Research Article

An Efficient Algorithm for LCS Problem between Two Arbitrary Sequences

Yubo Li

School of Data Science and Computer, Sun Yat-sen University, Guangzhou, China

Correspondence should be addressed to Yubo Li; liyb5@mail2.sysu.edu.cn

Received 10 April 2018; Revised 17 August 2018; Accepted 14 October 2018; Published 29 November 2018

The longest common subsequence (LCS) problem is a classic computer science problem. For the essential problem of computing LCS between two arbitrary sequences $s_1$ and $s_2$, this paper proposes an algorithm taking $O(n + r)$ space and $O(r + n^2)$ time, where $r$ is the total number of elements in the set $\{(i, j) | s_1[i] = s_2[j]\}$. The algorithm can be more efficient than relevant classical algorithms in specific ranges of $r$.

1. Introduction

The longest common subsequence (LCS) problem is a classic computer science problem and still attracts continuous attention [1–4]. It is the basis of data comparison programs and widely used by revision control systems for reconciling multiple changes made to a revision-controlled collection of files. It also has applications in bioinformatics and many other problems such as [5–7]. For the general case of an arbitrary number of input sequences, the problem is NP-hard [8]. When the number of sequences is constant, the problem is solvable in polynomial time [9]. For the essential problem of computing LCS between two arbitrary sequences ($LCS_2$), the complexity is at least proportional to the product of the lengths of sequences according to the conclusion as follows.

It is shown that unless a bound on the total number of distinct symbols [author’s note: the size of alphabet] is assumed, every solution to the problem can consume an amount of time that is proportional to the product of the lengths of the two strings [9].

The sizes of lengths of sequences make the quadratic time algorithms impractical in many applications. Hence, it is significant to design more efficient algorithm in practice. This paper is confined to $LCS_2$ and is to present an algorithm that can be more efficient than relevant classical algorithms in specific scenarios.

The following introduction is also confined to the case of two input sequences. Chvátal and Sankoff (1975) proposed a Dynamic Programming (DP) algorithm of $O(n^2)$ space and time [10]. It is the basis of the algorithms for LCS problem. Soon in the same year, D.S. Hirschberg (1975) posted a Divide and Conquer (DC) algorithm that is a variation of the DP algorithm taking $O(n)$ space and $O(n^2 \log n)$ time [11]. In 2000, Bergroth, Hakonen, and Raita contributed a survey [12] that shows in the past decades there is no theoretically improved algorithm based on Hirschberg’s DC algorithm [11] as it is so brilliant. In 1977, Hirschberg additionally proposed an $O(pn + n \log n)$ algorithm and an $O(p(m + 1 - p) \log n)$ algorithm where $p$ is length of LCS [13]. The first one is efficient when $p$ is small, while the other one is efficient when $p$ is close to $m$. Both of the two algorithms are more suitable when the length of LCS can be estimated beforehand. Then, Nakatsu, Kambayashi, and Yajima (1982) in [14] presented an algorithm suitable for similar sequences and having bound of $O(n(m - p + 1))$ and $O(m(m - p + 1) \log n)$. Let the two sequences be $s_1$ and $s_2$. Same in 1977, Hunt and Szymanski proposed an algorithm taking $O(r)$ space and $O((r + n) \log n)$ time, where $r$ is the total number of elements in the set $\{(i, j) | s_1[i] = s_2[j]\}$ [15]. The algorithm reduces $LCS_2$ to longest increasing subsequence (LIS) problem. Apostolico and Guerra (1987) in [16] proposed...
an algorithm based on [15] taking time $O(n \log s + d \log \log n)$, where $d$ is the number of dominant matches (as defined by Hirschberg [13]) and $s$ is minimum of $n$ and the alphabet size. Further, based on [16], Eppstein (1992) in [17] proposed an $O(n \log s + d \log \log \min(d, nm/d))$ algorithm when the problem is sparse. If the alphabet size is constant, Masek and Paterson (1980) in [18] proposed an $O(n^2 / \log^2 n)$ algorithm utilizing the method of four Russians (1970) [19]; Abboud, Backurs, and Williams (2015) in [20] showed an $O(n^{2-\varepsilon})$ algorithm where $\varepsilon > 0$. $O(n^2 / \log^2 n) / \log^2 n$ algorithms are also proposed by Bille and Farach-Colton (2008) in [21] and Grabowsky (2014) in [22], each of which has its own prerequisite. Restrained by the conclusion of [9, 20], Backurs, and Williams (2015) in [20] showed an $O(\frac{n}{\log n})$ method of four Russians utilizing the method of four Russians (1970) [19].

This paper is organized as follows. In Section 1, the current state of algorithms for LCS problem between two sequences including LCS$_2$ is introduced. The proposed algorithm of this paper is presented and exemplified in Section 2, where preliminary terminologies needed to understand most of the paper and the theoretical basis of the proposed algorithm are also given. In Section 3, efficiency of the proposed algorithm is analyzed.

2. Algorithm

The longest common subsequence (LCS) is the longest subsequence common to all sequences in a set of sequences. This subsequence is not necessarily unique or not required to occupy consecutive positions within the original sequences (e.g., $fa^3$ is a longest common subsequence between $nfa^2$ and $fan^5$). LCS$(seq_1, seq_2)$ is a defined function that returns a set containing all the LCSes between two sequences, while the longest increasing subsequence (LIS) is a subsequence of a given sequence in which the subsequence’s elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique (e.g., $\{1,2,3\}$ is a longest increasing subsequence of $\{1,4,2,3\}$). LIS$(seq)$ is also a defined function that returns a set containing all the LISs of a sequence. Assume $s_1 = xnfaf$ and $s_2 = yfanf$. For all $s_1[i] = s_2[j]$, assume there is a sequence $l$, of which the elements are vectors in the form of $(i, j)$ (see Figure 1). The left part of an element of $l$ ($l[0]$) is the position of a symbol in $s_1$, and the right part of the element ($l[1]$) is the position of the symbol in $s_2$. $l$ is sorted according to $l[0]$ as the first key in ascending order and according to $l[1]$ as the second key in descending order. Define $(i_u, j_u, i_v, j_v) \in \mathcal{L}$, $i_u < i_v$ and $j_u < j_v$. Associating $\mathcal{L}(l)$ with LCS$(s_1, s_2$), it is bijective mapping between $\mathcal{L}(l)$ and LCS$(s_1, s_2$. Hence, LCS$(s_1, s_2)$ can be reduced to LIS$(l)$ [25].

According to the theoretical basis, Algorithm 1 is proposed for LCS$_2$. The algorithm is designed to reduce LCS$_2$ to LIS problem.

2.1. Example. Reuse $s_1 = xnfaf$, $s_2 = yfanf$ that are given previously. The process of computing LCSes between $s_1$ and $s_2$ using Algorithm 1 is illustrated in Figure 2 and presented as follows. $l[0]$ from left to right. The right part of $l[0]$ = (1, 3) is 3, $3 + 1 = 4$; then $s_2[4]$ is going to be computed. $s_2'[4][0] = s_2'[3][0] + 1 = 1$; $s_2'[4][1]$ is the position of (1, 3) in $l$; therefore $s_2'[4][1] = (1, 0)$. 

<table>
<thead>
<tr>
<th>$l$</th>
<th>$n$</th>
<th>$f$</th>
<th>$a$</th>
<th>$a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(1,3)</td>
<td>(2,4)</td>
<td>(2,1)</td>
<td>(3,5)</td>
</tr>
</tbody>
</table>

Figure 1: $s_1, s_2$ and the conceived new data $l$. 

\[
\begin{array}{c|ccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
s_1 & x & n & f & a & f \\
\hline
s_2 & y & f & a & n & f \\
\end{array}
\]
1. ALG (s1, s2) ⇔ n = |s2|
2. ▷ Step 1: Construct new data
3. s1, s2 ←→ l ⇔ r = |l|
4. ▷ Step 2: The main procedure
5. s2[0 . . n] ←→ (0, −)
6. pre[0 . . r − 1] ←→ -
7. end ←→ [ ]
8. for i = 0 to r − 1 do
9. s2[ξ[1] + 1][1] ←→ i ⇔ ξ = l[i][1]
10. s2[ξ[1] + 1][0] ←→ s2[ξ][0] + 1
11. pre[i] ←→ s2[ξ][1]
12. end ←→ s2[ξ + 1]
13. for j = ξ + 2 to r − 1 do
14. if s2[j][j][0] < s2[ξ + 1][0]
15. s2[j][0] ←→ s2[ξ + 1]
16. else
17. break
18. ▷ Step 3: Compute LIS of I
19. l, pre, end ←→ lis ▷ lis ∈ LCS(l)
20. ▷ Step 4: Compute LCS between s1 and s2
21. s1 or s2, lis ←→ les ▷ les ∈ LCS(s1, s2)
22. return les

Algorithm 1: Algorithm proposed in this paper.

The right part of s2[4] is 0; then pre[0] = s2[3][1][1]. end records the information of s2[4][1] = (1, 0).

end : [1] → 0

Then, s2[5][0] < s2[4][0]; therefore s2[5] = s2[4]; s2[6][0] < s2[4][0]; therefore s2′[6] = s2′[4].

For l[1] = (2, 4), the right part of (2, 4) is 4; then s2′[5] is going to be computed. s2′[5][0] = s2′[4][0] + 1 = 2; s2′[5][1] is the position of (2, 4) in l; therefore s2′[5] = (2, 1).

The right part of s2′[5] is 1; then pre[1] = s2′[4][1]. end records the information of s2′[5] = (2, 1).

end : [1] → 0

Then, s2′[6][0] < s2′[5][0]; therefore s2′[6] = s2′[5].

For l[2] = (2, 1), s2′[2][0] = s2′[1][0] + 1 = 1; s2′[2][1] is the position of (2, 1) in l; therefore s2′[2] = (1, 2).

The right part of s2′[2] is 2; then pre[2] = s2′[0][1]. end records the information of s2′[2] = (1, 2).

end : [1] → 0 [2] → 1

Then, s2′[3][0] < s2′[2][0]; therefore s2′[3] = s2′[2]. s2′[4][0] ≠ s2′[2][0], s2′[4] is kept unchanged, and the rest of the elements s2′[5] and s2′[6] are not going to be checked.

The rest of the elements of l can be computed in the same way. Figure 2(d) is the final result of pre and end.

From the auxiliary data end, it can be seen that there is only one LIS in l. The length of the LIS is 4.

end[3] points to 7; therefore the last element of the LIS is l[7] = (5, 5).

Table I: Complexity of each procedure of Algorithm 1.

<table>
<thead>
<tr>
<th>Procedure of algorithm</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>O(n + r)</td>
<td>O(\max(r, n \log n))</td>
</tr>
<tr>
<td>Step 2</td>
<td>O(n + r)</td>
<td>O(\frac{n^2 - n}{2})</td>
</tr>
<tr>
<td>Step 3</td>
<td>O(r)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Step 4</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Assume r is the number of match vectors between s1 and s2. Step 1 of Algorithm 1 is a process of O(n + r) space and O(max(r, n log n)) time. As the length of LCS is O(n), step 3 is a process of O(r) space and O(n) time. Step 4 takes O(n) space and O(n) time. Write operations in s2 are for all element of l are listed together in Figure 2(c). In pre and end (see Figure 2(d)), the time of write operation is r. In s2', the time of write operation of dark gray block is r; the time of write operation of gray block is at most \( \sum_{i=1}^{n-1} i = n(n-1)/2 = (n^2 - n)/2 \), which is illustrated in Figure 3. Therefore, step 2 takes O(n + r) space and O(r + (n^2 - n)/2) time. Complexities of every step of Algorithm 1 are listed in Table 1. The whole algorithm takes O(n + r) space and O(r + (n^2 - n)/2) time, which is dominated by step 2.

3. Efficiency

The algorithm proposed in this paper is designed to compute LCS between two arbitrary sequences, which is the same as the original intention of the classical algorithms: Chvátal-Sankoff algorithm [10], Hirschberg algorithm [11], and Hunt-Szymanski algorithm [15]. The proposed algorithm can be more efficient in specific range of r compared with the classical algorithms, where r is the total number of elements in the set \{ |i, j|s1[i] = s2[j] | \} assuming two arbitrary sequences are s1 and s2.
3.1. Comparison with Hunt-Szymanski Algorithm. As the original position in $s_2$ of each element of $l$ is not used in the process of computing, in Figure 4 Hunt-Szymanski algorithm needs to utilize binary search to locate the position in $s_2'$ for write operation for each element of $l$. The time of binary search in $s_2'$ of Hunt-Szymanski algorithm is at most $\sum_{i=1}^n \log i + (r - n)\log n$, which is illustrated in Figure 5. Using Stirling’s approximation [26–28], $\sum_{i=1}^n \log i + (r - n)\log n = \log[\sum_{i=1}^n i] + (r - n)\log n = \log(n!) + (r - n)\log n = \log(n!)/n + (r - n)\log n = r\log n$. If the demand is only returning one LCS or the length of LCS, array $l$ of the algorithm proposed in this paper can be replaced with the MATCHLIST that is used in Hunt-Szymanski algorithm. Therefore, the algorithm proposed in this paper can take $O(n)$ space that is the same as the one Hunt-Szymanski algorithm takes. The main difference between them is the time consumed in $s_2'$.

In Figure 3, the total time of write operation of both dark gray and light gray blocks is at most $r + (n^2 - n)/2$. As $0 \leq r \leq n^2$, if $r + (n^2 - n)/2 < r\log n \Rightarrow (n^2 - n)/2(\log(n - 1)) < r \leq n^2$,

the algorithm proposed in this paper is more efficient in time than Hunt-Szymanski algorithm (see Figure 7).

3.2. Comparison with Chvátal-Sankoff Algorithm. Chvátal-Sankoff algorithm needs $n^4$ times of comparison in $n^2$ space, which is illustrated in Figure 6. To simplify the analysis, only the $r + (n^2 - n)/2$ time consumed in $s_2'$ of the algorithm proposed in this paper is going to be compared with the $n^2$ time of Chvátal-Sankoff algorithm. As $0 \leq r \leq n^2$, if $r + (n^2 - n)/2 < n^2 \Rightarrow 0 \leq r < (n^2 + n)/2$, the algorithm proposed in this paper is more efficient in time than Chvátal-Sankoff algorithm (see Figure 7). In this case of $r$, the proposed algorithm is also more efficient in space than Chvátal-Sankoff algorithm.

3.3. Comparison with Hirschberg Algorithm. Hirschberg algorithm takes $O(n)$ space and $O(n^2\log n)$ time. As $0 \leq r \leq n^2$,
the algorithm proposed in this paper takes $O(n + r)$ space and $O(r + (n^2 - n)/2) = O(n^2)$ time. Therefore, the proposed algorithm has lower time complexity than Hirschberg algorithm.

**Data Availability**

This submission is about an algorithm of an engineering problem. The efficiency of the algorithm is proven mathematically in theory.

**Conflicts of Interest**

The author declares that they have no conflicts of interest.

**References**


