Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020

Sequence Analysis

A Path Recorder Algorithm for Multiple Longest Common Subsequences (MLCS) Problems

Shiwei Wei [1,2] ,Yuping Wang [1,*], Yuanchao Yang [1] and Sen Liu [1]

[1] School of Computer Science and Technology, Xidian University, Xian, Shaanxi, China.
[2] School of Computer Science and Engineering, Guilin University of Aerospace Technology, Guilin, Guangxi, China.

* To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Searching the Longest Common Subsequences of many sequences is called a Multiple Longest Common Subsequence (MLCS) problem which is a very fundamental and challenging problem in many fields of data mining. The existing algorithms cannot not applicable to problems with long and large-scale sequences due to their huge time and space consumption. To efficiently handle large-scale MLCS problems, a Path Recorder Directed Acyclic Graph (PRDAG) model and a novel Path Recorder Algorithm (PRA) are proposed.
Results: In PRDAG, we transform the MLCS problem into searching the longest path from the Directed Acyclic Graph (DAG), where each longest path in DAG corresponds to an MLCS. To tackle the problem efficiently, we eliminate all redundant and repeated nodes during the construction of DAG, and for each node, we only maintain the longest paths from the source node to it but ignore all non-longest pathes. As a result, the size of the DAG becomes very small, and the memory space and search time will be greatly saved. Empirical experiments have been performed on a standard benchmark set of both DNA sequences and protein sequences. The experimental results demonstrate that our model and algorithm outperform the related leading algorithms, especially for large-scale MLCS problems.
Contact: ywang@xidian.edu.cn

1 Introduction

The biological data can be usually represented by sequences of different symbols. For example, protein sequences can be represented as sequences of 20 different amino acids, and DNA sequences can be represented as sequences of 4 bases: A, C, G and T. One of the most fundamental problems in bioinformatics is to measure the similarity of biological sequences, which is very important in the fields such as identification of cancers (Aravanis *et al.*, 2017), detection of the common origin of different species (Zvelebil and Baum, 2007), pattern recognition and molecular biology (Yang *et al.*, 2013), etc. Finding the MLCS of sequences is a commonly used method to measure the similarity of sequences. So designing effective model and developing efficient algorithm for dealing with the MLCS problem are becoming more and more important. However, Maier has proved that it is an NP-hard problem (Maier, 1978). As the number and the length of sequences increase, the time complexity and space complexity of solving such problems grow exponentially.

According to the different number of sequences, these problems can be generally classified into two classes: (1) The problem of finding the Longest Common Subsequences (LCS) between only *two* sequences is called an LCS problem. (2) The problem of finding the Longest Common Subsequences among *three or more* sequences is called an MLCS problem.

In the past few decades, efforts have been mainly concentrated on the LCS problem, and many methods (Sankoff, 1972; Hirschberg, 1977; Masek and Paterson, 1980; Smith and Waterman, 1981; Hsu and Du, 1984; Apostolico *et al.*, 1992) have been proposed. One of the typical methods is called Dynamic Programming approach (DP), but it is inefficient for MLCS problems. However, with the development of next generation of human genome project and gene sequencing techniques, the number of biological sequences (as well as other kinds of sequences from various applications) is growing rapidly. More and more real world problems require to search MLCS from many sequences. It becomes urgent to develop more effective models and efficient algorithms to address such MLCS problems.

In 1977, Hunt (Hunt and Szymanski, 1977) proposed a new type of methods called dominant point approach for computing MLCS, in which

© The Author(s) (2020). Published by Oxford University Press. All rights reserved. For Permissions, please email: journals.permissions@oup.com

a Directed Acyclic Graph (DAG) is constructed and searching MLCS is transformed into looking for the longest paths from the source node to the ending node in the DAG. This type of methods has been proved to be much more efficient than DP approach for MLCS problems with more than 3 sequences. However, the original dominant point method (Hunt and Szymanski, 1977) is only for problems with two sequences and same length, and its time complexity is $O((n+r)log^n)$, where n is the length of sequences and r is the number of nodes in the built DAG. To improve the efficiency, a parallel algorithm for MLCS problems is proposed by Korkin (Korkin, 2001). Its time complexity is $O(d|\Sigma|)$, where d is the number of sequences and Σ is the set of symbols. To further improve the efficiency, Chen (Chen *et al.*, 2006) proposed an efficient MLCS algorithm for DNA sequences called FAST-LCS. It involves a new table structure called the successor table, which enables the node's successors to be created in a constant time. It also uses a new strategy to delete the dominated nodes in the DAG. These strategies can save the space and speed up the search. To further improve the efficiency of the FAST-LCS, Wang (Wang *et al.*, 2011) proposed a parallel algorithm called *Quick-DP*. Li (Li *et al.*, 2012) proposed a parallel algorithm for the LCS problem, which can be efficiently executed on GPUs. Yang (Yang *et al.*, 2010) designed a parallel MLCS algorithm for the cloud computing platform. These algorithms show a better performance than the typical dominant point approach because they adopted some schemes for saving space cost or parallelization, but all of them utilize the non-dominated sorting method to delete the dominated nodes in the construction of DAG, which is very time consuming. Thus they are inefficient when handling MLCS problems with long and a large number of sequences. Recently, Li and Wang, et al (Li *et al.*, 2016a) designed a new model called NCSG and proposed a dominant points based algorithm called *Top-MLCS* to reduce the time complexity in constructing DAG. Although *Top-MLCS* has a better performance in terms of time consuming, it needs much more memory space to store the whole DAG. When the required memory exceeds the limit of the physic memory (this often happens for large scale problems), the algorithm will fail to find MLCS (even the algorithm cannot be executed).

At present, the number of sequences in MLCS problems is becoming larger and larger (up to 100 even up to 1000). There are more challenges and difficulties to find out MLCS from a large-number of sequences. Usually we call MLCS problems with more than 100 sequences as large-scale MLCS problems (LMLCS for short). Although many methods (e.g., Rick, 1994, Chen *et al.*, 2006, Yang *et al.*, 2010, Wang *et al.*, 2011, Yang *et al.*, 2013, Yang *et al.*, 2014, Li *et al.*, 2016a, Li *et al.*, 2016b) have been proposed for processing MLCS problems, they are not efficient or even can not work on LMLCS due to the high time and space consumption. Therefore, it is very necessary and valuable to design more efficient algorithms to handle LMLCS. So in this paper, we design a Path Recorder Directed Acyclic Graph (PRDAG) model and propose a Path Recorder Algorithm (PRA). Comprehensive experiments are performed on two kinds of biological sequences (DNA sequences and protein sequences) with different numbers and lengths of sequences, and the experimental results show that the proposed algorithm PRA performs better than related leading algorithms, especially for LMLCS. The main contributions are summarized as follows.

1. A Path Recorder Directed Acyclic Graph (PRDAG) model is designed to transfer searching MLCS among the sequences into searching the longest path in graph PRDAG, which can significantly simplify the procedure of finding MLCS for LMLCS. Before a new match point (corresponding to a node in PRDAG) is created according to its precursor, we will check whether it already exists in PRDAG and ensure that the generated PRDAG do not contain any redundant node. Thus, the scale of PRDAG is much smaller, and this can reduce both time consuming and space consuming.
2. Based on the PRDAG model, a fast dominant point based algorithm called Path Recorder Algorithm (briefly PRA) is proposed. For each node in PRDAG, we only record its such precursor nodes whose the longest distance to the current point is 1 as the key precursor nodes, but ignore other precursor nodes (called its non-key precursor nodes) because they contribute nothing to the longest paths. Therefore, the size of constructed DAG is very small, and more running time and memory consumption are saved. In addition, once the construction of DAG is completed, all MLCSs can be quickly found out through searching the longest paths in DAG by traversing from the ending node to source node.

The rest of this paper is organized as follows. Next section introduces some backgrounds and related works about LCS and MLCS. Section 3 describes the new path recorder directed acyclic graph model PRDAG and the proposed algorithm PRA in details. In section 4, to evaluate performance of the proposed algorithm, we compare it with some of the best performance algorithms by experiments on a comprehensive benchmark set. Finally, in Section 5 we make the conclusion.

2 Backgrounds and Related Works

2.1 Definition of the MLCS Porblem

To better understand the MLCS problem, we first define some notations and terminologies to be used in this paper and then review some related works for the MLCS problem.

Definition 1. *Let Σ be the symbol set and $s = c_1c_2...c_n$ be a sequence on symbol set Σ ($c_i \in \Sigma, 1 \leq i \leq n$). Let $|s|$ denote the length of s, i.e., $|s| = n$. A sequence s' is called a subsequence of sequence s if and only if s' is obtained by removing zero or at least one symbol from s, i.e., $s' = c_{i_1}c_{i_2}...c_{i_m}$ satisfying $1 \leq i_1 < i_2 < \cdots < i_m \leq n$. It is denoted as $s' \in Subseq(s)$.*

Definition 2. *Let $S = \{s_1, s_2, ..., s_d\}$ be a set of d sequences $s_1, s_2, ..., s_d$ on Σ. A sequence s' is called a Longest Common Subsequence (*LCS*) of the d sequences if s' satisfies the following conditions:*
(1) $s' \in Subseq(s_i)$ for $i = 1, 2, \cdots, d$, i.e., s' is a common subsequence of all d sequences s_i ($1 \leq i \leq d$).
(2) $\neg\exists s'' \in Subseq(s_i)$ for $1 \leq i \leq n$ satisfying $|s''| > |s'|$, i.e., s' is the longest subsequence of all these d sequences s_i ($1 \leq i \leq d$).

Let $|S| = d$ denote the cardinality of set S and $LCS(S)$ denote the set of all Longest Common Subsequences of S.

Usually there are more than one LCS for given d sequences. For example, for two sequences $s_1 = GAAGCGTA$ and $s_2 = AGTCTGAC$, both subsequences $AGCTA$ and $AGCGA$ are their LCS.

2.2 Dominant Point Based Approaches

Dominant point based approaches have been considered to be one of the most efficient approaches for obtaining exact MLCSs from multiple sequences (e.g., Wang *et al.*, 2011; Yang *et al.*, 2013; Li *et al.*, 2016a). Before discussing this type of methods in details, we first introduce some terminologies (Peng and Wang, 2017) to be used in this paper:

Definition 3. *Given d sequences s_1, s_2, ..., s_d on a symbol set Σ, if a symbol $\delta \in \Sigma$ is a common symbol in d sequences, i.e., $s_1[p_1] = s_2[p_2] = ... = s_d[p_d] = \delta$, where $s_i[p_i]$ represents δ being the p_i-th symbol from the left of sequence s_i. Then vector $p = (p_1, p_2, ..., p_d)$ is called a **match point** of these d sequences. Each match point $p =$*

Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020

$(p_1, p_2, ..., p_d)$ *corresponds to a unique symbol* δ*. So we also often use* $p = \delta(p_1, p_2, ..., p_d)$ *to denote the match point, where* δ *is the corresponding symbol of p, denoted by* $C(p) = \delta$*.*

Definition 4. *Given two match points* $p = (p_1, p_2, ..., p_d)$ *and* $q = (q_1, q_2, ..., q_d)$ *of d sequences on a symbol set* Σ*, we call:*

(1) ***p = q*** *iff* $\forall i(1 \le i \le d)$, $p_i = q_i$.

(2) p ***dominates*** *q (denoted by* $p \preceq q$*), if* $p_i \le q_i$ $(1 \le i \le d)$ *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 a ***successor*** *of p or p is a* ***precursor*** *of q, if* $p \prec q$ *and there is no other match point* $v(v \ne q)$ *satisfying* $p \prec v \preceq q$ *and* $C(q) = C(v) = \delta$*, denoted by* $Suc_\delta(p) = q$*.*

Note that a match point p has at most $|\Sigma|$ successors.

Definition 5. *Let* $P = \{P_1, P_2, ..., P_m\}$ *be a set of match points of d sequences, where each* $P_i(1 \le i \le m)$ *is a match point of the d sequences. For a match point* $P_j \in P$*, If* $\neg\exists P_i \preceq P_j$, $1 \le i, j \le m, i \ne j$*, then* P_j *is called a non-dominated point (dominant point for short) on* P*. The set of all dominant points on* P *is called the dominant set of* P*.*

The main idea of dominant point based approaches is as follows. Given d sequences, initially, the specific match point $(0, 0, ..., 0)$ with no incoming edges is defined as the source match point, and it forms the 0th level set (denoted by *level-0*). Then all successors of the source match point can be determined, and form the 1th level set *level-1*. A directed edge from the source match point to each of it's successors is drawn to represent the relationship between the match point and each of its successors. In this way, a Directed Acyclic Graph (DAG) containing *level-0* and *level-1* is formed, where the source match point is in *level-0* and its all successors are in *level-1*. Here, *level-1* indicates that the corresponding symbols of the match points in *level-1* may be the 1th character of an MLCS of the d sequences. However, through further analysis, we can find that the corresponding symbol of all dominated match points in *level-1* will appear later during the construction of DAG. So only the non-dominated match points in *level-1* need to be kept during the construction of DAG. All non-dominated match points in *level-1* forms the dominant set D^1 and it can be obtained by the non-dominated sorting method(Chen *et al.*, 2006; Wang *et al.*, 2011). Next, calculate all successors of each dominant in D^1. These successors form *level-2* and draw directed edges from each dominant in D^1 to their successors. Then a DAG from *level-0* to *level-2* is formed. Similarly, we only keep all dominants from *level-2* and form the dominant set D^2. Repeat this process until no successor can be found. We then set point $(\infty, \infty, ..., \infty)$ as the final successor of all dominnats without successor. The point $(\infty, \infty, ..., \infty)$ is defined as ending point and has no successor. The construction of DAG is completed. Once DAG is gotten, the longest paths, which respond to all MLCS, will be obtained. So the key issue of MLCS problems is how to effectively construct DAG. Algorithm 1 shows the main process of classical dominant point based approaches.

Fig.1 is an example which shows the process of Algorithm 1 when dealing with two sequences $GAAGCGTA$ and $AGTCTGAC$, and the detailed description of the process is provided in supplementary material.

From the example above we can see that, classical dominant point based approaches have the following main shortcomings:

- One match point can appear many times during the process of constructing DAG, even may appear in different levels. For example, match point $(5, 4)$ appears not only twice in *level-2*, but also in *level-1* and *level-3*. Computing and storing these points will consume time and memory space.
- All of redundant points (marked by black or gray background) in each level will be generated first, then identified, and finally deleted. However, these operations waste a lot of space and computation time.
- In order to find out set D^k of non-dominated points by nondominated sorting in *level-k*, pairwise comparisons of d-dimensional points have to be made among all points in set *Candi* in *level-k*, and this operation is quite time-consuming especially when the number d of sequences is large or the cardinality of set *Candi* is large.

Algorithm 1 Framework of Classical Dominant Point Based Approaches

Input:
The d sequences.
Output:
The generated DAG and All MLCS(s) obtained from d sequences.

1: $k = 0$, $D^k \leftarrow O(0, 0, ..., 0)$
2: $DAG \leftarrow O(0, 0, ..., 0)$
3: **while** $D^k \ne \emptyset$ **do**
4: $Candi = \emptyset$
5: **for** each element p of D^k **do**
6: calculate all successors of p: $Suc(p)$
7: Tentative $Candi \leftarrow Candi \cup Suc(p)$
8: $Candi \leftarrow$ Remove repeated points in $Candi$, i.e., each point in $Candi$ will be stored in one copy.
9: **end for**
10: $D^{k+1} \leftarrow NondominatedSorting(Candi)$
11: $DAG \leftarrow DAG \cup D^{k+1}$
12: draw a directed arrow(edge) from each point in D^k to each of its successors in D^{k+1}
13: $k \leftarrow k + 1$
14: **end while**

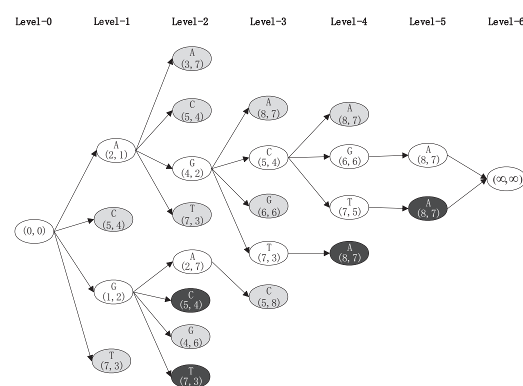

Fig. 1. Construction of DAG for two sequences $GAAGCGTA$ and $AGTCTGAC$, where the black and gray nodes are repeated and dominated nodes, respectively.

Due to these shortcomings, the classical dominant point based approaches are unsuitable to deal with large-scale MLCS problems. In order to overcome the defects of the classical dominant point based approaches, this paper designs a more effective model and develops a more efficient algorithm PRA for large-scale MLCS problems.

3 The Path Recorder Directed Acyclic Graph Model and Path Recorder Algorithm

3.1 The Path Recorder Directed Acyclic Graph Model: PRDAG

PRDAG is a graph model and our main task is to construct a complete PRDAG graph from the source match point. Given d sequences $s1, s2, \cdots, sd$. We can construct graph model PRDAG as follows.

Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020

Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020

Construction of Graph Model: PRDAG

1. Initially, PRDAG contains one point: the source point $O(0, 0, \cdots, 0)$. Add O to a first-in-first-out queue Q, set the length of the tentative longest path from source point O to O as $L(O(0, 0, \cdots, 0)) = 0$.
2. Take an element, say p, from the head of queue Q. Find out p's successors: $q_1, q_2, \cdots, q_t$. For each of successors $q_i (1 \leq i \leq t)$, check whether it is a newly created point. If yes, add it to the queue Q and add it as a successor point of p into PRDAG. Draw a directed edge from p to q_i and set its L value as: $L(q_i) = L(p) + 1$. Otherwise, i.e., q_i has existed in PRDAG, then there are three cases:

   1) If $L(q_i) < L(p) + 1$, then there is at least one longest path from O to q_i passing through p whose length is $L(p) + 1$. So we update L value of q_i as $L(q_i) = L(p) + 1$, set p as the key precursor of q_i, and remove all of q_i's previous precursors (they now become the non-key precursors) and all directed edges from these non-key precursors to q_i.

   2) If $L(q_i) = L(p) + 1$, then paths from O to q_i passing through p are also the longest ones, and accordingly p is a key precursor of q_i. Thus, draw a directed edge from p to q_i.

   3) If $L(q_i) > L(p) + 1$, then all paths from O to q_i passing through p are not the longest ones, so nothing need to do.
3. If Q is empty, the construction of PRDAG is finished. Otherwise, goto Step 2.

Let's use an example to explain the construction of PRDAG. For given two sequences $s1 = GAAGCGTA$ and $s2 = AGTCTGAC$.

1. Initially, PRDAG contains one point: the source point $O(0, 0)$. Add $O(0, 0)$ to a first-in-first-out queue Q and set $L(O(0, 0)) = 0$.
2. Take an element, in this case, only $O(0, 0)$, from the head of queue Q. Find out O's successors: $A(2, 1)$, $C(5, 4)$, $G(1, 2)$ and $T(7, 3)$. Now all successors are newly created points (i.e., they do not exist in PRDAG). Add them to Q one by one and add them as the successors of O into PRDAG. Draw a directed edge from O to each of them, set the lengths of the tentative longest paths from O to them (briefly called their L values) as $L(A(2, 1)) = 1$, $L(C(5, 4)) = 1$, $L(G(1, 2)) = 1$ and $L(T(7, 3)) = 1$, respectively (as shown at the bottom left of each point in Figure 2 (a)). Up to now, Q contains 4 points: $A(2, 1), C(5, 4), G(1, 2)$ and $T(7, 3)$, while PRDAG contains 5 points: $O(0, 0)$, $A(2, 1), C(5, 4), G(1, 2)$ and $T(7, 3)$. O is the key precursor of each of $A(2, 1), C(5, 4), G(1, 2)$ and $T(7, 3)$.
3. Take element $A(2, 1)$ from the head of queue Q. Find out all its successors $A(3, 7)$, $C(5, 4)$, $G(4, 2)$ and $T(7, 3)$. Because successors $A(3, 7)$ and $G(4, 2)$ are newly created points, we first add them to Q, and add them as the successors of $A(2, 1)$ into PRDAG. Draw a directed edge from $A(2, 1)$ to each of them. Set their L values as: $L(A(3, 7)) = L(A(2, 1)) + 1 = 2$ and $L(G(4, 2)) = L(A(2, 1)) + 1 = 2$, respectively, and set $A(2, 1)$ as a key precursor of $A(3, 7)$ and $G(4, 2)$. For successor $C(5, 4)$, because $L(C(5, 4)) < L(A(2, 1)) + 1$, the tentative longest path from O to point $C(5, 4)$ now becomes $O(0, 0) \rightarrow A(2, 1) \rightarrow C(5, 4)$. So $A(2, 1)$ is a key precursor of point $C(5, 4)$ while point $O(0, 0)$ becomes a non-key precursor of point $C(5, 4)$. The L value of point $C(5, 4)$ is updated to $L(C(5, 4)) = L(A(2, 1)) + 1 = 2$. Add a directed edge from $(2, 1)$ to $(5, 4)$ and erase the directed edge from O to $(5, 4)$. For successor $T(7, 3)$, The situation is same as that of $(5, 4)$. $A(2, 1)$ becomes a key precursor of point $C(7, 3)$ while point $O(0, 0)$ is *no longer* a key precursor of point $C(7, 3)$. So add a directed edge from $(2, 1)$ to $(7, 3)$ and erase the directed edge from O to $T(7, 3)$. Update $L(T(7, 3)) = L(A(2, 1)) + 1 = 2$. Up to now, Q holds 5 points: $C(5, 4), G(1, 2)$, $T(7, 3)$, $A(3, 7)$ and $G(4, 2)$, while PRDAG contains 7 points: $O(0, 0), A(2, 1), C(5, 4), G(1, 2), T(7, 3), A(3, 7)$ and $G(4, 2)$.
4. Then point $C(5, 4)$ is taken from Q and its successors are $A(8, 7), G(6, 6)$ and $T(7, 5)$. They are all newly created points. Add them to Q and add them as successors of $C(5, 4)$ into PRDAG. Since $(5, 4)$ is the key precursor of $A(8, 7), G(6, 6)$ and $T(7, 5)$, draw a directed edge from $C(5, 4)$ to each of them. Set the L values of them as $L(A(8, 7)) = L(C(5, 4)) + 1 = 3$, $L(G(6, 6)) = L(C(5, 4)) + 1 = 3$, $L(T(7, 5)) = L(C(5, 4)) + 1 = 3$.
5. Point $G(1, 2)$ is taken from Q, and its successors are $A(2, 7)$, $(5, 4)$, $G(4, 6)$ and $T(7, 3)$. Among these successors, $A(2, 7)$ and $G(4, 6)$ are newly created points. Add them as successors of $G(1, 2)$ into PRDAG and draw a directed edge from $G(1, 2)$ to each of them. Set the L values of them as $L(A(2, 7)) = L(G(1, 2)) + 1 = 2$ and $L(G(4, 6)) = L(G(1, 2)) + 1 = 2$. But for $C(5, 4)$ and $T(7, 3)$, they have already been in PRDAG. Add a directed edge from $G(1, 2)$ to each of them, because $L(G(1, 2)) + 1 = L(C(5, 4))$ and $L(G(1, 2)) + 1 = L(T(7, 3))$. $G(1, 2)$ is also a key precursor of $C(5, 4)$ and $T(7, 3)$.
6. Point $T(7, 3)$ is now at the head of Q. Take it from Q and calculate its successor, i.e., $A(8, 7)$. This successor has already been in PRDAG with $L(T(7, 3)) + 1 = L(A(8, 7)) = 3$. So $T(7, 3)$ is also a key precursor of $A(8, 7)$. Draw a directed edge from $T(7, 3)$ to $A(8, 7)$.

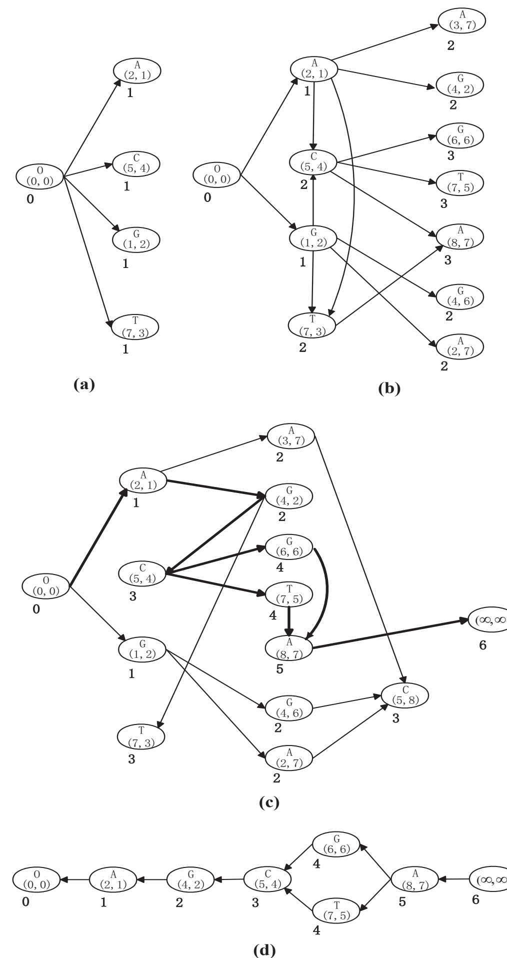

Fig. 2. The process of PRDAG construction for sequences GAAGCGTA and AGTCTGAC. The head and tail in a directed edge '→' denote the key precursor and successor, respectively. LCSs are obtained by the recorded paths from the ending point to the source point node.

Up to now, Q holds 7 points and PRDAG contains 12 points as shown in Figure 2(b).

7. Continue to take elements from Q, repeat the same operation as Step 3 to Step 6. When Q becomes an empty queue, the construction of PRDAG is completed as shown in Figure 2(c).
8. Backward search the key precursors from ∞. Successively find out the key precursor $A(8,7)$ of ∞, the key precursors $G(6,6)$ and $T(7,5)$ of $(8,7)$, the key precursor $C(5,4)$ of $G(6,6)$ and $T(7,5)$, the key precursor $G(4,2)$ of $C(5,4)$, the key precursor $A(2,1)$ of $G(4,2)$ and the key precursor $O(0,0)$ of $A(2,1)$. Then we get two longest common subsequences as shown in Figure 2(d).

Based on the relationship of the *key* precursor and the successor, starting from the ending point ∞ and successively searching the precursor(s) of the current point(s) in the final PRDAG, we can easily find out the longest paths from the source point to the ending point (as shown in Figure 2(d)), which correspond to the MLCSs: $AGCGA$ and $AGCTA$.

3.2 The Path Recorder Algorithm (PRA)

An efficient algorithm based on the model PRDAG is proposed in this section and is called Path Recorder Algorithm (PRA). Before introducing PRA, we first introduce the key data structure to be used in the algorithm.

3.2.1 Node structure

For a match point (node) t in DAG, it contains the following information:

- The match point vector corresponding to t.
- $L(t)$: the *length* of the longest path(s) from the source point to t.
- $Suc(t)$: all *successors* of t.
- $Pre(t)$: all *key precursors* of t, i.e., $Pre(t) = \{s \mid t \in Suc(s)$ and $L(t) = L(s) + 1\}$.

In our PRA algorithm, The match point is represented by its coordinates. The $Pre(t)$ is used to mark those key precursors of point t which just locate on the longest path from the source point to current point t.

3.2.2 Path Recorder Algorithm(PRA)

Algorithm 2 shows the main framework of PRA, which consists of three components as follows: At the first step, successor tables are built on the given d sequences. At the second step, starting from the source point $O(0,0,\cdots,0)$, graph (PRDAG) will be generated by continuously expanding successors of match point, which is implemented by *Constuct_PRDAG()*. At the third step, when the construction of PRDAG is finished, all MLCSs can be found out from PRDAG by performing the *Traverse_PRDAG()* method.

Algorithm 2 Main Framework of PRA

```
Input: The sequences: S_d = {s_1, s_2, ..., s_d}
Output: All MLCSs from sequences S_d: mlcs.
1: sucTables ← BuildSuccTalbes(S_d)
2: ending ← Construct_PRDAG(sucTables)
3: mlcs ← Traverse_PRDAG(ending)
```

In Algorithm 2, *Constuct_PRDAG()* is used for constructing PRDAG, the pseudocode of which is given in Algorithm 3.

In Algorithm 3, there are several special data structures. With the help of them, PRDAG can be generated, where Dom is a set to store all match points in the PRDAG, and Q is a first-in-first-out queue to temporarily store all newly created successors of match points. During the process

Algorithm 3 *Construct_PRDAG()*

```
Input: sucTables
Output: The generated PRDAG
1: source = O(0,0,...,0), ending = ∞(∞,∞,...,∞)
2: L(source) ← 0, L(ending) ← 0
3: Pre(source) ← ∅ , Suc(ending) ← ∅
4: Dom ← {source, ending}
5: Q ← {source}
6: while Q ≠ ∅ do
7:   t ← pick up the head element of Q
8:   if t has no successor then
9:     Update_Mark(t, ending)
10:  else
11:    for each successor s of t do
12:      if s ∉ Dom then
13:        Dom ← Dom ∪ {s}
14:        Pre(s) ← {t}
15:        L(s) ← L(t) + 1
16:        append s to the tail of Q.
17:      else
18:        Update_Mark(t, s)
19:      end if
20:    end for
21:  end if
22: end while
```

of construction of PRDAG, we always pick up the first point in queue Q(the first element in Q) and then remove it from Q after we find out all successors of it. These successors will be appended to the tail of queue Q one by one. By Q, PRDAG can be expanded gradually until the construction is completed.

Initializations are shown from $line1 \sim line5$. Then, for each point t in Dom, check if it has successors. If not, we assume that the ending point $\infty(\infty,\infty,\cdots,\infty)$ is t's only successor, and then update the path recorder of $ending$(as shown in $line8 \sim line9$). Otherwise, for t, by referring to the Successor Tables $sucTables$, we calculate all successors of t to expand PRDAG. Suppose s is any of these successors. Before we add s into PRDAG (i.e., add s to set Dom), we should identify whether s has already been in Dom.

1. If s has not been in Dom (i.e., the point s does not exist in the PRDAG), it will be added into Dom and the relationship between it and its key precursor t will be established. Then length of the tentative longest paths from the source node to s is also updated to $L(t)+1$, and s will be appended to the tail of Q for further expanding.
2. If s has already been in Dom, there is no need to add s into Dom again. We only need to update the key precursor of s by performing $Update_Mark()$ which is a program to update the tentative longest path and the key precursor of s.

This process is shown in $line11 \sim line20$ in Algorithm 3. However, there is a key issue for the above procedure. Once a successor s is created, we must check whether s is in Dom ($line12$ in Algorithm 3) before updating PRDAG. This operation is performed very frequently in PRA, and it will be quite time-consuming when the number of points in Dom is large. To deal with this problem, a hash table is adopted in our algorithm, which can quickly check whether Dom contains s or not, and can quickly take it out of Dom if it exists in Dom. Empirical experiments also show that the time spent on checking whether a point exists in Dom is almost negligible.

During the construction of PRDAG, in order to update the tentative longest paths from the source point to the current point in real time, a method called $Update_Mark()$ is developed to update the set of the key precursors ($Pre(s)$) and the L value of current point s according to different cases:

Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020

Algorithm 4 *Update_Mark(t, s)*

Input: The point s and its precursor t.
Output: The updated L value of s, and the set of the key precursors of s.
1: **if** $L(s) < L(t) + 1$ **then**
2: $Pre(s) \leftarrow \{t\}$
3: $L(s) \leftarrow L(t) + 1$
4: **else if** $L(s) = L(t) + 1$ **then**
5: $Pre(s) \leftarrow Pre(s) \cup \{t\}$
6: **else**
7: continue;
8: **end if**

1. If $L(s) < L(t) + 1$, update the set of the key precursors of s by $Pre(s) \leftarrow \{t\}$ and the length of the tentative longest path from O to s by $L(s) \leftarrow L(t) + 1$. In this case, a longer path than the tentative longest path is found which is the tentative longest path from O to t plus the directed edge from t to s. Then, t is currently only key precursor of s and the tentative longest path is updated.
2. If $L(s) = L(t) + 1$, update the set of key precursors by $Pre(s) \leftarrow Pre(s) \cup \{t\}$. In this case, the longest path from O to t plus the directed edge from t to s is an additional tentative longest path and t is another key precursor of s.
3. If $L(s) > L(t) + 1$, this case indicates that the longest paths from O to t plus the directed edge from t to s is shorter than the current longest paths from O to s. Thus t is a non-key precursor of s and nothing needs to do.

The pseudo code of method $Update_Mark()$ is presented in Algorithm 4 which shows the process of updating L value and records the tentative longest paths.

When the construction of PRDAG is finished, the longest pathes can be easily found out from the completed PRDAG. By tracing the key precursors from the ending point to the source point, all the longest paths corresponding to *MLCSs* can be immediately obtained by using tree's depth traversal approach $Traverse_PRDAG()$.

4 Experiments and Analysis

4.1 Experimental Setups and Compared Algorithms

All of experiments are run on a workstation equipped with Intel(R) Xeon(R) Gold 6138 CUP(2.00GHz) and 704GB memory. The operating system is Windows 7 professional 64. We make the comparisons of the proposed algorithm *PRA* with three best performed algorithms *Top_MLCS*(Li *et al.*, 2016a), *Quick-DP* (Wang *et al.*, 2011) and *Fast_LCS* (Chen *et al.*, 2006) in these experiments. Biological sequences from *NCBI* [1] and *DIP* corpus [2] are selected as the test sets and usually have two kinds of sequences: protein sequences and DNA sequences (for the limitation of the journal pages, the experimental results on DNA sequences are not given in this paper but shown in supplementary materials).

To evaluate the performance of the proposed algorithm *PRA*, two types of experiments are conducted: 1) Fix the sequence length and change the number of sequences; 2) Fix the number of sequences and change the length of sequences. For the first type of experiments, the length of protein sequences is fixed to 240, and the number of protein sequences varies from 3 to 9000. The experimental results including run time and memory consumption consumed by the compared algorithms are listed in Table 1, where 'd' represents the number of sequences, 'l' represents the length of the obtained MLCS. For the second type of experiments, the number of protein sequences is fixed to 10 and the length of protein sequences varies from 80 to 250. The experimental results are listed in Table 2, where 'L' represents the Length of sequences, 'l' represents the length of the obtained MLCS.

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

[2] http://dip.doe-mbi.ucla.edu/dip/Download.cgi

Table 1. The run time (millisecond) / memory (megabyte) consumed by the compared algorithms on protein sequences with length fixed to 240.

d	l	Protein ($\|\Sigma\| = 20$)			
		FAST_LCS	Quick-DP	Top_MLCS	PRA
3	45	379 / 14	157 / 17	189 / 66	159 / 66
5	30	37616 / 21	3392 / 36	5555 / 534	3244 / 381
7	25	1996261 / 184	169122 / 187	123414 / 4295	64673 / 2601
9	24	– / –	3599302 / 2964	933081 / 25897	616202 / 15295
10	23	– / –	– / –	2119472 / 60655	1413280 / 38266
20	18	– / –	– / –	2177137 / 96125	1268958 / 56968
40	15	– / –	– / –	247959 / 14151	90146 / 6611
60	14	– / –	– / –	51084 / 4555	23001 / 2470
80	12	– / –	2882690 / 10397	4434 / 861	1771 / 482
100	11	– / –	1235160 / 2373	3633 / 561	1475 / 244
200	9	9450 / 25	1713 / 28	179 / 49	105 / 25
500	8	947 / 20	360 / 22	151 / 34	107 / 23
1000	6	252 / 23	180 / 21	164 / 31	142 / 28
5000	3	687 / 63	612 / 63	640 / 64	578 / 61
9000	0	1112 / 119	1163 / 109	1050 / 111	1001 / 105

Table 2. The run time (millisecond) / memory (megabyte) consumed by the compared algorithms. Each of data sets contains 10 protein sequences.

L	l	Protein ($\|\Sigma\| = 20$)			
		FAST_LCS	Quick-DP	Top_MLCS	PRA
80	6	38 / 1	28 / 1	23 / 1	17 / 1
110	9	142 / 5	66 / 7	66 / 14	41 / 6
140	13	14434 / 18	1568 / 116	654 / 153	213 / 80
170	16	6859997 / 1081	249010 / 2288	42153 / 2241	22526 / 1114
180	17	– / –	534724 / 3028	88490 / 4228	40506 / 2069
200	20	– / –	3357801 / 8406	584819 / 19025	252729 / 8539
210	21	– / –	– / –	1418219 / 40882	741791 / 18904
230	23	– / –	– / –	6451866 / 176502	2937844 / 72265
240	25	– / –	– / –	– / –	8480665 / 153142
250	27	– / –	– / –	– / –	9616040 / 275924

4.2 Experimental Results and Analysis

From Table 1 we can see that, the *FAST_LCS* algorithm always fails to handle protein sequences when the number of them varies from 9 to 100 due to the extremely long run time. While the *Quick-DP* algorithm also fails to deal with protein sequences with number varying from 10 to 60. The reason is that, in each point, the dimension of the match point grows quickly as the number of sequences grows. The pruning strategy adopted by these two algorithms needs a large number of comparisons among match points, which will be very time-consuming. *Top_MLCS* and *PRA* are much faster than *Quick-DP* and *FAST_LCS*. Both of them can successfully process all protein sequences. But, relatively, the proposed algorithm *PRA* always performs better than *Top_MLCS*. In particular, *PRA*

Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020

is 4.67% to 63.64% faster than *Top_MLCS*, and in general *PRA* is 33.34% in average faster than *Top_MLCS*. This demonstrates that algorithm *PRA* is more suitable for large-scale MLCS problems. Also It is can been seen from the experimental results that the memory consumption of *FAST_LCS* (*Quick-DP*) is less than that of *Top_MLCS* and *PRA* when the number of protein sequences is less than 7(9), but the memory consumption of *FAST_LCS* (*Quick-DP*) is greater than that of algorithm *Top_MLCS* and *PRA* when the number of protein sequences is greater than 200 (80). This also demonstrates that when dealing with large-scale MLCS problems, *Top_MLCS* and *PRA* outperform *FAST_LCS* and *Quick-DP* in terms of memory consumption. Compared with algorithm *Top_MLCS*, *PRA* always consumes fewer memory. To be specific, the memory consumption of *PRA* is 72.9% of *Top_MLCS* on average when processing protein sequences. Especially, when the number of protein sequences is 40, the memory consumption of *PRA* is only 46.72% of *Top_MLCS*. This is because *PRA* only records the key precursors of each match point and ignores non-key precursors during the construction of PRDAG, and many non-longest paths and useless match points can be removed from PRDAG timely. So *PRA* performs best among all compared algorithms in terms of memory consumption when handling large-scale MLCS problems.

From experimental results in Table 2 we can see that, the time consumption of *FAST_LCS* and *Quick-DP* grows sharply as the length of protein sequences grows, but the time consumption of *Top_MLCS* and *PRA* grows much more slowly. And *PRA* always spends much fewer time than *Top_MLCS*. To be specific, it is faster than *Top_MLCS* about 48.8% in average when processing proteins sequences. What is more, *FAST_LCS* can only process protein sequences with length no more than 170 in a reasonable time (3 hours), *Quick-DP* with length no more than 200, and *Top_MLCS* with length no more than 230. But *PRA* can process protein sequences with length up to 250. In addition, compared with *Top_MLCS*, *PRA* always consumes less memory. Its memory consumption is only 53.34% of that of *Top_MLCS*. Furthermore£¬as the length of sequences increases, the ratio of memory consumption of *PRA* and that of *Top_MLCS* becomes much smaller. When the length of protein sequences is 230, *Top_MLCS* consumes 176502MB memory space, but *PRA* consumes 72265MB, which only accounts for 40.94% of that of *Top_MLCS*.

5 Conclusion

In this paper, a new path recorder directed acyclic graph model, termed PRDAG, is designed for solving large scale MLCS problems. In model PRDAG, there is no repeated match points and every match point is assigned a special path recorder (a key precursor pointer) to record the longest paths from the source point to itself. So the scale of generated PRDAG is much smaller than that generated by the traditional DAG models, and a lot of memory space can be saved. Based on this model, a fast algorithm called path recorder algorithm (PRA) is proposed. During the construction of PRDAG, PRA will construct and update path recorders in real time during the expansion of PRDAG. Once the construction is completed, all MLCSs can be immediately found out by traversing the generated PRDAG from the ending point to the source point. And what is more, the algorithm PRA has good performance in terms of computing time, because we employ path recorder technology rather than the pruning strategy which needs a large number of comparisons among match points. Experimental results on a set of benchmarks of both DNA sequences and protein sequences also demonstrate that the proposed PRA outperforms compared state-of-the-art dominant point based algorithms, and can efficiently tackle large-scale MLCS problems.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No.61872281).

References

Apostolico, A., Browne, S., and Guerra, C. (1992). Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, **92**(1), 3 – 17.

Aravanis, A. M., Lee, M., and Klausner, R. D. (2017). Next-generation sequencing of circulating tumor dna for early cancer detection. *Cell*, **168**(4), 571–574.

Chen, Y., Wan, A., and Liu, W. (2006). A fast parallel algorithm for finding the longest common sequence of multiple biosequences. *BMC BIOINFORMATICS*, **7**(4). Symposium of Computations in Bioinformatics and Bioscience in Conjunction with the International Mult-Symposium on Computer and Computational Sciences, Hangzhou, PEOPLES R CHINA, JUN 20-24, 2006.

Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *J. ACM*, **24**(4), 664–675.

Hsu, W. and Du, M. (1984). Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, **24**(1), 45–59.

Hunt, J. W. and Szymanski, T. G. (1977). A fast algorithm for computing longest common subsequences. *Commun. ACM*, **20**(5), 350–353.

Korkin, D. (2001). A new dominant point-based parallel algorithm for multiple longest common subsequence problem. *Technical Report TR01-148, Univ. of New Brunswick, Tech. Rep.*

Li, Y., Wang, Y., and Bao, L. (2012). FACC: A Novel Finite Automaton Based on Cloud Computing for the Multiple Longest Common Subsequences Search. *MATHEMATICAL PROBLEMS IN ENGINEERING*.

Li, Y., Wang, Y., Zhang, Z., Wang, Y., Ma, D., and Huang, J. (2016a). A Novel Fast and Memory Efficient Parallel MLCS Algorithm for Long and Large-Scale Sequences Alignments. In *2016 32ND IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE)*, IEEE International Conference on Data Engineering, pages 1170–1181, 345 E 47TH ST, NEW YORK, NY 10017 USA. IEEE; IEEE Comp Soc; Aalto Univ, Sch Sci, IEEE. 32nd IEEE International Conference on Data Engineering (ICDE), Helsinki, FINLAND, MAY 16-20, 2016.

Li, Y., Li, H., Duan, T., Wang, S., Wang, Z., and Cheng, Y. (2016b). A real linear and parallel multiple longest common subsequences (mlcs) algorithm. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1725–1734, New York, NY, USA. ACM.

Maier, D. (1978). The complexity of some problems on subsequences and supersequences. *J. ACM*, **25**(2), 322–336.

Masek, W. J. and Paterson, M. S. (1980). A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, **20**(1), 18 – 31.

Peng, Z. and Wang, Y. (2017). A novel efficient graph model for the multiple longest common subsequences (mlcs) problem. *Frontiers in genetics*, **8**, 104.

Rick, C. (1994). New algorithms for the longest common subsequence problem.

Sankoff, D. (1972). Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, **69**(1), 4–6.

Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, **147**(1), 195 – 197.

Wang, Q., Korkin, D., and Shang, Y. (2011). A Fast Multiple Longest Common Subsequence (MLCS) Algorithm. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, **23**(3), 321–334.

Yang, J., Xu, Y., and Shang, Y. (2010). An efficient parallel algorithm for longest common subsequence problem on gpus. In *Proceedings of the world congress on engineering, WCE*, volume 1.

Yang, J., Xu, Y., Sun, G., and Shang, Y. (2013). A new progressive algorithm for a multiple longest common subsequences problem and its efficient parallelization. *IEEE Transactions on Parallel and Distributed Systems*, **24**(5), 862–870.

Yang, J., Xu, Y., Shang, Y., and Chen, G. (2014). A space-bounded anytime algorithm for the multiple longest common subsequence problem. *IEEE Transactions on Knowledge and Data Engineering*, **26**(11), 2599–2609.

Zvelebil, M. J. and Baum, J. O. (2007). *Understanding bioinformatics*. Garland Science.

Downloaded from https://academic.oup.com/bioinformatics/advance-article-abstract/doi/10.1093/bioinformatics/btaa134/5771335 by The Edward G Miner Library user on 04 March 2020