Solving the Longest Common Subsequence Problem with the Lost Score Scheme

Kuo-Tsung Tseng[a] and Chang-Biau Yang[b*]

[a]Department of Shipping and Transportation Management
National Kaohsiung Marine University, Kaohsiung, Taiwan
tsengkt@nkmu.edu.tw

[b]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
cbyang@cse.nsysu.edu.tw

Abstract

Given two sequences A and B of lengths m and n respectively, the longest common subsequence (LCS) problem is to find the common subsequence of both A and B with the maximum length. Traditionally, an LCS lattice is built, where $LCS_{i,j}$ represents the LCS length of $A_{1..i}$ and $B_{1..j}$. In this paper, we present a novel concept, the lost score matrix *(LSM), for solving the LCS problem. $LSM_{i,j}$ is defined as the score that has been lost for calculating the LCS length of $A_{1..i}$ and $B_{1..j}$. Though the time complexity of our algorithm is $O(mn)$ in the worst case, our computation time is reduced if A and B are very similar. Besides, we can filter out quickly those sequences which are not so similar.*

1 Introduction

It is crucial to measure the similarity of two sequences in many applications, such as voice recognition, pattern matching, plagiarism detection and computational biology. The *longest common subsequence* [2–7, 11, 12] problem and the *edit distance* [1, 10] problem are often applied to the similarity measurement of two sequences and they have been extensively studied for several decades.

Given two sequences A and B, with lengths m and n respectively, where $m \leq n$, the *longest common subsequence* (LCS) problem is to find the common subsequence of both A and B with the maximum length. Here, a subsequence of sequence S is a sequence that can be obtained by deleting arbitrary number of elements at arbitrary positions of S.

Finding the minimum number of editing operations (insertion, deletion and substitution) to transform sequence A into B is defined as the *edit distance* problem [1, 10, 13]. If the cost of one insertion or one deletion is 1, and the cost of one substitution is 2 (viewed as one insertion plus one deletion), then this special edit distance of two sequences A and B can be obtained by their LCS length with the formula $E = m+n-2L$, where E denotes the edit distance and L denotes the LCS length.

To solve the alignment problem of DNA sequences or protein sequences, Needleman and Wunsch first defined the LCS problem, and they designed a primitive algorithm for solving it [9]. The time complexity of the primitive algorithm is $O(mn(m+n))$. Then several improved algorithms were proposed. The most famous dynamic approach for solving the LCS problem was proposed by Hirschberg in 1975 [4]. The time complexity is $O(mn)$. Besides, Hirschberg also proposed a divide and conquer approach with $O(mn)$ time and $O(m+n)$ space. To reduce the computational time, Hunt and Szymanski collected the matching pairs of the two input sequences and then invoked the algorithm for solving the longest increasing subsequence problem [6]. The time complexity is $O((R+n)\log n))$, where R denotes the number of matching pairs. Since only matching pairs are considered, this algorithm is more efficient if R is small. On the other hand, if the two input sequences are very similar, then R becomes large

*Corresponding author (Chang-Biau Yang).

and hence the efficiency decays dramatically.

To calculate the LCS of two similar sequences, Nakatsu *et al.* proposed an algorithm with time complexity $O(n(m-L))$ [8], where L denotes the LCS length. As we can see, when the input sequences are similar, L will be large and thus the required time becomes little. If we know in advance that the two input sequences are similar, the algorithm can run efficiently. However, if the sequence similarity is low, the algorithm will become inefficient.

Imagine the scenario that among a set of sequences, our goal is to find similar sequence pairs. In this scenario, we have no idea about the sequence similarity in advance. The LCS algorithm of Nakatsu *et al.* cannot be applied well since the similarity may be high, and may also be low. It is a dilemma for deciding whether the algorithm of Nakatsu *et al.* is used or not. Inspired by this scenario, in this paper, we propose a different idea for solving the LCS problem. Traditionally, the LCS algorithms are based on the concept that the gained score is accumulated gradually, and finally the gained score represents the similarity. On the contrary, we calculate the lost score of the two input sequences. Initially, the lost score is set to zero. And, the lost score is also increased gradually during the execution of the algorithm. With this idea, we can preset a threshold for determining whether two sequences are similar or dissimilar. When the lost score exceeds the threshold, the two sequences will become dissimilar and the algorithm terminates. In other words, once the algorithm decides that the two sequences are dissimilar, it no longer calculates the final LCS length and thus the computation time can be reduced. Besides, if the two sequences are similar, the LCS length can be found more quickly. Note that time complexity of our algorithm is still $O(mn)$ in the worst case.

The rest organization of this paper is as follows. In Section 2, some preliminaries are given. In Section 3, we present our lost score scheme for solving the LCS problem. Finally, we give our conclusions in Section 4.

2 Preliminaries

A sequence S is denoted by $s_1s_2s_3\dots s_l$, and $|S|$ represents its length ($|S| = l$). s_i denotes the ith element of S. In addition, let $S_{i..j} = s_is_{i+1}s_{i+2}\dots s_j$ denote a substring from the ith element to the jth element of S.

Given two sequences $A = a_1a_2a_3\dots a_m$ and $B = b_1b_2b_3\dots b_n$ of lengths m and n, respectively, where $m \leq n$. Let $LCS_{i,j}$ represent the LCS length of $A_{1..i}$ $(1 \leq i \leq m)$ and $B_{1..j}$ $(1 \leq j \leq n)$. The dynamic programming approach for calculating the LCS length [4] is given in Equation 1.

Boundry Conditions
$LCS_{i,j} = 0$, **if** $i = 0$ **or** $j = 0$

If $a_i = b_j$
$$LCS_{i,j} = LCS_{i-1,j-1} + 1 \qquad (1)$$

Otherwise
$$LCS_{i,j} = \max\begin{cases} LCS_{i,j-1} \\ LCS_{i-1,j} \end{cases}$$

Table 1 shows an example of the LCS lattice for computing the LCS length of sequences $A = cecedec$ and $B = fecdfddec$.

Table 1: The LCS lattice of sequences $A = cacadac$ and $B = bacdbddac$.

		0	1	2	3	4	5	6	7	8	9
			f	e	c	d	f	d	d	e	c
0		0	0	0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1	1	1
2	e	0	0	1	1	1	1	1	1	2	2
3	c	0	0	1	2	2	2	2	2	2	3
4	e	0	0	1	2	2	2	2	2	3	3
5	d	0	0	1	2	3	3	3	3	3	3
6	e	0	0	1	2	3	3	3	3	4	4
7	c	0	0	1	2	3	3	3	3	4	5

Nakatsu *et al.* proposed an algorithm with time complexity O($n(m-L)$) for solving the LCS problem [8], where L denotes the LCS length of A and B. In their paper, the closer m and L are, the quicker their algorithm will do. In other words, if two given sequences are very similar, the L will be determined quickly.

We use an example to illustrate the algorithm of Nakatsu *et al.* Suppose sequences $A = cecedec$ and $B = fecdfddec$, where $m = 7$ and $n = 9$. A simplified result of Nakatsu's algorithm is shown in Table 2. Here, we have modified their algorithm for ease of comparing with the traditional lattice.

The yth element of round x with value z ($M_{x,y} = z$) in Table 2 represents that the LCS length of $A_{1..x+y-1}$ and $B_{1..z}$ is y, where $B_{1..z}$ is the shortest prefix of B satisfying that. For example, $M_{1,3} = 9$ since $B_{1..9} = fecdfddec$ is the shortest prefix of B that has a common subsequence of length 3 with $A_{1..1+3-1} = cec$. As another example, $M_{2,3} = 8$ since $B_{1..8} = fecdfdde$ is the shortest prefix of B that has a common subsequence of length 3 with $A_{1..2+3-1} = cece$.

It is easy to compute round 1 in Table 2. First, $a_1 = c$, so the algorithm finds the first c in B which is b_3. Second, $a_2 = e$, so the algorithm finds the first e after b_3 in B which is b_8. Then, $a_3 = c$, the 3rd element in round 1 is b_9. Now it is impossible to go any further. Thus, round 1 terminates.

Round 2 begins with $a_2 = e$. The algorithm finds the first e in B which is b_2, and 2 is smaller than $M_{1,1} = 3$, so $M_{2,1} = 2$.

Round 3 starts with $a_3 = c$. The first c in B is b_3, but 3 is larger than than $M_{2,1} = 2$, so $M_{3,1} = 2$. The final round (round 3) in Table 2 is $2, 3, 4, 8, 9$, which means that $b_2b_3b_4b_8b_9 = ecdec$ is the LCS of A and B.

Table 2: A simplified result of sequences $A = cecedec$ and $B = fecdfddec$ by using the algorithm of Nakatsu *et al.*

round	1	2	3	4	5	(max length)
1	3	8	9			
2	2	3	8			
3	2	3	4	8	9	

3 Lost Score Matrix

In this section, we propose the *lost score matrix* (LSM) to denote the score that has been lost at position (i, j) of the lattice for calculating the LCS length of sequences A and B. It is clear that the possible maximum LCS length of A and B at position (i, j) is $LCS_{i,j} + \min(m - i, n - j)$. Here, $LCS_{i,j}$ means the accumulated LCS length and $\min(m - i, n - j)$ means the possible maximum length we may get in the future. Accordingly, the definition of LSM of $A_{1..i}$ and $B_{1..j}$, denoted as $LSM_{i,j}$, is given in Equation 2.

$$LSM_{i,j} = m - LCS_{i,j} - \min(m - i, n - j), \quad (2)$$

where $|A| = m$, $|B| = n$ and $m \leq n$.

By the above definition, we illustrate an example of LSM as shown in Table 3, which uses the same example as Table 1.

Table 3: The LSM of sequences $A = cecedec$ and $B = fecdfddec$.

		0	1	2	3	4	5	6	7	8	9
			f	e	c	d	f	d	d	e	c
0		0	0	0	1	2	3	4	5	6	7
1	c	1	1	1	0	1	2	3	4	5	6
2	e	2	2	1	1	1	2	3	4	4	5
3	c	3	3	2	1	1	1	2	3	4	4
4	e	4	4	3	2	2	2	2	3	3	4
5	d	5	5	4	3	2	2	2	2	3	4
6	e	6	6	5	4	3	3	3	3	2	3
7	c	7	7	6	5	4	4	4	4	3	2

Though the definition of LSM is based on the LCS length, we invoke dynamic programming to compute LSM directly, without the LCS information. ”The procedure for calculating the $LSM_{i,j}$ is given as follows.

Boundry Conditions
$LSM_{i,0} = i, \ 0 \leq i \leq m$
$LSM_{0,j} = \max(0, j - n + m), \ 1 \leq j \leq n$

If $a_i = b_j$
$LSM_{i,j} = LSM_{i-1,j-1}$

Otherwise
$X = (m - i) - (n - j)$

$$LSM_{i,j} = \begin{cases} \min\begin{cases} LSM_{i,j-1} \\ LSM_{i-1,j} + 1 \end{cases} & \text{if } X < 0 \\ \min\begin{cases} LSM_{i,j-1} \\ LSM_{i-1,j} \end{cases} & \text{if } X = 0 \\ \min\begin{cases} LSM_{i,j-1} + 1 \\ LSM_{i-1,j} \end{cases} & \text{if } X > 0 \end{cases}$$

(3)

The correctness is given in the following theorem.

Theorem 1. *Equation 3 correctly calculates the LSM defined by Equation 2.*

Proof. With Equations 1 and 2, we have following.

(i) Boundary Conditions
$LCS_{i,j} = 0$ when $j = 0$, and $\min(m-i, n) = m-i$ because $m \leq n, \ 0 \leq i \leq m$. We have
$LSM_{i,0} = m - 0 - \min(m - i, n)$
$= m - 0 - (m - i) = i, \ 0 \leq i \leq m.$

$LCS_{i,j} = 0$ when $i = 0$.

$$\min(m, n-j) = \begin{cases} m & 0 \le j \le n-m \\ n-j & n-m+1 \le j \le n \end{cases}$$

$LSM_{0,j} = m - 0 - \min(m, n-j)$
$= \max(0, j-n+m),\ 1 \le j \le n.$

(ii) If $a_i = b_j$
$LCS_{i,j} = LCS_{i-1,j-1} + 1.$
$LCS_{i-1,j-1}$
$= m - LSM_{i-1,j-1} - \min(m-(i-1), n-(j-1)).$
$LSM_{i,j} = m-(LCS_{i-1,j-1}+1)-\min(m-i, n-j)$
$= m - (m - LSM_{i-1,j-1} - \min(m-(i-1), n-(j-1)) + 1) - \min(m-i, n-j)$
$= LSM_{i-1,j-1}.$

(iii) Otherwise
$LSM_{i,j}$

$$= m - \max\begin{cases} LCS_{i,j-1} \\ LCS_{i-1,j} \end{cases} - \min(m-i, n-j)$$

$$= \min\begin{cases} LSM_{i,j-1} + \begin{cases} 0 & \text{if } m-i \le n-j \\ 1 & \text{if } m-i > n-j \end{cases} \\ LSM_{i-1,j} + \begin{cases} 0 & \text{if } m-i \ge n-j \\ 1 & \text{if } m-i < n-j \end{cases} \end{cases}$$

$$= \begin{cases} \min\begin{cases} LSM_{i,j-1} \\ LSM_{i-1,j}+1 \end{cases} & \text{if } m-i < n-j \\ \min\begin{cases} LSM_{i,j-1} \\ LSM_{i-1,j} \end{cases} & \text{if } m-i = n-j \\ \min\begin{cases} LSM_{i,j-1}+1 \\ LSM_{i-1,j} \end{cases} & \text{if } m-i > n-j \end{cases}$$

Therefore, this theorem holds.

■

The traditional order for calculating the LCS lattice usually uses the row-major order or the column-major order. If we follow the traditional order, we have to compute the whole LSM. Observing the values in LSM of Table 3, one can find that the upper right part and the lower left need not be calculated, since the lost scores in these parts are too large to get the optimal LCS score. Thus, we can use the *lost score order* (LSO) to compute the elements in LSM. In this order, the lost scores are computed with the order from 0 to m.

By examining Equation 3, there is one fact that elements of value k in LSM can only be derived from the elements of value $k-1$ or k, $1 \le k \le m$. Since the elements of value 0 can only be derived from those elements of value 0, and elements of value 1 can only be derived by elements of value 0 or 1 and so on, it is clear to see that LSO is right.

Our algorithm for computing $LSM_{m,n}$ with LSO is shown in Algorithm 1. Table 4 shows the LSM calculation of sequences $A = cacadac$ and $B = bacdbddac$ with LSO. Obviously, the higher similarity of sequences A and B, the quicker Algorithm 1 will be.

Table 4: The LSM with LSO of sequences $A = cecedec$ and $B = fecdfddec$. Elements with question marks mean that the answer $LSM_{m,n}$ has been obtained before those elements were computed.

		0	1	2	3	4	5	6	7	8	9
			f	e	c	d	f	d	d	e	c
0		0	0	0	1	2	3	4	5	6	7
1	c	1	1	1	0	1	2	?	?	?	?
2	e	2	2	1	1	1	2	3	?	?	?
3	c	3	3	2	1	1	1	2	?	?	?
4	e	4	?	?	2	2	2	2	3	?	?
5	d	5	?	?	?	2	2	2	2	?	?
6	e	6	?	?	?	?	3	3	3	2	?
7	c	7	?	?	?	?	?	?	?	?	2

4 Conclusion

In this paper, we propose a novel scheme, the lost score matrix, for solving the LCS problem. Though the improvement of time complexity may not be much, it is practical if sequences A and B are very similar, or if we desire to filter out those sequences which are not so similar. It should do a good job in plagiarism detection if we set lost score k as our detecting threshold. The proposed algorithm can be easily modified to stop when the lost score is greater than k, which means that it is not a case of plagiarism.

The idea of the lost score is novel, however, the required computation time is not reduced so much. This is because the decision of the lost score order is heavy overhead. We are working on the decision scheme. In addition, we are also studying on the scheme that only matching pairs are calculated.

References

[1] H.-Y. Ann, C.-B. Yang, Y.-H. Peng, and B.-C. Liaw, "Efficient algorithms for the block edit problems," *Information and Computation*, Vol. 208(3), pp. 221–229, 2010.

[2] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "A fast and simple algorithm for

Algorithm 1 Computing $LSM_{m,n}$ with LSO

```
Initiation:
for $i = 0 \to m$ do $LSM_{i,0} = i$
end for
for $j = 1 \to n$ do $LSM_{0,j} = \max(0, j - n + m)$
end for

$k = 0$, $R = 0$
while $LSM_{m,n}$ is unknown. do
    $l$=Index of leftmost element of value $k$ in row $R$
    $r$=Index of rightmost element of value $k$ in row $R$
    for $c = l \to r$ do
        if $LSM_{c+1,R+1}$ is unknown, then compute $LSM_{c+1,R+1}$
        end if
    end for

    if there exists any element of value k in row $R + 1$ then
        $R = R + 1$ // calculate the next row
    else
        $R = 0$
        $k = k + 1$ // calculate the next value
    end if
end while
```

computing the longest common subsequence of run-length encoded strings," *Information Processing Letters*, Vol. 108, pp. 360–364, 2008.

[3] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "Fast algorithms for computing the constrained lcs of run-length encoded strings," *Theoretical Computer Science*, Vol. 432, pp. 1–9, 2012.

[4] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Vol. 18, pp. 341–343, 1975.

[5] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng, "Efficient algorithms for finding interleaving relationship between sequences," *Information Processing Letters*, Vol. 105(5), pp. 188–193, 2008.

[6] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20(5), pp. 350–353, 1977.

[7] C. S. Iliopoulos and M. S. Rahman, "Algorithms for computing variants of the longest common subsequence problem," *Theoretical Computer Science*, Vol. 395, pp. 255–267, 2008.

[8] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similar text strings," *Acta Informatica*, Vol. 18, No. 2, pp. 171–179, 1982.

[9] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, Vol. 48, pp. 443–453, 1970.

[10] M. Pawlik and N. Augsten, "RTED: a robust algorithm for the tree edit distance," *Proceedings of the VLDB Endowment*, Vol. 5, No. 4, pp. 334–345, 2011.

[11] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor, "Efficient sparse dynamic programming for the merged LCS problem with block constraints," *International Journal of Innovative Computing, Information and Control*, Vol. 6, pp. 1935–1947, 2010.

[12] C.-T. Tseng, C.-B. Yang, and H.-Y. Ann, "Efficient algorithms for the longest common subsequence problem with sequential sub-

string constraints," *Journal of Complexity*, Vol. 29, pp. 44–52, 2013.

[13] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.