# Longest Common Subsequence in $k$ Length Substrings

Gary Benson[1,⋆], Avivit Levy[2], and B. Riva Shalom[2]

[1] Department of Computer Science, Boston University, Boston, MA 02215
gbenson@bu.edu
[2] Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel
{avivitlevy,rivash}@shenkar.ac.il

**Abstract.** In this paper we define a new problem, motivated by computational biology, *LCSk* aiming at finding the maximal number of $k$ length *substrings*, matching in both input string while preserving their order of appearance in the input strings. The traditional LCS definition is a spacial case of our problem, where $k = 1$. We provide an algorithm, solving the general case in $O(n^2)$ time, where $n$ is the length of the input, equaling the time required for the special case of $k = 1$. The space requirement is $O(kn)$. In order to enable backtracking of the solution $O(n^2)$ space is needed.

**Keywords:** Longest Common Subsequence, Similarity of strings, Dynamic Programming.

## 1 Introduction

The *Longest Common Subsequence* problem, whose first famous dynamic programming solution appeared in 1974 [18], is one of the classical problems in Computer Science. The widely known string version appears in Definition 1.

**Definition 1.** The String Longest Common Subsequence ($LCS$) Problem*:*
*Input:* *Two sequences* $A = a_1a_2 \dots a_n$, $B = b_1b_2 \dots b_n$ *over alphabet* $\Sigma$.
*Output:* *The length of the longest subsequence common to both strings, where a subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.*

For example, for the sequences appearing in Figure 1. $LCS(A, B)$ is 5, where a possible such subsequence is $T\ T\ G\ T\ G$.

The LCS problem has been very well studied. For a survey, see [6]. The problem is mainly motivated by the need to measure similarity over the input sequences. The well known dynamic programming solution [16] requires running time of $O(n^2)$, for two input strings of length $n$.

⋆ This work was partially funded by NSF grant IIS-1017621.

The LCS problem had also been investigated on more general structures such as trees and matrices [3], run-length encoded strings [4], weighted sequences [2], [7] and more. Many variants of the LCS problem were studied as well, among which LCS alignment [15], [14], [13], constrained LCS [17], [8], restricted LCS [10] and LCS approximation [12].

**Motivation.** The LCS has been used as a measure of sequence similarity for biological sequence comparison. Its strength lies in its simplicity which has allowed development of an extremely fast, bit-parallel computation which uses the bits in a computer word to represent adjacent cells a row of the LCS scoring matrix and computer logic operations to calculate the scores from one row to the next [1], [9], [11]. For example, in a recent experiment, 25,000,000 bit-parallel LCS computations (sequence length = 63) took approximately 7 seconds on a typical desktop computer [5] or about 60 times faster than a standard algorithm. This speed makes the LCS attractive for sequence comparison performed on high-sequencing data. The disadvantage of the LCS is that it is a crude measure of similarity because consecutive matching letters in the LCS can have different spacings in the two sequences, i.e., there is no penalty for insertion and deletion. What is proposed here is a definition of LCS that makes the measure of similarity more accurate because it forces adjacent letters in the LCS to be adjacent in both sequences. In our problem, the common subsequence is required to consist of k length substrings. A formal definition appears in Definition 2.

**Definition 2.** The Longest Common Subsequence in k Length Substrings Problem (*LCSk*)*:*
*Input:* *Two sequences* $A = a_1a_2 \ldots a_n$, $B = b_1b_2 \ldots b_n$ *over alphabet* $\Sigma$.
*Output:* *The maximal* $\ell$ *s.t. there are* $\ell$ *substrings,*
$a_{i_1}...a_{i_1+k-1} \ldots a_{i_\ell}...a_{i_\ell+k-1}$, *identical to* $b_{j_1}b..j_{1+k-1} \ldots b_{j_\ell}...b_{j_\ell+k-1}$
*where* $\{a_{i_f}\}$ *and* $\{b_{j_f}\}$ *are in increasing order for* $1 \le f \le \ell$ *and*
*where two k length substrings in the same sequence, do not overlap.*

We demonstrate *LCSk* considering the sequences appearing in Figure 1.

$$A = \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathbf{G} & \mathbf{T} & \mathrm{G} & \mathbf{T} & \mathbf{G} \end{array}$$

$$B = \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \mathbf{G} & \mathbf{T} & \mathrm{T} & \mathrm{G} & \mathbf{T} & \mathbf{G} & \mathrm{C} & \mathrm{C} \end{array}$$

**Fig. 1.** An LCS2 example

A possible common subsequence in pairs ($k = 2$) is $GTTG$ obtained from $a_4$, $a_5$, $a_7$, $a_8$ and $b_1, b_2, b_5, b_6$. Though $a_6 = b_4$, and such a match preserves the order of the common subsequence, it cannot be added to the common subsequence in pairs, since it is a match of a single symbol. For $k = 3$, one of the possible solutions is $TGC$ achieved by matching $a_1$, $a_2$, $a_3$, with $b_5, b_6, b_7$. For $k = 4$ a possible solution is $TGTG$ obtained from matching $a_5$, $a_6$, $a_7$, $a_8$ and $b_3, b_4, b_5, b_6$. Note that in the last two cases the solution contains a single triple, quadruplet.

The paper is organized as follows: Section 2 gives some preliminaries. The solution for the *LCSk* problem is detailed in Section 3. Section 4 concludes the paper.

## 2 Preliminaries

The LCSk problem is a generalization of the LCS problem. We might consider using the solution of the latter in order to solve the former. If we perform the LCS algorithm on the input sequences, we can backtrack the dynamic programming table and mark the symbols participating in the common subsequence. We can then check whether those symbols appear in consecutive $k$ length substrings in both input sequences, and delete them if not. Such a procedure guarantees a common subsequence in $k$ length substrings but not necessarily the optimal length of the common subsequence. For example consider $LCS2$ of the sequences appearing on Figure 1. Applying the LCS algorithm on these strings may yield $T\ T\ G\ T\ G$, containing a single non-overlapping pair matching while there exists LCS2 of $T\ G\ T\ G$ having two pair matchings. Hence, a special algorithm designed for *LCSk* is required.

As the LCSk problem considers matchings of $k$ consecutive symbols, we call such a matching, throughout this paper, a *$k$ matching* and define the following definitions:

**Definition 3**

$$kMatch(i,j) = \begin{cases} 1\ if\ a_{i+f} = b_{j+f},\ \ for\ every\ 0 \le f \le k-1 \\ 0\ Otherwise \end{cases}$$

*If $kMatch(i,j) = 1$, the matching substring is denoted (i,j).*

**Definition 4.** ***Predecessors.*** *Let $candidates(i,j)$ be the set of all longest common subsequences, consisting of k matchings, of prefix $A[1...i+k-1]$ and prefix $B[1...j+k-1]$. Then let $pred(i,j)$ be all the possible last k matchings in $candidates(i,j)$. That is, $pred(i,j) = \{(s,t)|\exists c \in candidates(i,j),\ where\ (s,t)\ is\ the\ last\ \ k\ matching\ in\ c\}$.*

*We define the length of $p \in pred(i,j)$ derived from candidate c, to be the number of k matchings in c and denote it by $|p|$.*

**Example.** Consider LCS2 of the sequences of Figure 1. Let $candidates(5,3)$ be the common subsequences in pairs of $B[1...4] = G\ T\ T\ G$ and of $A[1...6] = T\ G\ C\ G\ T\ G$, thus, $candidates(6,4)$ contains$\{TG, GT\}$. $TG$ can be obtained in two ways: $a_1a_2$ matched to $b_3b_4$, or $a_5a_6$ matched to $b_3b_4$, and $GT$ by $a_4a_5$ matched to $b_1b_2$ therefore, we have $pred(i,j) = \{(1,3),(5,3),(4,1)\}$. In this example all predecessors are of length 1. Keeping the predecessors enables backtracking to reveal the longest common subsequence in $k$ length substrings itself.

The following Lemma is necessary for the correctness of the solution.

**Lemma 1.** *Let $p_1$,$p_2 \in pred(i,j)$, then if $|p_1| + 1 = |p_2|$, then any maximal common subsequence of k length substrings using the k matching $p_2$ has length greater or equal to that using the k matching $p_1$.*

Proof: Suppose $p_1 = (s,t)$ and $p_2 = (s',t')$. Several cases are possible for $p_1, p_2$:

1. If $s' < s$ and $t' < t$, then the candidate whose last $k$ matching is $p_2$ might be further extended till $A[s]$ and $B[t]$, enlarging the difference between $p_1$ and $p_2$.
2. If $s' = s$ and $t' = t$ both predecessors have the same opportunities for extension.
3. If $s+k-1 < s'$ and $t+k-1 < t'$, the $k$ matching $(s',t')$ can be added to the candidate whose last $k$ matching is $(s,t)$, contradicting its maximality.
4. If there is an overlap between the $k$ matchings represented by the predecessors, $s < s' < s+k$ or $t < t' < t+k$, starting from $a_{s'+k}$, every $k$ matching can be used to extend the common subsequence in $k$ length substring, represented by both predecessors. However, the subsequence using $p_1$ cannot have an additional $k$ matching before $a_{s'+k}$, as overlaps are forbidden. Consequently, the difference between the length of $p_1$ and $p_2$ is preserved in the extended maximal common subsequences. ∎

## 3 Solving the LCSk Problem

As in other LCS variants, we solve the problem using a dynamic programming algorithm. We denote by $LCSk_{i,j}$ the longest common subsequence, consisting of $k$ matchings in the prefixes $A[1...i+k-1]$ and $B[1...j+k-1]$. Lemma 2 below, formally describes the computation of $LCSk_{i,j}$.

**Lemma 2.** ***The Recursive Rule***

$$LCSK_{i,j} = max \begin{cases} LCSk_{i,j-1}, \\ LCSk_{i-1,j}, \\ LCSk_{i-k,j-k} + kMatch(i,j) \end{cases}$$

Proof: $LCSK_{i,j}$ contains the maximal number of common $k$ length substrings, preserving their order in the input sequences. A possible subsequence can be constructed by matching the substrings $a_i, \ldots, a_{i+k-1}$ with $b_j, \ldots, b_{j+k-1}$, in case all $a_{i+f}$ and all $b_{j+f}$, for $0 \leq f \leq k-1$, are not part of previous $k$ *matching*s. This is guaranteed when considering the prefixes $A[1..i-k]$ and $B[1..j-k]$ while trying to extend by one the common subsequence for cell $LCSk_{i,j}$. Another option of extending the subsequence is by using the $k$ matching $(s,j)$ , for $s < i$. Similarly, we can use the k matching $(i,t)$ for $t < j$. Note that the options of extending $LCSk_{i-f,j-f}$, for $1 \leq f \leq k-1$ is included in both $LCSk_{i,j-1}$ and $LCSk_{i-1,j}$. These claims can be easily proven using induction. ∎

According to Lemma 2 we can solve the $LCSk$ problem using a dynamic programming algorithm working on a two dimensional table of size $(n-k+1)^2$ where the rows represent the $A$ sequence and the columns stand for sequence $B$. Cell $LCSk[i,j]$ contains the value $LCSk_{i,j}$ and the appropriate predecessors. Nevertheless, when we wish to attain the common subsequence itself, we encounter a complication.

In the original LCS algorithm, computing the common subsequence, requires maximizing three options of possible prefixes of the LCS. When some of these prefixes have the same length, there is no significance which of them is chosen, as a single common subsequence is sought and the selection has no effect on future matches. However, in the $LCSk$ problem the situation is different. For example, consider $LCS2$. Let $A = a\ c\ a\ b$ and $B = c\ a\ c\ a\ b$. LCS2[3,3] equals 1 due to the 2 matching $(1, 2)$ (matching $a_1 a_2$ to $b_2, b_3$) ("ac"), or by the 2 matching $(2, 1)$ ("ca"). In spite of the fact that both common subsequences, share the same length, the former is part of the final solution as it enables a further 2 matching $(3, 4)$ while the latter cannot be extended due to the overlap restriction. It, therefore, seems that all possibilities of common subsequence in $k$ length substrings, that is, all predecessors should be saved at every calculation in order to enable a correct backtracking of the optimal solution. As the dynamic programming proceeds, this information can exponentially increase. Nevertheless, we prove in Lemma 3 that in the $LCSk$ problem only one maximal previously computed subsequence is required.

The three options of forming $LCSk_{i,j}$, appearing in Lemma 2 form $candidates(i, j)$, hence $pred(i, j)$. Therefore, the $pred(i, j)$ set should be updated after computing $LCSk_{i,j}$.

**Corollary 1.** *If* $LCSk_{i,j-1} = LCSk_{i-1,j} = LCSk_{i-2,j-2}+1$*, and kMatch(i,j)=1 then* $pred(i, j) = pred(i, j-1) \bigcup pred(i-1, j) \bigcup (i, j)$.

*If* $LCSk_{i,j-1} = LCSk_{i-1,j}$ *and kMatch(i,j)=0 then* $pred(i, j) = pred(i, j-1) \bigcup pred(i-1, j)$ .

*In both cases, if one or more of the relevant* $LCSk_{x,y}$, $x \leq i, y \leq j$ *has shorter length, its corresponding pred is not included in* $pred(i, j)$.

Proof: Note, that the length of a predecessor $p \in pred(i, j)$ equals the value of $LCSk_{i,j}$. Due to Lemma 1 there is no necessity to consider the shorter predecessors. Suppose all three sets contain predecessors representing common subsequences of the same length. Without further information, we cannot determine which common subsequence ending in $pred(i, j-1)$, $pred(i-1, j)$, or in $k$ matching of $(i, j)$, will be in the maximal output, therefore, all predecessors must be considered. ∎

## 3.1 The Backtrack Process

Using the recursive rule of Lemma 2, the value computed for $LCSk_{i-k+1,j-k+1}$ is the length of the common subsequence in $k$ length substrings of sequences $A$ and $B$. In order to obtain the common subsequence itself we perform the following procedure. Consider the value saved in cell $LCSk[i, j]$, where $i$ and $j$ are initialized by $n-k+1$. We suppose that each cell contains a single predecessor, as will be proven hereafter in Lemma 3. Let the predecessor saved in the current cell be $(x, y)$. Two cases are regarded as long as $i, j > 0$.

1. if $x = i$ and $y = j$, then a $k$ matching starts in these indices, therefore $a_{i+f}$ *for every* $0 \leq f \leq k-1$ can be added to the constructed output, preserving the increase of the indices. In order to proceed we decrease both $i, j$ by $k$ to avoid previous $k$ matchings overlapping $(i, j)$.
2. Otherwise, no $k$ matching occurs in the current indices. The predecessor $(x, y)$ directs us to the cell containing a $k$ matching which is part of an LSCk with the value $LCSk_{i,j}$. Therefore, we decrease the indices $i = x$ and $j = y$

## 3.2 Predecessors Elimination

We aim at minimizing the number of predecessors per $LCSk[i, j]$ and therefore define a process of predecessors elimination. Eliminating a predecessor $p$, that is, deleting it from $pred(i, j)$ can be safely done if a maximal common subsequence in $k$ length substrings of the same length can be attained using another predecessor from $pred(i, j)$. Lemma 3 provides the elimination procedure and its correctness.

**Lemma 3.** ***Elimination Lemma*** *Let* $p_1$, $p_2 \in pred(i, j)$ *be* $k$ *matchings, where* $|p_1| = |p_2|$, *then one of* $p_1, p_2$ *can be arbitrarily eliminated.*

Proof: Let $p_1 = (s, t)$ , $p_2 = (s', t')$. In case $kMatch(i, j) = 0$ then, if the backtracking pass through table cell [i,j] it implies that the previously found $k$ matching is $(i + k, j + k)$ due to the second case of the backtracking procedure. Moreover, according to Corollary 1, both $\{s, s'\} \leq i$ and $\{t, t'\} \leq j$. As a consequence, there is no preference to one of the equal length predecessors as both cannot overlap the previous $k$ matching.

Suppose then that $kMatch(i, j) = 1$ and $p_2 = (i, j)$. According to the backtracking procedure, we get to cell $[i, j]$ either by the first case of the procedure where there is a $k$ matching $(i+k, j+k)$ or by its second case where at cell $[i', j']$ there is no $k$ matching but it contains a predecessor (i,j). The latter implies that the previously found $k$ matching is $(i + k + f, j + k + h)$ for $f, h > 0$.

There are two cases to consider.

1. If no optimal solution uses the $k$ matching $(i, j)$ it implies that the optimal solution includes $k$ matchings $(i', j')$ and $(i'', j'')$ where $i' < i < i' + k$ and $i'' - k < i < i''$ or $j' < j < j' + k$ and $j'' - k < j < j''$. If only one inequality holds for $i$ or $j$ then some optimal solution will include $(i, j)$, contradicting the case definition. According to the first case of the backtrack procedure, when backtracking from cell [i", j"], including the $k$ matching $(i'', j'')$, we decrease both indices by $k$. Since $i'' - k < i$ and $j'' - k < j$ cell [i,j] will not be considered, therefore even if we saved $p_2$, that is we eliminated $p_1$, it has no consequence on the optimal solution.
2. If there exists an optimal solution including $p_2$ but we arbitrarily eliminated it. Since we proved that the previously found $k$ matching is $(i+k+f, j+k+h)$ for $f, h \geq 0$ there is no preference to $p_2$ over $p_1$ as they are both of the same length and both do not overlap the previously found $k$ matching according to Corollary 1. Apparently, $p_1$ is included in another optimal solution. ∎

**Example.** Figure 2 depicts an LCS2 table. We demonstrate the two cases in Lemma 3 where $kMatch(i, j) = 1$. For the first case, consider cell $LCS2[5, 6]$ including $pred_{5,6} = \{(4, 5), (5, 6)\}$. Suppose we arbitrarily eliminate $(4, 5)$. The LCS2 may contain the 2 matching $(5, 6)$ that overlaps with (4,1),(4,5) and on the same time overlaps also $(6, 7)$ what can decrease the length of the solution. Nevertheless, according to the backtracking procedure, after considering $LSC2[6, 7]$ we decrease the indices and go to $LCS2[4, 5]$ in which $(5, 6)$ cannot exist, due to Corollary 1.

For the second case consider cell $LCS2[2, 3]$ including $pred_{2,3} = \{(1, 1), (2, 3)\}$. (2,3) is included in one of the optimal solution. Suppose we eliminated it and retained $(1, 1)$. The backtracking path goes through cells $[7, 7]$ to $[6, 7]$ to $[4, 5]$ to $[2, 3]$ where it finds a non overlapping predecessor $(1, 1)$ with the same length as the deleted $(2, 3)$.

**Theorem 1** *The $LCSK(A, B)$ problem can be solved in $O(n^2)$ time and $O(kn)$ space, where $n$ is the length of the input sequences $A$, $B$. Backtracking the solution requires time of $O(\ell)$ where $\ell$ is the number of $k$ matchings in the solution, and $O(n^2)$ space.*

Proof: The algorithm fills a table of size $(n-k+1)^2$. Each entry is filled according to Lemma 2 in constant time as we perform a constant number of comparisons. We assume that $k$ is rather a small constant thus computing $kMatch(i, j)$ is done in constant time. In addition unifying three *pred* sets of size one each, does not increase the time requirements per entry. The Elimination procedure requires also constant time according to Lemma 3. All in all, constant time

| | 1 C | 2 T | 3 T | 4 G | 5 C | 6 T | 7 T | 8 T |
|---|---|---|---|---|---|---|---|---|
| 1 C | 1 (1,1) | 1 (1,1) | 1 (1,1) | 1 (1,1) | 1 (1,1)(1,5) | 1 (1,5) | 1 (1,5) | - - |
| 2 T | 1 (1,1) | 1 (1,1) | 1 (1,1)(2,3) | 1 (1,1) | 1 (1,1)(1,5) | 1 (1,5) | 1 (1,5) | - - |
| 3 G | 1 (1,1) | 1 (1,1) | 1 (1,1) | 1 (1,1) | 1 (1,1)(1,5) | 1 (1,5) | 1 (1,5) | - - |
| 4 C | 1 (4,1) | 1 (4,1) | 1 (1,1),(4,1) | 1 (4,1) | 2 (4,5) | 2 (4,5) | 2 (4,5) | - - |
| 5 T | 1 (4,1) | 1 (4,1),(5,2) | 1 (4,1) | 1 (4,1) | 2 (4,5) | 2 (4,5),(5,6) | 2 (4,5),(5,7) | - - |
| 6 T | 1 (4,1) | 1 (4,1),(6,2) | 1 (4,1) | 1 (4,1) | 2 (4,5) | 2 (4,5),(6,6) | 3 (6,7) | - - |
| 7 T | 1 (4,1) | 1 (4,1) | 2 (7,3) | 2 (7,3) | 2 (7,3),(4,5) | 2 (7,3)(6,6) | 3 (6,7) | - - |
| 8 G | - | - | - | - | - | - | - | - |

**Fig. 2.** An LCS2 Table. The numbers represent the length of the common subsequence. The pairs in parenthesis stand for the predecessors. Each cell contains all possible predecessors according to Corollary 1. Due to the Elimination Lemma only one predecessor is retained in the to following cells.

operations are performed for each of the table entries, concluding in $O(n^2)$ time requirement for computing the optimal length of the common subsequence in $k$ length substrings.

During the backtracking process we go through the cells representing the $k$ matchings of one optimal solution. If the difference between two such $k$ matchings is more than $k$, we will go through an intermediate cell whose predecessor directs us to the next $k$ matching. Hence finding the common subsequence in $k$ length substrings requires $O(\ell)$ where $\ell$ is the number of $k$ matchings in the solution.

Regarding space: Each of the $n^2$ entries contains, according to Corollary 1 three predecessors and the Eliminate function, due to Lemma 3, results in a single predecessor before considering further entries, implying $O(n^2)$ space requirement. Nevertheless, due to Lemma 2, during the computation of $LCSk[i, j]$ we need only row $i - k$ and column $j - k$. As a consequence, at each step we save only $k$ rows and columns implying the space requirement is $O(kn)$. In order to backtrack the solution, the whole table is needed, implying $O(n^2)$ space requirement. ∎

## 4 Conclusion

In this paper we defined a generalization of the LCS problem, where each matching must consist of $k$ consecutive symbols. We proved that using the known LCS algorithm does not always output an optimal solution. However, by thoroughly understanding the traits of the problem we proved a similar algorithm with the same time complexity can solve the problem. Due to the importance of the LCS problem as a measure of similarity between the inputs, more generalizations may be thought of.

## References

1. Allison, L., Dix, T.I.: A bit-string Longest-Common-Subsequence. Information Processing Letters 23(5), 305–310 (1986)
2. Amir, A., Gotthilf, Z., Shalom, R.: Weighted LCS. J. Discrete Algorithms 8(3), 273–281 (2010)
3. Amir, A., Hartman, T., Kapah, O., Shalom, R., Tsur, D.: Generalized LCS. Theor. Comput. Sci. 409(3), 438–449 (2008)
4. Apostolico, A., Landau, G.M., Skiena, S.: Matching for run-length encoded strings. Journal of Complexity 15(1), 4–16 (1999)
5. Benson, G., Hernandez, Y., Loving, J.: A bit-parallel, general integer-scoring sequence alignment algorithm. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 50–61. Springer, Heidelberg (2013)
6. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proc, 7th Symposium on String Processing and Information Retrieval (SPIRE), pp. 39–48 (2000)
7. Blin, G., Jiang, M., Vialette, S.: The Longest Common Subsequence Problem with Crossing-Free Arc-Annotated Sequences. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 130–142. Springer, Heidelberg (2012)

8. Chen, Y.C., Chao: On the generalized constrained longest common subsequence problems. Journal of Combinatorial Optimization 21(3), 383–392 (2011)
9. Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J., Reid, J.F.: A fast and practical bit-vector algorithm forthe longest common subsequence problem. Information Processing Letters 80(6), 279–285 (2001)
10. Gotthilf, Z., Hermelin, D., Landau, G.M., Lewenstein, M.: Restricted LCS. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 250–257. Springer, Heidelberg (2010)
11. Hyyro, H.: Bit parallel LCS- length computation revisited. In: Proc. 15th Australasian Workshop on Combinatorial Algorithms, AWOCA (2004)
12. Landau, G.M., Levy, A., Newman, I.: LCS approximation via embedding into locally non-repetitive strings. Inf. Comput. 209(4), 705–716 (2011)
13. Landau, G.M., Myers, E.W., Ziv-Ukelson, M.: Two Algorithms for LCS Consecutive Suffix Alignment. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 173–193. Springer, Heidelberg (2004)
14. Landau, G.M., Schieber, B., Ziv-Ukelson, M.: Sparse LCS Common Substring Alignment. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 225–236. Springer, Heidelberg (2003)
15. Landau, G.M., Ziv-Ukelson, M.: On the Common Substring Alignment Problem. J. Algorithms 41(2), 338–359 (2001)
16. Hirschberg, D.S.: A Linear space algorithm for Computing Maximal Common Subsequences. Commun. ACM 18(6), 341–343 (1975)
17. Tsai, Y.T.: The constrained longest common subsequence problem. Information Processing Letters 88(4), 173–176 (2003)
18. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM 21, 168–173 (1974)