A Branch Elimination-based Efficient Algorithm for Large-scale Multiple Longest Common Subsequence Problem

Shiwei Wei, Yuping Wang*, *Senior Member, IEEE,* and Yiu-ming Cheung, *Fellow, IEEE*

Abstract—It is a key issue to find out all longest common subsequences of multiple sequences over a set of finite alphabets, namely MLCS problem, in computational biology, pattern recognition and information retrieval, to name a few. However, it is very challenging to tackle the large-scale MLCS problem effectively and efficiently due to the high complexity of time and space. To this end, this paper will therefore propose a Branch Elimination-based Space and Time efficient algorithm called *BEST-MLCS*, which includes the following four key strategies: 1) Estimation scheme for the lower bound of the length of MLCS. 2) Estimation scheme for the upper bound of the length of the paths through the current match point. 3) Branch elimination strategy by finding all useless match points and removing the branches not on the longest paths. 4) A new Directed Acyclic Graph (DAG) construction method for constructing the smallest DAG among the existing ones. As a result, the proposed algorithm *BEST-MLCS* can save a lot of space and time and can handle much larger scale MLCS problems than the existing algorithms. Extensive experiments conducted on biological DNA sequences show that the performance of the proposed algorithm *BEST-MLCS* outperforms three state-of-the-art algorithms in terms of run-time and memory consumption.

Index Terms—multiple longest common subsequences(MLCS), dominant point-based approach, useless match point detection, branch elimination, smaller DAG.

1 Introduction

Searching Longest Common Subsequences (LCS) of Multiple (i.e., three or more) sequences (MLCS for short [1]) is a fundamental but challenging problem in many areas such as file comparison [2], pattern recognition [3], [4], distance metric learning [5], [6], computational biology [7], [8] and information retrieval [9], [10]. In the literature, Sankoff [11] presented a dynamic programming (DP) method to find out the LCS of two sequences (also simply called LCS without further distinction). DP can handle the LCS problem in $O(n^2)$ running time and memory space, respectively, where n is the length of the sequences.

Basically, an MLCS problem is more challenging than an LCS problem. Many methods designed for LCS problems are not suitable for MLCS problems. In the past decades, a number of studies have been focused on dealing with MLCS problems within the DP framework (e.g., see [12]–[17]), but these DP-type algorithms are ineffective and inefficient for MLCS problems, especially for large-scale MLCS problems (i.e., a problem with the large number of and long sequences). In fact, as the number and length of sequences increase, the consumption of run-time and memory space will grow exponentially due to their high time and space complexity of $O(n^d)$ [18], where d ($d \geq 2$) is the number of sequences and n is the length of sequences.

Alternatively, the dominant point-based approach, of which the main idea was presented by Hakata and Imai [19], [20], is a kind of more effective and efficient algorithms for the MLCS problem. It is based on the observation that most points in the dynamic table of DP-type algorithms are useless, and only the key points, i.e., the so-called dominant points, need to be computed and stored [18]. Definitely, the resulted search space of the dominant point-based approach is much smaller than that of DP-type methods. It turns out that a lot of run-time and memory space can be saved. In addition, the dominant point-based approach transforms the MLCS problem into the problem of finding the longest path in a Directed Acyclic Graph (DAG). Subsequently, many existing longest path algorithms for the DAG can be applied to solve the MLCS. To further improve the performance of the dominant point-based approach, a number of its variants have been proposed, e.g., see literature [21]–[24]. Especially, Chen et al. [22] proposed a fast dominant point-based algorithm, denoted as *FAST_LCS*, in which a special data structure called successor table is designed to speed up calculating the successors of the match point. Furthermore, to speed up the construction of DAG, Wang et al. [24] presented a new algorithm called *Quick-DP* by a divide-and-conquer technique. Quick-DP has a better performance than *FAST_LCS* in terms of time complexity. Nevertheless, as the number and the length of sequences increase, the DAG constructed by *FAST_LCS* and *Quick-DP* will become larger and larger. It turns out that *FAST_LCS* and *Quick-DP* often get stuck before they finish the construction of DAG. This is because the time complexity of the non-dominated sorting technology employed by both algorithms is $O(N^2)$, where

- *S. Wei is with the School of Computer Science and Technology, Xidian University, Xian, Shaanxi, China, and the School of Computer Science and Engineering, Guilin University of Aerospace Technology, Guilin, Guangxi, China. E-mail: swwei_2001@163.com.*
- *Y. Wang (corresponding author) is with the School of Computer Science and Technology, Xidian University, Xian, Shaanxi, China. E-mail: ywang@xidian.edu.cn.*
- *Y.M. Cheung is with Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR, China. E-mail: ymc@comp.hkbu.edu.hk.*

the number N of match points in the DAG is much larger than n and d.

Recently, Li et al. [25] presented a novel algorithm called *Top_MLCS* which designs a new method to construct DAG and utilizes a forward-and-backward topological sorting technology to find out the longest paths in the DAG. This method shows a better performance on time consumption due to the topological sorting technology. To further improve it, Peng and Wang [26] proposed a *Leveled-DAG* approach to reduce the size of DAG by removing outdated nodes, Wei et al. [27] presented a *path recorder* method to eliminate redundant and repeated nodes of DAG, and Liu et al. [28] designed a *character merging* algorithm to shorten the length of DNA sequences by merging consecutively repeated characters for MLCS problems. All these algorithms utilize topological sorting technology instead of non-dominated sorting technology to construct DAG. The DAGs constructed by them are much smaller than those by dominant point-based approaches including *FAST_LCS* and *Quick-DP*. Hence, they have a better performance than *FAST_LCS* and *Quick-DP* in terms of time and space consumption. Nevertheless, as the size of DAG increases, the topological sorting based algorithms still consume a large amount of memory space because they have to store the whole DAG including all match points and directed edges (i.e., useless match points and non-longest paths cannot be identified and removed in time). Therefore, they cannot effectively solve the large-scale MLCS problem due to the memory overflow.

In order to deal with large-scale MLCS problems, we will propose a Branch Elimination-based Space and Time efficient algorithm called BEST-MLCS, which can eliminate branches that are not on the longest paths. As a result, we can construct much smaller DAG than all existing algorithms, thereby handling much larger scale MLCS problems. The main contributions of this paper are summarized as follows:

1) An estimation scheme for the lower bound L_{mlcs} of the length of MLCS is designed.
2) An estimation scheme for the upper bound $U_{(O,p,\infty)}$ of the length of paths from the starting match point to the ending match point through the current match point p is presented.
3) A branch elimination strategy for finding all useless match points and removing the branches that are not on the longest paths is proposed (Theorem 2). Specifically, first, a useless match point detection scheme (Theorem 2) finds out useless match points as follows. For the current match point p, if $U_{(O,p,\infty)} < L_{mlcs}$, p must not be on any longest path corresponding to MLCS and is a useless match point. Then, a branch elimination strategy is proposed, which removes all paths (i.e., branches) from starting match point O to the useless match point p. Note that with the increase of the number and the length of sequences, the number of useless match points and the branches through these useless match points will become very huge. The useless match point detection scheme can identify these useless match points promptly and the branch elimination strategy can remove all branches from O to them on DAG.
4) A new DAG construction method for constructing the smallest DAG among the existing ones is proposed. Given a starting match point, the new DAG is built level by level without putting the useless match points on DAG, meanwhile removing all existing branches through useless match points. As a result, the resulted DAG will be much smaller than the existing ones (including the DAGs constructed by the state-of-the-art MLCS algorithms), and its time and memory space will be greatly saved.

The main difference between the proposed algorithm and the state-of-the-art algorithms is that, *BEST-MLCS* designs a novel branch elimination strategy based on the lower bound and upper bound estimation to remove a lot of useless match points and non-longest paths from DAG, avoiding using non-dominates sorting method and topological sorting approach which are very time-consuming and need a huge amount of memory space. Thus, the size of the constructed DAG is very small, and *BEST-MLCS* can efficiently find the longest paths corresponding to MLCSs from the DAG with less consumption of run-time and memory space.

The rest of this paper is organized as follows: Section 2 introduces the preliminary of the LCS and MLCS problems. Section 3 reviews related works for the LCS and MLCS problems. Section 4 describes the proposed algorithm *BEST-MLCS* in detail. Section 5 presents the experimental results to compare the proposed algorithm with three state-of-the-art algorithms. Finally, we draw a conclusion in Section 6.

2 PRELIMINARY

2.1 Basic Definitions

Definition 1. *Let $s = c_1c_2\cdots c_n$ be a sequence on a character set Σ, where $c_i \in \Sigma, 1 \le i \le n$, $|\Sigma|$ denotes the cardinality of Σ, and $|s|$ denotes the length of s, i.e., $|s| = n$. If a sequence $s' = c_{i_1}c_{i_2}\cdots c_{i_m}$ satisfies $1 \le i_1 < i_2 < \cdots < i_m \le n$, s' is called a subsequence of s, denoted by $s' \in subseq(s)$, where $subseq(s)$ is the set of all subsequences of s [22].*

Actually, a subsequence can be obtained by removing zero or more characters from the given sequence. For example, the sequences $GACT$, GAA and AT are all subsequences of the sequence $GAACT$, i,e., $GACT, GAA,$ and $AT \in subseq(GAACT)$.

Definition 2. *Given d $(d \ge 2)$ sequences $s_1, s_2, \cdots, s_d$ on character set Σ, a sequence s' is called an LCS if it satisfies the following conditions [22]:*

1) *For $\forall i, 1 \le i \le d$, $s' \in subseq(s_i)$, i.e., s' is a common subsequence of all d sequences.*
2) *There is no other common subsequence $s'' \in subseq(s_i)(i = 1, 2, \cdots, d)$ longer than s' (i.e., $|s''| > |s'|$).*

If $d = 2$, the problem of finding LCS is usually called LCS problem; otherwise, if $d >= 3$, the problem is called MLCS problem.

Obviously, an *LCS* problem is the simplest case of an *MLCS* problem. Given d sequences, there may be more than one *LCS*s or *MLCS*s. For example, for two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$, there are two *LCS*s: $AGCGA$ and $AGCTA$.

2.2 The DP Approach

Definition 3. *For a sequence $s = c_1c_2\cdots c_n$, its subsequence $c_1c_2\cdots c_j$ is called a prefix sequence of s and denoted by $s[1...j]$ [22].*

The DP approach is a traditional method for tackling LCS and MLCS problems [29]. Given d sequences s_1, s_2, ..., s_d with the length of n, it will recursively build a d dimension score table L with $n \times n \times ... \times n = n^d$ elements, where the element $L[i_1, i_2, ..., i_d]$ denotes the length of MLCS of the prefix sequences $s_1[1...i_1]$, $s_2[1...i_2]$, ..., $s_d[1...i_d]$, and can be calculated by the following formula [26]:

$$L[i_1, \cdots, i_d] = \begin{cases} 0 & \text{if } \exists i_j = 0, (1 \le j \le d) \\ L[i_1-1, \cdots, i_d-1]+1 & \text{if } s_1[i_1] = \cdots = s_d[i_d] \\ max(\bar{L}) & \text{otherwise} \end{cases} \quad (1)$$

where $\bar{L}$=$\{L[i_1, i_2, \cdots, (i_k - 1), \cdots, i_d] | k = 1, 2, \cdots, d\}$.

Once the score table L is built, the MLCS can be obtained by traversing from the bottom right element $L[n, n, ..., n]$ to the top left element $L[0, 0, ..., 0]$. For instance, the score table L built for the sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$ is shown in Figure 1, and the LCS of these two sequences are obtained by traversing from $L[8, 8]$ to $L[0, 0]$.

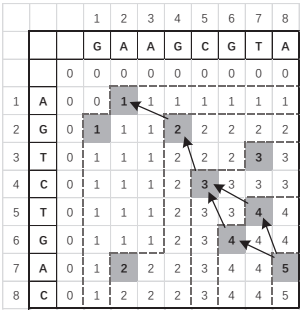


Fig. 1. The score table L for two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$. The LCS can be found out from the score table L by traversing from shadow number 5 to 1.

Given d sequences with the length of n, both time complexity and space complexity of the DP approach are up to $O(n^d)$ [14]. As d and n increase, the space and time consumption of these approaches will increase exponentially. That is, the scalability of a DP approach is limited, which is therefore essentially unsuitable for dealing with large-scale MLCS problems.

3 Related Works

3.1 Dominant Point-based Approach

Before discussing dominant point-based approach in detail, we first introduce some terminologies [25].

Definition 4. *Given d sequences $s_1, s_2, \cdots, s_d$ on a character set Σ, let $s_i[p_i]$ denote the p_i-th character of sequence s_i from left. If $s_1[p_1] = s_2[p_2] = ... = s_d[p_d] = \sigma \in \Sigma$, the vector $p = (p_1, p_2, ..., p_d)$ is called a **match point** of these d sequences, and σ is called a common character of p, denoted by $\sigma = cch(p)$ [22]. The match point $(p_1, p_2, \cdots, p_d)$ with its common character σ is represented as $\sigma(p_1, p_2, \cdots, p_d)$.*

For example, given two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$ in Figure 2, there are many match points with the form $\sigma(i, j)$. The common character $\sigma \in \Sigma$ linked with dotted line corresponds to its position indexes i and j in two sequences, i.e., $s_1[i] = s_2[j] = \sigma$, like $A(2, 1)$, $A(3, 1)$ and $G(1, 2)$. Since $cch((2, 1)) = A$, match point $A(2, 1)$ is also denoted by $(2, 1)$ for short. Similarly, $A(3, 1)$ and $G(1, 2)$ can be represented as $(3, 1)$ and $(1, 2)$, respectively.

Definition 5. *Given two match points $p = (p_1, p_2, \cdots, p_d)$ and $q = (q_1, q_2, \cdots, q_d)$ of d sequences, we call [22]*

1) *$p = q$, if $\forall i, 1 \le i \le d, p_i = q_i$.*
2) *p **dominates** q (denoted by $p \preceq q$), if $\forall i, 1 \le i \le d, p_i \le q_i$ and $\exists j, 1 \le j \le d, p_j < q_j$.*
3) *p **strongly dominates** q (denoted by $p \prec q$), if $\forall i, 1 \le i \le d, p_i < q_i$.*
4) *q is called a **successor** of p with respect to σ, if there is no other match point r satisfying that $p \prec r \preceq q$ and $cch(r) = cch(q) = \sigma$, denoted by $q = succ_\sigma(p)$.*

In general, a match point p has at most $|\Sigma|$ successors. That is, the set of all successors of p, denoted by $succ(p) = \{succ_\sigma(p) | \sigma \in \Sigma\}$, contains at most $|\Sigma|$ elements, e.g., $succ(A(2, 1)) = \{A(3, 7), C(5, 4), G(4, 2), T(7, 3)\}$. If q is a successor of p, i.e., $q \in succ(p)$, p is called a precursor of q. For example, $A(2, 1)$ is a precursor of $A(3, 7)$.

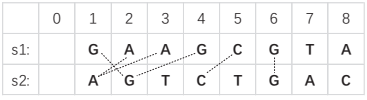


Fig. 2. An example about the match points of two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$, where each two dimensional point $\sigma(i, j)$ consisting of two numbers (columns) i and j linked by dotted lines forms a match point, where i and j are the indexes of common character σ in s_1 and s_2, i.e., $s_1[i] = s_2[j] = \sigma$.

Definition 6. *Given a set of match points $P = \{P_1, P_2, ..., P_m\}$, for a match point $P_j \in P$, If $\neg\exists P_i \preceq P_j$, $1 \le i, j \le m, i \ne j$, P_j is called a non-dominated point (dominant point for short) on P [22]. All of dominant points on P form the dominant set of P.*

For the convenience of description, a special match point $(0, 0, \cdots, 0)$ is defined as the starting match point and denoted as O, and $(\infty, \infty, \cdots, \infty)$ denoted as ∞ is defined as the ending match point.

As the most effective and efficient method, the dominant point-based approach mainly consists of the following steps to find out all MLCSs of d sequences:

Step 1) Initialization: Set the initial level number $k = 0$ and $D^k = \{O\}$.

Step 2) Compute the set of all successors of each match point in D^k, i.e., $succ(D^k)$. All of these successors form

Level-(k+1) of DAG, denoted by $L^{(k+1)}$. Connect each match point in D^k to its each successor by a directed edge.

Step 3) Find out the dominant set $D^{(k+1)}$ of $L^{(k+1)}$ by non-dominated sorting method [30]. The $(k+1)$-th character of any MLCS will come from these common characters of dominant points in $D^{(k+1)}$.

Here, we use the aforementioned example to describe the process of dominant point-based approach as shown in Figure 3.

$O(0,0)$ is the starting match point and its all successors form the match points in *Level-1* of DAG, i.e., match points in *Level-1* are $L^1 = \{A(2,1), C(5,4), G(1,2), T(7,3)\}$. The dominant points in L^1 (i.e., A(2,1) and G(1,2)) depicted with white background are then selected by non-dominated sorting to form the dominant set D^1, i.e., $D^1 = \{A(2,1), G(1,2)\}$. Note that the first character of MLCSs will come from the common characters of dominant points in D^1 (i.e., A or G), but not from the common characters of dominated match points in L^1 (i.e., $C(5,4)$ and$(T(7,3))$) which are marked by grey background in Figure 3.

Step 4) Suppose we have gotten the dominant set $D^{(k+1)}$. Let $k = k+1$. If some match point in D^k has successor, go to Step 2), otherwise, each match point in D^k has no successor, connect each match point in D^k to ∞. The construction of DAG is completed.

Now we have got D^1 in the aforementioned example. Compute all successors of each match point in D^1, and all of these successors consist of $L^2 = \{A(3,7), C(5,4), G(4,2), T(7,3); A(2,7), C(5,4), G(4,6), T(7,3)\}$. Find out the dominant set of L^2, i.e., $D^2 = \{G(4,2), A(2,7)\}$. Note that the second character of the MLCSs must come from the match points in D^2 instead of L^2. Repeat Steps 2) to 4) until the complete DAG is created by the dominant point-based approach as shown in Figure 3. The MLCSs correspond to all longest paths from the starting point O to the ending point ∞. One can find all longest path/MLCS (i.e., $AGCGA$ and $AGCTA$) by traversing back from ∞ to O of DAG.

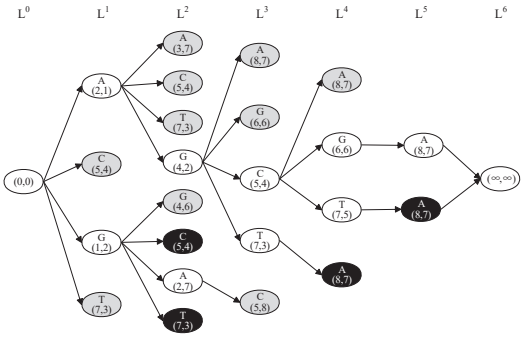


Fig. 3. The DAG constructed by the dominant point-based approach on two sequences $GAAGCGTA$ and $AGTCTGAC$, where the black and gray nodes are duplicated and dominated match points, respectively.

However, the bottleneck of the dominant point-based approach is that it has to construct a huge DAG first, while computer will run out of memory before the construction of DAG is completed. Specifically, the dominant point-based approach has two major drawbacks:

1) There may be many repeated match points and dominated points in each level (e.g., C(5,4) repeatedly appears twice and six points {A(3,7), C(5,4), T(7,3), C(5,4), G(4,6), T(7,3)} are dominated points in L^2), and a match point appearing in one level may appear many times in the later levels (e.g., C(5,4) appears in L^1-L^3) and only it in the latest level is useful. Thus, the constructed DAG will be very huge so that the computer has not enough memory to store DAG.

2) To find D^k, the non-dominated sorting method will take a lot of computation. Its time complexity is $O(dN^2)$ in level k, where N is the number of match points in L^k and d is the number of sequences. Note that N will be very large (at the worst case, $N = |\Sigma|^k$ increases exponentially with the increase of level k). Hence, when the length n, the number d of sequences and $|\Sigma|$ are large, i.e., when the MLCS problem becomes a large-scale problem, the non-dominated sorting method will be very time consuming.

FAST_LCS [22] and *Quick-DP* [24] are two typical examples of this kind of algorithms.

3.2 Top_MLCS

In order to reduce the time and space consumption, *Top_MLCS*, as one of the best existing state-of-the-art algorithms, is designed [25], which consists of three main steps.

Step 1) In order to save the storage, *Top_MLCS* avoids the repeated match points appearing in DAG. Before a match point p is put on DAG, it is checked whether p has been on DAG. If not, p will be put on DAG and draw an edge to p from its precursor. Otherwise, additional p will not be put on DAG. It only needs to draw an edge to original p from its precursor. As a result, the constructed DAG will not contain the repeated match points. That is, it is a non-redundant DAG.

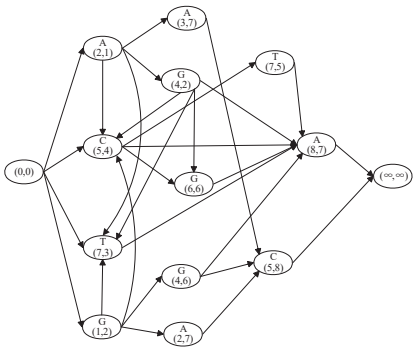


Fig. 4. The construction of the non-redundant DAG for $GAAGCGTA$ and $AGTCTGAC$ by using the topological sorting approach.

Here, we use the aforementioned example to describe this procedure as shown in Figure 4. Initially, DAG contains only the starting match point (0,0), and then put its successors (2,1), (5,4), (7,3) and (1,2) on DAG directly because these successors have not been on DAG. Draw a directed edge from (0,0) to each of its successors. Search successors of each of these four match points. For example, match point (2,1) has four successors (5,4), (7,3), (3,7) and (4,2), but (5,4) and (7,3) have existed on the DAG. Thus, there is no need to put them again, and we only need to draw directed edges from (2,1) to them and create new successors (3,7) and (4,2). For other three match points, we can continue to construct DAG in the similar way. Repeat the procedure until DAG

is constructed completely. Obviously, DAG in Figure 4 has much fewer match points than the one in Figure 3, but DAG in Figure 4 destroys the levels of match points. It is not easy to find MLCS. To redefine the levels of points, *Top_MLCS* uses a forward topological sorting scheme.

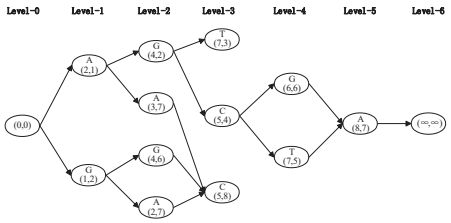


Fig. 5. The obtained DAG after the forward topological sorting operation is executed.

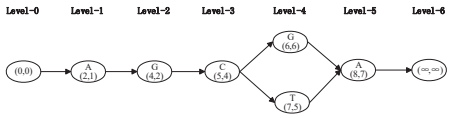


Fig. 6. The final longest paths corresponding to MLCSs are obtained after the backward parallel topological sorting operation.

Step 2) Define the level of each match point of DAG (see Figure 4) by a forward topological sorting scheme as follows. Set the level of the starting match point (0,0) as 0 in Figure 4. Suppose we have defined level k (currently level-0). Remove all directed edges from match points in level-k. Identify the math points with indegree 0 and define their level as $(k+1)$ (currently the level of (1,2) and (2,1) is defined as $k+1=1$) and add a directed edge to each match point in level $(k+1)$ from its precursor in level k (i.e., from (0,0) to (1,2) and (2,1), respectively). Repeat this process until the levels of all match points are defined as shown in Figure 5.

Step 3) Get the final DAG by a backward topological sorting scheme as follows. From the ending point ∞, find its precursor(s) (i.e., (8,7) in Figure 5). For each precursor, successively find its precursor(s) backward until the starting match point O is reached. If a match point ((7,3) or (5,8) in Figure 6) cannot be reached by the backward topological sorting from (∞,∞), it and all paths through it will be removed. The final DAG is shown in Figure 6. All MLCSs will be contained in this DAG and can be obtained from paths from O to ∞.

Although *Top_MLCS* greatly improves the performance of the existing dominant point-based approaches, it still has the following two main problems:

1) It first needs to construct and stores a large DAG (see Figure 4), which will consume a huge computer memory for the large-scale MLCS problem. Usually, the computer has not enough memory to store it. Thus its scalability is limited.

2) The forward/backward topological sorting operation will consume a lot of computation for the large-scale problem.

4 The Proposed BEST-MLCS

4.1 The Main Framework of BEST-MLCS

As mentioned before, the existing approaches fail to deal with large-scale MLCS problems due to huge time and space consumption [27]. The fundamental reason is that, as the number and length of sequences increase, the size of DAG to be built becomes larger and larger. It turns out that the computational time and storage space exceed the maximum limits. To overcome these shortcomings, during the construction of DAG, the proposed *BEST-MLCS* promptly identifies the useless match points and non-longest paths, then removes them in time to reduce the size of the DAG.

To be specific, before obtaining the final MLCS of sequences, *BEST-MLCS* first quickly estimates a lower bound L_{mlcs} of the length of the true MLCS. Then, *BEST-MLCS* estimates an upper bound $U_{(O,p,\infty)}$ of the length of any path from the starting match point O to the ending match point ∞ through a match point p before match point p is added to the DAG. If $U_{(O,p,\infty)} < L_{mlcs}$, we can judge that any path through p is not the longest path. Thus, p is a useless match point, and all paths passing through it are non-longest paths in the DAG. Based on this observation, all useless match points and non-longest paths can be removed promptly, a much smaller DAG than the existing ones will be constructed.

In summary, the proposed algorithm *BEST-MLCS* contains four key strategies: 1) *Lower bound estimation.* Estimate a lower bound L_{mlcs} of the length of MLCS; 2) *Upper bound estimation.* Estimate an upper bound $U_{(O,p,\infty)}$ of the length of path from O to ∞ through a current match point p; 3) *Branch elimination.* Determine whether the current match point p is a useless match point before it is put on DAG. Any useless match point p is not put on DAG and all branches through p are removed from DAG; 4)*Smaller DAG construction.* Construct a much smaller DAG than the existing ones. The details of these key strategies are introduced as follows.

4.1.1 Quick Estimation of Lower Bound L_{mlcs}

Before the MLCS (or the longest path in DAG) is obtained, we do not know the true length of MLCS, but we can get a lower bound L_{mlcs} of it by finding an approximate MLCS. Then the length of this approximate MLCS is a lower bound of the length of the true MLCS. The longer the approximate MLCS, the better it approaches to the true MLCS. Our purpose is to find an approximate MLCS as long as possible, and the approximate MLCS should be searched quickly. Based on these considerations and inspired by [31] and [32], a fast heuristic method for estimating the lower bound L_{mlcs} is designed. The main steps are given below.

For a d-dimensional match point $p = (p_1, p_2, \cdots, p_d)$, denote $sum(p) = \sum_{i=1}^{d} p_i$. Please note that, if a successor q of p has the smallest value of $sum()$, q must be a non-dominate point among all successors of p. A path from O through p to q can be longer than a path from O through p to any other successor. Thus a path through p and q is more possible to be better than a path through p and other successor. Based on this idea, when finding an approximate MLCS, we can only select a few successors (e.g., θ successors) with the first θ smallest values of $sum()$ as the candidates of next point after point p. In this way, we only consider θ

successors as the candidates of the next point on each level and can quickly find an approximate MLCS and its length L_{mlcs}. Details are as follows.

Let θ be a small positive integer (for example, set $\theta = 2$) and E be a set of θ selected match points with the first θ smallest values of $sum()$.

1) Initialization: Set $O = (0, 0, \cdots, 0)$ as a selected match point, i.e.,$E = \{O\}$, and set $L_{mlcs} = 0$.
2) Update L_{mlcs}: Search all successors of each match point in E, and select θ successors with the first θ smallest values of $sum()$ among all successors (if there are only β successors with $\beta < \theta$, then let $\theta = \beta$). Update E by removing its all current elements and putting the θ selected successors into it, and let $L_{mlcs} \leftarrow L_{mlcs} + 1$.
3) If E is empty, return L_{mlcs}; otherwise, goto step 2).

Here, we use the aforementioned example to illustrate the process in Figure 7. Initially, $L_{mlcs} = 0$, and $E = \{O\}$. O has successors $(2, 1)$, $(1, 2)$, $(5, 4)$ and $(7, 3)$. Among these successors, match points $(2, 1)$ and $(1, 2)$ have the smallest $sum()$ value 3. Thus, we update E by $E = \{(2, 1), (1, 2)\}$ and set $L_{mlcs} = L_{mlcs} + 1 = 1$. The successors $(5, 4)$ and $(7, 3)$ are not selected into E as shown in Figure 7. For each match point in E, calculate their successors, and there are 6 successors in total, where two successors $(4, 2)$ and $(2, 7)$ with the first two smallest $sum()$ values are selected (note that the smallest $sum()$ value match point is $(4, 2)$, and the second smallest $sum()$ value match point has two: $(2, 7)$ and $(5, 4)$ with the same $sum()$ value. In this case, we only need to randomly select one from $(2, 7)$ and $(5, 4)$, supposing $(2, 7)$ is selected) and update E by $E = \{(4, 2), (2, 7)\}$ and update L_{mlcs} by $L_{mlcs} = 2$. Similarly, when $L_{mlcs} = 3$, the corresponding $E = \{(5, 4), (7, 3)\}$. When $L_{mlcs} = 4$, the corresponding $E = \{(6, 6), (7, 5)\}$. Finally, we can get $L_{mlcs} = 5$ and the corresponding $E = \{(8, 7)\}$.

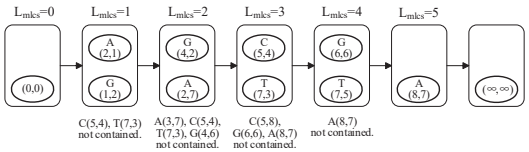


Fig. 7. The process of estimating a lower bound L_{mlcs} of the length of the longest paths in DAG.

During the process of estimating L_{mlcs}, only two small sets (E and its updated set) need to be maintained each time. Hence, the lower bound L_{mlcs} can be estimated very fast with little memory consumption. Also, it is worth noting that L_{mlcs} is often close to the length of the longest path. Thus, L_{mlcs} is usually a good lower bound of the length of the true MLCS.

4.1.2 Efficient Estimation of Upper Bound $U_{(O,p,\infty)}$

Suppose that p is a current point on DAG and we want to know the lengths of all paths through p from O to the ending match point. But it is impossible to know the lengths of these paths before we complete the construction of these paths. However, if we can estimate an upper bound $U_{(O,p,\infty)}$ of the lengths of these paths, and know this upper bound is smaller than the lower bound L_{mlcs} of the length of true MLCS (i.e., $U_{(O,p,\infty)} < L_{mlcs}$), we can judge that these paths through p are not the longest paths and can be excluded from DAG. In this way, the DAG constructed will be much smaller than the existing ones.

Note that the current match point p has been constructed on DAG, the length of the longest path from O to p can be computed. In fact, it is the level of p (denoted by $lev(p)$) on DAG and an efficient method to compute $lev(p)$ will be given in Subsection 4.1.4.

Also note that the true length $dist(p)$ of the longest path from the current match point p to the ending match point ∞ is not known. A feasible way is to estimate an upper bound $U_{(p,\infty)}$ of $dist(p)$. Then $U_{(O,p,\infty)} = lev(p) + U_{(p,\infty)}$ is an upper bound of the length of any path through p. In the following, we will design a specific method to quickly estimate $U_{(p,\infty)}$ and make it as close as possible to the true value $dist(p)$ (i.e., make it as small as possible).

For a sequence $s = c_1c_2\cdots c_n$ on a character set Σ, the times of appearance of the character $\sigma \in \Sigma$ after the position i in s can be easily computed and is denoted by $num_s(\sigma, i)$. For instance, given a sequence $s = GAAGCGTA$, the character A appears three times after the position 1 in s, thus $num_s(A, 1) = 3$, while the character G appears two times after the position 3 in s, so $num_s(G, 3) = 2$.

Given d sequences $s_1, s_2, \cdots, s_d$ on a character set Σ and a match point $p = (p_1, p_2, \cdots, p_d)$ of them, we have the following result:

Theorem 1. *For any longest path from match point $p = (p_1, p_2, \cdots, p_d)$ to the ending match point ∞, and for any $\sigma \in \Sigma$, the times of appearance of σ in this longest path is not greater than $\min\{num_{s_1}(\sigma, p_1), num_{s_2}(\sigma, p_2), \cdots, num_{s_d}(\sigma, p_d)\}$. Hence,*

$$U_{(p,\infty)} = \sum_{\sigma\in\Sigma} \min\{num_{s_1}(\sigma, p_1), num_{s_2}(\sigma, p_2), \cdots, num_{s_d}(\sigma, p_d)\} \quad (2)$$

is an upper bound of the length of any longest path from match point $p = (p_1, p_2, \cdots, p_d)$ to the ending match point ∞.

$$U_{(O,p,\infty)} = lev(p) + U_{(p,\infty)} \quad (3)$$

is an upper bound of the length of the longest path from O to ∞ through p.

Proof. Denote

$$v(\sigma) = \min\{num_{s_1}(\sigma, p_1), num_{s_2}(\sigma, p_2), \cdots, num_{s_d}(\sigma, p_d)\}.$$

Obviously $num_{s_i}(\sigma, p_i) \geq v(\sigma)$ for any p_i. This means that σ appears at least $v(\sigma)$ times in each sequence s_i after position p_i and only appears $v(\sigma)$ times in some sequence(s). Thus σ can appear at most $v(\sigma)$ times on any longest path from match point $p = (p_1, p_2, \cdots, p_d)$ to the ending match point ∞. Therefore, $v(\sigma)$ is an upper bound of the times of appearance of σ in the longest paths, and the sum of these $v(\sigma)$,

$$U_{(p,\infty)} = \sum_{\sigma\in\Sigma} v(\sigma), \quad (4)$$

is an upper bound of the length of the longest path from p to ∞. Note that $U_{(p,\infty)}$ is also an upper bound of the true

number of match points on the longest paths from p to the ending match point ∞.

For the match point p, the length of the path from the starting match point O to p is known, i.e., $lev(p)$. Thus, an upper bound of the length of the path from O to ∞ through p is estimated by

$$U_{(O,p,\infty)} = lev(p) + U_{(p,\infty)}. \quad (5)$$

□

For example, for two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$, and for a match point $p = (1, 2)$ (see Figure 2), an upper bound $U_{(p,\infty)}$ of the length of the longest path from p to ∞ can be calculated according to Formula 2:

$$\begin{aligned} U_{(p,\infty)} = \; & \min\{num_{s_1}(A,1), num_{s_2}(A,2)\} \\ & + \min\{num_{s_1}(C,1), num_{s_2}(C,2)\} \\ & + \min\{num_{s_1}(G,1), num_{s_2}(G,2)\} \\ & + \min\{num_{s_1}(T,1), num_{s_2}(T,2)\} \\ = \; & \min\{3,1\} + \min\{1,2\} + \min\{2,1\} + \min\{1,2\} \\ = \; & 4. \end{aligned}$$

In fact, the length of the longest paths from $p = (1, 2)$ to the ending match point is actually 4. Thus, $U_{(p,\infty)}$ is an appropriate upper bound of the length of the longest path from p to ∞.

4.1.3 Branch Elimination by Finding Useless Match Points

Theorem 2. *For any match point $p = (p_1, p_2, \cdots, p_d)$, If*

$$U_{(O,p,\infty)} < L_{mlcs}, \quad (6)$$

p is a useless match point and should not be included in DAG. Furthermore, all paths (branches) constructed from O to p should be eliminated.

Proof. If $U_{(O,p,\infty)} < L_{mlcs}$, we can ensure that all paths though p are not the longest path. Thus, p must be a useless match point and should not be included in DAG. Therefore, all paths constructed from O to p should be eliminated. The proof is completed. □

Let us use the aforementioned example in Figure 7 to illustrate the scheme of determining useless match points in details. The estimated lower bound of the length of the MLCS is known, i.e., $L_{mlcs} = 5$, as shown in Figure 7. $U_{(p,\infty)}$ can be computed according to Formula 2, and $lev(p)$ can be obtained during the construction of DAG as shown in Figure 8.

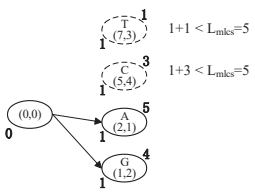


Fig. 8. An example to identify useless match points. The value $lev()$ is marked at the bottom left of the match point, and the value $U_{(p,\infty)}$ is marked at the top right of the match point. Useless match points T(7,3) and C(5,4) are marked with dotted lines and not contained on DAG.

Initially, $lev(O)$ of the starting match point O is defined as 0. O has four successors: $(1, 2)$, $(2, 1)$, $(5, 4)$ and $(7, 3)$. For its each successor p, set $lev(p) = 1$, i.e., set $lev((2, 1)) = lev((5, 4)) = lev((1, 2)) = lev((7, 3)) = 1$, because the length of the longest paths from the starting match point to each of the successors is 1. From Figure 8, it can be seen that match points $(5, 4)$ and $(7, 3)$ can be easily determined to be useless match points because they satisfy

$$U_{(O,p,\infty)} = lev(p) + U_{(p,\infty)} < L_{mlcs} = 5,$$

and all paths (branches) from O to $(5, 4)$ and $(7, 3)$ will not be contained on DAG.

4.1.4 Construct Smaller DAG

Based on the above branch elimination scheme, we construct the smaller DAG level by level. First, level zero L^0 consists of only the starting match point O, and then level 1 to level $|MLCS|$, denoted by L^1, L^2, $\cdots$, $L^{|MLCS|}$, respectively, are sequentially constructed, where $|MLCS|$ represents the length of the final MLCS. To save the time and space, we only construct and store one level each time.

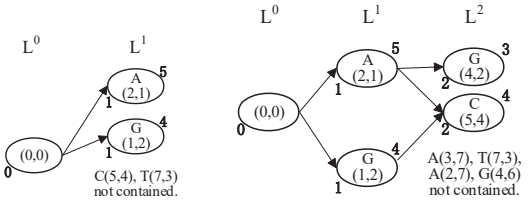
After L^k is constructed (currently, L^0 is constructed), L^{k+1} can be constructed by the following steps:

1) Select any match point $p \in L^k$, search its successor set $succ(p)$.
2) For each successor $q \in succ(p)$, check whether q has already existed in DAG. If not, set the level of q (i.e., the length of the current longest path from O to q) as $lev(q) = k + 1$, go to step 3). Otherwise, compute the level $lev(q)$ of q in two cases:
   - If $lev(q) < k + 1$, it indicates that the existing longest path(s) from O to q is (are) shorter than the new longest path from O to q through p. Update $lev(q) = k + 1$ and shift q from $L^{lev(q)}$ to L^{k+1}. Remove all existing paths through q. Go to step 4).
   - If $lev(q) = k + 1$, it indicates that the path from O to q through p is also a new longest path from O to q, keep $lev(q)$ unchanged. Go to step 4).
3) Identify whether q is a useless match point according to Theorem 2. If yes, do not put q in DAG, go to setp 5). Otherwise, put q into L^{k+1}.
4) Add a directed edge from p to q in DAG.
5) If successors of all match points in L^k have been checked, the construction of L^{k+1} is finished. Otherwise, go to step 1).

By using the above procedure, we can construct a smaller DAG than the existing ones. For easily understanding the process, we use the aforementioned example to construct DAG in detail.

Initially, level zero $L^0 = \{(0, 0)\}$ of DAG is constructed with $lev(O) = 0$. The estimated lower bounder L_{mlcs} is obtained by $L_{mlcs} = 5$ as shown in Figure (7). Successors of $(0, 0)$ are $(2, 1)$, $(5, 4)$, $(1, 2)$ and $(7, 3)$. As $(5, 4)$ and $(7, 3)$ are useless match points, they are not put on DAG. Instead, put $(2, 1)$ and $(1, 2)$ into L^1 directly, set their $lev() = 1$, as shown in Figure 9 (a).

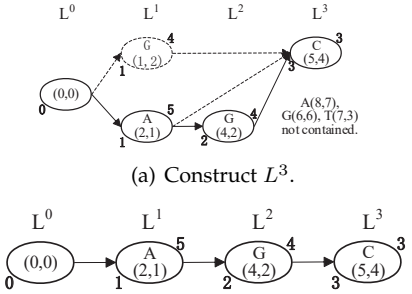
Construct L^2: Calculate successors of each match point in L^1. $(2, 1)$ has successors $(3, 7)$, $(5, 4)$, $(4, 2)$ and $(7, 3)$, and $(1, 2)$ has successors $(5, 4)$, $(7, 3)$, $(2, 7)$ and $(4, 6)$. It can be



(a) Construct L^1. (b) Construct L^2.

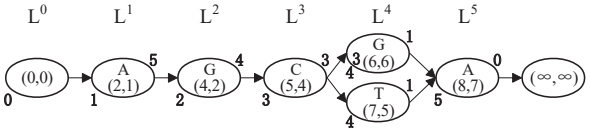
Fig. 9. The process of constructing the smaller DAG, where useless match points are not put on DAG.

seen that only $(4, 2)$ and $(5, 4)$ are not useless match points. Thus, only these two match points are put in L^2 of DAG and set their level as $lev() = 2$, as shown in Figure 9 (b).



(a) Construct L^3.

(b) G(1,2) and dotted lines are not on the longest paths and removed.



(c) The final DAG is constructed.

Fig. 10. The process of constructing the smaller DAG, where useless match points are not put on DAG. When the construction of the smaller DAG is finished, all MLCSs are gotten by finding all paths from O to ∞.

Construct L^3: L^2 contains two match points $(4, 2)$ and $(5, 4)$. $(4, 2)$ has successors $(8, 7)$, $(6, 4)$, $(6, 6)$ and $(5, 4)$, as shown in Figure 10 (a). It can be seen that only $(5, 4)$ is not a useless match point. Thus, $(5, 4)$ is put in L^3 on DAG and set $lev((5, 4)) = 3$. As $(5, 4)$ has existed in L^2, and the previous longest paths from O to $(5, 4)$, which are marked with the dotted edges in Figure 10 (a), are shorter than the current one, delete all previous path from O to $(5, 4)$, the pruned DAG is obtained as shown in Figure 10 (b).

Repeat the above steps until no new level set can be created. Then, the final DAG is obtained, as shown in Figure 10 (c). Each path from O to ∞ is corresponding to an MLCS, and vice versa.

Also note that in each time of the construction of DAG, BEST-MLCS only stores the part of DAG in two successive levels, e.g., L^k and L^{k+1} instead of the whole DAG from L^0 to L^{k+1} (only two successive levels in each sub-figure of Figures 9-10). Thus, the space and time consumption by BEST-MLCS is much smaller than that by the existing ones.

4.2 Fast Implementation of the Proposed Algorithm

4.2.1 Key data structures

In order to fast construct DAG by using as small time and space as possible, several data structures are adopted for the proposed algorithm.

Successor Table Tachnique. Successor Table Technique [22] is employed. Chen [22] has proved that the time complexity of calculating all successors of the given match point p is $O(d|\Sigma|)$. The DAG can be constructed fast by using Successor Tables. Note that Successor Tables can be built before the proposed algorithm is performed and thus they have little effect on memory and time consumed during constructing DAG.

Hash Technique. In order to avoid adding duplicated match points into the DAG, it is necessary to quickly check whether a match point has already existed in DAG. In other words, before a match point is going to be added in DAG, the usual way is to compare it with every match point on DAG. If this operation is performed frequently, many comparisons between two d-dimensional vectors will be made and will cost a lot of time. To circumvent this problem, we put all match points that have been on the DAG into a hash table denoted by H, in which different match points usually have different hash values. Before adding a match point into the DAG, we first check whether it has been in H. If yes, it indicates that the match point has been on the DAG. It is well known that the hash table H can always use a constant time to get the result (identify whether a match point exists in it by comparing the hash values) regardless of the number of match points it contains, which can greatly speed up the identification process. The empirical studies also show that the time spent on the identification process is hardly affected by the increasing number of compared match points.

Compute $U_{(p,\infty)}$ quickly by Distance Tables. For any given match point p, to compute an upper bound of the path from p to ∞, i.e., $U_{(p,\infty)}$, as quickly as possible and by using the computation as less as possible, we design a new data structure called Distance Table as follows.

Note that the Distance Tables are designed before the algorithm is performed, so that we can get $U_{(p,\infty)}$ for any p before the algorithm begins.

For d sequences with length of n, let DT_k denote the Distance Table for the k-th $(1 \le k \le d)$ sequence $s_k = c_1 c_2 \cdots c_n$. It is a $|\Sigma| \times (n + 1)$ matrix and its (i, j) element can be defined by the following formula:

$$DT_k[i,j] = |\{m \mid c_m = \sigma_i,\ m > j,\ 0 \le j \le n,\ 1 \le i \le |\Sigma|,\ \sigma_i \in \Sigma\}|, \tag{7}$$

where σ_i is the i^{th} character in Σ, and $DT_k[i, j]$ represents the number of characters c_m after position j in s_k.

Once Distance Tables are defined, for any match point $p = (p_1, p_2, \cdots, p_d)$, we can quickly compute the $U_{(p,\infty)}$ according to the following formula.

$$U_{(p,\infty)} = \sum_{1 \le i \le |\Sigma|} \min\{DT_1[i, p_1], DT_2[i, p_2], \cdots, DT_d[i, p_d]\}. \tag{8}$$

For example, given two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$, Figure 11 gives the Distance Tables of these two sequences according to Eq. (7).

			G	A	A	G	C	G	T	A
		0	1	2	3	4	5	6	7	8
A	1	3	3	2	1	1	1	1	1	0
C	2	1	1	1	1	1	0	0	0	0
G	3	3	2	2	2	1	1	0	0	0
T	4	1	1	1	1	1	1	1	0	0

(a) DT_1

			A	G	T	C	T	G	A	C
		0	1	2	3	4	5	6	7	8
A	1	2	1	1	1	1	1	1	0	0
C	2	2	2	2	2	1	1	1	1	0
G	3	2	2	1	1	1	1	0	0	0
T	4	2	2	2	1	1	0	0	0	0

(b) DT_2

Fig. 11. The distance tables of the given sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$, which are built according to Eq. (7).

For match point $(5, 4)$, by Eq. (8), one has $U_{((5,4),\infty)} = min\{DT_1[1,5], DT_2[1,4]\} + min\{DT_1[2,5], DT_2[2,4]\} + min\{DT_1[3,5], DT_2[3,4]\} + min\{DT_1[4,5], DT_2[4,4]\} = min\{1,1\} + min\{0,1\} + min\{1,1\} + min\{1,1\} = 3$.

By using distance tables, we can compute $U_{(p,\infty)}$ of any match point quickly. The computation complexity is $O(d * |\Sigma|)$ for each match point.

4.2.2 The Pseudo-code of The Proposed Algorithm

In order to describe the new algorithm in detail, a pseudo-codes of algorithm *BEST-MLCS* is given in Algorithm 1.

At the beginning, the preparation and initialization are shown in $line1 \sim line4$, where Successor Tables ST_i $(1 \leq i \leq d)$ and Distance Tables DT_i $(1 \leq i \leq d)$ are built. The estimated lower bound L_{mlcs} is calculated. $line5 \sim line32$ are the key steps of the proposed algorithm, which shows how a smaller DAG is constructed level by level. Finally, the longest paths corresponding to MLCSs can be obtained from the smaller DAG and all MLCSs will be returned in $line33 \sim line34$.

Algorithm 1 Pseudo-codes of Algorithm *BEST-MLCS*

Input: d given sequences.
Output: The MLCSs of d sequences.
Pre-processing (1-4):
1: Build successor tables $ST_i (1 \leq i \leq d)$ and distance tables $DT_i (1 \leq i \leq d)$ on d sequences.
2: Define the initial and end match points $O = (0, 0, \cdots, 0)$ and $\infty = (\infty, \infty, \cdots, \infty)$.
3: Compute the estimated lower bound of the length of the longest paths L_{mlcs}.
4: $L^k \leftarrow \{O\}$, $O.lev \leftarrow 0$, $k \leftarrow 0$, $H \leftarrow \{O\}$; // H indicates the hash table
5: **while** $L^k \neq \emptyset$ **do**
6: **for** $p \in L^k$ && $p.lev == k$ **do**
7: compute p's successors $succ(p)$;
8: **for** $q \in succ(p)$ **do**
9: **if** $q \in H$ **then**
10: take q from H;
11: **if** $q.lev < k + 1$ **then**
12: remove all previous paths passing through q;
13: $q.lev \leftarrow k + 1$;
14: **else if** $q.leve == k + 1$ **then**
15: $q.precs \leftarrow q.precs \cup \{p\}$;
16: continue;
17: **end if**
18: **else**
19: $q.lev \leftarrow k + 1$; // if $q \notin H$
20: **end if**
21: compute $U_{(q,\infty)}$ according to Equation 8;
22: **if** $q.lev + U_{(q,\infty)} < L_{mlcs}$ **then**
23: remove q;
24: **else**
25: $H \leftarrow H \cup \{q\}$;
26: $L^{k+1} \leftarrow L^{k+1} \cup \{q\}$; //add q to the L^{k+1}
27: $q.prec \leftarrow \{p\}$;
28: **end if**
29: **end for**
30: **end for**
31: $k \leftarrow k + 1$;
32: **end while**
33: the smaller DAG is constructed and the longest paths can be found; Accordingly, the corresponding $MLCS$s are obtained.
34: return $MLCS$s;

5 Experimental Results and Analysis

In order to verify the good performance of the proposed algorithm *BEST-MLCS* on large-scale MLCS problems (to be specific, in this paper, the large-scale MLCS problem refers to that the number of DNA sequences is greater than 10000 and the length is not less than 100), we compare it with three state-of-the-art algorithms *FAST_LCS* [22], *Quick-DP* [24] and *Top_MLCS* [25] by experiments. All the compared algorithms are implemented in Java and run on a Dell T7920 workstation equipped with an Intel(R) Xeon(R) Gold 6138 CPU (2.00GHz) and 704GB of memory. The experiments are conducted on the widely used real DNA sequences in the bio-informatics domain [1] and the random synthetic DNA sequences with the alphabet $\{A, C, G, T\}$. We conduct the following four types of experiments:

1) Fix the number of sequences to 40000 and change the length of the sequences from 60 to 165 in order to compare the performance of the 4 compared algorithms on problems with a large number of sequences. In this type of experiments, we conduct experiments on total 22 test instances and make the following two comparisons:
   - Comparison of time and space consumption of 4 compared algorithms. The results are given in Table 1, Figure 12, and Figure 13.
   - Comparison of sizes of DAG (number of match points in DAG) constructed by 4 compared algorithms. The results are given in Table 2 and Figure 14.
2) Change the number of sequences from 70000 to 1000000 and fix the length of sequences to 110 in order to compare the performance of 4 compared algorithms on problems whose number of sequences is very large and changes. In this type of experiments, we conduct experiments on total 24 test instances and make the comparison of time and space consumption of 4 compared algorithms. The results are given in Table 3.
3) Comparison of the longest length of sequences which 4 compared algorithms can deal with when the number of sequences changes from 70000 to

1. http://www.ncbi.nlm.nih.gov/nuccore/110645304?report=fasta

1000000. The results are given in Table 4, Figure 15, and Figure 16.

4) Robustness of estimating the lower bound L_{mlcs} of the length of MLCS with the change of parameter θ, and the consumed time of the estimation for problems with 40000 sequences and the length from 60 to 165. In this type of experiments, we conduct experiments on total 22 test instances. The results are given in Table 5 and Figure 17.

In these Tables, $|MLCS|$ represents the length of MLCSs obtained, '-' indicates that the algorithm cannot find out the final results in a pre-assigned execution time, which is set to 2 hours in the experiments, and '+' indicates that the algorithm cannot be executed successfully due to memory overflow.

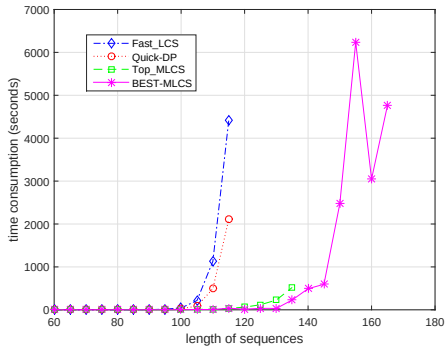


Fig. 12. The run-time (seconds) consumed by 4 compared algorithms on 22 test instances with 40000 sequences of the length varying from 60 to 165.

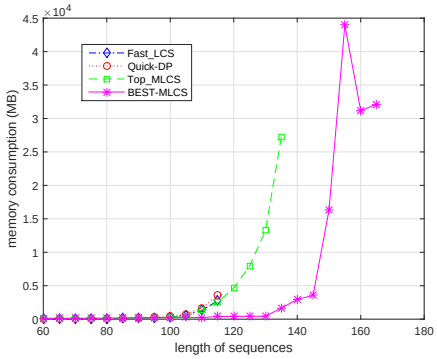


Fig. 13. The memory (MB) consumed by 4 compared algorithms on 22 test instances with 40000 sequences of the length varying from 60 to 165.

For the first type of experiments, from Table 1 and Figure 12, it can be seen that, for the same test problem, *FAST_LCS*, *Quick-DP* and *Top_MLCS* always require more time to find MLCS than *BEST-MLCS*. For example, when the length of sequences is 110, *FAST_LCS*, *Quick-DP* and *Top_MLCS* consume 1130.58, 490.95 and 16.73 seconds, respectively, while *BEST-MLCS* only consumes within 7.94 seconds, which is only 0.7%, 1.6% and 47.5% of time used by three compared algorithms, respectively. Among four algorithms, *FAST_LCS* and *Quick-DP* perform the worst on time consumption, i.e., they usually need much more time than the other two algorithms. The reason is that both of them adopt a non-dominated sorting technology to identify the dominated points on every level. With the increase of the number of level, the number of points will increase exponentially. Thus, the time consumed will grow exponentially, which results in that they cannot find out the final results in the pre-assigned time limit (say 2 hours) and memory limit (say 50G) when the length of sequences is greater than 110 or the length of MLCSs is greater than 14. Also, *Top_MLCS* performs better than *FAST_LCS* and *Quick-DP* on time consumption. The reason is that it employs a topological sorting technology rather than non-dominated sorting technology, thus a lot of computing time can be saved. However, *Top_MLCS* has to build a big DAG firstly and then utilises two topological sorting operations (i.e., forward and backward topological sorting) to get a smaller DAG. These operations also consume a lot of computing time. While *BEST-MLCS* directly constructs the smaller DAG. Thus, *Top_MLCS* always requires more time than *BEST-MLCS*. When the length of sequences varies from 60 to 130, the computing time taken by *Top_MLCS* is about 2.34 times that of *BEST-MLCS* on average. When the length of sequences is larger than 135, *Top_MLCS* uses much more time than *BEST-MLCS* and it cannot find MLCS. When the length of sequences is larger than 115, *FAST_LCS* and *Quick-DP* cannot find MLCS.

For the memory consumption, it can be seen from Table 1 and Figure 13 that, although *Top_MLCS*, *FAST_LCS* and *Quick-DP* consume little memory than *BEST-MLCS* when the length of sequences is not greater than 95 (this is because *BEST-MLCS* has to require additional memory to estimate the lower bound L_{mlcs}), memory space consumed by *Top_MLCS*, *FAST_LCS* and *Quick-DP* grows rapidly even exponentially as the length of sequences increases. Obviously, *Top_MLCS* performs better than *FAST_LCS* and *Quick-DP*. *FAST_LCS* and *Quick-DP* always fail to find MLCS for problems with length larger than 115 due to the timeout, and *Top_MLCS* always fails to find MLCS for problems with the length larger than 135 due to memory overflow (i.e., there is not enough memory to store the DAG). For larger-scale problems with 40000 sequences and the sequence length being larger than 95, *BEST-MLCS* performs better than three compared algorithms on memory consumption. Also, it can be seen from Figure 13 that the curve of memory consumption by *BEST-MLCS* grows much slower than that by three compared algorithms. Thus, the proposed algorithm *BEST-MLCS* greatly outperforms three compared algorithms *FAST_LCS*, *Quick-DP* and *Top_MLCS* on time and memory consumption when handling large-scale MLCS problems.

Note that, in general, as the length of sequences increases, the run-time and memory space consumed by algorithm *BEST-MLCS* will grow. However, much more run-time and memory space are consumed when the length is 155 instead of 160 or 165 (see Table 1 or Figures 12-13). This is because the Lower Bound L_{mlcs} estimated by our estimation scheme on the problem of the 40000 sequences with length of 155 is less precise than those on the problems of 40000 sequences with lengths 160 and 165, respectively, resulting in that more useless match points cannot be found for the problem with length 155 than those for the problems with lengths 160 and 165, respectively. Thus the size of DAG

TABLE 1
The time and memory consumption of 4 compared algorithms on 22 test instances with 40000 sequences of the length varying from 60 to 165.

Lengths of sequences	|MLCS|	Time(seconds)				Memory(Metabytes)			
		FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS	FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS
60	4	0.22	0.22	0.22	0.09	49.2	46.9	54.7	123.8
65	4	0.25	0.24	0.24	0.11	51.2	50.7	55.0	136.4
70	5	0.34	0.28	0.31	0.26	61.3	59.7	58.6	136.3
75	6	0.49	0.33	0.59	0.27	76.8	72.2	58.2	148.4
80	7	0.80	0.46	0.63	0.41	109.0	93.0	96.7	157.1
85	8	1.87	0.65	0.85	0.45	138.8	129.2	83.0	168.1
90	9	3.72	2.55	1.36	0.99	169.8	201.4	127.4	181.1
95	10	15.51	7.12	2.24	1.39	255.8	268.3	174.0	192.1
100	11	43.75	25.82	4.24	2.95	267.2	426.4	295.8	254.9
105	12	216.11	86.18	7.86	6.41	612.7	762.4	497.4	278.0
110	14	1130.58	490.95	16.73	7.94	1328.1	1667.0	1392.1	241.0
115	14	4408.69	2107.50	45.02	38.38	2773.0	3620.2	2533.2	401.6
120	15	-	-	62.61	15.49	-	-	4682.7	403.2
125	16	-	-	118.12	35.19	-	-	7930.5	422.1
130	16	-	-	227.62	31.96	-	-	13282.0	441.3
135	18	-	-	521.04	229.73	-	-	27249.9	1648.7
140	19	-	-	+	492.20	-	-	+	2954.1
145	20	-	-	+	595.12	-	-	+	3561.1
150	21	-	-	+	2472.51	-	-	+	16271.4
155	22	-	-	+	6234.20	-	-	+	43973.2
160	22	-	-	+	3048.15	-	-	+	31178.7
165	23	-	-	+	4760.16	-	-	+	32142.3

for the problem with length 155 is larger than those for the problems with lengths 160 and 165, respectively. Therefore, *BEST-MLCS* spent more run-time and memory space for the problem with length 155. But any way, *BEST-MLCS* performs better than three compared algorithms.

TABLE 2
The number of match points on each level of DAG constructed by *FAST_LCS*, *Quick-DP*, *Top_MLCS* and *BEST-MLCS* for problem with 40000 DNA sequences and length 110.

Level number	Number of match points			
	FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS
0	1	1	1	1
1	3	3	3	3
2	9	9	9	9
3	23	23	23	17
4	52	52	51	31
5	111	111	109	50
6	230	230	229	61
7	457	457	452	83
8	823	823	778	102
9	1254	1254	1092	124
10	1535	1535	1176	124
11	1426	1426	866	118
12	917	917	344	103
13	346	346	48	48
14	48	48	1	2
15	1	1	1	1

The key reason why *BEST-MLCS* can perform better than three compared algorithms with less time and memory consumption and can deal with the large-scale MLCS problems is that *BEST-MLCS* constructs much smaller DAG than three compared algorithms. In fact, Table 2 and Figure 14 show the number of match points in every level of DAG constructed by *FAST_LCS*, *Quick-DP*, *Top_MLCS* and *BEST-MLCS* for problems with 40000 sequences and length 110. It can be seen from these results that the size of DAG constructed by *BEST-MLCS* is much smaller than that by three compared algorithms. For example, at level-10, the number of match points created by *BEST-MLCS* is 124, while those by *FAST_LCS*, *Quick-DP*, *Top_MLCS* are 1535, 1535 and 1176, respectively. Please note that, since the dominant set found by *Quick-DP* is the same as that by *FAST_LCS* (the proof is given in [24]), the number of nodes on each level

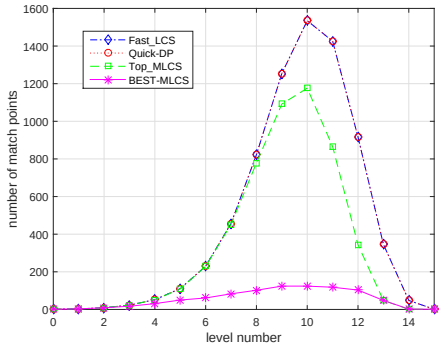


Fig. 14. The number of match points on each level of DAG constructed by *FAST_LCS*, *Quick-DP*, *Top_MLCS* and *BEST-MLCS* for problem with 40000 DNA sequences and length 110.

of DAG constructed by *FAST_LCS* is the same as that by *Quick-DP* as shown in Table 2 and Figure 14.

To further compare the performance of 4 algorithms, the second type of experiments is conducted on 12 problems with the number of sequences varying from 70000 to 1000000 and the length fixed to 110. Table 3 shows the experimental results on the condition that the maximum memory space is set to 256 Gigabyte and the maximum run-time is set to 3 hours. Neither *FAST_LCS* nor *Quick-DP* can find MLCSs for these problems. Also, *Top_MLCS* can only obtain MLCSs on the problems with the number of sequences varying from 70000 to 100000. Once the number of sequences is greater than 100000, *Top_MLCS* always fails because the memory consumed by it is very huge and often exceeds the maximum limit. By contrast, *BEST-MLCS* can find MLCSs on all problems we have tried so far. The time and memory consumed by *BEST-MLCS* are relatively not large compared to the scale of the problems.

For the third type of experiments, when the memory limit is set to 100 Gigabytes and the run-time limit is set to 10 hours, the longest lengths of 70000 to 1000000 sequences which can be handled by four compared algorithms

TABLE 3
Time and space consumption of *FAST_LCS*, *Quick-DP*, *Top_MLCS* and *BEST-MLCS* on problems with 70000 to 1000000 sequences and the length of sequences fixed to 110.

Number of sequences	|MLCS|	Time(seconds)				Memory(Metabytes)			
		FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS	FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS
70000	20	-	-	4841.10	2393.38	-	-	209074.4	5120.9
90000	20	-	-	6235.69	2816.85	-	-	252804.9	7126.4
100000	19	-	-	5910.01	2257.07	-	-	219772.6	7212.1
200000	18	-	-	+	5061.26	-	-	+	14125.5
300000	18	-	-	+	5545.13	-	-	+	16597.1
400000	18	-	-	+	7793.38	-	-	+	25172.7
500000	18	-	-	+	8032.28	-	-	+	27282.2
600000	17	-	-	+	8723.03	-	-	+	24306.3
700000	17	-	-	+	8214.76	-	-	+	21836.6
800000	17	-	-	+	7486.52	-	-	+	22455.7
900000	17	-	-	+	9733.57	-	-	+	30868.5
1000000	17	-	-	+	5767.76	-	-	+	17069.3

TABLE 4
The length of sequences which 4 algorithms can handle within 10 hours and 100G memory for problems with 70000 to 1000000 sequences, and the length of MLCS obtained.

Number of sequences	Length of sequences				Length of MLCSs			
	FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS	FAST_LCS	Quick-DP	Top_MLCS	BEST-MLCS
70000	90	95	105	120	15	16	19	22
90000	90	90	105	125	14	14	18	23
100000	95	95	105	125	15	15	18	23
200000	90	95	105	120	14	15	17	21
300000	95	95	100	115	14	14	16	19
400000	90	95	100	120	13	14	15	20
500000	95	95	100	120	14	14	15	20
600000	95	95	95	120	14	14	14	20
700000	95	95	95	115	13	13	13	18
800000	95	95	95	115	13	13	13	17
900000	95	95	95	115	13	13	13	18
1000000	95	95	95	115	13	13	13	18

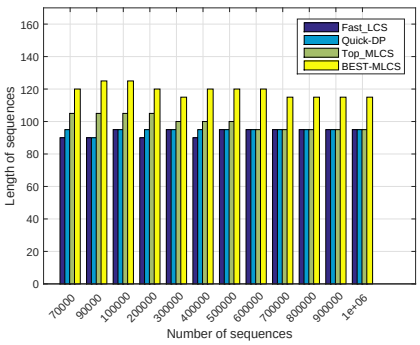


Fig. 15. The length of sequences which 4 algorithms can handle within 10 hours and 100G memory for problems with 70000 to 1000000 sequences.

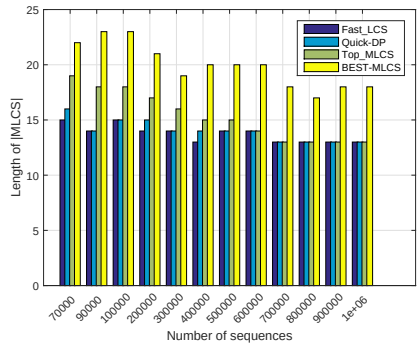


Fig. 16. The length of MLCSs obtained by 4 algorithms within 10 hours and 100G memory for problems with 70000 to 1000000 sequences.

are given in Table 4 and Figure 15. For less than 200000 sequences, *FAST_LCS* and *Quick-DP* can only handle the problems with the length no more than 95, and *Top_MLCS* can handle the problems with the length no more than 105. By contrast, *BEST-MLCS* can handle the problems with the length up to 125. For 300000 to 500000 sequences, *FAST_LCS* and *Quick-DP* can only handle the problems with the length no more than 95, and *Top_MLCS* can handle the problems with the length no more than 100, but *BEST-MLCS* can handle the problems with the length up to 120. For 600000 to 1000000 sequences, *FAST_LCS*, *Quick-DP* and *Top_MLCS* can only handle the problems with the length no more than 95, but *BEST-MLCS* can handle the problems with the length up to 115.

Furthermore, the maximum lengths of MLCSs obtained by four compared algorithms are given in Table 4 and Figure 16. From the experimental results, it can be seen that *BEST-MLCS* can always find much longer MLCS than *FAST_LCS*, *Quick-DP* and *Top_MLCS*. For example, for problem with 1000000 sequences, the length of sequence *BEST-MLCS* can handle is 115 and the length of MLCS is 18, while the maximum length of sequences the compared algorithms can handle is only 95 and the length of MLCS is only 13. In fact, to the best of our knowledge, the scale of the problems handled by *BEST-MLCS* is the largest and the results obtained by *BEST-MLCS* are the best so far.

In the proposed algorithm, there is one parameter θ in estimating the lower bound L_{mlcs} of the length of MLCS. For the fourth type of experiments, to test the robustness of the estimation method, we investigate the effect of θ on L_{mlcs} by varying the values of θ through the experiments.

TABLE 5
Robustness of estimated lower bound L_{mlcs} with the change of θ and the consumed time for 40000 sequences with length from 60 to 165.

Length of sequences	|MLCS|	L_{mlcs}						Consumed Time (Milliseconds)					
		θ=32	θ=64	θ=128	θ=256	θ=512	θ=1024	θ=32	θ=64	θ=128	θ=256	θ=512	θ=1024
60	4	4	4	4	4	4	4	13	12	15	12	13	12
65	4	4	4	4	4	4	4	16	17	17	16	15	17
70	5	5	5	5	5	5	5	30	30	30	30	30	30
75	6	6	6	6	6	6	6	53	52	52	52	53	54
80	7	7	7	7	7	7	7	95	90	93	93	90	96
85	8	8	8	8	8	8	8	130	134	138	139	139	145
90	9	9	9	9	9	9	9	200	290	335	320	338	354
95	10	10	10	10	10	10	10	258	365	557	588	588	610
100	11	11	11	11	11	11	11	323	511	837	1254	1259	1319
105	12	11	11	11	11	12	12	377	618	1062	1656	2516	2538
110	14	13	13	13	13	13	14	435	730	1282	2155	3402	5758
115	14	13	13	13	13	14	14	504	892	1619	2825	4630	7571
120	15	14	14	14	15	15	15	599	1061	1954	3624	5969	9601
125	16	13	14	14	15	15	15	662	1217	2264	4359	7515	13159
130	16	15	16	16	16	16	16	736	1401	2663	5344	9473	16677
135	18	16	17	17	17	17	17	845	1582	3039	5885	10851	20444
140	19	18	18	18	18	18	18	964	1812	3437	7088	12446	24438
145	20	16	16	19	19	19	19	983	1758	3471	6852	12634	24528
150	21	19	19	19	19	19	20	1135	2125	4061	8471	15865	29851
155	22	19	19	19	19	19	20	1207	2238	4404	9479	17156	33252
160	22	19	20	20	21	21	21	1240	2306	4450	9278	17424	34360
165	23	19	19	19	21	21	21	1337	2475	4826	12065	20108	42033

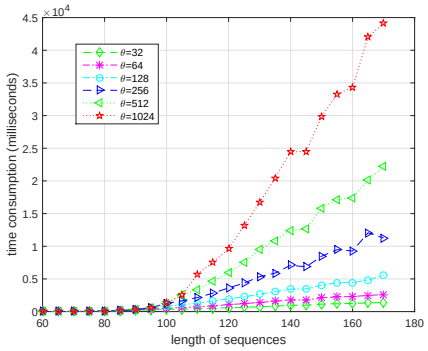


Fig. 17. The comparisons about the consumed time required by the fast heuristic method to estimate the lower bound value on 40000 DNA sequences with length varying from 60 to 165 when the parameter θ has different values. The data are taken from Table 5.

Also, we study the time consumed by taking the different values of θ. The results are given in Table 5 and Figure 17.

The more precise the estimated lower bound L_{mlcs}, the much more useless match points can be identified and can be excluded from DAG, and thus the more smaller DAG constructed will be. The experiments are conducted on problems with 40000 sequences and the length varying from 60 to 165. From experimental results, it can be seen that for each test problem with the fixed length among total 22 test problems, the values of L_{mlcs} are almost unchanged (or at most little changed) with the change of the values of θ. This indicates the effect of θ on L_{mlcs} is small and the estimation method of L_{mlcs} is robust.

However, as θ increases, the time consumed by the estimation method will increase. Thus, it is better to choose a small value of θ. On the other hand, note that generally, the larger the value of θ, the more precise the estimated lower bound. This implies that we should take a large value of θ. To balance the time consumed and the precision of the lower bound estimation and note the robustness of the estimation method, it is better to take a moderate value of θ. In all experiments aforementioned, we take the value of θ to be 256.

6 Conclusion

This paper has proposed a novel *BEST-MLCS* algorithm to tackle the large-scale MLCS problems effectively and efficiently, which has the following four key components: 1) a method to precisely estimate the lower bound of the length of MLCS; 2) a scheme to estimate the upper bound of the longest path through the current match point; 3) a branch elimination strategy by identifying the useless match points; 4) a method to construct the smallest DAG (i.e., DAG constructed by the proposed algorithm is much smaller than that constructed by the existing state-of-the-art algorithms).

The proposed algorithm *BEST-MLCS* outperforms the existing state-of-the-art *FAST_LCS*, *Quick-DP* and *Top_MLCS*, and can tackle the large-scale MLCS problems. The experimental results on 68 test problems have shown that the time consumption and space consumption of the proposed algorithm are much smaller than those of *FAST_LCS*, *Quick-DP* and *Top_MLCS*, especially for large-scale MLCS problems.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grants 61872281 and 61672444.

References

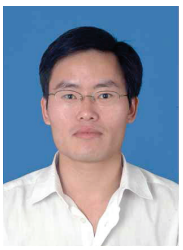
[1] Q. Wang, M. Pan, Y. Shang, and D. Korkin, "A fast heuristic search algorithm for finding the longest common subsequence of multiple strings," in *Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, Usa, July*, 2010.

[2] J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," in *Computing Science Technical Report 41*. AT & T Bell Laboratories, 1975.

[3] S.-Y. Lu and K. S. Fu, "A sentence-to-sentence clustering procedure for pattern analysis," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 8, no. 5, pp. 381–389, 1978.

[4] D. Sankoff and M. Blanchette, "Phylogenetic invariants for genome rearrangements," *Journal of Computational Biology*, vol. 6, no. 3-4, pp. 431–445, 1999.

[5] Y. P. Wang and T. Pavlidis, "Optimal correspondence of string subsequences," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 11, pp. 1080–1087, 1990.
[6] B. Su and Y. Wu, "Learning meta-distance for sequences by learning a ground metric via virtual sequence regression," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. [Online]. Available: https://doi.org/10.1109/TPAMI.2020.3010568
[7] G. Bourque and P. A. Pevzner, "Genome-scale evolution: Reconstructing gene orders in the ancestral species," *Genome Research*, vol. 12, no. 1, pp. 26–36, 2002.
[8] A. M. Aravanis, M. Lee, and R. D. Klausner, "Next-generation sequencing of circulating tumor DNA for early cancer detection," *Cell*, vol. 168, no. 4, pp. 571–574, 2017.
[9] T. K. Attwood and J. B. C. Findlay, "Fingerprinting g-protein-coupled receptors," *"Protein Engineering, Design and Selection"*, vol. 7, no. 2, pp. 195–203, 1994.
[10] J. B. Kruskal, "An overview of sequence comparison: Time warps, string edits, and macromolecules," *SIAM Review*, vol. 25, no. 2, pp. 201–237, 1983.
[11] D. Sankoff, "Matching sequences under deletion/insertion constraints," *Proceedings of the National Academy of Sciences*, vol. 69, no. 1, pp. 4–6, 1972.
[12] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, vol. 24, no. 4, pp. 664–675, 1977.
[13] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, 1980.
[14] W. J. Hsu and M. W. Du, "Computing a longest common subsequence for a set of strings," *BIT*, vol. 24, no. 1, pp. 45–59, 1984.
[15] A. Apostolico, S. Browne, and C. Guerra, "Fast linear-space computations of longest common subsequences," *Theoretical Computer Science*, vol. 92, no. 1, pp. 3–17, 1992.
[16] J. Gregor and M. G. Thomason, "Dynamic programming alignment of sequences representing cyclic patterns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 2, pp. 129–135, 1993.
[17] 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*, vol. 102, no. 2-3, pp. 99–103, 2007.
[18] J. Yang, Y. Xu, Y. Shang, and G. Chen, "A space-bounded anytime algorithm for the multiple longest common subsequence problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2599–2609, 2014.
[19] K. Hakata and H. Imai, "The longest common subsequence problem for small alphabet size between many strings," in *Proceedings of the Third International Symposium on Algorithms and Computation*, ser. ISAAC '92. Berlin, Heidelberg: Springer-Verlag, 1992, p. 469478.
[20] ——, "Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima," *Optimization Methods and Software*, vol. 10, no. 2, pp. 233–260, 1998.
[21] D. Korkin, "A new dominant point-based parallel algorithm for multiple longest common subsequence problem," *Technical Report TR01-148, Univ. of New Brunswick, Tech. Rep.*, 2001.
[22] Y. Chen, A. Wan, and W. Liu, "A fast parallel algorithm for finding the longest common sequence of multiple biosequences," *BMC Bioinformatics*, vol. 7, no. S4, 2006.
[23] D. Korkin, Q. Wang, and Y. Shang, "An efficient parallel algorithm for the multiple longest common subsequence (MLCS) problem," in *2008 37th International Conference on Parallel Processing*. IEEE, September 2008, pp. 354–363.
[24] Q. Wang, D. Korkin, and Y. Shang, "A fast multiple longest common subsequence (MLCS) algorithm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 3, pp. 321–334, 2011.
[25] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, and J. Huang, "A novel fast and memory efficient parallel MLCS algorithm for long and large-scale sequences alignments," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, May 2016, pp. 1170–1181.
[26] Z. Peng and Y. Wang, "A novel efficient graph model for the multiple longest common subsequences (MLCS) problem," *Frontiers in Genetics*, vol. 8, 2017.
[27] S. Wei, Y. Wang, Y. Yang, and S. Liu, "A path recorder algorithm for Multiple Longest Common Subsequences (MLCS) problems," *Bioinformatics*, vol. 36, no. 10, pp. 3035–3042, 2020. [Online]. Available: https://doi.org/10.1093/bioinformatics/btaa134
[28] S. Liu, Y. Wang, W. Tong, and S. Wei, "A fast and memory efficient MLCS algorithm by character merging for DNA sequences alignment," *Bioinformatics*, vol. 36, no. 4, pp. 1066–1073, 10 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz725
[29] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.
[30] X. Zhang, Y. Tian, R. Cheng, and Y. Jin, "A decision variable clustering-based evolutionary algorithm for large-scale many-objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 1, pp. 97–112, 2018.
[31] C. Blum, M. J. Blesa, and M. López-Ibáñez, "Beam search for the longest common subsequence problem," *Computers & Operations Research*, vol. 36, no. 12, pp. 3178–3186, 2009.
[32] C. Blum and M. J. Blesa, *Beam search for the longest common subsequence problem*. Elsevier Science Ltd., 2009.



Shiwei Wei is a Ph.D. student in School of Computer Science and Technology, Xidian University, Xi'an, China, and an associate professor in the School of Computer Science and Engineering, Guilin University of Aerospace Technology, Guilin, China. His research interests include data mining, intelligent computation and bioinformatics.

Yuping Wang received the PhD degree from the Department of Mathematics, Xian Jiaotong University, China, in 1993. He has been a full professor since 1997 in the Department of Applied Mathematics and School of Computer Science and Technology, Xidian University, China. His research interests include optimization modeling for problems in computer Science, evolutionary computation, and optimization algorithms. He has published more than 200 papers.

Yiu-ming Cheung (SM'06-F'18) received the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong. He is currently a Full Professor with the Department of Computer Science, Hong Kong Baptist University, Hong Kong. His current research interests include machine learning, pattern recognition, visual computing, and optimization. Dr. Cheung is a fellow of IET, British Computer Society (BCS), and Royal Society of Arts (RSA) and a Distinguished Fellow of International Engineering and Technology Institute (IETI). He is the Founding Chair of the Computational Intelligence Chapter of the IEEE Hong Kong Section and the Chair of the Technical Committee on Intelligent Informatics of the IEEE Computer Society. He serves as an Associate Editor for the IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, the IEEE TRANSACTIONS ON CYBERNETICS, Pattern Recognition, Knowledge and Information Systems, and Neurocomputing, to name a few.