Pattern Recognition 119 (2021) 108059

Contents lists available at ScienceDirect

Pattern Recognition

journal homepage: www.elsevier.com/locate/patcog

A branch and bound irredundant graph algorithm for large-scale MLCS problems

Chunyang Wang[a], Yuping Wang[a,*], Yiuming Cheung[b]

[a] School of Computer Science and Technology, Xidian University, Xian, Shaanxi, China
[b] Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR, China

ARTICLE INFO

Article history:
Received 14 November 2020
Revised 6 April 2021
Accepted 18 May 2021
Available online 28 May 2021

Keywords:
Multiple longest common subsequences
Small DAG
Branch and bound
Gene alignment

ABSTRACT

Finding the multiple longest common subsequences (MLCS) among many long sequences (i.e., the large scale MLCS problem) has many important applications, such as gene alignment, disease diagnosis, and documents similarity check, etc. It is an NP-hard problem (Maier et al., 1978). The key bottle neck of this problem is that the existing state-of-the-art algorithms must construct a huge graph (called direct acyclic graph, briefly DAG), and the computer usually has no enough space to store and handle this graph. Thus the existing algorithms cannot solve the large scale MLCS problem. In order to quickly solve the large-scale MLCS problem within limited computer resources, this paper therefore proposes a branch and bound irredundant graph algorithm called Big-MLCS, which constructs a much smaller DAG (called Small-DAG) than the existing algorithms do by a branch and bound method, and designs a new data structure to efficiently store and handle Small-DAG. By these schemes, Big-MLCS is more efficient than the existing algorithms. Also, we compare the proposed algorithm with two state-of-the-art algorithms through the experiments, and the results show that the proposed algorithm outperforms the compared algorithms and is more suitable to large-scale MLCS problems.

1. Introduction

Sequences over a finite alphabet Σ are a common type of data. A typical example of such sequences is DNA sequences consisting of four characters ($\Sigma = \{A, C, G, T\}$). One basic problem in the fields of pattern recognition, gene alignment, disease diagnosis, and documents similarity check is to find the longest common subsequences of many sequences. According to the number of the involved sequences, this problem can be divided into two categories: finding the longest common subsequences (LCS) of two sequences is called an LCS problem, and finding the longest common subsequence among three or more sequences is called an MLCS problem, where the length and number of the longest common subsequences are commonly used to measure the similarity between sequences. In this paper, we study MLCS problem, which has a number of attractions, such as cancer detection [1], cancer treatment [2], protein sequence alignment [3], protein sequence classifying [4], mRNA processing [5], body sensor network researching [6], gene data searching [7], and gene data analysing [8]. Unfortunately, the MLCS problem is an NP-hard problem [9]. In the decades, a number of methods have been developed to solve LCS and MLCS problems. Roughly these methods can be classified into two classes: Exact algorithms and Approximation algorithms. Exact algorithms are ones which try to find all MLCS in a single run, while approximation algorithms are ones which try to find an approximate/true MLCS quickly, however, the approximate algorithms can only get a part of the approximate/true MLCS. Blum et al. [10] proposed a beam search method to solve MLCS problem. Yang et al. proposed a parallel algorithm called Pro-MLCS [11], and SA-MLCS algorithm [12] to further reduce space overhead. Etminan et al. [13] proposed a method called FAME to speed up the search process. In this paper, we will focus our attention to the exact algorithms. There are two kinds of exact algorithms: dynamic programming based algorithms and dominant point based algorithms. The dynamic programming based algorithm [14] was first used to quickly find LCS of two sequences, and the time complexity of the algorithm is O (nm), where n and m are the lengths of two sequences, respectively. Apostolico et al. [15] proposed a method to reduce the space cost of the algorithm without affecting the time complexity of the original algorithm. Tchendji et al. [16] proposed a parallel algorithm to reduce the execution time of the algorithm. However, the time complexity of traditional dynamic programming algorithm will increase exponentially with the number of sequences, which is obviously not suitable for solving

* Corresponding author.
E-mail addresses: 867174762@qq.com (C. Wang), ywang@xidian.edu.cn (Y. Wang), ymc@comp.hkbu.edu.hk (Y. Cheung).

https://doi.org/10.1016/j.patcog.2021.108059
0031-3203/

MLCS problems. With the advancement of gene sequencing technology, the scale of the existing human gene database has grown rapidly [17], and algorithms based on dynamic programming can no longer meet the needs of finding the longest common subsequence in multiple sequences. In order to effectively solve the MLCS problem, some scholars have put forward the algorithms based on dominant point in recent years. The core idea of these algorithms is to construct a Directed Acyclic Graph (DAG) through non-dominated sorting, thereby transforming the MLCS problem into searching the longest path in a DAG. Experiments show that, compared with the dynamic programming based algorithms, the dominant point based algorithms greatly reduce the search space and thus achieve a great performance improvement. The first dominant point based algorithm was proposed by Hunt in 1977 [18], and its time complexity is $O((r+n)logn)$, where r is the number of nodes in DAG. Hakata et al. [19] proposed a strategy to solve larger scale MLCS problem. Korkin [20] gave a parallel version to reduce the time for searching for LCSs. In order to further improve the performance of the dominant point based algorithms, Chen [21] proposed FAST-LCS algorithm. Although a part of the space storage is sacrificed by constructing a Successor Table in this algorithm, the speed of constructing DGA is accelerated. Further, as a representative of the efficient dominant point based algorithms, the Quick-DPAR algorithm [22] reduces the times of non-dominated sorting by using a new rule. Gustavsson and Syberfeldt [23] accelerated the non-dominated sorting by using non-dominated tree. Nevertheless, these dominant point based algorithms will still quickly use up its memory space before a MLCS is found. In fact, during the search process, the number of nodes in each layer of the DAG will increase exponentially, resulting in a huge space overhead, even the memory space being exhausted. Also, as the nodes in each layer increase exponentially, the time required for non-dominated sorting in each layer will also increase exponentially, resulting in the severe heavy time cost. In 2016, Li et al. [24] proposed an algorithm named Top-MLCS, which has an overwhelming advantage over the existing dominant point based algorithms. The merit of Top-MLCS algorithm is that it created a DAG without redundant nodes, which is called Irredundant Common Subsequences Graph (ICSG). That is, the same point appears only once, and the non-dominated sorting between points is avoided, thus saving a lot of time and space. However, it needs forward and backward topological sorting schemes. For large-scale MLCS problems, DAG constructed by TOP-MLCS is still very huge and TOP-MLCS still faces the problem of memory exhausted. The main reasons are as follows: 1) To circumvent redundant nodes in the graph, Top-MLCS has to store all the built nodes of ICSG (implemented by Hash table). As construction of ICSG continues, the number of nodes in the DAG and Hash table increases rapidly, resulting in too large ICSG and the memory exhausted. 2) After the construction of the ICSG is completed, it needs to use topological sorting twice to find the MLCS, which increases the time cost. To construct small DAG, Liu et al. [25] designed a character merging scheme which merges the consecutively repeated characters in the sequences. However, when there are fewer consecutive repeated characters in sequences, the improvement of the strategy will be limited. To overcome the shortcomings of the existing dominant point based algorithms and effectively solve large-scale MLCS problems in a limited memory, we design a branch and bound irredundant graph algorithm called Big-MLCS. Our main contributions are as follows:

- **Design a branch and bound strategy for identifying non-contributed points and non-longest paths.** If we can judge that a match point is unlikely to appear in MLCS, this point and all paths through this point will not have contribution to search the longest path, and they are called non-contributed point and non-longest paths, respectively. Therefore, we do not put them on DAG, thereby reducing the scale of DAG. To do so, we design a branch and bound strategy to identify them.
- **Construct a much smaller DAG than those constructed by the existing algorithms.** We use the proposed branch and bound strategy to identify the non-contributed point and non-longest paths, and do not include them in DAG. As a result, the constructed DAG will be much smaller than the existing ones and is called Small-DAG.
- **Design a strategy for deleting points in the Hash table timely.** The establishment of the Hash table is to prevent the same points from appearing repeatedly in Small-DAG. Also, if we can judge that a point is unlikely to appear in subsequent searches, we can safely delete it from the Hash table without any impact on the final result, thereby reducing the scale of Hash table and memory usage.
- **Propose a new data structure for storing Small-DAG to avoid topological sorting.** In constructing Small-DAG, we only store the longest path, and do not store non-longest paths in Small-DAG. Therefore, in Small-DAG, all the paths from the end point to the start point are the longest paths. Both time and space costs are smaller than those in the existing DAGs including ICSG.
- **Propose a new algorithm for larger-scale MLCS problems with lower time and space cost.** We propose a branch and bound irredundant graph MLCS algorithm called Big-MLCS for large-scale MLCS problems and make the comparison with the state-of-the-art algorithms. The results show that our algorithm performs better than the counterparts and is more suitable to large-scale MLCS problems.

The rest of this paper is organized as follows. Section 2 introduces some preliminaries and makes an overview of the related works, mainly on the dominant point based algorithms including TOP-MLCS. Section 3 describes the proposed algorithm Big-MLCS in detail. Section 4 analyzes the performance of our algorithm in solving large-scale MLCS problems and compares it with the state-of-the-art algorithms. Finally, we draw a conclusion in Section 5.

2. Preliminaries and related work

2.1. Notations and definitions

In this Section, we will give some common notations and definitions first. Then, we will introduce the dominant point based algorithms including the algorithm based on ICSG with some examples.

Definition 1. Let Σ be a alphabet set ($\Sigma = \{A, C, G, T\}$ for a DNA sequence), $S = c_1c_2 \cdots c_n$ ($c_i \in \Sigma, 1 \le i \le n$) be a sequence on alphabet set Σ, and $|S|$ be the length of the sequence. If S' is a sequence formed by deleting non or some character(s) from the sequence S, we call S' a subsequence of S.

For example, if we delete all the characters A from sequence $S = ACGGTGA$, sequence $S' = CGGTG$ is a subsequence of sequence S.

Definition 2. For two sequences S_1 and S_2, if S' is a subsequence of both sequence S_1 and sequence S_2, then S' is called the common subsequence of S_1 and S_2. Among all common subsequences, a longest one is called a longest common subsequence of S_1 and S_2. The number of longest common subsequences is not necessarily only one. For example, for sequences $S_1 = ACGGTAGA$ and $S_2 = TACGAGTC$, both $S'_1 = ACGGT$ and $S'_2 = ACGAG$ are their longest common subsequences.

Definition 3. The problem of finding all the longest common subsequence of a given number of d ($d \geq 3$) sequences is called MLCS problem.

As mentioned earlier, dominant point based methods are the current most efficient methods for solving MLCS problems. In the following sections, we will introduce these approaches.

2.2. Dominant point based approaches

The reason why the algorithms based on dominant points can solve the MLCS problem more efficiently is that they reduce the search space through the non-dominated sorting, thereby reducing the size of the directed acyclic graph (DAG). In order to facilitate our understanding of the core idea of the algorithms, we will give some definitions first, and then illustrate the process of constructing directed acyclic graph (DAG) by a simple example.

Definition 4. For a set of sequences $I = \{S_1, \cdots, S_i, \cdots, S_d\}$ on alphabet set Σ, we use $S_i[j]$ to represent the jth character of the sequence S_i ($S_i[j] \in \Sigma$, $1 \leq i \leq d$, $1 \leq j \leq |S_i|$). If $S_1[c_1] = \cdots = S_d[c_d] = \alpha \in \Sigma$, we call point $p = (c_1, c_2, \cdots, c_d)$ a match point of α.

For example, for sequences: $S_1 = ACGGTAGA$
$S_2 = TACGAGTC$
$S_3 = TCGAGTAC$
$p = (1, 2, 4)$ is a match point of character A.

Definition 5. Given a set of sequences $I = \{S_1, \cdots, S_i, \cdots, S_d\}$ on alphabet set Σ. For two matching points $p = (p_1, p_2, \cdots, p_d)$ and $q = (q_1, q_2, \cdots, q_d)$, we define:

1. $p = q$ if $\forall i (1 \leq i \leq d), p_i = q_i$.
2. p weakly dominates q, if $\forall i (1 \leq i \leq d), p_i \leq q_i$ and $\exists i, p_i < q_i$ (denoted by $p \preceq q$).
3. p dominates q or q is dominated by p, if $\forall i (1 \leq i \leq d), p_i < q_i$ (denoted by $p \prec q$).
4. q is called a successor of p if $p \prec q$. Further, if there is no match point r to satisfy $p \prec r \prec q$, then q is called an immediate successor of p.
5. If q is a successor of p, we call p a predecessor of q.

Definition 6. Given a set of match points $P = \{P_1, P_2, \cdots, P_n\}$, if a match point P_i ($1 \leq i \leq n$) is not dominated by any and other match points in set $P = \{P_1, P_2, \cdots, P_n\}$, P_i is a non-dominated point of set P. If there is a point P_j which dominates P_i, P_i is called a dominated point.

The principle of the dominant point based algorithms is based on the following observation: in the set of all successor nodes, only non-dominated points are likely on the longest path (corresponding to MLCS) on the DAG, so one can only keep non-dominated points by non-dominated sorting, and delete the dominated points from the set, thus reducing the search space and making DAG constructed much smaller.

For d given sequences, we first create a d dimensional initial node (also called source node) $O = (0, 0, \cdots, 0)$ and a d dimensional infinite node (also called end node) $E = (\infty, \infty, \cdots, \infty)$, and set the level of the source node to 0 in the initialization phase. Afterward, we look for all successor nodes of the source node, and use the non-dominated sorting method to find the non-dominated points in the set of these successors, delete the dominated nodes, draw an edge from the original (initial) node to the remaining successors (each remaining successor is used as a node), and set the level of these remaining successors to 1. For each node in level 1, we look for its all successors. All successors of all nodes in level 1 form a set of successors of level 1. We use the non-dominant sorting method to find the non-dominated points in the set of successors of level 1, and delete the all dominated points. The remaining successors in the set of successors of level 1 are the nodes in level 2. Draw an edge from any node in level 1 to its remaining successor in level 2. We can construct nodes in the followed levels by repeating the above process until any node in a level has no successor, and then draw an edge from each node in this level to the end point. This finishes the construction of the directed acyclic graph (DAG). Each longest path in the DAG graph is corresponding to a MLCS.

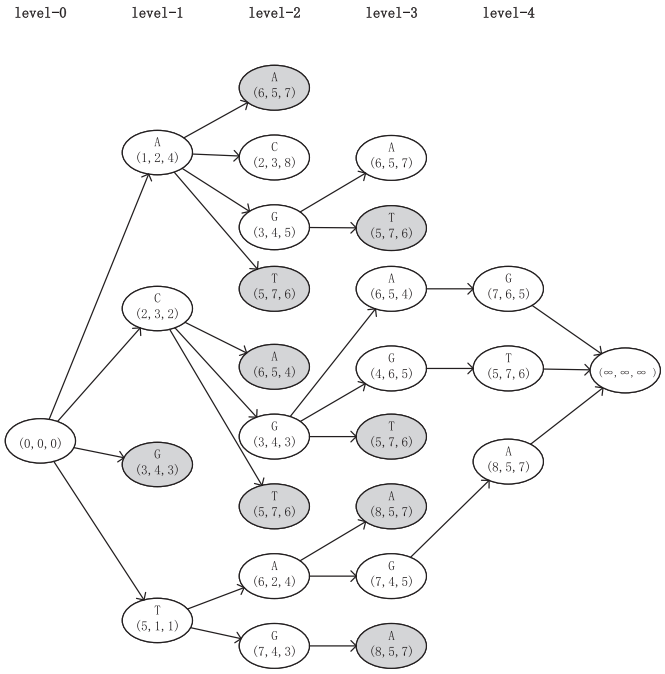


Fig. 1. Construction process of the DAG by the dominant point based algorithm.

We will use an example (the example given in Definition 4) to illustrate the framework of the aforementioned dominant point based algorithm.

Example 1. Given three sequences:
$S_1 = ACGGTAGA$
$S_2 = TACGAGTC$
$S_3 = TCGAGTAC$
Figure 1
shows the details of the construction process. That is:

1. Create the original node and infinite node, and then set the level of the original node to 0.
2. Find the successors of the original node, here we can find $A(1, 2, 4)$, $C(2, 3, 2)$, $G(3, 4, 3)$, and $T(5, 1, 1)$. After non-dominated sorting on successor nodes, we can find that $G(3, 4, 3)$ is dominated by $C(2, 3, 2)$. So we delete the successor $G(3, 4, 3)$ (marked in gray), draw an edge from the original point to its three remaining successor nodes, and mark the level of the remaining successor nodes as level 1.
3. For each node of all nodes $A(1, 2, 4)$, $C(2, 3, 2)$ and $T(5, 1, 1)$ in level 1, e.g., for node $A(1, 2, 4)$, find all its successors $A(6, 5, 7)$, $C(2, 3, 8)$, $G(3, 4, 5)$ and $T(5, 7, 6)$. Similarly, we can find the successors of nodes $C(2, 3, 2)$ and $T(5, 1, 1)$. After using non-dominant sorting method, we can delete 4 dominated points (the successors $A(6, 5, 7)$ and $T(5, 7, 6)$ of $A(1, 2, 4)$, and the successors $A(6, 5, 4)$ and $T(5, 7, 6)$ of $C(2, 3, 2)$) in gray and keep 5 non-dominated points ($C(2, 3, 8)$, $G(3, 4, 5)$, $G(3, 4, 3)$, $A(6, 2, 4)$, $G(7, 4, 3)$) in white. Set the level of these 5 non-dominated points in white to 2.
4. Repeat Step 3 until all the nodes in the level 4 have no successor nodes. At this time, we draw an edge from each node

in level 4 to end node. The construction of the entire directed acyclic graph (DAG) is completed.
5. All the paths from the original node to the end node are all longest common subsequences. In this example, there are three longest common subsequences: $S'_1 = CGAG$, $S'_2 = CGGT$, $S'_3 = TAGA$

Although the dominant point based algorithm described above is more efficient than the dynamic programming based algorithms, there are two main shortcomings below:

- Repeatedly appeared points waste a lot of space and time. For example, as shown in Fig. 1, point $G(3, 4, 3)$ in the first level appears again in the second level. Similarly, point $T(5, 7, 6)$ appears two times in the second level and also appears two times in the third level and one time in the fourth level. It appears repeatedly 5 times. Obviously, these repeated points needs to be stored repeatedly and deleted repeatedly (if they repeat more than two times and are dominated), which wastes a lot of space and time.
- The non-dominated sorting on the successor nodes created in each level needs huge time consumption. In fact, it needs comparison of each pair of high dimensional points in each level, and the comparison between one pair of points needs d comparisons of two real numbers if there are d sequences (in the example, $d = 3$). With the increase of the problem scale (the number of sequences and the length of sequences), the number of nodes in each level will increase exponentially, which will results in huge time consumption.

Unfortunately, with the increase of the length and number of sequences, the time and space consumptions of algorithms based on dominant point are very large. Hence, these algorithms are difficult to deal with large-scale MLCS problems. Recently, Li et al. [24] put forwards the Top-MLCS algorithm. Its most notable innovation is to construct a much smaller DAG, i.e., ICSG, without redundant nodes and cleverly avoid the non-dominated sorting among successors. After ICSG is constructed, we can get the MLCS through topological sorting. In the next section, we will introduce the framework of Top-MLCS algorithm.

2.3. Top-MLCS algorithm

In order to easily understand the Top-MLCS algorithm, we introduce the definition of the in-degree of nodes in graph theory first.

Definition 7. For a node in a direct graph, the sum of all edges to this node from other nodes is called the in-degree of the node.

For example, in Fig. 1, the in-degree of the original node $O = (0, 0, \cdots, 0)$ is 0 because no edge to it, and the in-degree of the infinite node $E = (\infty, \infty, \cdots, \infty)$ is 3 because there are three edges to it.

The framework of the Top-MLCS algorithm is mainly composed of the following three parts:

2.3.1. Construction of ICSG

One of the most important techniques of Top-MLCS algorithm is constructing ICSG. In the process of constructing the ICSG, in order to avoid redundant nodes in the graph, one needs to store the information of the constructed points by a Hash table. When a new successor q of a node p appears, one has to judge whether this successor q has been created in the ICSG. If it has not been created, it will be added as a new node in the ISCG, and an edge from the predecessor p to its successor q is drawn. This predecessor-successor relation information is stored in the Hash table. If it has already been created, it needs not to be added to the ICSG. One only has to draw an edge from p to it.

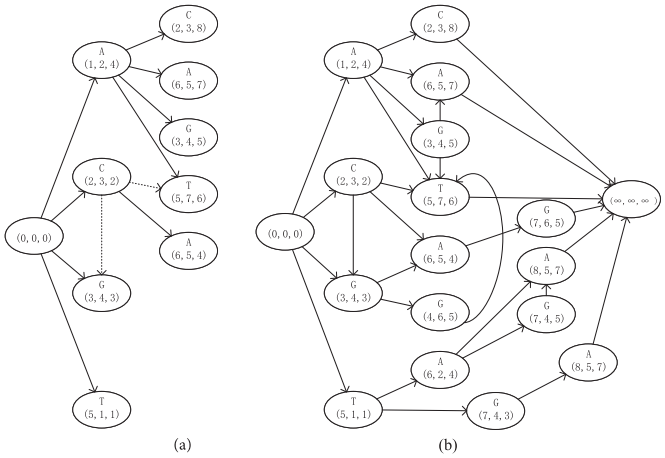


Fig. 2. Construction of ICSG, where (a) shows the process of building ICSG, and (b) shows the constructed ICSG.

Let us use the example in Definition 4 to show the process of ICSG construction. Given three sequences:

$S_1 = ACGGTAGA$
$S_2 = TACGAGTC$
$S_3 = TCGAGTAC$

1. Create the original node, and let infinite node denote the end node.
2. Find successors of the original node, and we can find four successors $A(1, 2, 4)$, $C(2, 3, 2)$, $G(3, 4, 3)$,and $T(5, 1, 1)$. Draw the edges from the original point to the successor nodes, and put them all in the Hash table.
3. For each new generated node, Find its all successors. For example, for node $A(1, 2, 4)$, find its all successors $C(2, 3, 8)$, $A(6, 5, 7)$, $G(3, 4, 5)$ and $T(5, 7, 6)$. Identify whether these four successors have been created in ICSG (check whether the successor exists in hash table). Because these four successors have not been created, we connect the edges from $A(1, 2, 4)$ to each of them, and store all the successors into the Hash table. For node $C(2, 3, 2)$, find its all successors $T(5, 7, 6)$, $A(6, 5, 4)$ and $G(3, 4, 3)$. Because $T(5, 7, 6)$ and $G(3, 4, 3)$ have been created in the ICSG, we only need to draw edges from $C(2, 3, 2)$ to each of these two successors. While $A(6, 5, 4)$ has not been created, we draw an edge from $C(2, 3, 2)$ to its this successor $A(6, 5, 4)$ and store $A(6, 5, 4)$ into the Hash table, as shown in Fig. 2(a).
4. Repeat above process until all new generated points in the graph have no successors. Draw an edge from each of the new generated points to the infinite node. So far, the entire ICSG is constructed, as shown in Fig. 2(b).

2.3.2. Forward topological sorting

Note that there are no redundant (repeated) nodes in ICSG and a large amount of time required for non-dominated sorting is avoided. However, the ICSG constructed has the following problem: The level of each node in the DAG constructed by other dominant point based algorithms is very clear, while the level of each node in ICSG is not clear or even a node in the ICSG may have no level because there may be more than one path from the source node to it (see Fig. 2b). This results in more difficult and more computation cost to find all longest paths from the source node to the end node. In order to determine the level of nodes in ICSG, Top-MLCS algorithm uses a topological sorting scheme as follows.

1. Initially, we set the level of the source node as 0. Then, starting from each node in the current level (only the source node in level 0), we delete all edges originating from each node in the current level (represented by dashed edges in Fig. 3(a)).

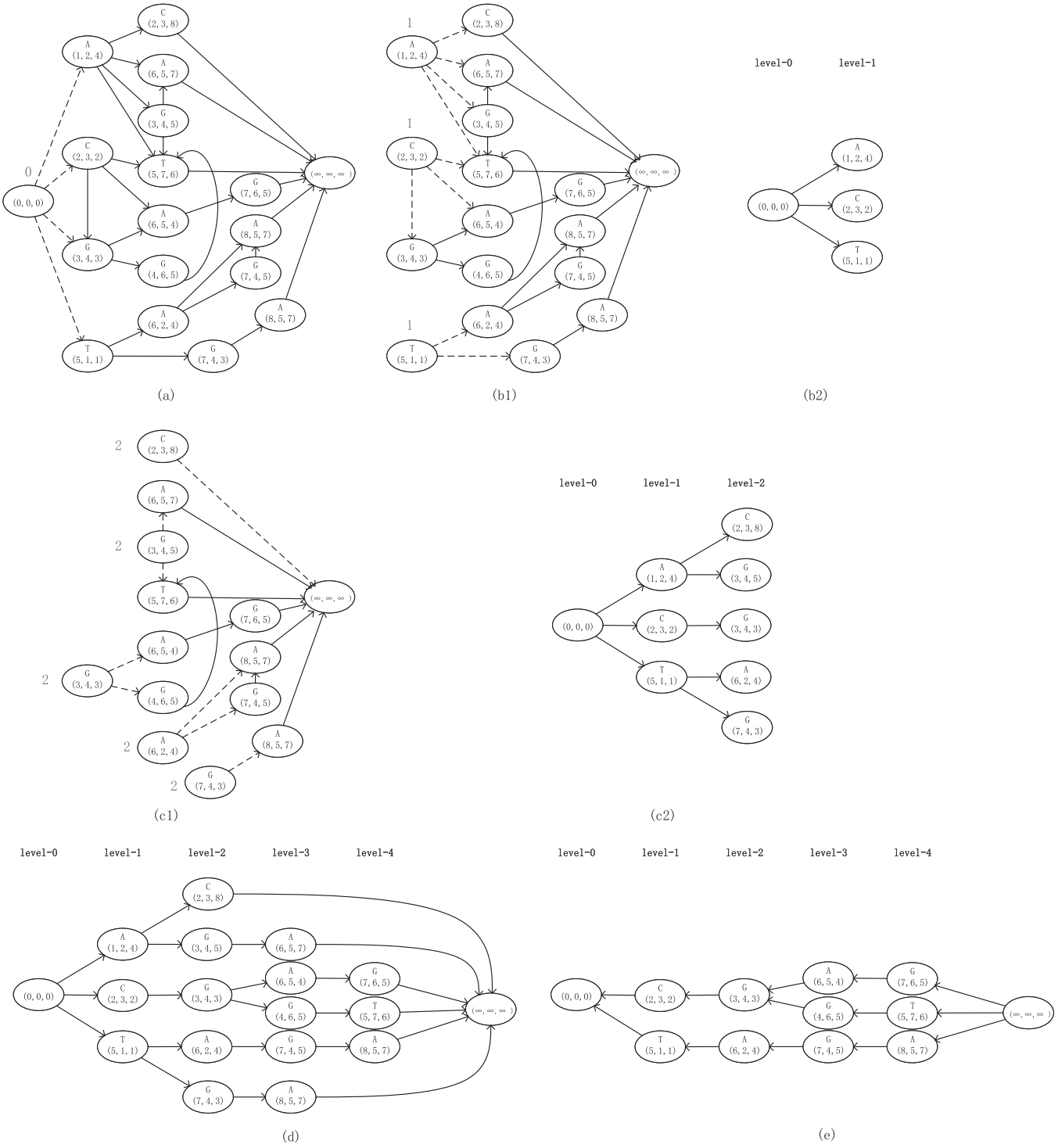


Fig. 3. An example to optimize ICSG using topological sorting, where (a), (b1), (b2), (c1), and(c2) show the process of forward topological sorting, (d) shows the forward topological sorted ICSG, and (e) shows the backward topological sorted ICSG.

Find all successor nodes of the each current level node (the source node in this example) in ICSG whose in-degrees are 0 ($A(1,2,4)$, $C(2,3,2)$ and $T(5,1,1)$). These nodes are the nodes in the next level (in level 1) as shown in Fig. 3(b1). Now we can get a part of the forward topological sorted ICSG, as shown in Fig. 3(b2).

2. Starting from each node in level 1, we delete all edges originating from the nodes in level 1 (represented by dashed edges in Fig. 3(b1)). Find all nodes in the ICSG with the in-degree being 0. Here we can find $C(2,3,8)$, $G(3,4,5)$, $G(3,4,3)$, $A(6,2,4)$ and $G(7,4,3)$. Set the level of these 5 nodes to 2, as shown in Fig. 3(c1). Our current forward topological sorted ICSG is expanded to Fig. 3(c2).
3. Repeat the above process by starting from each node (in Fig. 3(c1)) in level 2 until the levels of all nodes in ICSG are determined, and we can get a new ICSG with each node being defined a level by the forward topology sorting scheme, as shown in Fig. 3(d).

2.3.3. Backward topological sorting and analysis

When one gets the leveled ICSG shown in Fig. 3(d), one can get all MLCSs by a backward topological sorting as follows: Searching successively level by level from end node to source node. If a path does not go through any point in some level, then this path is not the longest path and it is not corresponding to any MLCS. Thus, this path can be deleted from ICSG in Fig. 3(d), which will result in a simplified ICSG as shown in Fig. 3(e). Each path from source node to end node in this simplified ICSG is the longest path and any MLCS is corresponding to the longest path from source node to end node in this simplified ICSG. Thus, finding all MLCS is transformed into searching all longest paths in this simplified ICSG.

However, Top-MLCS has the following disadvantages, illustrated by the aforementioned example:

- Although there are no redundant nodes in the graph, and we need not do non-dominated sorting for points of the same level, we have to use a lot space to store all these non-repeated nodes created and the relation of each pair of the predecessor and the successor of these non-repeated nodes. In this example, in order to avoid duplicate creation of the same points, we need to store 16 points of d dimensions in the hash table, where d is the number of sequences. As the number of sequences increases, the cost of storing these nodes will increase greatly.
- After ICSG with non-repeated nodes is constructed as shown in Fig. 3, one needs to set up the exact level of each point in this ICSG through forward topological sorting (the process shown from Fig. 3(a)–(d)), which will take a lot of time especially when the scale of this ICSG is large. In order to get the final solution, one has to further use the backward topological sorting to delete the non-longest path from leveled ICSG (shown in Fig. 3(d))to get the simplified ICSG (shown in Fig. 3(e)). This process is very time-consuming.
- Top-MLCS algorithm lacks branch and bound strategy in the process of constructing ICSG. We can see from the above example that not all points can form the longest path, such as the path composed of points $T(5,1,1)$, $G(7,4,3)$ and $A(8,5,7)$ (Fig. 3(d)), whose length is only 3, but computing resources and storage resources are still wasted on them.

To overcome these drawbacks, Section III will propose the Big-MLCS algorithm for MLCS Problems.

3. The Big-MLCS algorithm

We will introduce our strategy for deleting points in the Hash table timely in Section 3.1, a new data structure for storing Small-DAG in Section 3.2, and a branch and bound method for deleting no-contributed points and non-longest paths in Section 3.3.

3.1. Reduce storage space by deleting points in vector Hash tables timely

In order to introduce our algorithm more clearly, we will first give a definition of minimum value of a matching point.

Definition 8. For a match point $p = (p_1, p_2, \cdots, p_d)$, We call $m(p) = min\{p_1, p_2, \cdots, p_d\}$ the minimum value of p.

Note that there are no repeated nodes in the constructed ICSG in Top-MLCS, but one needs to store all the points that have been constructed into the Hash table. However, with the increasing number of nodes, the memory occupied by the Hash table will continually increase, and the efficiency of searching elements in the Hash table will decline as well. When the number of sequences (i.e., the dimension of nodes in Hash table) is large, the space consumption of Hash table will be even larger. Here, we design a new Hash Table strategy called Vector Hash Table (VHT) by deleting the points in VHT timely, which has a significant reduction in the storage of original Hash table. The theoretical basis of this operation is that, if the minimum value of the two match points are equal, they will not dominate each other, which means they will not become each other's successor node. For the example mentioned earlier, the minimum value of all components of both $A(1,2,4)$ and $T(5,1,1)$ is 1, so it is impossible for one to be a successor of the other.

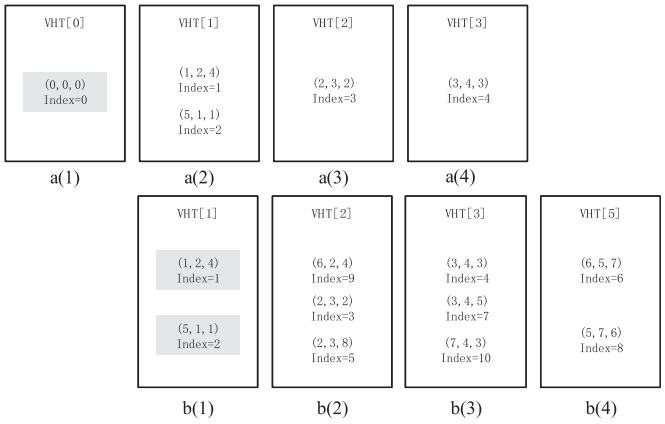


Fig. 4. Strategy for deleting non-contribution points in VHT.

Based on this observation, we store successors in different VHTs according to the minimum component value of a match point. We show the steps to implement this strategy in Algorithm 1 (We give each point a unique Index, which will be used in Section 3.2).

To easily understand this strategy, we will briefly explain the process of Algorithm 1 through the example mentioned earlier.

Algorithm 1 Strategy for deleting points in VHT timely

```
VHT[0] store {(0, 0, ··· , 0), Index = 0}
Index = Index + 1
for i = 0 to |S| do
    while VHT[i] ≠ ∅ do
        get p from VHT[i]
        for each k in Successor(p) do
            if k ∉ VHT[m(k)] then
                VHT[m(k)] Store {k, Index}
                Index = Index + 1
            end if
        end for
    end while
    Delete VHT[i]
end for
```

1. Store original node (0,0,0) in VHT[0], as shown in Fig. 4a(1).
2. Find its all successors. If the minimal component value of a successor is i, this successor is stored in the corresponding VHT[i]. In this example, it has four successors (1,2,4), (5,1,1), (2,3,2) and (3,4,3). Successors (1,2,4) and (5,1,1) have the minimal component value $i = 1$. Thus these two successors are stored in VHT[1] (Fig. 4). Similarly, successor (2,3,2) has the minimal component value $i = 2$ and will be stored in VHT[2] (Fig. 4). (3,4,3) has the minimal component value $i = 3$ and will be stored in VHT[3] (Fig. 4a). Delete the previous VHT[0] shown in Fig. 4a(1) (marked in gray).
3. For nodes in the undeleted VHT[i] with the smallest i, find their all successors. For any successor of a node in VHT[i] with its minimal component value being j, put it in the VHT[j]. Delete VHT[i] after putting all successors of its all nodes. In this example, find the successor nodes of (1,2,4) and (5,1,1) in

VHT[1], store their successors in VHT[2]-[5] according to their minimum component values (Fig. 4b(2)–(4)), and then delete VHT[1] as shown in Fig. 4b(1).
4. For the undeleted VHT[j] with the smallest j, repeat the above process until all nodes in the undeleted VHT have no successors.

Using this VHT strategy, it is expected that we can delete a large number of nodes in VHT timely.

3.2. A new data structure for storing Small-DAG to avoid topological sorting

In the Top-MLCS algorithm, when the construction of ICSG is completed, the forward and backward topological sorting schemes should be used to get all MLCSs, which is laborious. Note that in Section 3.1, we design VHT which maps each node to its index. In this section, we design a new data structure called Graph Hash Table (GHT) which can maps the index of each node to its position information (level, predecessor and corresponding character) in Small-DAG. By using GHT, we can easily and quickly know the level, predecessor and corresponding character of each node from its index and only need to store the following three kinds of data for constructing Small-DAG:

1. Character α ($\alpha \in \Sigma$).
2. Predecessors of any node in Small-DAG.
3. Level of the node in Small-DAG.

To build Small-DAG, we need to update the position information: the predecessor and level of each node. The method is given in Algorithm 2.

Algorithm 2 Method of updating information of nodes.

```
get p and Index of p from VHT[i]
get Level p from GHT[Index of p]
for each k in Successor(p) do
    if k ∉ VHT[m(k)] then
        VHT[m(k)] Store {k, Index of k}
        Predecessor of k = Index of p
        Levelk = Levelp + 1
        GHT Store {Index of k , k}
    else
        get Index of k fromVHT[m(k)]
        get Levelk from GHT[Index of k]
        if Levelp + 1 > Levelk then
            Predecessor of k = Index of p
            Level of k = Levelp + 1
            GHT Store {Index of k , k}
        end if
        if Levelp + 1 = Levelk then
            Predecessor of k+ = Index of p
            GHT Store {Index of k , k}
        end if
        if Levelp + 1 < Levelk then
            DoNothing
        end if
    end if
end for
```

In order to better understand the updating process, we will briefly explain it through Fig. 5.

When we look for the successors of p, if successor k has not been created in the graph, we will directly create it, temporarily mark its level as $Level p + 1$, and connect an edge from k to p (Fig. 5(a)).

If k has already been created, it means that there are at least one precursor q of k and a path from q to k. In this case, in order to ensure that we only record the longest path, we need to compare the length of the new path with the existing one(s). There are three cases here:

1. If $levelp + 1 > levelk$, the new path is longer than the existing one(s). W have to update the value of $levelk$ by $levelp + 1$ and delete the existing path to k. Connect an edge from k to p (Fig. 5(b)).
2. If $levelp + 1 = levelk$, the length of the new path is same as the existing one(s), then we only need to connect one edge from k to p without updating the level of k. (Fig. 5(c)).
3. If $levelp + 1 < levelk$, the new path is shorter. So we don't need any action (Fig. 5(d)).

We will continue to use the previous example to introduce how to construct and store Small-DAG by the strategies in Sections 3.1 and 3.2. For given sequences:

$S_1 = ACGGTAGA$
$S_2 = TACGAGTC$
$S_3 = TCGAGTAC$

1. Create the original node and infinite node. Put the initial node in VHT[0], mark the level of the initial node as 0, and the Index of the initial node as 0.
2. Find successors of the original node: $A(1,2,4)$, $C(2,3,2)$, $G(3,4,3)$,and $T(5,1,1)$. Their Indexes are 1, 2, 3, 4 respectively. Put $A(1,2,4)$ and $T(5,1,1)$ in VHT[1], $C(2,3,2)$ in VHT[2], and $G(3,4,3)$ in VHT[3], and then mark their level as 1. Connect the edges from them to the original node and delete VHT[0] (Fig. 6(a) and Fig. 4(a)).
3. For each node in VHT[1], find its successors. For $A(1,2,4)$, we can find its successors $C(2,3,8)$, $A(6,5,7)$, $G(3,4,5)$ and $T(5,7,6)$. Their Indexes are 5,6,7,8 respectively. Store them in the VHTs[2],[5],[3] and [5], respectively, and mark their levels as $levelA(1,2,4)+1=2$ because they have not been created, and then connect the edges from them to $A(1,2,4)$. Similarly, find the successors $A(6,2,4)$ and $G(7,4,3)$ of $T(5,1,1)$ in VHT[1], put them in VHTs[2] and [3], respectively. Mark their levels as $levelT(5,1,1)+1=2$ because they have not been created, and then connect the edges from them to $T(5,1,1)$. Delete VHT[1] (Fig. 6(b)).
4. For nodes $C(2,3,2)$, $A(6,2,4)$ and $C(2,3,8)$ in VHT[2]. Find all successors $T(3,4,3)$, $T(5,7,6)$ and $A(6,5,4)$ of $C(2,3,2)$ first. Note that $T(5,7,6)$ already exists, and its current level 2 is equal to $levelC(2,3,2)+1$, so we only need to connect one edge from $T(5,7,6)$ to $C(2,3,2)$. Similarly, $T(3,4,3)$ already exists, but its level $1 < levelC(2,3,2)+1=2$, so we update its level to 2, and delete the edge from it to its previous precursor $O(0,0,0)$. $A(6,5,4)$ has not been created yet, so we store it in VHT[4], mark its level as $levelC(2,3,2)+1=2$, and connect the edge from it to $C(2,3,2)$. Then we find the successors of $C(2,3,8)$ and $A(6,2,4)$. Since $C(2,3,8)$ has no successor, we connect an edge from infinite node to it, and temporarily mark the level of infinite node as 3. $A(6,2,4)$ has successors $G(7,4,5)$ and $A(8,5,7)$. Since these two successors are new created ones, let their level is equal to $levelA(6,2,4)+1=3$ and put them in VHTs[4] and [5]. At this point, we have processed all nodes in VHT[2], so we can delete VHT[2] (Fig. 6(c)).
5. For nodes in VHT[3], VHT[4], and so on, successively repeat above steps until all nodes have no successors. In the end we can get the result of Fig. 6(d).

By comparing the results of Fig. 6 and Fig. 3, we can find that our new strategies have the following advantages:

1) During the construction of Small-DAG, we delete useless information in Hash table and useless edges and only store the edges

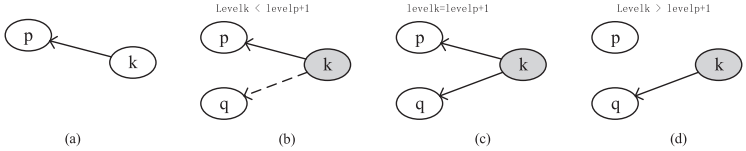
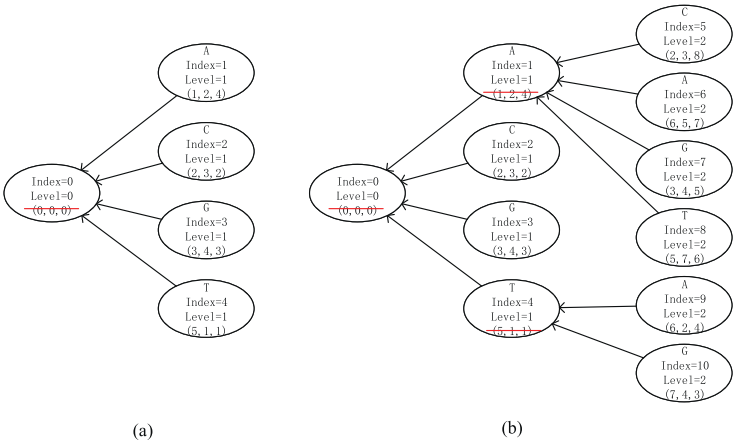
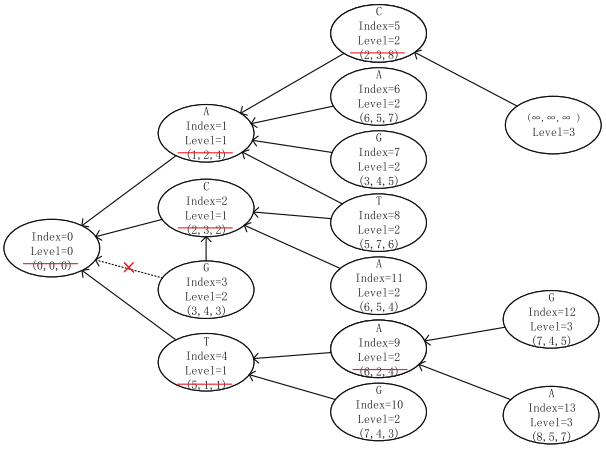


Fig. 5. Method of updating nodes' data.





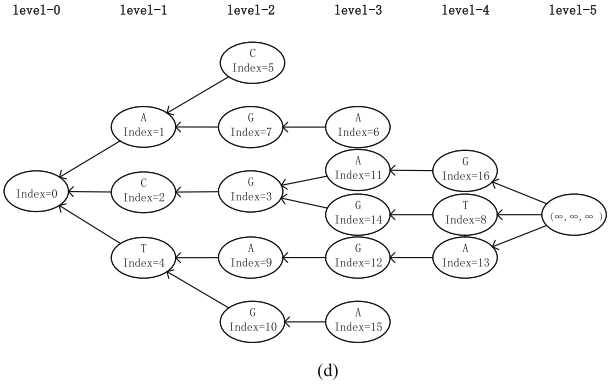


Fig. 6. Construct and store Small-DAG by the strategies in Sections 3.1 and 3.2.

on the longest paths, while during the construction of ICSG by Top-MLCS, only repeated nodes are avoided and ICSG still contains many useless edges. Also, Top-MLCS should store the d-dimensional vectors of all points, as shown in Fig. 3(a), but Small-DAG constructed by Big-MLCS dose not need to store any d-dimensional vectors, as shown in Fig. 6(d).

2) To get Fig. 3(d) from Fig. 3(a), Top-MLCS has to use the forward topology sorting, while Big-MLCS can directly get Fig. 6(d), which is similar to Fig. 3. But Fig. 3(d) has to store a d-dimensional vector on each node, while Fig. 6(d) needs not to do so.

3) To obtain all MLCS from Fig. 3, Top-MLCS should use the backward topology sorting form Fig. 3(d) to get Fig. 3(e), while Big-

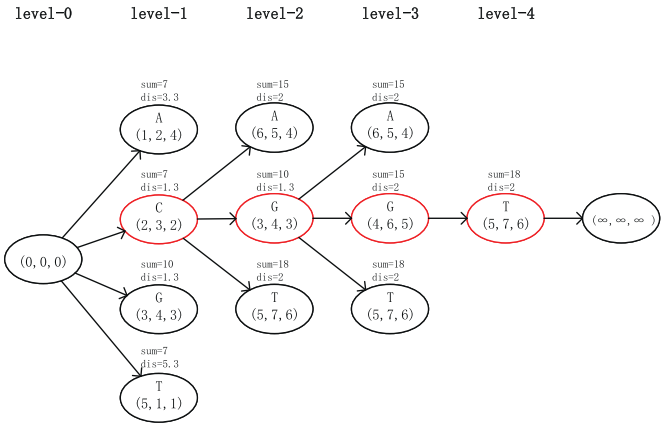


Fig. 7. Approximately estimate the lower bound of the length of MLCS.

MLCS can directly obtain all MLCS from the end node to the source node in Small-DAG. Thus, Big-MLCS dose not need the forward and backward topological sorting schemes. Therefore, the time cost of Top-MLCS is much higher than that of Big-MLCS.

To further reduce the space and time cost, we design a branch and bound method to delete non-contributed points and non-longest paths during the construction of Small-DAG.

3.3. Branch and bound method and Big-MLCS

Note that Small-DAG constructed in the previous subsections often contains non-contributed points and non-longest paths. Generating and storing them will consume a lot of computer resources. But they are worthless for searching MLCS. How to identify such paths including nodes on them is not an easy task. In this section, we will propose a branch and bound method which can identify whether a node is on the longest path. The method consists of two parts:

1. Find a true path as long as possible.
2. Judge whether a point is on the MLCS.

In order to find a true path as long as possible, we choose only one matching point which is most possible to be on the longest path in each level according to the following two criteria. For a matching point $p = (p_1, p_2, \cdots, p_d)$, we calculate:

$$sum = \sum_{i=1}^{d} p_i \tag{1}$$

$$dis = \sum_{i=1}^{d} \left| p_i - \frac{sum}{d} \right| \tag{2}$$

Among all matching points in each level, we pick up one matching point with minimum *sum* first. If there are more than one matching points with the minimal *sum*, we will choose the one with minimum *dis*. If there are more than one matching points with both minimal *sum* and minimal *dis*, we will randomly choose one matching point among them. In this way, we can choose one matching point in each level and these matching points from level 1 to the final level form a path. The length of this path is a temporary maximum lower bound of the length of the longest path. We call this as **Temporary Maximum Length**.

Let us briefly show this process based on the example mentioned earlier. As shown in Fig. 7, we calculate *sum* and *dis* of nodes in the first level respectively, and select $C(2, 3, 2)$(Its *sum* and *dis* attain the minimal value simultaneously), then continue to find its successors and compute their *sum* and *dis*. Select $G(3, 4, 3)$ (Its *sum* and *dis* attain the minimal value simultaneously) at level 2. Repeat the above operations. Finally, we can get a path ($CGGT$) with **Temporary Maximum Length** 4.

For estimating **Temporary Maximum Length**, when both *sum* and *dis* of more than one match points are same, we have to randomly choose one match point. But note that there are few cases for both *sum* and *dis* of more than one match points to be same, and even if there are some such cases, the experiments have shown that the impact of randomly choosing match points on **Temporary Maximum Length** is not large.

Suppose that the current matching point put on Small-DAG is $P = (p_1, p_2, \cdots, p_d)$. We can get a set of subsequences $\bar{I} = \{\bar{S}_1, \bar{S}_2, \cdots, \bar{S}_d\}$, where the ith subsequence $\bar{S}_i$ consists of the characters after the p_ith character in original sequence S_i. We estimate an upper bound of the length of path from the source node to the end node through node P as follows. First, we divide this path into two parts. The first part is the first half path from the source node to node P, and its length is $level_P$ which has been known. The second half path is from node P to the end node. We estimate an upper bound of the length of the second half path, marked as $UpperBoundtoEndNode_P$. Then we use their sum as an upper bound of the length of path from the source node to the end node through node P, expressed as $UpperBound_P$. So our estimation method can be formalized as:

$$UpperBound_P = level_P + UpperBoundtoEndNode_P \tag{3}$$

If this estimated upper bound is smaller than **Temporary Maximum Length**, then this path through P must not be on the MLCS and can be deleted.

To estimate an upper bound of the length of the second half path, note that the length of MLCS of $\bar{I}$ will not exceed the length of the LCS of any two subsequences in set $\bar{I}$. This means that we can roughly estimate an upper bound of the length of MLCS of $\bar{I}$ through the length of the LCS of two subsequences, and this upper bound of the length of MLCS of $\bar{I}$ can be seen as an upper bound of the second half path. However, there are $C_d^2 = \frac{d(d-1)}{2}$ cases for selecting two sequences from set $\bar{I}$. How to choose two sequences from $\bar{I}$? Since the length of the LCS of two sequences is an upper bound of the length of MLCS, it is better to choose the two sequences such that the length of their LCS is as small as possible. For this purpose, we select the shortest sequence in set $\bar{I}$ as one sequence marked as S_* (if there are multiple sequences with the shortest length, then choose one randomly). As for another sequence, it is better to choose one which is as most different as possible to the first one. To do so, we have to give a metric to measure the difference between two sequences. We call this metric as diversity metric.

We let $N_{\bar{S}_i}^{\alpha}$ $(a \in \Sigma)$ represent the number of the character α in sequence $\bar{S}_i$, and $len\bar{S}_i$ denote the length of the sequence $\bar{S}_i$. Note that the larger the value of $(N_{S_*}^{\alpha} - N_{\bar{S}_i}^{\alpha})$, the bigger the diversity between S_* and $\bar{S}_i$, and the bigger the difference between these two sequences. Also, the fewer the character α contained in S_*, the fewer the match points it can form with $\bar{S}_i$ than other character in S_*, and thus smaller contribution to the diversity. By considering these two factors, the diversity between S_* and $\bar{S}_i$ is defined as:

$$diversity(S_*, \bar{S}_i) = \sum_{\alpha \in \Sigma} \frac{N_{S_*}^{\alpha}}{lenS_*} (N_{S_*}^{\alpha} - N_{\bar{S}_i}^{\alpha}) \tag{4}$$

Based on the diversity metric, we can choose the second sequence having the greater diversity with sequence S_*. We use the following example to explain the calculation process of diversity. For given sequences

$\bar{S}_1 = AGATAT$

$\bar{S}_2 = CCTAGCC$

$\bar{S}_3 = AATTCGC$ since the length of $\bar{S}_1$ is 6, which is less than the length of the other two, we take $S_* = \bar{S}_1$. And then we count the number of each character in these sequences follows

$N^A_{S_*} = 3,\ N^C_{S_*} = 0,\ N^G_{S_*} = 1,\ N^T_{S_*} = 2$

$N^A_{\bar{S}_2} = 1,\ N^C_{\bar{S}_2} = 4,\ N^G_{\bar{S}_2} = 1,\ N^T_{\bar{S}_2} = 1$

$N^A_{\bar{S}_3} = 2,\ N^C_{\bar{S}_3} = 2,\ N^G_{\bar{S}_3} = 1,\ N^T_{\bar{S}_3} = 2$

$diversity(S_*, \bar{S}_2) = \frac{3}{6}(3-1) + \frac{0}{6}(0-4) + \frac{1}{6}(1-1) + \frac{2}{6}(2-1) = \frac{4}{3}$

$diversity(S_*, \bar{S}_3) = \frac{3}{6}(3-2) + \frac{0}{6}(0-2) + \frac{1}{6}(1-1) + \frac{2}{6}(2-2) = \frac{1}{2}$

So we choose the length of LCS between $\bar{S}_1$ and $\bar{S}_2$ to approximately estimate the upper bound of the length of MLCS of these three sequences (also the upper bound of the length of the second half path). We use the dynamic programming algorithm mentioned earlier to calculate the LCS length of these two sequences quickly, and give the analysis of the time complexity in Section IV.

As mentioned before, for the current matching point P, we have to identify whether P is not on the MLCS. We will first compare the upper bound of the length of the path through it with **Temporary Maximum Length**. If the upper bound is smaller than **Temporary Maximum Length**, then P is not on the MLCS and can be deleted directly in Small-DAG. We use the previous example to illustrate the branch and bound strategy in more detail. For given sequences:

$S_1 = ACGG\mathbf{T}AGA$

$S_2 = \mathbf{T}ACGAGTC$

$S_3 = \mathbf{T}CGAGTAC$

Suppose that the current matching point is $P = T(5, 1, 1)$. The sequences $\bar{I} = \{\bar{S}_1, \bar{S}_2, \bar{S}_3\}$ after P are:

$\bar{S}_1 = AGA$

$\bar{S}_2 = ACGAGTC$

$\bar{S}_3 = CGAGTAC$

We compute the upper bound of the length of MLCS of $\bar{I}$ which is 3 and the length of the first half path which is $levelT(5, 1, 1) = 1$. Since

$levelT(5, 1, 1) + 3 = 4 >= \textbf{Temporary Maximum Length}$,

we cannot identify $P = T(5, 1, 1)$ is not on the MLCS and need to put P on Small-DAG as a node and search its successors.

But for another matching point $P = G(7, 4, 3)$, the sequences $\bar{I} = \{\bar{S}_1, \bar{S}_2, \bar{S}_3\}$ after P are:

$\bar{S}_1 = A$

$\bar{S}_2 = AGTC$

$\bar{S}_3 = AGTAC$

We can calculate the upper bound of MLCS of $\bar{I}$ which is 1 and the length of the first half path which is $levelG(7, 4, 3) = 2$. Since

$levelG(7, 4, 3) + 1 = 3 < \textbf{Temporary Maximum Length}$,

we can identify that $P = G(7, 4, 3)$ is a non-contributed point and needs not to be put on Small-DAG as a node. The framework of our proposed Big-MLCS is shown in Algorithm 3.

In general, the Small-DAG built by the Big-MLCS algorithm has a significant reduction in scale compared to ICSG. There are two main reasons for this: 1) In the process of building Small-DAG, we delete the high-dimensional of nodes through VHT strategy, and record Small-DAG with GHT with less space cost. 2) We delete a large number of nodes that are not on the longest path (non-contribution points) through branch and bound strategy, so as to reduce the search space and further reduce the size of Small-DAG.

Algorithm 3 Big-MLCS.

```
Estimate Temporary Maximum Length
for i = 0 to |S| do
    while VHT[i] ≠ ∅ do
        get p and Index of p from VHT[i]
        get Level p from GHT[Index of p]
        Upperbound_p ← estimate the upper bound through p
        if Upperbound_p ≥ Temporary Maximum Length then
            Find Successor(p) and update related information
            if Level p + 1 > Temporary Maximum Length then
                Temporary Maximum Length = Level p + 1
            end if
        end if
    end while
    Delete VHT[i]
end for
```

4. Time and space complexity

The time and space complexity of the proposed algorithm. First, We mark the length of the sequences with n and the number of sequences with d. In the initialization, we built the Successor Table proposed in Fast-LCS to quickly find successor nodes of a point, and the time complexity of constructing Successor Table is $O(d|\Sigma|n)$ [21]. Second, we estimate the time cost to find the successor nodes and store them in the Vector Hash Table. We use V to represent the set of all points in Small-DAG, then the time complexity of this part will be $O(|V|)$. Finally, we use E to represent the set of all edges in Small-DAG, and time complexity of this part is $O(|E|)$. Although we need to find a path in order to estimate **Temporary Maximum Length**, since we only keep one node in each layer, the time complexity of this part will not exceed $O(d|\Sigma||MLCS|) \le O(d|\Sigma|n)$. The time complexity of estimating upper bound $UpperBound_p$ is no more than the time complexity of finding the LCS lengths of all possible two sequences by Dynamic Programming for two sequences. In fact, even if we calculate the LCS lengths of all possible two sequences, the time complexity of this part is only $O(n^2d^2)$. So the time complexity of the entire search process is no more than $O(d|\Sigma|n) + O(n^2d^2) + O(|V|) + O(|E|)$. Since $O(d|\Sigma|n) \ll O(n^2d^2)$ and $O(|V|) = O(|E|)$, thus, the time complexity of Big-MLCS is $\max\{O(n^2d^2), O(|V|)\}$. Because $O(n^2d^2) < O(|V|)$, so the time complexity of our proposed algorithm is $O(|V|)$.

For the compared algorithms, the time complexity of Quick-DPPAR is $O\left(d(\log n)^{d-2}|V_1|\right)$ [22], where V_1 is the set of nodes in the DAG constructed by Quick-DPPAR, and the time complexity of Top-MLCS is $O(|V_2|)$ [24], where V_2 is the set of nodes in the ICSG constructed by Top-MLCS. It should note that, due to the lack of a reasonable scheme to reduce the search space, the DAG constructed by Top-MLCS is much larger than those constructed by Quick-DPPAR and Big-MLCS, and DAG constructed by Quick-DPPAR is larger than that constructed by Big-MLCS, i.e., $|V_2| \gg |V_1| > |V|$. But Quick-DPPAR uses the time-consuming non-dominated sorting method to reduce the search space, so $O\left(d(\log n)^{d-2}|V_1|\right) > O(|V_2|)$ [24]. Thus, $O\left(d(\log n)^{d-2}|V_1|\right) > O(|V_2|) \gg O(|V|)$.

The space complexity of Big-MLCS. The space consumed by Successor Table is $O(d|\Sigma|n)$, the space consumed by storage points is no more than $O(d|V|)$ (It should be noted here that since we delete the nodes in the Hash Table during the search, our actual memory consumption depends on the maximum number of nodes stored in the Hash Table which is no more than $|V|$), and the space consumed by storage edge is $O(|E|)$. Since $O(d|\Sigma|n) \ll O(d|V|) + O(|E|)$, the space complexity of Big-MLCS is $O(d|V|)$. The space complexity of Quick-DPPAR and Top-MLCS can be expressed as $O(d|V_1|)$ [22] and $O(d|V_2|)$ [24], respectively. For $|V_2| \gg |V_1| > |V|$, we can deduce that our Big-MLCS algorithm has lower space complexity than two compared algorithms due to the use of the VHT and GHT strategies as well as the branch and bound strategy.

5. Experiments and analysis

5.1. Experimental environment, data set and compared algorithms

In the experiments, all algorithms were run on a cloud server. The hardware of the cloud server is: inter (R) Xeon (R) Gold 5115 CPU (2.40GHz), and the maximum memory usage of each user is limited to 128G. The data set comes from the widely used well-known data set NCBI (Available: www.ncbi.nlm.nih.gov). This is a real DNA data set.

The compared algorithms are two state-of-the-art excellent performance Dominant-point based algorithms: Quick-DPPAR and Top-MLCS. In Section B, we will first show the effectiveness of the strategy of deleting nodes in hash table and the proposed Big-MLCS in the form of a line graph. Then we will test the performance of Big-MLCS by two kinds of experiments commonly used in the experiments of the existing algorithms: (1) Fix sequence length to 85, increasing the number of sequences from 5 to 30000. (2) Fix sequence number to 5, increasing the length of sequences from 50 to 470. In order to ensure the fairness of the experiments, we randomly selected 10 groups of samples for each experiment, and compared their average time and memory overhead. The test examples for different algorithms are exactly the same.

5.2. Results and analysis

To test the effectiveness of the strategy of deleting the nodes in Hash table and Big-MLCS, we do the following experiments: We randomly choose 10 groups of samples with the number of sequences being 5 and the length of sequences being 100. We compare the number of nodes used in the Hash table in three strategies. Strategy 1: the strategy which does not use any deletion (i.e.,does not delete any point in Hash Table). Strategy 2: the strategy which uses only the strategies in Section 3.1. Strategy 3: the proposed strategy which combine the strategies in Sections 3.1 and 3.3.

We can see that when Strategy 1 is used, we need to store information of about 1.6 million nodes, and when strategy 2 is used, the number of nodes we have to store will increase first and then gradually decrease to 0. Also, Strategy 2 only needs to store at most about 200000 nodes, only about 12.5% of storage of Strategy 1. However, Strategy 3 only needs to store about 32,000 nodes at most, only about 2% of storage of strategy 1 (Fig. 8). These results indicate that the strategy for deleting points in Section 3.1 and the branch and bound algorithm in Section 3.3 are very efficient.

In the following, we will compare the performance of our algorithm Big-MLCS with Quick-DPPAR and Top-MLCS by conducting two kinds of experiments. The results are summarized in Tables 1 and 2, where '$*$' indicates that the algorithm cannot get results in limited memory (128G), ' + ' indicates that the algorithm cannot get results in our limited time of 6000 seconds.

From Table 1, we can see that when the number of sequences is fixed at 5, as the sequence length increases, the time for all algorithms to solve the MLCS problem will increase as well. When the length of the sequence is less than 120, all three algorithms can get MLCS in a short time. The slowest one is Quick-DPPAR, and the time required by the Top-MLCS algorithm is about 1.5 to 16 times of that required by the proposed algorithm Big-MLCS. When sequence length is between 120 and 200, Quick-DPPAR is obviously cannot get the final result in the given time limit, while Top-MLCS and Big-MLCS can still get the final results. But the time used by Top-MLCS is about 18 to 29 times of that used by the proposed algorithm Big-MLCS. When the sequence length increases to more than 200, Both Quick-MLCS and Top-MLCS algorithm can not solve the problem in the 128G memory space or the given limited time, while the proposed algorithm Big-MLCS can still get the MLCS until the length of sequences reaches 470 in the given space and time limits. This indicates Big-MLCS can solve the much longer MLCS problems in the same environment. The length of the MLCS problems solved by Big-MLCS is more than 2 times of that by Top-MLCS and more than 3 times of that by Quick-MLCS.

For problems with the fixed number of sequences, it can also be seen that the longer the sequences, the more difficult the MLCS problems, and the more time and space an algorithm will consume.

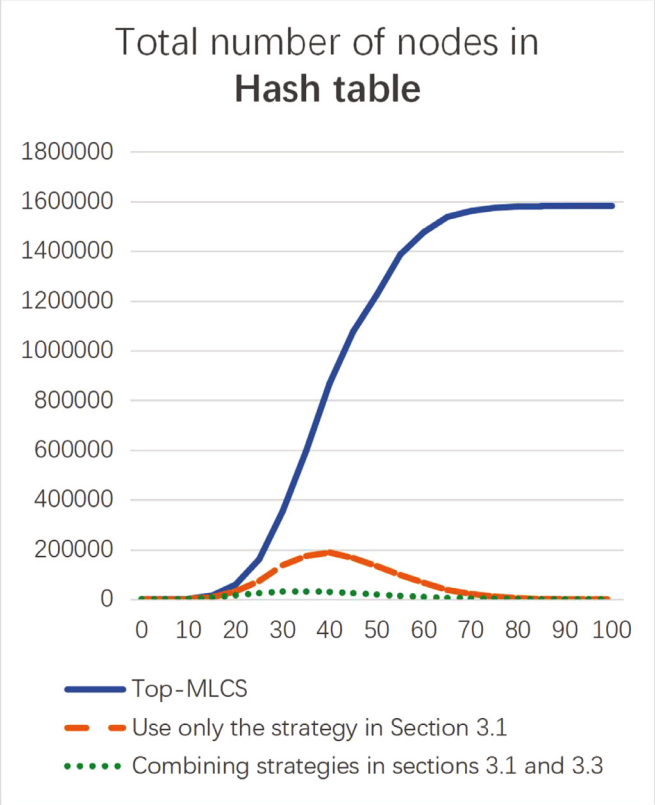


Fig. 8. Total number of nodes in Hash table.

Table 1
The Running Time (in second) of Quick-DPPAR (R1), Top-MLCS (R2), and Big-MLCS (R3) on 5 Sequences with Various Lengths.

Length of sequences	DNA ($\|\Sigma\| = 4$A)		
	R1	R2	R3
50	0.361	0.116	0.084
75	19.431	1.085	0.181
90	77.187	3.052	0.334
100	147.141	5.667	0.822
120	1980.077	26.434	1.648
140	+	82.681	2.855
160	+	188.481	5.032
180	+	369.641	12.749
200	+	781.881	42.952
220	+	*	57.351
250	+	*	123.994
270	+	*	191.525
300	+	*	358.516
320	+	*	539.699
350	+	*	954.201
370	+	*	1331.873
400	+	*	2412.356
420	+	*	2901.176
450	+	*	4578.232
470	+	*	5486.221

Table 2
The Running Time (in second) of Quick-DPPAR (R1), Top-MLCS (R2), and Big-MLCS (R3) for different number of Sequences with fixed length 85.

Number of sequences	DNA ($\|\Sigma\| = 4$)			
	$\|LCS\|$	R1	R2	R3
5	35	39.496	3.011	0.307
10	27	+	632.233	38.763
20	23	+	*	586.441
40	21	+	*	249.311
60	19	+	*	192.739
100	18	+	*	201.879
150	17	+	*	183.253
200	17	+	992.672	189.072
250	16	+	931.968	175.545
300	14	+	521.583	116.603
500	14	+	292.577	57.331
700	14	+	215.792	67.806
1000	13	+	349.783	103.431
3000	12	+	239.094	87.462
5000	11	+	244.049	94.342
10000	10	+	304.792	125.442
30000	9	+	352.388	153.451

Table 3
Total number of Points and Memory Overhead (in Gigabytes) in construction of ICSG by Top-MLCS (R2) and Small-DAG by Big-MLCS (R3) of D Sequences with Length 85.

Number of sequences	DNA ($\|\Sigma\| = 4$)			
	Points(R2)	Points(R3)	Memory(R2)	Memory(R3)
5	622333	89488	0.243	0.161
10	114680000	3620073	34.567	2.347
20	*	36210254	*	14.454
40	*	11985235	*	8.645
60	*	8650605	*	8.246
100	*	6557299	*	8.971
150	*	4374551	*	6.540
200	26551691	4175879	89.622	6.433
250	18198590	3177579	47.868	5.679
300	8633164	1922892	39.341	3.671
500	3127653	630890	16.435	2.542
700	1675006	706902	32.304	1.981
1000	1803448	715636	38.572	3.243
3000	324373	151484	20.185	2.134
5000	160634	79333	17.913	1.787
10000	74244	46410	20.541	2.033
30000	20342	13511	15.657	1.651

For the problems with the length of the sequence being fixed at 85, it can be seen from Table 2 that Quick-MLCS can only find the MLCS for one problem, i.e., the problem with 5 sequences in the given time and space limits and uses much more time than that used by Top-MLCs and Big-MLCS. While the time used by Top-MLCS is about 10 times of that used by Big-MLCS.

It can also be seen that for problems with at least 10 but less than 200 sequences, Top-MLCS can find the MLCs for only one problem, i.e., the problem with 10 sequence. But the time used by Top-MLCS is about 16 times of that used by Big-MLCS. For problems with more than 10 but less than 200 sequences, Top-MLCS cannot solve them, but Big-MLCS still can get the MLCS in a short time. When number of sequences increases from 200 to 30000, Quick-MLCS still cannot solve these problems, while Top-MLCS and Big-MLCS can. However, the time used by Top-MLCS is from about 2.3 to 5.3 times of that used by Big-MLCS.

Note that with the increase of the number of sequences, the overall trend of time used to solve the MLCS problem will first increase, and then gradually decrease. The reason for this phenomenon is that with the increase of the number of sequences to larger than a certain scale (In this case, when the number of aligned sequences is more than 100), the length of the final MLCS and the number of match points in graph will gradually decrease and the problem becomes less time consuming. This means that for the problems with the fixed length, it is not the case that the more sequences, the more difficulty of the MLCS problems. The most difficult MLCS problems for the fixed sequence length are with a middle number of sequences. For example, for the problems with the length of sequences fixed to 85, the most difficult problems are the problems with 20 to 200 sequences.

We have also recorded the total number of points and the memory overhead of graph ICSG constructed by Top-MLCS and those of graph Small-DAG constructed by the proposed algorithm Big-MLCS, respectively, and listed them in Table 3. We can see from Table 3 that for the problems with fixed length 85 and different number of sequences, when the number of sequences gradually increases, the total number of points and memory consumption of both Top-MLCS and Big-MLCS first increase and then decrease. However, because Big-MLCS uses the strategies in Sections 3.1 and 3.3, the total points and memory consumption of Small-DAG constructed by Big-MLCS are much smaller than those of ICSG constructed by Top-MLCS.

For problem with 5 sequences, the number of points in ICSG of Top-MLCS is about 7 times of that in Small-DAG and memory overhead of ICSG of Top-MLCS is about 2 times of that of Small-DAG. For problem with 10 sequences, the number of points in ICSG of Top-MLCS is about 31 times of that in Small-DAG and memory overhead of ICSG of Top-MLCS is about 17 times of that of Small-DAG. For problems with more than 10 but less than 700 sequences, the number of points in ICSG of Top-MLCS is about 4.7 to 6 times of that in Small-DAG and memory overhead of ICSG of Top-MLCS is about 7 to 14 times of that of Small-DAG. For problems with more than 700 but no more than 30000 sequences, the number of points in ICSG of Top-MLCS is about 2 times of that in Small-DAG and memory overhead of ICSG of Top-MLCS is about 10 to 16 times of that of Small-DAG.

6. Conclusion

In this paper, we have proposed a new fast and memory efficient algorithm called Big-MLCS for large-scale MLCS problems. By deleting points in Hash Table timely during the search process and application of new data structures, we have reduced the time cost by avoiding the topological sorting in Top-MLCS, and further reduce DAG and the space consumption through the branch and bound method. As a result, a much Smaller DAG called Small-DAG than the existing ones is constructed. The experimental results have shown that the Big-MLCS outperforms the two state-of-the-art algorithms: Quick-MLCS and Top-MLCS, in terms of time and space cost, and can solve much larger-scale MLCS problems more quickly than Quick-MLCS and Top-MLCS.

However, there are some issues to be studied further: 1) In the branch and bound method, the lower bound of the length of the longest path is estimated one time, that is, once it is estimated, it is used from the beginning to the end without updating. This is not good enough in some time. In the future, we will try to adaptively update this lower bound by a heuristic method so that the algorithm is more efficient. 2) The upper bound estimation used many sequences, which can increase the time cost. In the future, it is necessary to design a method which uses only a few sequences with fewer time cost. In fact, we found that the length of the LCS formed by the two shorter sequences will be relatively shorter and closer to the true upper bound. Therefore, when estimating the upper bound, we can only consider selecting two of the shorter sequences, and calculate their LCS length as the upper bound of

the second part. 3) The current algorithm is not a parallel algorithm. Due to the large number of search nodes' successors and the operation of computing diversity during the running of the algorithm, we can speed up the execution of the algorithm through the strategy of parallel computing (calculating each successor of one node at the same time, and calculating diversity among multiple sequences concurrently).

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (61872281).

References

[1] A. Aravanis, M. Lee, R. Klausner, Next-generation sequencing of circulating tumor DNA for early cancer detection, Cell 168 (4) (2017) 571–574, doi:10.1016/0003-4916(63)90068-X.
[2] B. Nogrady, How cancer genomics is transforming diagnosis and treatment, Nature 579 (7800) (2020) S10–S11, doi:10.1038/d41586-020-00845-4.
[3] L. Chaabane, Lamiche, A hybrid solver for protein multiple sequence alignment problem, J. Bioinf. Comput.Biol. 11 (1) (2018) 9111–9116, doi:10.1142/S0219720018500154.
[4] D.S. Huang, X.M. Zhao, G.B. Huang, Y.M. Cheung, Classifying protein sequences using hydropathy blocks, Pattern Recognit. 39 (12) (2006) 2293–2300, doi:10.1016/j.patcog.2005.11.012.
[5] X. Pan, H.-B. Shen, Scoring disease-microRNA associations by integrating disease hierarchy into graph convolutional networks, Pattern Recognit. 105 (2020) 107385, doi:10.1016/j.patcog.2020.107385.
[6] D. Jarchi, C. Wong, R.M. Kwasnicki, B. Heller, Gait parameter estimation from a miniaturized ear-worn sensor using singular spectrum analysis and longest common subsequence, IEEE Trans. Biomed. Eng. 61 (4) (2014) 1261–1273, doi:10.1109/tbme.2014.2299772.
[7] T.D. Pham, Spectral distortion measures for biological sequence comparisons and database searching, Pattern Recognit. 40 (2) (2007) 516–529, doi:10.1016/j.patcog.2006.02.026.
[8] L. Ou-Yang, X.-F. Zhang, H. Yan, Sparse regularized low-rank tensor regression with applications in genomic data analysis, Pattern Recognit. 107 (2020) 107516, doi:10.1016/j.patcog.2020.107516.
[9] D. Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336, doi:10.1145/322063.322075.
[10] C. Blum, M.J. Blesa, L. Manuel, Beam search for the longest common subsequence problem, Comput. Oper. Res. 36 (12) (2009) 3178–3186, doi:10.1016/j.cor.2009.02.005.
[11] J. Yang, Y. Xu, G. Sun, Y. Shang, A new progressive algorithm for a multiple longest common subsequences problem and its efficient parallelization, IEEE Trans. Parallel Distrib.Syst. 24 (5) (2013) 862–870, doi:10.1109/TPDS.2012.202.
[12] J. Yang, Y. Xu, Y. Shang, G. Chen, A space-bounded anytime algorithm for the multiple longest common subsequence problem, IEEE Trans. Knowl. Data Eng. 26 (11) (2014) 2599–2609, doi:10.1109/TKDE.2014.2304464.
[13] N. Etminan, E. Parvinnia, A. Sharifi-Zarchi, FAME: fast and memory efficient multiple sequences alignment tool through compatible chain of roots, Bioinformatics 36 (12) (2020) 3662–3668, doi:10.1093/bioinformatics/btaa175.
[14] Hirschberg, S. Daniel, Algorithms for the longest common subsequence problem, J. ACM 24 (1) (1977) 664–675, doi:10.1145/322033.322044.
[15] A. Apostolico, S. Browne, C. Guerra, Fast linear-space computations of longest common subsequences, Theor. Comput. Sci. 92 (1) (1992) 3–17, doi:10.1016/0304-3975(92)90132-Y.
[16] V.K. Tchendji, A.N. Ngomade, J.L. Zeutouo, J.F. Myoupo, Efficient CGM-based parallel algorithms for the longest common subsequence problem with multiple substring-exclusion constraints, Parallel Comput. 91 (2020) 102598, doi:10.1016/j.parco.2019.102598.
[17] S.C. Schuster, Next-generation sequencing transforms today's biology, Nat. Methods 5 (1) (2008) 16, doi:10.1038/nmeth1156.
[18] J. Hunt, T. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (5) (1977) 350–353, doi:10.1145/359581.359603.
[19] K. Hakata, H. Imai, Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima, Optim. Methods Softw. 10 (2) (1998) 233–260, doi:10.1080/10556789808805713.
[20] D. Korkin, A New Dominant Point-Based Parallel Algorithm for Multiple Longest Common Subsequence Problem, Technical Report, Department of Computer Science, University of New Brunswick, New Brunswick, Canada, 2001.
[21] Y. Chen, A fast parallel algorithm for finding the longest common sequence of multiple biosequences, BMC Bioinf. 7 (Suppl 4) (2006) 1–12, doi:10.1080/10556789808805713.
[22] Q. Wang, D. Korkin, Y. Shang, A fast multiple longest common subsequence (MLCS) algorithm, IEEE Trans. Knowl. Data Eng. 23 (3) (2011) 321–334, doi:10.1109/TKDE.2010.123.
[23] P. Gustavsson, A. Syberfeldt, A new algorithm using the non-dominated tree to improve non-dominated sorting, Evol. Comput. 26 (1) (2018) 89–116, doi:10.1162/evco_a_00204.
[24] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, J. Huang, A novel fast and memory efficient parallel MLCS algorithm for long and large-scale sequences alignments, in: 32nd IEEE International Conference on Data Engineering (ICDE), Helsinki, Finland, 2016, pp. 1170–1181, doi:10.1109/ICDE.2016.7498322.
[25] S. Liu, Y. Wang, W.N. Tong, S.W. Wei, A fast and memory efficient MLCS algorithm by character merging for DNA sequences alignment, Bioinformatics 36 (4) (2020) 1066–1073, doi:10.1093/bioinformatics/btz725.

Yuping Wang: received the PH.D. degree from the Department of Mathematics, Xian Jiaotong Unversity, Xi'an, China, in 1993. Currently, he is a full professor with the School of Computer Science and Technology of Xidian University, Xi'an, China. His research interests include optimization algorithms, optimization modeling for big data and network scheduling.

Yiuming Cheung: received Ph.D. degree at Department of Computer Science and Engineering from the Chinese University of Hong Kong in 2000. Currently, he is a Professor at the Department of Computer Science in Hong Kong Baptist University. His research interests include machine learning, information security, signal processing, pattern recognition, and data mining.

Chunyang Wang: received Bachelor degree at Department Computer Science and Technology, Northeastern University, Shenyang, China, in 2018. Currently, he is a PH.D student with the School of Computer Science and Technology of Xidian University, Xi'an, China. His research interests include big data, pattern recognition, and data mining.