Hindawi
Mathematical Problems in Engineering
Volume 2018, Article ID 4158071, 6 pages
https://doi.org/10.1155/2018/4158071

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

Academic Editor: Nicholas Chileshe

The longest common subsequence (LCS) problem is a classic computer science problem. For the essential problem of computing LCS between two arbitrary sequences $s1$ and $s2$, 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 $\{(i,j) \mid s1[i]=s2[j]\}$. 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* ($\boldsymbol{LCS_2}$), 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 $s1$ and $s2$. 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 $\{(i,j) \mid s1[i] = s2[j]\}$ [15]. The algorithm reduces LCS_2 to longest increasing subsequence (LIS) problem. Apostolico and Guerra (1987) in [16] proposed

	0	1	2	3	4	5
$s1$:	x	n	f	a	f	a

	0	1	2	3	4	5
$s2$:	y	f	a	n	f	a

	n	f		a		f		a	
	0	1	2	3	4	5	6	7	8
l :	(1, 3)	(2, 4)	(2, 1)	(3, 5)	(3, 2)	(4, 4)	(4, 1)	(5, 5)	(5, 2)

Figure 1: $s1$, $s2$ and the conceived new data l.

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 $\varepsilon > 0$. $O(n^2(\log \log n)/\log^2 n)$ algorithms are also proposed by Bille and Farach-Colton (2008) in [21] and Grabowski (2014) in [22], each of which has its own prerequisite. Restrained by the conclusion of [9, 20], in these decades an extensive amount of research keeps trying to achieve lower complexity than $O(n^2)$ of computing LCS between two condition-specific sequences for different applications, which also can be found in the survey [12]. For computing the length of LCS between two sequences over constant alphabet size, Allison and Dix (1986) presented an algorithm of $O(n^2/w)$, where w is the word-length of computer [23]. This algorithm uses bit-vector formula with 6 bit-wise operations. Although falling into the same complexity class as simple $O(n^2)$ DP algorithms, this algorithm is faster in practice. Crochemore, Iliopoulos, Pinzon, and Reid (2001) in [24] proposed a similar approach whose complexity is also $O(n^2/w)$. Due to the fact that only 4 bit-wise operations are used by the bit-vector formula, this approach gives a practical speedup over Allison and Dix's algorithm.

Compared with Chvátal-Sankoff algorithm [10], Hirschberg algorithm [11], and Hunt-Szymanski algorithm [15], most of the other algorithms for LCS problem between two sequences have more dependency, such as the following: the length of LCS is estimable beforehand [13, 14], two input sequences are similar [14, 16], problem is sparse enough [17], or the alphabet size is finite [16, 18, 20]. Some algorithms give speedup over classical algorithms in engineering [23, 24]. In this paper, an algorithm of $O(n + r)$ space and $O(r + n^2)$ time is proposed for LCS_2, where r is the total number of elements in the set $\{(i, j) | s1[i] = s2[j]\}$ assuming the two arbitrary sequences are $s1$ and $s2$. The algorithm also reduces LCS_2 to longest increasing subsequence (LIS) problem. Compared with relevant classical algorithms, the algorithm can be more efficient in specific range of r.

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., $fafa$ is a longest common subsequence between $nfafa$ and $fanfa$). $LCS(seq1, seq2)$ 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 $s1 = xnfafa$ and $s2 = yfanfa$. For all $s1[i] = s2[j]$, 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 ($l[u][0]$) is the position of a symbol in $s1$, and the right part of the element ($l[u][1]$) is the position of the symbol in $s2$. l is sorted according to $l[u][0]$ as the first key in ascending order and according to $l[u][1]$ as the second key in descending order. Define $(i_u, j_u), (i_v, j_v) \in l, (i_u < i_v) \wedge (j_u < j_v) \Longleftarrow (i_u, j_u) < (i_v, j_v)$. Associating $LIS(l)$ with $LCS(s1, s2)$, it is bijective mapping between $LIS(l)$ and $LCS(s1, s2)$. Hence, $LCS(s1, s2)$ **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 $s1 = xnfafa$, $s2 = yfanfa$ that are given previously. The process of computing LCSes between $s1$ and $s2$ using Algorithm 1 is illustrated in Figure 2 and presented as follows.

Scan l from left to right. The right part of $l[0] = (1, 3)$ is 3, $3 + 1 = 4$; then $s2'[4]$ is going to be computed. $s2'[4][0] = s2'[3][0] + 1 = 1$; $s2'[4][1]$ is the position of $(1, 3)$ in l; therefore $s2'[4] = (1, 0)$.

```
ALG(s1, s2)    ▷ n = |s2|
▷ Step 1 : Construct new data
s1, s2 ⟶ l   ▷ r = |l|
▷ Step 2 : The main procedure
s2′[0...n] ⟵ (0, -)
pre[0...r − 1] ⟵ -
end ⟵ [ ]
for i = 0 to r − 1 do
   s2′[ξ + 1][1] ⟵ i   ▷ ξ = l[i][1]
   s2′[ξ + 1][0] ⟵ s2′[ξ][0] + 1
   pre[i] ⟵ s2′[ξ][1]
   end ⟵(records) s2′[ξ + 1]
   for j = ξ + 2 to r − 1 do
      if s2′[j][0] < s2′[ξ + 1][0]
         s2′[j] ⟵ s2′[ξ + 1]
      else
         break
▷ Step 3 : Compute LIS of l
l, pre, end ⟶ lis   ▷ lis ∈ LIS(l)
▷ Step 4 : Compute LCS between s1 and s2
s1 or s2, lis ⟶ lcs   ▷ lcs ∈ LCS(s1, s2)
return lcs
```

ALGORITHM 1: Algorithm proposed in this paper.

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

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
2 → 1

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
2 → 1

Then, $s2'[3][0] < s2'[2][0]$; therefore $s2'[3] = s2'[2]$. $s2'[4][0] \not< s2'[2][0]$, $s2'[4]$ ***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 1: Complexity of each procedure of Algorithm 1.

Procedure of algorithm	Space	Time
Step 1	$O(n + r)$	$O(\max(r, n\log n))$
Step 2	$O(n + r)$	$O\left(r + \frac{n^2 - n}{2}\right)$
Step 3	$O(r)$	$O(n)$
Step 4	$O(n)$	$O(n)$

$pre[7] = 5$ and $l[5] = (4, 4)$; then the last two elements of the LIS are $(4, 4)$ $(5, 5)$.

$pre[5] = 4$ and $l[4] = (3, 2)$; then $(3, 2)$ $(4, 4)$ $(5, 5)$.

$pre[4] = 2$ and $l[2] = (2, 1)$; then $(2, 1)$ $(3, 2)$ $(4, 4)$ $(5, 5)$.

$pre[2]$ is null. Then the LIS is $(2, 1)$ $(3, 2)$ $(4, 4)$ $(5, 5)$.

Since it is bijective mapping between $LIS(l)$ and $LCS(s1, s2)$, $(2,1)$ $(3,2)$ $(4,4)$ $(5,5) \in LIS(l) \Longrightarrow s1[2]s1[3]s1[4]s1[5] = s2[1]s2[2]s2[4]s2[5] = fafa \in LCS(s1, s2)$. $fafa$ is the only LCS between $s1$ and $s2$.

2.2. Complexity. According to the conclusion of [15] (paragraph 3 page 4), we have the following.

> Step 1 [author's note: similar to step 1 of Algorithm 1 of this paper] can be implemented by sorting each sequence while keeping track of each element's original position. We may then merge the sorted sequences creating the MACHLISTs [author's note: similar to array l of this paper] as we go. This step takes a total of $O(n\log n)$ time and $O(n)$ space.

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 $s2'$ 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 light 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 $O(r + (n^2 - n)/2) = O(r + n^2)$ 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 $s2$.

	0	1	2	3	4	5
$s1$:	x	n	f	a	f	a

	0	1	2	3	4	5
$s2$:	y	f	a	n	f	a

(a) Two sequences: $s1$ and $s2$

	n	f		a		f		a	
	0	1	2	3	4	5	6	7	8
l:	(1, 3)	(2, 4)	(2, 1)	(3, 5)	(3, 2)	(4, 4)	(4, 1)	(5, 5)	(5, 2)

(b) New data l constructed from $s1$ and $s2$

		y	f	a	n	f	a
	0	1	2	3	4	5	6
$s2'$:	(0, –)	(0, –)	(0, –)	(0, –)	(0, –)	(0, –)	(0, –)
For $l[0]$ = (1, 3), $s2'$:	(0, –)	(0, –)	(0, –)	(0, –)	(1, 0)	(1, 0)	(1, 0)
For $l[1]$ = (2, 4), $s2'$:	(0, –)	(0, –)	(0, –)	(0, –)	(1, 0)	(2, 1)	(2, 1)
For $l[2]$ = (2, 1), $s2'$:	(0, –)	(0, –)	(1, 2)	(1, 2)	(1, 0)	(2, 1)	(2, 1)
For $l[3]$ = (3, 5), $s2'$:	(0, –)	(0, –)	(1, 2)	(1, 2)	(1, 0)	(2, 1)	(3, 3)
For $l[4]$ = (3, 2), $s2'$:	(0, –)	(0, –)	(1, 2)	(2, 4)	(2, 4)	(2, 1)	(3, 3)
For $l[5]$ = (4, 4), $s2'$:	(0, –)	(0, –)	(1, 2)	(2, 4)	(2, 4)	(3, 5)	(3, 3)
For $l[6]$ = (4, 1), $s2'$:	(0, –)	(0, –)	(1, 6)	(2, 4)	(2, 4)	(3, 5)	(3, 3)
For $l[7]$ = (5, 5), $s2'$:	(0, –)	(0, –)	(1, 6)	(2, 4)	(2, 4)	(3, 5)	(4, 7)
For $l[8]$ = (5, 2), $s2'$:	(0, –)	(0, –)	(1, 6)	(2, 8)	(2, 4)	(3, 5)	(4, 7)

(c) Write operation in $s2'$

	0	1	2	3	4	5	6	7	8
pre:	–	0	–	1	2	4	–	5	6

end:				
1	→	0	2	6
2	→	1	4	8
3	→	3	5	
4	→	7		

(d) Final result of pre and end

Figure 2: Example.

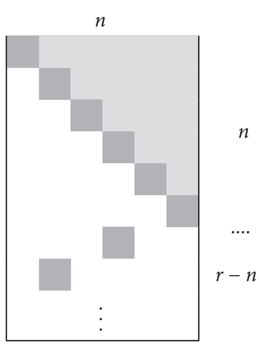

Figure 3: Maximum write operation of light gray block in $s2'$ of Alggorithm 1.

	n	f		a		f		a	
	0	1	2	3	4	5	6	7	8
l:	3	4	1	5	2	4	1	5	2

For $l[0] = 3$, $s2'$:	3			
For $l[1] = 4$, $s2'$:	3	4		
For $l[2] = 1$, $s2'$:	1	4		
For $l[3] = 5$, $s2'$:	1	4	5	
For $l[4] = 2$, $s2'$:	1	2	5	
For $l[5] = 4$, $s2'$:	1	2	4	
For $l[6] = 1$, $s2'$:	1	2	4	
For $l[7] = 5$, $s2'$:	1	2	4	5
For $l[8] = 2$, $s2'$:		2	4	5

Figure 4: Write operation in $s2'$ of Hunt-Szymanski algorithm.

3.1. Comparison with Hunt-Szymanski Algorithm. As the original position in $s2$ 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 $s2'$ for write operation for each element of l. The time of binary search in $s2'$ 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], $\sum_{i=1}^{n}\log i + (r-n)\log n = \log\prod_{i=1}^{n} i + (r-n)\log n = \log(n!) + (r-n)\log n \approx n\log n + (r-n)\log n = r\log n$. 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 $s2'$. 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 $0 \leqslant r \leqslant n^2$, if $r + (n^2 - n)/2 < r\log n \Longrightarrow (n^2 - n)/2(\log n - 1) < r \leqslant n^2$, 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^2 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 $s2'$ of the algorithm proposed in this paper is going to be compared with the n^2 time of Chvátal-Sankoff algorithm. As $0 \leqslant r \leqslant n^2$, if $r + (n^2 - n)/2 < n^2 \Longrightarrow 0 \leqslant r < (n^2 + n)/2$, 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 $0 \leqslant r \leqslant n^2$,

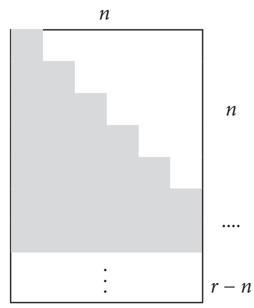

FIGURE 5: Maximum time of binary search of Hunt-Szymanski algorithm.

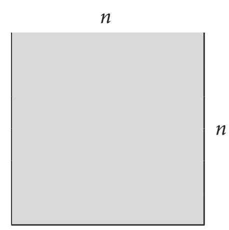

FIGURE 6: Time of comparison of Chvátal-Sankoff algorithm.

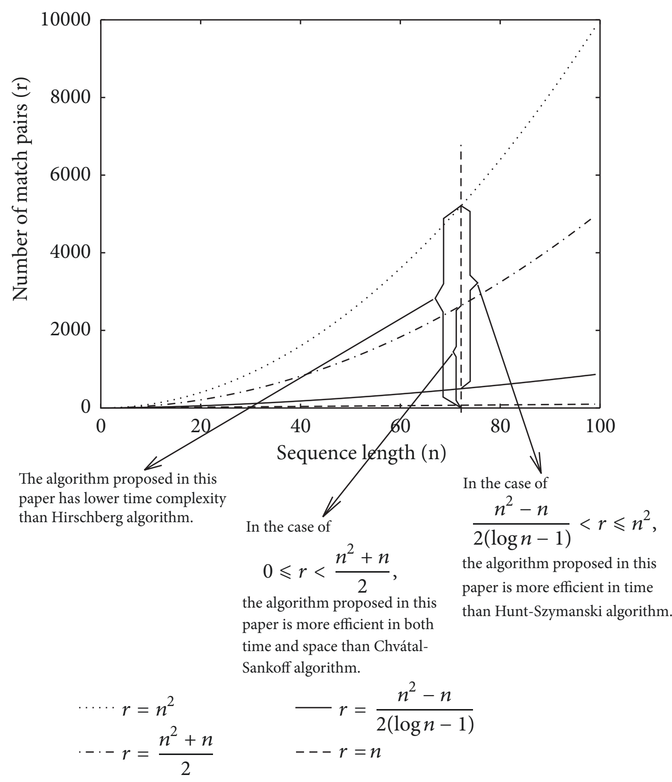

FIGURE 7: Comparison of efficiency against classical algorithms.

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

[1] D. Zhu, L. Wang, and X. Wang, "An improved O(Rlog log n + n) time algorithm for computing the longest common subsequence," *IAENG International Journal of Computer Science (IJCS)*, vol. 44, no. 2, pp. 166–171, 2017.

[2] J. Yang, Y. Xu, Y. Shang, and G. Chen, "A space-bounded anytime algorithm for the multiple longest common subsequence problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2599–2609, 2014.

[3] Y. Sakai, "A fast On-Line algorithm for the longest common subsequence problem with constant alphabet," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E-95-A, no. 1, pp. 354–361, 2012.

[4] M. Sazvar, M. Naghibzadeh, and N. Saadati, "Quick-MLCS: A new algorithm for the multiple longest common subsequence problem," in *Proceedings of the 5th International C Conference on Computer Science and Software Engineering,C3S2E 2012*, pp. 61–66, Canada, June 2012.

[5] R. F. Rahmat, F. Nicholas, S. Purnamawati, and O. S. Sitompul, "File Type Identification of File Fragments using Longest Common Subsequence (LCS)," *Journal of Physics: Conference Series*, IOP Publishing, p. 012054, 2017.

[6] A. Sorokin, "Using longest common subsequence and character models to predict word forms," in *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pp. 54–61, Berlin, Germany, August 2016.

[7] X. Xie, W. Liao, H. Aghajan, P. Veelaert, and W. Philips, "Detecting road intersections from GPS traces using longest common subsequence algorithm," *ISPRS International Journal of Geo-Information*, vol. 6, no. 1, 2017.

[8] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM*, vol. 25, no. 2, pp. 322–336, 1978.

[9] A. V. Aho, D. S. Hirschberg, and J. D. Ullman, "Bounds on the complexity of the longest common subsequence problem," *Journal of the ACM*, vol. 23, no. 1, pp. 1–12, 1976.

[10] V. Chvatal and D. Sankoff, "Longest common subsequences of two random sequences," *Journal of Applied Probability*, vol. 12, pp. 306–315, 1975.

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

[12] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings of the 7th International Symposium on String Processing and Information Retrieval, SPIRE 2000*, pp. 39–48, Spain, September 2000.

[13] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, vol. 24, no. 4, pp. 664–675, 1977.

[14] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similar test strings," *Acta Informatica*, vol. 18, no. 2, pp. 171–179, 1982/83.

[15] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[16] A. Apostolico and C. Guerra, "The longest common subsequence problem revisited," *Algorithmica*, vol. 2, no. 1–4, pp. 315–336, 1987.

[17] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano, "Sparse dynamic programming. I. Linear cost functions," *Journal of the ACM*, vol. 39, no. 3, pp. 519–545, 1992.

[18] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[19] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzhe, "The economical construction of the transitive closure of an oriented graph," *Doklady Akademii Nauk SSSR*, vol. 194, pp. 487-488, 1970.

[20] A. Abboud, A. Backurs, and V. V. Williams, "Tight Hardness Results for LCS and Other Sequence Similarity Measures," in *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 59–78, Berkeley, CA, USA, October 2015.

[21] P. Bille and M. Farach-Colton, "Fast and compact regular expression matching," *Theoretical Computer Science*, vol. 409, no. 3, pp. 486–496, 2008.

[22] S. Grabowski, "New tabulation and sparse dynamic programming based techniques for sequence similarity problems," *Discrete Applied Mathematics: The Journal of Combinatorial Algorithms, Informatics and Computational Sciences*, vol. 212, pp. 96–103, 2016.

[23] L. Allison and T. I. Dix, "A bit-string longest-common-subsequence algorithm," *Information Processing Letters*, vol. 23, no. 6, pp. 305–310, 1986.

[24] M. Crochemore, C. S. Iliopoulos, Y. . Pinzon, and J. F. Reid, "A fast and practical bit-vector algorithm for the longest common subsequence problem," *Information Processing Letters*, vol. 80, no. 6, pp. 279–285, 2001.

[25] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*, Cambridge University Press, New York, NY, USA, 1997.

[26] J. Dutka, "The early history of the factorial function," *Archive for History of Exact Sciences*, vol. 43, no. 3, pp. 225–249, 1991.

[27] L. Le Cam, "The central limit theorem around 1935," *Statistical Science*, vol. 1, no. 1, pp. 78–91, 1986.

[28] K. Pearson, "Historical Note on the Origin of the Normal Curve of Errors," *Biometrika*, vol. 16, no. 3-4, pp. 402–404, 1924.