Locating Longest Common Subsequences with Limited Penalty

Bin Wang(✉), Xiaochun Yang, and Jinxu Li

School of Computer Science and Engineering, Northeastern University, Shenyang 110169, Liaoning, China
{binwang,yangxc}@mail.neu.edu.cn, lijinxu92@gmail.com

Abstract. Locating longest common subsequences is a typical and important problem. The original version of locating longest common subsequences stretches a longer alignment between a query and a database sequence finds all alignments corresponding to the maximal length of common subsequences. However, the original version produces a lot of results, some of which are meaningless in practical applications and rise to a lot of time overhead. In this paper, we firstly define longest common subsequences with limited penalty to compute the longest common subsequences whose penalty values are not larger than a threshold τ. This helps us to find answers with good locality. We focus on the efficiency of this problem. We propose a basic approach for finding longest common subsequences with limited penalty. We further analyze features of longest common subsequences with limited penalty, and based on it we propose a filter-refine approach to reduce number of candidates. We also adopt suffix array to efficiently generate common substrings, which helps calculating the problem. Experimental results on three real data sets show the effectiveness and efficiency of our algorithms.

Keywords: Longest common subsequence · Penalty score · Common substring

1 Introduction

The longest common subsequence (LCS) problem is a classic and well studied problem in computer science with extensive applications in diverse areas ranging from spelling error corrections to molecular biology. Especially in bioinformatics, LCS is the most important metric in all of local alignments, which are used for comparing primary biological sequence information, such as the amino-acid sequences of proteins or the nucleotides of DNA sequences. Locating LCS enables a researcher to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence. The longest

This work is partially supported by the NSF of China for Outstanding Young Scholars under grant No. 61322208, the NSF of China under grant Nos. 61272178 and 61572122.

S. Candan et al. (Eds.): DASFAA 2017, Part II, LNCS 10178, pp. 187–201, 2017.
DOI: 10.1007/978-3-319-55699-4_12

common subsequence problem for two strings, is to find a common subsequence in both strings, having maximum possible length, where a subsequence of a string is obtained by deleting zero or more symbols of that string. In this paper, we require to get their matching regions for the strings.

The original version of LCS stretches a longer alignment between the query and the database sequence in the left and right directions, from the position where the exact match occurred. The extension does not stop until the accumulated threshold. However, in practice, the original LCS produces too many alignments, some of which are meaningless. For example, bio-scientists prefer to find matches of bio-sequences locally (i.e. within a small region). To tackle this problem, in this paper we propose a new version of LCS, called longest common subsequences with limited penalty, denoted LCSP. LCSP adopts a penalty threshold to maintain the same level of sensitivity for detecting sequence similarity.

The main challenges and contributions of this paper are listed as follows:

- In order to satisfy the requirement of real applications, we propose a new version of longest common subsequences with limited penalty score in Sect. 3. In order to be consistent with the alignment problem in bio-sequences, we adopt the flexible scoring scheme in bio-applications to quantify penalties in the LCS.
- Obviously, generating LCSs using dynamic programing and checking every generated LCS under the penalty threshold are time consuming. We propose an approach by concatenating common substrings to avoid the dynamic programming in Sect. 4. Furthermore, this could be help to generate small number of LCSs for checking using the penalty threshold. This algorithm can retrieve all correct results, and is thus an exact algorithm.
- The number of concatenated common substrings could be large, especially when the given strings are long. In order to reduce this number, we propose a filter-refine approach to further improve our algorithm, which can avoid useless concatenated common substrings and early terminate calculations in Sect. 5. We also in Sect. 6 to show how to efficiently find common substrings by constructing suffix array index structure.
- We conduct experimental evaluations on three real data sets with different alphabets, lengths, and distributions to test and analyze our algorithms in Sect. 7. The results demonstrate the effectiveness and efficiency of our proposed algorithms.

2 Related Work

A lot of research efforts have been made to design algorithms for string alignment, such as Needleman-Wunsch [14], Smith-Waterman [18], and their corresponding improvements OASIS [14], BWT-SW [11] and ALAE [21], all of which are based on dynamic programming. When conducting sequence comparison, these algorithms consider exact matching, as well as insertion, deletion and substitution, and assign a score scheme to these transformation operations. The goal

of sequence matching is to find out the optimal matching, i.e. maximizing the number of matches and minimizing the number of spaces and mismatches. These algorithms usually suffer large space consumption, requiring a space complexity of $O(mn)$ [17], where m and n are the lengths of the two strings.

Although improvements have been made to reduce space complexity and enhance running efficiency using suffix array, they might return unsatisfactory results, which is caused by inappropriate score setting. Such a result typically acquires a decent score in their forepart, but confronts a score drop in the mid-part because of mismatches, and gets a relatively high score in the last part. As a consequence, these results usually end up with high overall scores, but are still unsatisfactory since their mismatched mid-parts are inconsistent with users' actual demands. Edit distance based approaches [8] retrieve dissimilar parts of the two strings, and restore them to the original strings, based on which the string similarity is evaluated. Other algorithms like BLAST [10] firstly acquire exact matched part of the two strings, then expand it to left/right, and form high-score matched sequences. In spite of their higher efficiency compared to dynamic programming based algorithms, they cannot guarantee retrieving all high-score segments without omission.

The classic dynamic programming solution to LCS problem, invented by Wagner and Fischer [19], has $O(mn)$ worst case running time. To reduce the space complexity, Hirschberg [4] provide an algorithm with $O(n)$ worst space cost, using a divide-and-conquer approach. The fastest known algorithm by Masek and Paterson [13] runs in $O(n^2/\log n)$ time. However, faster algorithms exist with complexities depending on special cases, such as when the input consists of permutations or when the output is known to be very long or very short. For example, Myers in [15] and Nakatsu et al. in [16] presented an $O(nB)$ algorithm, where the parameter B is the simple Levenshtein distance between the two given strings [12]. Hunt and Szymanski [3] studied the complexity of the LCS problem in terms of matching index pairs, i.e., they defined t to be the number of index-pairs (i, j) with $a_i = b_j$ (such a pair is called a match) and designed an algorithm that finds the LCS of two sequences in $O(t \log n)$ time. For a survey on the LCS problem see [2].

The rest of this paper is structured as follows: Sect. 3 elaborates the preliminaries of this paper and the problem definition; Sect. 4 proposes a basic approach for finding longest common subsequences with limited penalty. In Sect. 5, we analyze features of longest common subsequences with limited penalty, and based on it we propose a filter-refine approach to reduce number of candidates. And in Sect. 6 we adopt suffix array to efficiently generate common substrings, which helps calculating the problem. In Sect. 7 we present experimental results on real data sets to demonstrate the accuracy and time efficiency of the proposed technique. Then finally in Sect. 8, we conclude the paper.

3 Preliminaries and Problem Definition

Let Σ be an alphabet. For a string X of the characters in Σ, we use $|X|$ to denote the length of X, $X[i]$ to denote the i-th character of X (starting from 1), and $X[i \ldots j]$ to denote the substring from its i-th character to its j-th character.

A subsequence of a string is obtained by deleting zero or more symbols of that string. The longest common subsequence problem for two strings, is to find a common subsequence in both strings, having maximum possible length.

Definition 1. *Longest common subsequence (LCS). Given two strings* $X = X[1]X[2]\dots X[m]$ *and* $Y = Y[1]Y[2]\dots Y[n]$. *A subsequence* $X[i_1]X[i_2]\dots X[i_r]$ *of* X $(0 < i_1 < i_2 < \dots < i_r \leq m)$ *is obtained by deleting* $m - r$ *symbols from* X. *A common subsequence of two strings* X *and* Y, *denoted* $cs(X, Y)$, *is a subsequence common to both* X *and* Y. *The longest common subsequence of* X *and* Y, *denoted* $lcs(X, Y)$ *or* $LCS(X, Y)$, *is a common subsequence of maximum length. We denote the length of* $lcs(X, Y)$ *by* $|lcs(X, Y)|$.

For example, for the two strings $X = \texttt{traobcybgsfd}$ and $Y = \texttt{tracycy raogsfdy}$, $lcs(X, Y) = \texttt{tracygsfd}$. Based on the alignment of longest common subsequence, there exist three common substrings `tra`, `cy`, and `gsfd` along the alignment of $lcs(X, Y)$. We use (X^a, Y^b, l) to represent that $X[a \dots a+l-1]$ and $Y[b \dots b+l-1]$ share a common substring, where a and b are start positions of the matching substring in X and Y, respectively. In between every two adjacent common substrings, there is an uncommon substring pair $\langle X_i, Y_i \rangle$. We use penalty to evaluate the difference between all uncommon substrings in a longest common subsequence.

Definition 2. *Penalty of an LCS. Given two strings* X *and* Y, *let* $\langle X_1, Y_1 \rangle, \dots, \langle X_k, Y_k \rangle$ *be the pairs of uncommon substrings in* $lcs(X, Y)$. *The penalty of* $lcs(X, Y)$ *is defined as:*

$$p(lcs(X, Y)) = \sum_{i=1}^{k} \alpha \cdot M(X_i, Y_i) + \beta \cdot S(X_i, Y_i), \tag{1}$$

where $M(X_i, Y_i)$ *and spaces* $S(X_i, Y_i)$ *represent the number of mismatches and spaces between* X_i *and* Y_i, *and* (α, β) *is the scoring scheme where* α *and* β *are penalty scores of a mismatch and a space, respectively.*[1]

For ease of presentation, we use Fig. 1 to show the penalties of different LCSs. From this figure, we can easily see that there are two alignments corresponding to the same LCS subsequence $lcs(X, Y) = \texttt{tracygsfd}$. The first alignment consists of common strings `tra` with $(X^1, Y^1, 3)$, `cy` with $(X^6, Y^4, 2)$, and `gsfd` with $(X^9, Y^{11}, 4)$, and its penalty is $2\beta + (\alpha + 4\beta)$. The second alignment consists of `tra` with $(X^1, Y^1, 3)$, `cy` with $(X^6, Y^6, 2)$, and `gsfd` with $(X^9, Y^{11}, 4)$, and its penalty is $2\alpha + (\alpha + 2\beta)$. When both α and β equals 1, these two alignment have different penalties 7 and 5.

[1] Notice that, the definition of penalty score of LCS is different from edit distance even when $\alpha = \beta = 1$. The edit distance between two strings represents the *minimal* number of edit operations transforming from one string to another string, which does not guarantee to find an alignment with longest common subsequences as LCS does.

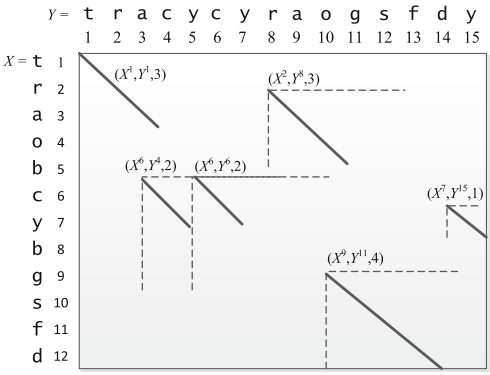


Fig. 1. Common substrings of $X = \texttt{traobcybgsfd}$ and $Y = \texttt{tracycyraogsfdy}$.

Problem Definition. The problem of longest common subsequence with limited penalty (a.k.a. LCSp) is to locate positions of every exact matching substrings along the longest common subsequences whose penalty is not greater than τ given two strings X and Y, and a penalty threshold τ, denoted $lcsp(X, Y, \tau)$.

4 A Basic Approach Based on Common Substrings

A straightforward approach of locating LCSp includes the following three steps: (i) calculate LCS using dynamic programming in $O(mn)$ time, where m and n are string lengths of the two given strings X and Y; (ii) get all possible alignments along the alignment of $lcs(X, Y)$ in $O(n \log n)$ (assuming $m \leq n$); and (iii) keep alignments whose penalty is not greater than the given penalty threshold. Therefore, the total cost is $O(mn)$.

Obviously, generating all LCSs firstly and then checking their penalties are time consuming. Now we propose our approach based on common substrings (We discuss how to get common substrings of X and Y in $O(n)$ time in Sect. 6). The basic idea of our approach is to start from the common substrings of X and Y since the final results must contain certain substring pair in the set of common substrings of X and Y. Then we concatenate the common substrings to get longer substring pairs of X and Y (lines 4–9), and verify each concatenated substring pair by calculating its penalty (lines 10–12). We call this baseline approach BasicLCSp (see Algorithm 1).

Reexamine the two strings $X = \texttt{traobcybgsfd}$ and $Y = \texttt{tracycyraogsfdy}$ and their common substrings `tra` with $(X^1, Y^1, 3)$, `rao` with $(X^2, Y^8, 3)$, `cy` with $(X^6, Y^4, 2)$ and $(X^6, Y^6, 2)$, `y` with $(X^7, Y^{15}, 1)$, and `gsfd` with $(X^9, Y^{11}, 4)$. The algorithm BasicLCSp firstly puts these common substrings in a candidate set C_{set}. Secondly, it gets 10 concatenated substring pairs from the above common substrings, which are $\langle X[1 \ldots 7], Y[1 \ldots 5] \rangle$, $\langle X[1 \ldots 7], Y[1 \ldots 7] \rangle$, $\langle X[1 \ldots 4], Y[1 \ldots 10] \rangle$, $\langle X[1 \ldots 7], Y[1 \ldots 15] \rangle$, $\langle X[1 \ldots 12], Y[1 \ldots 14] \rangle$, $\langle X[6 \ldots 12], Y[4 \ldots 14] \rangle$, $\langle X[6 \ldots 7], Y[4 \ldots 15] \rangle$, $\langle X[6 \ldots 7], Y[6 \ldots 15] \rangle$,

Algorithm 1. BasicLCSp

Input: X and Y: Two strings; C: A set of common substrings; τ: A given penalty threshold
Output: $lcsp(X, Y, \tau)$

```
1  Common substrings C_set ← CalComStr(X, Y);
2  Rank strings in C_set in the order of their start positions in ascending order;
3  k ← number of common substrings in C_set;
4  for i = 1; i < k; i++ do
5      str_c ← the i-th common substring (X^a, Y^b, l_i) in C_set;
6      for j = i + 1; j ≤ k; j++ do
7          Let the j-th common substring be (X^c, Y^d, l_j);
8          if a < c && b < d then
9              Generate a candidate substring X' start from str_c and end at the
               j-th common string in C_set;
10             if penalty of X' ≤ τ then
11                 Can ← X';
12                 str_c ← X';
13 return the longest string in Can;
```

$\langle X[2 \ldots 7],\ Y[8 \ldots 15]\rangle$, and $\langle X[2 \ldots 12], Y[8 \ldots 14]\rangle$. The algorithm keeps the concatenated substring as a candidate if its penalty $\leq \tau$. Finally it returns the longest candidate as $lcsp(X, Y, \tau)$.

The algorithm BasicLCSp is correct. Any LCS of two strings X and Y must contain their common substrings and it must start from one common substring and end at another common substring. The time complexity of BasicLCSp is $O(k^2)$, where k is the number of common substrings of X and Y.

5 Reducing Number of Concatenated Common Substrings

The algorithm BasicLCSp enumerates all possible concatenated common substrings. Some of them will not generate the LCSp. To locate LCSp efficiently, we propose a *filter-refine* approach, called ImprovedLCSp. We first analyze the feature of LCSp, based on which we carefully prune those common substrings that could not generate LCSp. We propose one filtering in Sects. 5.1 and an early termination approach to avoid useless calculations in Sect. 5.2.

5.1 Avoiding Useless Concatenation of Common Substrings

property 1. Let A be an alignment of an LCSp of X and Y under the penalty threshold τ. For any two common substrings C_1 with (X^a, Y^b, l_1) and C_2 with (X^c, Y^d, l_2) in A $(a < c, b < d)$. The penalty of the concatenated substring pair $\langle X[a, c+l_2-1], Y[b, d+l_2-1]\rangle$ must satisfy $p(X[a, c+l_2-1], Y[b, d+l_2-1]) \leq \tau$.

Lower Bound of Penalty. Let C_1 with (X^a, Y^b, l_1) and C_2 with (X^c, Y^d, l_2) be two common substrings of strings X and Y, if there does not exist any common substring C with (X^e, Y^f, l_3) $(a < e < c, b < f < d)$, the lower bound of penalty of concatenating C_1 and C_2 is

$$LB(C_1, C_2) = \min(\alpha, \beta) \cdot \max(c - a - l_1, d - b - l_1). \tag{2}$$

Theorem 1. *Two common substrings C_1 with (X^a, Y^b, l_1) and C_2 with (X^c, Y^d, l_2) cannot belong to the same alignment of an* LCSp *if there does not exist any common substring C with (X^e, Y^f, l_3) $(a < e < c, b < f < d)$ and $LB(C_1, C_2) > \tau$*

Proof. Assume C_1 and C_2 belong to the same alignment when Eq. 2 holds. Since there does not exist any common substring C with (X^e, Y^f, l_3) $(a < e < c, b < f < d)$, we let $\langle X_i, Y_i \rangle$ be the uncommon substring pair in between C_1 and C_2, then according to Eq. 1, we know $\alpha \cdot M(X_i, Y_i) + \beta \cdot S(X_i, Y_i) \leq \tau$. Since $\min(\alpha, \beta) \max(|X_i|, |Y_i|) \leq M(X_i, Y_i) + \beta \cdot S(X_i, Y_i)$, and $|X_i| = c - a - l_1$, $|Y_i| = d - b - l_1$, we can see the above assumption does not hold.

Based on Theorem 1, we can prune concatenated substrings that satisfy Eq. 2. For example, let $\alpha = 1, \beta = 1$. Given $\tau = 5$, it is useless to concatenate the two common substrings with $(X^6, Y^4, 2)$ and $(X^7, Y^{15}, 1)$ since $\min(\alpha = 1, \beta = 1) \cdot \max(7 - 6 - 2, 15 - 4 - 2) = 9 > \tau$. The same reason, we do not concatenate $(X^1, Y^1, 3)$ with $(X^7, Y^{15}, 1)$, and $(X^6, Y^6, 2)$ with $(X^7, Y^{15}, 1)$.

5.2 Early Termination of Calculations

Since we want to find the longest common subsequences with limited penalty, we prefer to check two common substrings with longest position distance. Therefore, instead of storing common substrings in an ordered set, we store them in a lattice such that we can use it to easily prune useless concatenation and early terminate the calculation of LCSp.

A Lattice Structure. We use a lattice to store all common substrings of X and Y. Each node in the lattice represents a common substring. Consider any two common substrings with (X^a, Y^b, l_1) and (X^c, Y^d, l_2). If $a < c$ and $b < d$, we call (X^a, Y^b, l_1) *dominates* (X^c, Y^d, l_2). Furthermore, if there does not exist any other common substring with (X^e, Y^f, l_3) such that $a < e < c$, $b < f < d$, then there is an edge from (X^a, Y^b, l_1) to (X^c, Y^d, l_2), we call (X^a, Y^b, l_1) *strictly dominates* (X^c, Y^d, l_2). We label the edge between any two common substrings with strictly dominate relationship using the penalty of its corresponding uncommon substring.

Figure 2 shows an example of the lattice for our running example. There is an edge in between $(X^1, Y^1, 3)$ and $(X^6, Y^4, 2)$, and the edge is labelled 2 since the penalty of concatenating these two common substrings is 2. Therefore, the penalty of concatenating strings along the path $(X^1, Y^1, 3)$, $(X^6, Y^4, 2)$, and $(X^7, Y^{15}, 1)$ is $2 + 7 = 9$.

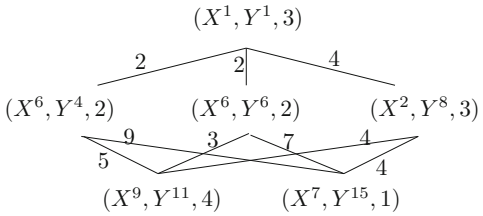


Fig. 2. A lattice of common substrings of $X = \texttt{traobcybgsfd}$ and $Y = \texttt{tracycyraogsfdy}$ when $\alpha = 1$ and $\beta = 1$.

Pruning Useless Concatenation Using the Lattice for Common Substrings. From the above example, we can see that the lattice structure can easily identify the strict dominate relationship between any two common substring pairs. When the summation of labels in a path is greater than τ, we do not need to concatenate common substrings along the path.

Algorithm 2 shows a pruning algorithm based on depth-first-search (DFS). For every search step, it adjusts the permitted penalty score so that we could avoid traversing those paths that could not generate LCSp.

Algorithm 2. PRUNE(L, τ)

Input: A lattice L for common substrings of X and Y, penalty threshold τ;
Output: A pruned lattice;

```
// start from the root of L and traverse L using DFS
if L is a single node then
    return L;
foreach node v pointed by the root r of L do
    if edge(r, v) > τ then
        remove the edge from r to v in L;
    PRUNE(L, τ − edge(r, v));
```

Choosing a Good Calculation Order. In fact, we are only interested in the longest common subsequences whose penalty is not greater than τ, therefore, it is no need to calculate those common subsequences with shorter lengths.

Aiming at this target, we reorganize children of each node in the lattice by ranking their lengths in descending order. Then by using the DFS, the path with longest untraversed common strings will take precedence. When the first $lcsp(X, Y, \tau)$ is found, we are safe to early terminate all calculations since the later calculations can only generate a common subsequence with shorter length.

6 Efficiently Constructing Common Substrings

We can use dynamic programming to get all common substrings of X and Y in $O(mn)$ time. In order to accelerate this process, we can also use the suffix tree [20]

0	1	2	3	4	5	6	7	8	9	10	11
a	b	f	a	b	$\#_1$	a	b	e	a	b	$\#_2$

(a) Strings with the text array.

i	$SA[i]$	SA^{-1}	$LCP[i]$	$text[i]$
0	5	5	0	1
1	11	9	0	2
2	3	11	0	1
3	9	2	2	2
4	6	6	2	2
5	0	0	2	1
6	4	4	0	1
7	10	8	1	2
8	7	10	1	2
9	1	3	1	1
10	8	7	0	2
11	2	1	0	1

(b) The SA and LCP array for T=abfab$\#_1$abeab$\#_2$.

Fig. 3. An example of SA array.

or suffix array [5]. Compared with the suffix tree, suffix array can be configured in linear time [9] and small space cost [7], so we consider the establishment of suffix array index structure.

Given a string X, its suffix array SA records the start positions of all the suffixes of one string. Since the suffixes are sorted lexicographically, $SA[i]$ is the start position of the i-th suffix based on the lexicographical order. The suffix array of string T, denoted as SA, is actually an array with integer from 1 to n, revealing the dictionary order of n suffixes. $T_{SA[i]}$ denotes the $SA[i]$th suffix $T[i \ldots n]$. The inverse SA^{-1} of suffix array is also an integer array, satisfying $SA^{-1}[SA[i]] = i$ $(1 \leq i \leq n)$. Obviously, the inverse of suffix array can also be constructed in linear time.

LCP array is used for maintaining the length of the longest common prefix of two adjacent suffix in SA. Suppose we use $lcp(u, v)$ to denote the length of the longest common prefix of u and v, then $LCP[1] = 0$ and $LCP[i] = lcp(T_{SA[i-1]}, T_{SA[i]})$ where $2 \leq i \leq n$. Based on the suffix array and its inverse, this LCP array can be constructed in linear time [6].

Given two strings X and Y, we add $\#_1$ and $\#_2$ to their ends, respectively, forming a new string $T = X\#_1Y\#_2$. Suppose $\#_1 < \#_2$, and all characters in the string collection are larger than these separators according to dictionary order. We define the suffix array of the new string T and the related LCP array as the generalized augmented suffix array [1], which is consistent with the suffix arrays of X and Y. This approach can be done in $O(m + n)$ time.

For example, let $X =$ `abfab` and $Y =$ `abeab`, then $T =$ `abfab`$\#_1$`abeab`$\#_2$, with their index starting from 0 as shown in Fig. 3(a). The SA array, SA^{-1} array and LCP array of T are all shown in Fig. 3. The position and length of the common substring in the outer matrix can be figured out according to the LCP array,

which reduce the computation of mismatch in outer matrix. We can compute four common substrings ab with $(X^0, Y^6, 2)$, $(X^0, Y^9, 2)$, $(X^3, Y^6, 2)$, and $(X^3, Y^9, 2)$ in linear time, with the help of LCP array and the dynamic programming programming for common substrings of X and Y.

7 Experiments

In this section, we evaluated the effect of the different factors on the performance and used the following three data sets in the experiments.

- **Genome data set.** This data set contains human's first genome data, from which we randomly selected 1000 strings of various lengths as data strings. The query strings were generated similarly from mice genome data. We generated a query workload with 50 query strings.
- **DBLP data set.** We generated this data set from DBLP. It includes 1,632,442 papers, and each paper contains some of the properties of paper, such as title, author, abstract, and etc. We randomly selected 1000 strings of various lengths as data strings and randomly picked up 50 strings to construct a query workload.
- **AOL query log data set.** It contains the web pages from a large number of users Query records sorted by anonymous user IDs. Each record includes anonymous user ID, the contents of the query, and query time. The length of the records are from 20 and 100. We randomly chose 50 contents of queries to construct its query workload.

Our experimental results were run on Ubuntu (Linux) 13.10 with Interl (R) Core (TM) i7 CPU 870@2.93 GHZ 8 GB RAM. All the algorithms were implemented using GNU C++.

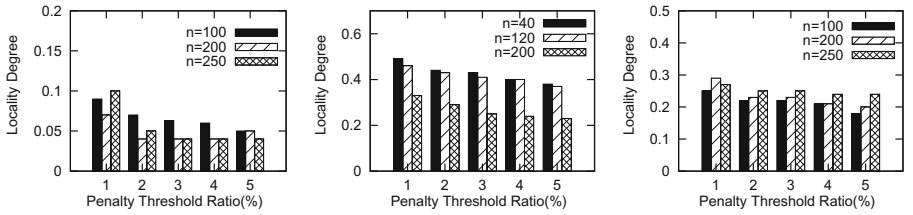
7.1 Evaluation of Effectiveness

We define *Locality Degree LD* to evaluate the effectiveness of LCSP as follows.

$$LD = \frac{avg(\sum l(lcs_p))}{avg(\sum l(lcs) - \sum l(lcs_p))},$$

where $l(lcs_p)$ represents the length of matching substrings generated by using LCSP, $l(lcs)$ represents the length of matching substrings generated by using LCS, and $avg(\cdot)$ is the average value. Notice that, we require that both LCSP and LCS generate the same longest common subsequences under the given penalty threshold.

Figure 4 shows the Locality Degree when increasing the penalty threshold ratio, which is the percentage of average data string length. We can see when getting the same longest common subsequence, LCSP prefers to find meaningful matching substrings with shorter lengths. Figure 4(a) shows the locality degree LD was very low, only less than 0.1 on DNA data set, which means that LCS



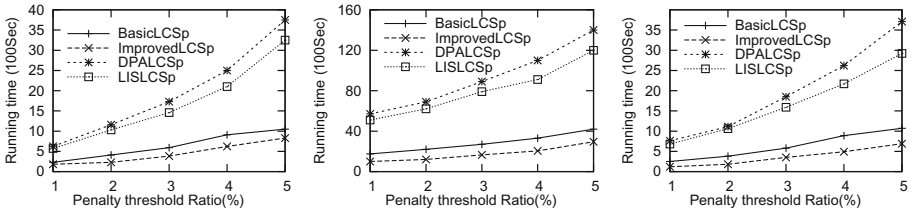
(a) DNA. (b) DBLP. (c) AOL query log.

Fig. 4. Effectiveness of LCSp.

generates much more meaningless results compared with LCSp. The locality degree on DBLP data set was less than 0.5, which was higher than it on DNA data set, since the selectivity of DBLP data was much less than the selectivity of DNA data. As the penalty threshold ratio increased, the locality degree on three data sets decreased since the smaller penalty threshold was, the more locality was required.

7.2 Comparison with Other Algorithms

We chose two state-of-the-art LCS algorithms DPA [2] and LIS [1], and modified them to support LCSp as discussed in Sect. 4. We call these modified LCS-based algorithms DPALCSp and LISLCSp. We compared the running time of our two algorithms (i.e. BasicLCSp and ImprovedLCSp) with these two LCS-based algorithms.



(a) DNA. (b) DBLP. (c) AOL query log.

Fig. 5. Comparison of different algorithms.

Figure 5 shows the comparison results of the four algorithms. We can see that the algorithm DPALCSp was the slowest algorithm. Both BasicLCSp and ImprovedLCSp algorithms ran much faster than LCS-based algorithms. When increasing the penalty threshold ratio, the running time increased. This is consistent with our expectation since more concatenated common substrings would be generated when the penalty threshold increases, incurring larger time cost for calculation. The running time of our algorithms kept more stable than both DPALCSp and LISLCSp since our algorithms can generate the longest

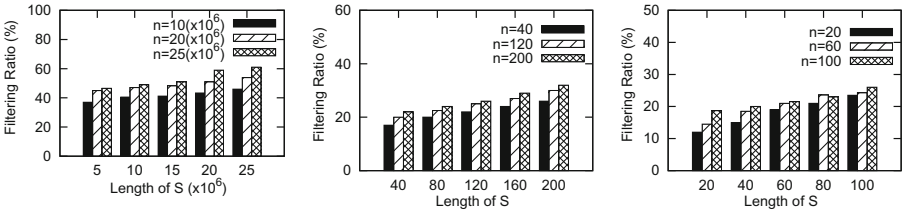
common subsequences directly based on common substrings, whereas the LCS-based algorithms had to back chasing all possible alignments to calculate results, which were costly.

7.3 Evaluation of LCSP

In order to compare the effects of the query filtering among different algorithms, we define *Filtered Ratio* (FR) and *Early Terminate Ratio* (ETR) as follows. Filtered Ratio (FR) is the proportion that a number of pruned concatenated common substrings to the whole number of concatenated common substrings. Early Terminate Ratio (ETR) is the proportion that traversed nodes to the nodes in the lattice.

To measure the performance of algorithms, we take filtered ratio, early terminate ratio, and running time as the three metrics to evaluate pruning power, effect of early termination, and efficiency of our algorithms, respectively. It is obvious that a favored algorithm with high efficiency should have large filtered area ratio, larger early terminate ratio, and small running time. We use the scoring scheme $\alpha = 1$ and $\beta = 1$. We got similar results when varying α and β.

Pruning Power. We conducted experiments on strings with different lengths. The lengths of query string and data string were comparable. The detailed filtering ratios for three data sets are shown in Fig. 6. In Fig. 6(a), when the length of one string is fixed to 20×10^6, and the length of another string increases from 5×10^6 to 25×10^6, the filtering ratio is significant, increasing from 36.9% to 42.7%. Also, by comparing results on three different data sets in Fig. 6, we can see that the filtering ratios raise when increasing the lengths of strings.



(a) DNA. (b) DBLP. (c) AOL.

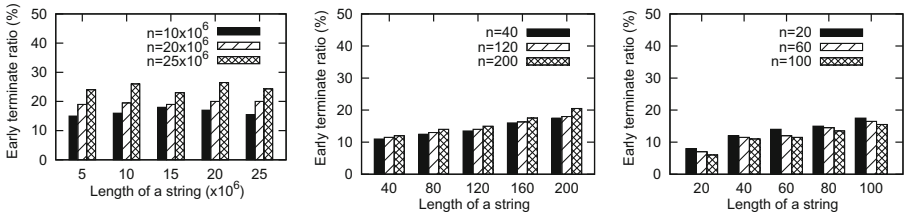
Fig. 6. Pruning power.

Effect of Early Termination. The effect of early termination is related to string length, as well as the penalty threshold. Therefore we evaluated the impact of these two factors in this section.

Figure 7 shows how early terminate ratio would be affected with increasing string lengths. From Fig. 7, we can see that when the lengths of strings are generally comparable, the early terminate ratio increases with the increase of string

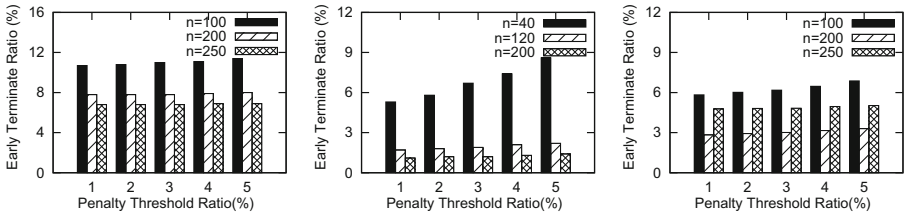
length. The reason is that, for both strings, longer strings generally indicate larger probabilities of more common substrings.

Figure 8 shows the impact of the variance of penalty threshold ratios on the early termination. To be more specific, seen from Fig. 8, when string length is fixed, the larger the penalty threshold, the greater the early terminate ratio, especially for DBLP and ALO query log data sets (see Fig. 8(b) and (c)). It is because the larger the penalty threshold, the more the candidate starting from the same common substring, leading to a large number concatenated substrings, thus a larger early terminate ratio. Notice that, the early terminate ratio did not increase significantly when increasing the penalty threshold ratio since the distribution of frequencies for different substrings were similar, therefore, the number of concatenated substrings kept stable when increasing the penalty threshold ratio.



(a) DNA. (b) DBLP. (c) AOL query log.

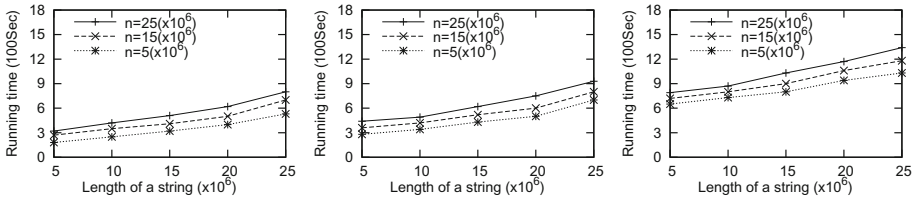
Fig. 7. Effect of early termination with different string lengths.



(a) DNA. (b) DBLP. (c) AOL query log.

Fig. 8. Effect of early termination with different penalty threshold.

Running Time. We also test the efficiency of our algorithms when varying the lengths of strings. Figure 9 reports the running time of ImprovedLCSp for different lengths of strings on DNA sequences when the penalty threshold $\tau = 2,6,10$, respectively. The results on DBLP and AOL query log data sets are similar.



(a) $\tau = 2\%|s|$. (b) $\tau = 6\%|s|$. (c) $\tau = 10\%|s|$.

Fig. 9. The performance of ImprovedLCSp on DNA data set.

Figure 9(a) shows when $\tau = 2\%$ of string length, with the increase of the lengths of strings, the running time also increased from 259 s to 676 s. As can be seen in Fig. 9(b), when $\tau = 6\%$ of string length, with the increase of the lengths of strings, the running time increased from 380 s to 814 s. In Fig. 9(c), it can be seen that when $\tau = 10\%$ of string length, the running time increased from 722 s to 1,183 s. In a word, when τ is fixed, the running time of the algorithm ImprovedLCSp is linear to the lengths of strings.

8 Conclusion

In this paper, we propose a new problem, the longest common subsequence with limited penalty to get LCSs with good locality. We show that the existing LCS-based algorithms are not efficient since they have to back chasing alignments to do verifications. In order to avoid checking each generated LCS using the penalty threshold, we propose an approach based on common substrings. By improving the basic algorithm, we propose a filter-refine approach that can reduce the number of concatenated common substrings. It can efficiently prune useless concatenations of common substrings and early terminate calculations. Our experimental study demonstrate its effectiveness and efficiency.

References

1. Arnold, M., Ohlebusch, E.: Linear time algorithms for generalizations of the longest common substring problem. Algorithmica **60**(4), 806–818 (2011)
2. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, pp. 39–48, 27–29 September 2000
3. Brodal, G.S., Kaligosi, K., Katriel, I., Kutz, M.: Faster algorithms for computing longest common increasing subsequences. In: Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006, Barcelona, Spain, pp. 330–341, 5–7 July 2006
4. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Commun. ACM **18**(6), 341–343 (1975)

5. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003). doi:10.1007/3-540-45061-0_73
6. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001, Jerusalem, Israel, pp. 181–192, 1–4 July 2001
7. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003, Morelia, Michocán, Mexico, pp. 186–199, 25–27 June 2003
8. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
9. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discrete Algorithms **3**(2–4), 143–156 (2005)
10. Korf, I., Yandell, M., Bedell, J.A.: BLAST - An Essential Guide to the Basic Local Alignment Search Tool. O'Reilly, Sebastopol (2003)
11. Lam, T.W., Sung, W., Tam, S., Wong, C., Yiu, S.: Compressed indexing and local alignment of DNA. Bioinformatics **24**(6), 791–797 (2008)
12. Levenshtein, V.I.: Binary codes capable of correcting spurious insertions and deletions of ones. Probl. Inf. Transm. **1**(1), 817 (1965)
13. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. **20**(1), 18–31 (1980)
14. Meek, C., Patel, J.M., Kasetty, S.: OASIS: an online and accurate technique for local-alignment searches on biological sequences. In: VLDB, pp. 910–921 (2003)
15. Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica **1**(2), 251–266 (1986)
16. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. Acta Inf. **18**, 171–179 (1982)
17. Overill, R.E.: Book review: "time warps, string edits, and macromolecules: the theory and practice of sequence comparison" by David Sankoff and Joseph Kruskal. J. Log. Comput. **11**(2), 356 (2001)
18. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
19. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)
20. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, pp. 1–11, 15–17 October 1973
21. Yang, X., Liu, H., Wang, B.: ALAE: accelerating local alignment with affine gap exactly in biosequence databases. PVLDB **5**(11), 1507–1518 (2012)