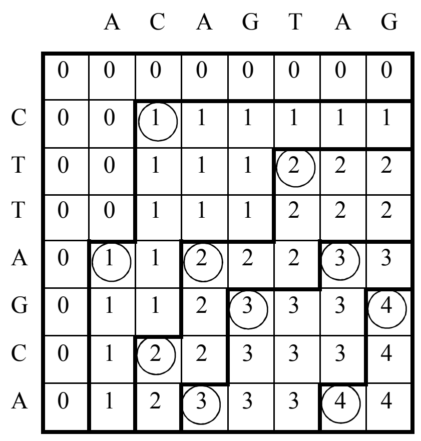
The boundary condition simply means that the length of an LCS between any string and a null string is zero. The definition of the matrix L can be naturally generalized to a case of d sequences: for each position $L[i_1,i_2,\ldots,i_d]$, its value is defined through the immediately preceding positions.

The maximal value in the matrix L is the length of LCS (i.e. $|LCS|$). LCS itself can be found by backtracking from $L[|\mathbf{a}|,|\mathbf{b}|]$ and, at each step, by either (a) following pointers which were set during the calculation of the values or (b) by recalculating the predecessor which yielded the value of the current matrix position. Each time a match is made (the first rule applies), a symbol to the LCS is found. In general, there may be several such paths because the LCS is not necessarily unique. As an example, the score matrix L corresponding to two sequences, **a** = “CTTAGCA” and **b** = “ACAGTAG”, is shown in Figure 1.

The resulting dynamic programming algorithm has a time and space complexity of $O(n^d)$ for d sequences of the length n [11]. Various approaches have been introduced to improve the efficiency of dynamic programming [10][16][17][2]. Unfortunately, however, these approaches are not applicable to the general MLCS problem with any number of strings.

2.3 Dominant Point Approach

In contrast to dynamic programming, in the dominant point approach only some positions of the score matrix L are calculated. The basic concept of this approach is that the longest common subsequence is uniquely defined on the match points; therefore, there is no need to compute all the matrix points of L [4]. Let L be the score matrix for a set of d sequences $a_1, a_2, \ldots, a_d$ over a finite alphabet $\sum$. A point p in the matrix L is denoted as $p = [p_1, p_2, \ldots, p_d]$, where p_i is the ith coordinate of the position p in L. The value at position p of the matrix L is denoted as $L[p]$.



		A	C	A	G	T	A	G
	0	0	0	0	0	0	0	0
C	0	0	1	1	1	1	1	1
T	0	0	1	1	1	2	2	2
T	0	0	1	1	1	2	2	2
A	0	1	1	2	2	2	3	3
G	0	1	1	2	3	3	3	4
C	0	1	2	2	3	3	3	4
A	0	1	2	3	3	3	4	4

Figure 1. The score matrix of two sequences a = “CTTAGCA” and b = “ACAGTAG”.

Definition 3. A point $p = [p_1, p_2, \ldots, p_d]$ in L is a *match point* iff

$$a_1[p_1] = a_2[p_2] = \cdots = a_d[p_d]$$

Definition 4. A point $p = [p_1, p_2, \ldots, p_d]$ *dominates* a point $q = [q_1, q_2, \ldots, q_d]$, if $p_i \le q_i$, for all $i = 1,2,\ldots,d$ (denoted by $p \preccurlyeq q$). For example, in Figure 1, point [1,2] dominates point [4,3].

It is clear that a common subsequence of $a_1, a_2, \ldots, a_d$ corresponds to a chain of dominant relations, and a LCS of the strings corresponds to the longest such chain.

Definition 5. A point $p = [p_1, p_2, \ldots, p_d]$ is a *predecessor* of another point $q = [q_1, q_2, \ldots, q_d]$, if $p_i < q_i$, for all $i = 1,2,\ldots,d$ (written as $p \prec q$). Alternatively, it is said that q is a *successor* of p.

A point $p = [p_1, p_2, \ldots, p_d]$ does not dominate a point $q = [q_1, q_2, \ldots, q_d]$ (denoted as $p \not\preccurlyeq q$), if $\exists i, 1 \le i \le d$, for which $q_i < p_i$. Note that $p \not\preccurlyeq q$ does not necessarily imply $q \preccurlyeq p$, i.e. for some points p and q; $p \not\preccurlyeq q$ and $q \not\preccurlyeq p$ may be true at the same time. In these situations, points p and q are said to be *independent*. For example, in Figure 1, neither [1,2] dominates [4,1] nor [4,1] dominates [1,2].

Definition 6. A match p is *dominant* or *k-dominant* iff $L[p] = L[p_1,p_2,\ldots,p_d] = k$ and $L[q] < k$ for all q such that $[0,0,\ldots,0] \preccurlyeq q \preccurlyeq p$ and $q \ne p$, that is, the dominants on the level k.

Define D^k as the set of k-dominants. The set of all dominant points (that is, k-dominants for all k) are denoted as D. In Figure 1, the regions of the same entry values are bounded by contours. The corner points of these contours are dominant points. These are the critical points which are sufficient to define the overall contour shape.

Definition 7. A match q is a $(k+1)$-dominant iff there is a k-dominant p with $p \prec q$ and there is no match r such that $p \prec r \preccurlyeq q$ and $r \ne q$ for any k-dominant p with $p \prec q$.

As a general iteration step, a set of $(k+1)$-dominants is calculated from a set of k-dominants, $1 \le k \le |LCS| - 1$. Thus, by advancing from one contour to another, all dominant points can be obtained.

Algorithm 1 Find Multiple Longest Common Subsequence

```
1:  procedure Quick-MLCS({$s_1$,$s_2$,...,$s_d$}, $\Sigma$)
2:      // Step 1: Calculation of dominant points
3:      Preprocessing;
4:      $D^0$ = {[0, 0, ..., 0]}; k = 0;
5:      while $D^k$ not empty do
6:          $D^{k+1}$ = Minima(Succ($D^k$, $\Sigma$));
7:          k = k + 1;
8:      end while
9:      // Step 2: Calculation of MLCS-optimal path
10:     pick a point p = [$p_1$, $p_2$, ..., $p_d$] ∈ $D^{k-1}$;
11:     while k – 1 > 0 do
12:         current LCS position = $a_1[p_1]$;
13:         pick a point q ∈ $D^{k-2}$ such that p ∈ Succ(q, $\Sigma$);
14:         p = q;
15:         k = k – 1;
16:     end while
17: end procedure
```

Figure 2. The pseudocode of Quick-MLCS

Figure 1 illustrates the process of the dominant point approach for the two sequences **a** and **b**. In Figure 1, the dominant points of the same level are encircled and tagged with the level number. Two dominant points, [1,2] and [4,1], of level 1 (1-dominant) are found at the first step. At the second step, three dominant points, [2,5], [4,3] and [6,2], of level 2 are detected based on the previous dominant points set {[1,2], [4,1]}, and so on.

The dominant point approach has been successfully applied to the case of two sequences [2][7][12]. In [9], three dominant point algorithms for three or more sequences were proposed. Recently, Wang et al. [22] presented a dominant point based algorithm for the MLCS problem with any number of strings. The sequential version of this algorithm, Quick-DP, is the fastest algorithm found in literature. Quick-DP uses an iterative method for computing dominant points in each level and then, after sorting these points, it employs a divide and conquer technique to minimize them. The performance of the minimization algorithm in Quick-DP decreases with the growth of the cardinality of the dominant points set. In this paper, Quick-DP is utilized as the base method.

3. THE NEW ALGORITHM, QUICK-MLCS

In this section, a new dominant point based algorithm is presented for the MLCS problem with any number of sequences, the Quick-MLCS. However, before presenting this algorithm, a few assumptions and definitions are needed.

It is assumed that $\mathbf{a}_1,\mathbf{a}_2,\ldots,\mathbf{a}_d$ are sequences from the Σ alphabet, and that $|\mathbf{a}_1| = |\mathbf{a}_2| = \ldots = |\mathbf{a}_d| = n$ are only taken for convenience.

Definition 8. A match q is called an *σ-successor* (a successor with respect to the σ symbol $\sigma \in \Sigma$) of a point p, if $p \prec q$ and there is no other match r of σ, such that $p \prec r \prec q$. The *σ-successor* of p is denoted as $p(\sigma)$. The set of *σ-successor* for p is indicated as $Succ(p,\sigma)$, i.e., $Succ(p,\sigma) = \{p(\sigma)\}$. The set of all *σ-successors* for

Algorithm 2 Compute the Minima of Dominant Points

```
1:  function Minima(A)
2:      if (size(A) = 1) then
3:          return A;
4:      end if
5:      n = max value in ith-dimensional coordinate;
6:      sorted = array [n][d];
7:      for all p ∈ A do
8:          if (sorted[p[i]] is empty) or (p < sorted[p[i]]) then
9:              sorted[p[i]] = p;
10:         end if
11:     end for
12:     min = [∞, ∞, ..., ∞];
13:     A = ∅;
14:     for all p ∈ sorted do
15:         if (p < min) then
16:             min = p;
17:             A = A ∪ p;
18:         end if
19:     end for
20:     return A;
21: end function
```

Figure 3. The pseudocode of Minima algorithm

a set of points A is denoted as $Succ(A,\sigma)$. The set of all successors, $\bigcup_{\sigma\in\Sigma} Succ(A,\sigma)$, for A is written as $Succ(A,\Sigma)$.

Definition 9. A point p in a set of points A is called a *minimal* element of A, if, for all $q \in A - \{p\}$, $q \nprec p$. The *minima* problem consists of finding all the minimal elements of A.

Define $Succ(D^k, \sigma)$ as $\{p(\sigma) \mid p \in D^k\}$ and $Succ(D^k,\Sigma)$ as $\{p(\sigma) \mid p \in D^k, \sigma \in \Sigma\}$. $Succ(D^k,\Sigma)$ is the set of the candidates of (k+1)-dominants.

It was proven in [14] that (k+1)-dominants, $D^{(k+1)}$, $0 \le k \le |MLCS| - 1$, constitute exactly the minima of the successor set $Succ(D^k,\Sigma)$ of k-dominants D^k, i.e.

$$D^{k+1} = minima\ Succ(D^k, \Sigma),$$

where *minima* is an algorithm that returns the minima of a set of points.

3.1 Main Idea

The Quick-MLCS consists of two main parts (Figure 2). In the first part, the set of all dominants is calculated iteratively, starting from the *0*-dominant set (containing one element). The set of (k+1)-dominants, $D^{(k+1)}$, is obtained based on the set of k-dominants D^k. In the second part, a path corresponding to an MLCS is found by tracing back through the sets of dominant points obtained in the first part of the algorithm, starting with an element from the last dominant set. If needed, all MLCS, can be enumerated systematically.

The most important part of the Quick-MLCS algorithm is the minimization part. This follows a two-step procedure to compute the minima of the successor set $Succ(D^k,\Sigma)$. In the first step, all points are linearly sorted based on the ith dimensional coordinate. In this process, if the space of a key is already occupied, then a new point is inserted if it can dominate the existing one. This means that, among points with duplicate keys, only the minimal points will remain. In the second step, a linear algorithm passes through the sorted points and removes those that are dominated by the previous ones (Figure 3).

To better understand the logic behind the Quick-MLCS algorithm, an example is given in Table 1. This example shows the trace table of the Quick-MLCS for two sequences, **a** = "CTTAGCA" and **b** = "ACAGTAG".

Table 1. The trace table of Quick-MLCS for two sequences a="CTTAGCA" and b="ACAGTAG"

k	D^k	$Succ(D^k,\Sigma)$	**minima** $Succ(D^k,\Sigma)$
0	*{[0, 0]}*	*{[1, 2], [4, 1], [2, 5], [5, 4]}*	*{[1, 2], [4, 1]}*
1	*{[1, 2], [4, 1]}*	*{[4, 3], [2, 5], [5, 4], [6, 2], [7, 3]}*	*{[2, 5], [4, 3], [6, 2]}*
2	*{[2, 5], [4, 3], [6, 2]}*	*{[4, 6], [5, 7], [5, 4], [7, 6], [7, 3]}*	*{[4, 6], [5, 4], [7, 3]}*
3	*{[4, 6], [5, 4], [7, 3]}*	*{[5, 7], [7, 6]}*	*{[5, 7], [7, 6]}*
4	*{[5, 7], [7, 6]}*	*{}*	*{}*

3.2 Successor Table

The Quick-MLCS needs a preprocessing step at the beginning of its algorithm (as in [22]) that efficiently calculates all the successors of each dominant point. In this step, a successor matrix **T** = $\{T[\sigma,j,i]\}$, $\sigma \in \Sigma$, $0 \le j \le max_{1\le k\le d}\{|a_k|\}$, $1 \le i \le d$, where each element $T[\sigma,j,i]$ specifies the position of the first occurrence of the character σ in the ith sequence, starting from the (j+1)st position in that sequence. If σ no longer occurs in the ith sequence, the value of $T[\sigma,j,i]$ is equal to $1 + max_{1\le k\le d}\{|a_k|\}$. With the matrix **T**, the σ-successor $p = [p_1, p_2, \dots, p_d]$ of a point $q = [q_1, q_2, \dots, q_d]$ can be calculated in O(d) time using the formula

$$p_i = T(\sigma, q_i, i), 1 \le i \le d$$

The calculation of this successor matrix **T** takes O($n|\Sigma|d$) time, where $|\Sigma|$ is the size of the Σ alphabet. The successor matrix **T**, calculated for the two sequences **a** = "CTTAGCA" and **b** = "ACAGTAG", is shown in Figure 4.

3.3 Complexity Analysis

Let $|D|$ be the size of the dominant point set D. Since it was already stated that $Succ(q,\sigma)$ can be calculated in O(d) time, it can be concluded that it takes O($|\Sigma|d$) time to compute the successor set $Succ(q,\Sigma)$, $q \in \Sigma$. The time for computing the minima of a point set is linear. Therefore, O($|D|d$) is the time needed to compute the minima of each σ-successor set for all $\sigma \in \Sigma$. Hence, the complexity of minima is O($|D||\Sigma|d$).

As was said previously in section 3.2, the calculation of the successor matrix **T** takes O($n|\Sigma|d$) time; thus, the overall complexity of the Quick-MLCS is

$$O(n|\Sigma|d + |D||\Sigma|d).$$

The space complexity of the algorithm can be easily estimated as O($n|\Sigma|d + nd + |D|d$): O($n|\Sigma|d$) for storing the successor matrix **T**, O(nd) for linearly sorting points and, finally, O($|D|d$) for minimizing the dominant points.

a = "CTTAGCA"									**b** = "ACAGTAG"								
	-	C	T	T	A	G	C	A		-	A	C	A	G	T	A	G
A	4	4	4	4	7	7	7	8	A	1	3	3	6	6	6	8	8
C	1	6	6	6	6	6	8	8	C	2	2	8	8	8	8	8	8
T	2	2	3	8	8	8	8	8	T	5	5	5	5	5	8	8	8
G	5	5	5	5	5	8	8	8	G	4	4	4	4	7	7	7	8

Figure 4. The successor matrix T for two sequences a = "CTTAGCA" and b = "ACAGTAG"

Table 2. The Average Running Time (in Milliseconds) of Quick-MLCS and Quick-DP Algorithms for Random Two-Sequence MLCS Problems of lengths ranging between 100 and 1,000

Sequence Length	**|Σ| = 4**		**|Σ| = 20**	
	Quick-DP	**Quick-MLCS**	**Quick-DP**	**Quick-MLCS**
100	14	4	17	5
200	31	17	35	20
300	54	32	81	49
400	93	59	147	94
500	157	99	253	161
600	228	148	400	259
700	317	219	620	427
800	468	327	877	599
900	634	446	1289	914
1000	803	582	1712	1259

4. EXPERIMENTAL EVALUATION

In this paper's experiments, the algorithms were run on a machine with the Linux Fedora 15 Lovelock Kernel 2.6.38 with Intel Core2 Duo P8400 2.26 GHz and 3 GB memory. The programming environment was GNU C++. The algorithms were tested on a set of strings of lengths ranging between 100 and 4,000, on alphabets of size 4 (e.g. DNA sequences) and 20 (e.g. protein sequences).

The new algorithm Quick-MLCS is compared with the Wang et al. [22] algorithm, the Quick-DP. The Quick-DP algorithm is designed for any number of strings and is among the fastest algorithms for the general MLCS problem. This paper's algorithm can work with any number of sequences and the Quick-DP algorithm was implemented according to Wang et al. [22]. All the source codes used for our experiments are available by request.

Since the MLCS method can be applied to many areas of bioinformatics and computational genomics, and well beyond the biological domain, the sequence representations of the objects from each application domain, as well as the distribution of the letters in the sequences, can be drastically different. Therefore, for an unbiased assessment of this paper's algorithms, a set of strings randomly and independently generated from the alphabet was used as a test set.

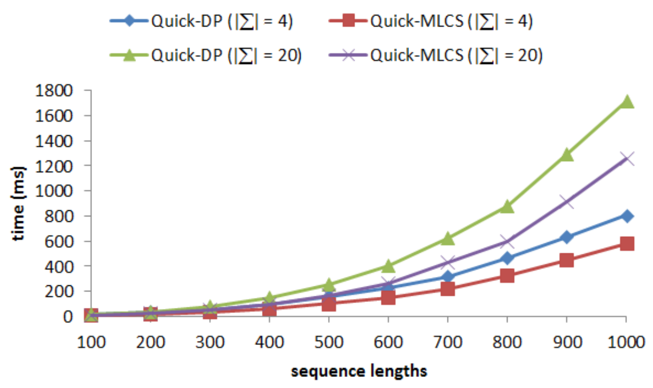


Figure 5. The Average Running Time (in Milliseconds) of Quick-MLCS and Quick-DP Algorithms for Random Two-Sequence MLCS Problems of lengths ranging between 100 and 1,000

In the experiments on two-sequence MLCS problems, 10 sets of two random strings for each string length were generated. Quick-MLCS and Quick-DP algorithms were tested on the same data sets and their average running times for strings of lengths ranging between 100 and 1,000 are shown in Table 2 and Figure 5.

The experimental results for strings of lengths ranging between 1,500 and 4,000 are presented in Table 3 and Figure 6.

Figure 5 and Figure 6 show that Quick-MLCS is slightly faster than the Quick-DP algorithm on two strings. The results in Table 2 and Table 3 indicate that the performance of the Quick-MLCS algorithm of this paper exceeds that of the Quick-DP by about 20%.

Table 3. The Average Running Time (in Seconds) of Quick-MLCS and Quick-DP Algorithms for Random Two-Sequence MLCS Problems of lengths ranging between 1,500 and 4,000

Sequence Length	|Σ| = 4		|Σ| = 20	
	Quick-DP	Quick-MLCS	Quick-DP	Quick-MLCS
1500	2.66	1.89	5.54	3.94
2000	5.75	4.35	12.07	9.06
2500	11.37	8.38	23.49	17.47
3000	18.28	13.94	39.15	29.78
3500	29.04	22.09	63.04	48.47
4000	42.48	33.57	94.95	75.2

5. CONCLUSIONS

The MLCS problem is a common task in the sequence comparison of DNA sequences and amino acid sequences of proteins [3][5][18][14][7][8]. Therefore, the direction of further improvement and development of this paper's algorithms for this field will be guided by the needs of the bioinformatics and computational genomics community.

The main contribution of this paper is the design of a new efficient algorithm, the Quick-MLCS, for solving a general case of the MLCS problem. The comparison with the currently best method, the Quick-DP, through various experiments, suggests that the Quick-MLCS is presently the fastest sequential general MLCS algorithm, with a speed significantly higher than that of the existing methods. The author's next steps will be towards presenting a parallel version of the Quick-MLCS which can handle larger sequences.

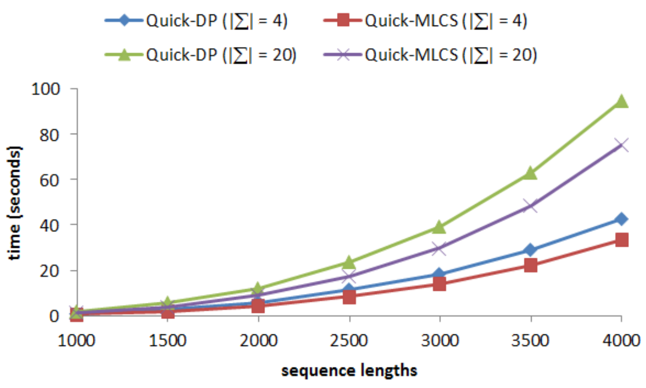


Figure 6. The Average Running Time (in Seconds) of Quick-MLCS and Quick-DP Algorithms for Random Two-Sequence MLCS Problems of lengths ranging between 1,500 and 4,000

6. ACKNOWLEDGMENTS

The authors would like to thank the four anonymous reviewers for their excellent and supportive comments.

7. REFERENCES

[1] Aho, A., Hopcroft, J., Ullman, J. 1983. Data structures and algorithms. Addison-Wesley.

[2] Apostolico, A., Browne, S. and Guerra, C. 1992. Fast Linear-Space Computations of Longest Common Subsequences. Theoretical Computer Science 92, 1, 3-17. DOI=10.1016/0304-3975(92)90132-Y http://dx.doi.org/10.1016/0304-3975(92)90132-Y

[3] Attwood, T.K. and Findlay, J.B.C. 1994. Fingerprinting G Protein-Coupled Receptors. Protein Eng. 7, 2, 195-203. DOI=10.1093/protein/7.2.195.

[4] Bergroth, L., Hakonen, H. and Raita, T. 2000. A Survey of Longest Common Subsequence Algorithms. Proc. Int'l Symp. String Processing Information Retrieval (SPIRE '00), IEEE Computer Society, Washington, DC, USA, 39-48.

[5] Bourque, G. and Pevzner, P.A. 2002. Genome-Scale Evolution: Reconstructing Gene Orders in the Ancestral Species. Genome Research 12, 26-36.

[6] Chen, Y., Wan, A. and Liu, W. 2006. A Fast Parallel Algorithm for Finding the Longest Common Sequence of Multiple Biosequences. BMC Bioinformatics 7, S4.

[7] Chin, F.Y. and Poon, C.K. 1990. A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabet Size. J. Information Processing 13, 4, 463-469.

[8] Dayhoff, M.O. 1969. Computer Analysis of Protein Evolution. Scientific Am. 221, 1, 86-95.

[9] Hakata, K. and Imai, H. 1998. Algorithms for the Longest Common Subsequence Problem for Multiple Strings Based on Geometric Maxima. Optimization Methods and Software 10, 233-260.

[10] Hirschberg, D.S. 1977. Algorithms for the Longest Common Subsequence Problem. J. ACM 24, 664-675. DOI=10.1145/322033.322044 http://doi.acm.org/10.1145/322033.322044

[11] Hsu, W.J. and Du, M.W. 1984. Computing a Longest Common Subsequence for a Set of Strings. BIT Numerical Math. 24, 1, 45-59.

[12] Hunt, J.W. and Szymanski, T.G. 1977. A Fast Algorithm for Computing Longest Common Subsequences. Comm. ACM 20, 5, 350-353. DOI=10.1145/359581.359603 http://doi.acm.org/10.1145/359581.359603

[13] Korkin, D. 2001. A New Dominant Point-Based Parallel Algorithm for Multiple Longest Common Subsequence Problem. Technical Report TR01-148, Univ. of New Brunswick.

[14] Korkin, D., Wang, Q. and Shang, Y. 2008. An Efficient Parallel Algorithm for the Multiple Longest Common Subsequence (MLCS) Problem. Proc. 37th Int'l Conf. Parallel Processing (ICPP '08), 354-363.

[15] Maier, D. 1978. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM* 25, 2 (April 1978), 322-336. DOI=10.1145/322063.322075 http://doi.acm.org/10.1145/322063.322075.

[16] Masek, W.J. and Paterson, M.S. 1980. A Faster Algorithm Computing String Edit Distances. J. Computer and System Sciences 20, 18-31.

[17] Rick, C. 1994. New Algorithms for the Longest Common Subsequence Problem. Technical Report No. 85123-CS, Computer Science Dept., Univ. of Bonn.

[18] Sankoff, D. and Blanchette, M. 1999. Phylogenetic Invariants for Genome Rearrangements. J. Computational Biology 6, 431-445.

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

[20] Sankoff, D. 1972. Matching Sequences Under Deletion/Insertion Constraints. Proc. Nat'l Academy of Sciences USA 69, 4-6.

[21] Smith, T.F. and Waterman, M.S. 1981. Identification of Common Molecular Subsequences. J. Molecular Biology 147, 195-197.

[22] Wang, Q., Korkin, D. and Shang, Y. 2011. A Fast Multiple Longest Common Subsequence (MLCS) Algorithm. IEEE Transactions on Knowledge and Data Engineering 23, 3, 321-334. DOI=10.1109/TKDE.2010.123 http://dx.doi.org/10.1109/TKDE.2010.123