LCSk++: Practical similarity metric for long strings

Filip Pavetić[1], Goran Žužić[1], and Mile Šikić[1,2]

[1] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
[2] Bioinformatics Institute 30 Biopolis Street, #07-01 Matrix, 138671 Singapore
`fpavetic@gmail.com`
{`goran.zuzic, mile.sikic`}`@fer.hr`

Abstract. In this paper we present *LCSk*++: a new metric for measuring the similarity of long strings, and provide an algorithm for its efficient computation. With ever increasing size of strings occuring in practice, e.g. large genomes of plants and animals, classic algorithms such as Longest Common Subsequence (LCS) fail due to demanding computational complexity. Recently, Benson et al. defined a similarity metric named *LCSk*. By relaxing the requirement that the k-length substrings should not overlap, we extend their definition into a new metric. An efficient algorithm is presented which computes *LCSk*++ with complexity of $O((|X| + |Y|)\log(|X| + |Y|))$ for strings X and Y under a realistic random model. The algorithm has been designed with implementation simplicity in mind. Additionally, we describe how it can be adjusted to compute *LCSk* as well, which gives an improvement of the $O(|X||Y|)$ algorithm presented in the original *LCSk* paper.

Keywords: Efficient longest common subsequence, Similarity of long strings, Bioinformatics, Sparse dynamic programming

1 Introduction

Measuring the similarity of strings is the fundamental problem which arises in many applications including DNA sequence comparison [1], differential file analysis and plagiarism detection [2]. Metrics such as Longest Common Subsequence [3] or Edit Distance [4] are usually used for solving this type of problems. Still, even advanced variants of these approaches don't cope well with long input strings (e.g. the size of the human genome).

A general approach of approximating the Longest Common Subsequence is given by Baker and Giancarlo [5]. They assume a list of matching substring pairs of various lengths as input and combine them to approximate the LCS between two long strings. The two variants of their algorithm have $O(T \log T)$ and $O(T \log\log min(T, nm/T))$ time complexities, where T denotes the number of matching fragments and n, m denote lengths of the two strings. We simplify some of their ideas and address the question of how to select these matching fragments.

Recently, significant effort was directed towards defining new similarity metrics. Benson et al. [1] defined a metric called $LCSk$, which is computed between strings X of length m and Y of length n. It counts the maximal number of nonoverlapping matching k-length substrings[3] in the two strings (see Example 1). An $O(mn)$ time and space algorithm is proposed for computing and reconstructing the optimal $LCSk$. Deorowicz and Grabowski [6] correctly observed that $LCSk$ can be computed more efficiently. Out of several approaches, the proposed *Sparse* method allows both the computation of $LCSk$ metric and its reconstruction in $O(m + n + r \log l)$ time and $O(r)$ memory complexity, where l is the length of the optimal solution and r is the total number of matching k-length substring pairs between the input strings. In their approach they adapt the Hunt-Szymanski [3] paradigm in a way that makes them rely on the usage of persistent red-black binary tree.

A serious drawback to the $LCSk$ definition is that it considers only nonoverlapping matches with length of **exactly** k, thereby possibly ignoring substring matches with lengths of at **least** k (see Example 1). Therefore, we propose the $LCSk$++ measure, the longest common subsequence which removes this restriction. We still use the substrings of length k for computation, but they are allowed to overlap which results that the resulting common subsequence consists of nonoverlapping matching substrings with length of at least[4] k. We give an efficient $O(m + n + r \log r)$ time complexity algorithm in section 3.2. The algorithm utilizes only on a light-weight Fenwick tree [7] data structure. In section 4 we demonstrate the ability of $LCSk$++ to separate pairs of strings which are similar from unrelated ones under a realistic random model[5]. We discuss the influence of the parameter k on the performance and the separability. To the best of our knowledge, such discussion didn't appear in any previous related work. We conclude the paper with an $O((m+n) \log(m+n))$ time and $O(m+n)$ memory complexity algorithm with good separability under the presented random model.

Example 1. Consider three strings: $X{=}ABCBA$, $Y{=}ABCBA$ and $Z{=}ABCDE$ and let $k = 3$. $LCS3$ between every pair of these strings is equal to 1. The fact that X and Y are more similar than X and Z is not captured. That is because the k-length substrings are forbidden to overlap.

2 Preliminaries

In this section we formalize the concepts used in the remainder of the paper.

Definition 1 (Common subsequence). *Given two strings X and Y consider two sets of distinct indices $I = \{i_1, i_2, ..., i_n\}$ and $J = \{j_1, j_2, ..., j_n\}$ such that*

[3] Similarly as in [1], we use the term substring to denote a consecutive part of the string, while a subsequence is obtained by deleting symbols from arbitrary indices.

[4] Hence the plusses in the $LCSk$++.

[5] We call this property separability.

$i_1 < i_2 < ... < i_n$, $j_1 < j_2 < ... < j_n$ and $X_{i_x} = Y_{j_x}$ for $x = 1...n$. Sets I and J determine a ***common subsequence*** *of X and Y whose length is equal to n.*

Definition 2 (k++ common subsequence). *Consider a common subsequence of strings X and Y. Such subsequence uniquely determines two sets of indices I and J (as in definition 1). If both I and J can be partitioned into families of sets of consecutive indices such that every set has a size of at least k, this subsequence is called a* ***k++ common subsequence****.*

Definition 3 (LCSk++). *LCSk++ of two strings X and Y is the length of their k++ common subsequence with maximal number of elements.*

Definition 4. *We denote a substring of string X starting at index i and ending at index j by $X_{i...j}$. If $i > j$ then $X_{i...j}$ denotes an empty string.*

Example 2. Consider the same strings as in Example 1. Now $LCS3{+}{+}(X, Y)$ equals 5 and $LCS3{+}{+}(X, Z)$ equals 3, which reflects the fact that X and Y are more similar than X and Z. That is because of the k-length substrings are allowed to overlap.

3 Computation

3.1 Basic dynamic programming

The basic dynamic programming algorithm sequentially computes the values of $dp(i, j) = LCSk{+}{+}(X_{0...i-1}, Y_{0...j-1})$ via the following recursive relation:

$$dp(i,j) = \max \begin{cases} 0 & \\ dp(i-1,j) & i \geq 1 \\ dp(i,j-1) & j \geq 1 \\ dp(i-q,j-q)+q & \text{for all } q \geq k \text{ s.t. } X_{i-q...i-1} = Y_{j-q...j-1} \end{cases} \tag{1}$$

The 2nd and 3rd terms in the above formula correspond to inheriting the $LCSk{+}{+}$ value from previously computed values while the last term tries to extend the $LCSk{+}{+}(X_{0...i-q-1}, Y_{0...j-q-1})$ with $X_{i-q...i-1}$ and $Y_{j-q...j-1}$ if they are equal. Those terms contribute $|X_{i-q...i-1}| = |Y_{j-q...j-1}| = q$ to the resulting length. A direct implementation of the above idea leads to an algorithm with time complexity $O(nm \cdot \min(n, m))$.

3.2 Efficient algorithm

Definition 5 (Match pair[6]). *For a given strings X, Y and integer $k \geq 1$ we define:*

$$kMatch(i,j) = \begin{cases} 1 \; if X_{i+f} = Y_{j+f}, for\ every\ 0 \leq f \leq k-1 \\ 0 \; otherwise \end{cases} \tag{2}$$

[6] Original definition given in [1].

If kMatch(i,j)=1, we call (i,j) *a **match pair**. In other words, kMatch(i,j)=1 when the substring of* A *starting at* i *and having an length of **exactly*** k *is equal to the substring of* B *starting at* j *with the same length.* (i,j) *is also called the **start** and* $(i+k, j+k)$ *is called the **end** of the match pair.*

When the number of match pairs r is less than quadratic (consult Section 4 for the analysis), it is possible to compute $LCSk$++ efficiently. The time complexity of the algorithm we will describe in this section is $O(n+m+r\log r)$.

For every match pair $P = (i_P, j_P)$ we use dynamic programming to compute $dp(P) = LCSk{+}{+}(X_{0\ldots i_P+k-1}, Y_{0\ldots j_P+k-1})$, which represents the value of $LCSk$++ ending with P. The following definitions will be useful:

Definition 6 (Precedence of match pairs). *Let* $P{=}(i_P, j_P)$ *and* $G{=}(i_G, j_G)$ *be k-match pairs. Then* G ***precedes*** P *if* $i_G + k \leq i_P$ *and* $j_G + k \leq j_P$*. In other words,* G *precedes* P *if the end of* G *is on the upper left side of the start of* P *in the dynamic programming table (see Figure 1).*

Definition 7 (Continuation of match pairs). *Let* $P{=}(i_P, j_P)$ *and* $G{=}(i_G, j_G)$ *be k-match pairs. Then* P ***continues*** G *if* $i_P - j_P = i_G - j_G$ *(i.e. they are on the same primary diagonal) and* $i_P - i_G = 1$ *(*P *is only one down-right position from* G*, see Figure 1).*

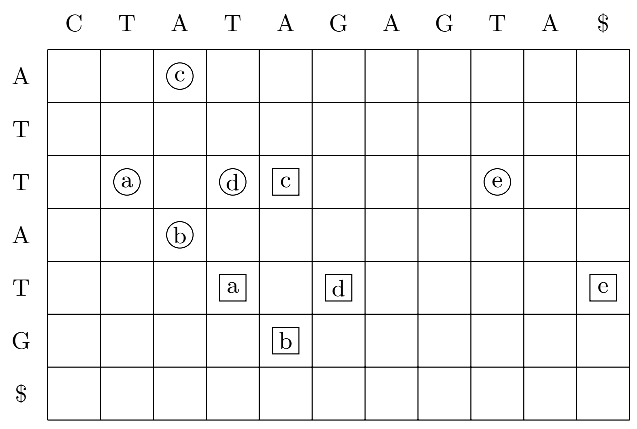


Fig. 1: $k = 2$; strings $X = ATTATG$ and $Y = CTATAGAGTA$ construct exactly five 2-match pairs denoted a to e. Starts are represented by circles, ends are represented by squares. The following holds: "b continues a", "c precedes e", while the following does not hold: "a precedes b", "c precedes d", "a continues b".

We can express the $dp(P) = LCSk{+}{+}(X_{0 \ldots i_P+k-1}, Y_{0 \ldots j_P+k-1})$ via the following formula which will be the basis for the efficient algorithm:

$$dp(P) = \max \begin{cases} k \\ \max_G dp(G) + k \text{ over all G preceding P} \\ dp(G) + 1 \qquad\qquad \text{if P continues G} \end{cases} \tag{3}$$

In other words, a k-match pair P can either start its own k-common subsequence, extend a k-common subsequence ending with a match G such that G precedes P or extend a k-common subsequence ending with a match pair G such that P continues G. In the second case the k-common sequence is enlarged by k (e.g. $c \to e$ in figure 1), while in the latter it's enlarged by 1 (e.g. $a \to b$ in figure 1).

Algorithm 1 Efficient $LCSk{+}{+}$ computation

```
 1: $MaxColDp \leftarrow$ 1D array filled with $n$ zeros
 2: $MatchPairs \leftarrow$ find all k-match pairs between $X$ and $Y$
 3: $events \leftarrow$ all starts and ends of $MatchPairs$ sorted in row-major order, if some
    start $S = (i_S, j_S)$ and some end $E = (i_E, j_E)$ share the same indices, $E$ should
    come first
 4: for all $event \in events$ do
 5:     if $event$ is a start $P = (i_P, j_P)$ then
 6:         $dp(P) \leftarrow k + max_{x \in 0 \ldots j_P}\ MaxColDp(x)$
 7:     else if $event$ is an end $P = (i_P + k, j_P + k)$ then
 8:         if $\exists G$ s.t. $P$ continues $G$ then
 9:             $dp(P) \leftarrow \max\{dp(P), dp(G) + 1\}$
10:         end if
11:         $MaxColDp(j_P + k) \leftarrow \max\{MaxColDp(j_P + k), dp(P)\}$
12:     end if
13: end for
14: return $\max_P dp(P)$
```

Algorithm 1 starts by extracting all of the r match pairs on line 2. This can be done in two ways: we can employ a suffix array in the exact same manner as in [6] to get the time complexity of $O(n+m+r)$. However, as k is small in practice[7] we can find all match pairs using a simple hash table in $O(n+m+kr) = O(n+m+r)$.

Line 3 creates events and sorts them, which ensures the correctness of the sweeping algorithm on lines 4-14. This can be accomplished in $O(r \log r)$ using a standard comparison based sorting algorithm. Line 8 can be implemented as a binary search over the *events* array. If $MaxColDp$ is implemented as a Fenwick tree [7], the operations on lines 6 and 11 have a cost of $O(\log n)$ which implies that the sweep algorithm runs in $O(r \log n)$. Overall complexity is $O(m+n+r \log r)$[8].

[7] Usually k will be small enough such that every k-length substring can be perfectly hashed using 64 bits, see section 4 for details.

[8] This is assuming that r is at least as big as n. In the other case the correct complexity is $O(m + n + r \log r + r \log n)$.

The memory complexity is $O(n+m+r)$ because we only need the space to save the match pairs and the $MaxColDp$ structure. If we would like to reconstruct the sequence, the dp array has to store $O(1)$ additional information per match pair: a pointer to the previous match pair in case some other match pair G preceded P or P continued some G in the optimal solution. We remark that by removing lines 8-10 this algorithm computes $LCSk$.

4 How to choose k?

In this section we will analyze the performance of the $LCSk{+}{+}$ measure on the following classification problem: given a pair of strings X and Y, decide whether they are similar to each other. To formalize this problem, next sections proposes two simple random models: one for the unrelated pairs of strings and one for similar pairs. We use this setting to demonstrate the influence of the parameter k to the performance and separability of $LCSk{+}{+}$.

On an efficiency note, it is natural to expect that the time complexity of the algorithm will decrease as the size of the alphabet increases (due to the diminishing number of match pairs). The interesting questions thus emerge when the size of the alphabet is small and the lengths of the strings are large. Such a setting is naturally found in DNA sequence alignment, which is an inspiration for the presented model.

4.1 Similarity model

We model a pair of **unrelated strings** $X^{(n)}$ and $Y^{(n)}$ of length n as random strings over the alphabet $\{A, C, G, T\}$ where each character has a known and mutually independent probability of appearing. This construction yields a constant $e_{unrelated}$ defined as $e_{unrelated} = P[X_i^{(n)} \neq Y_i^{(n)}]$.

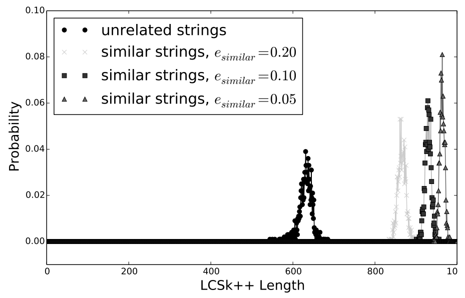
The model for **similar pairs** depends on a fixed parameter $e_{similar} < e_{unrelated}$. Pairs of strings are generated in a way that one of them is created randomly over $\{A, C, G, T\}$ using the same distributions as above, and the other one is its mutated copy satisfying restriction $P[X_i^{(n)} \neq Y_i^{(n)}] = e_{similar}$. The model reflects simplified evolutionary mutations (only substitutions are considered).

In order to distinguish between two classes, we would expect the following to hold (see Figure 2 for confirmation of this claim):

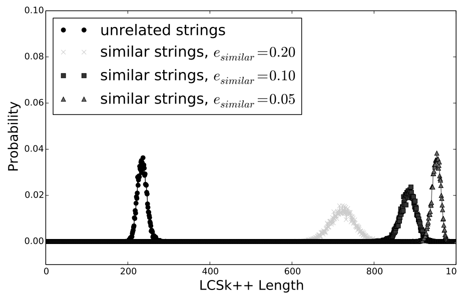
$$\frac{E[\text{LCSk++}(X_{similar}^{(n)}, Y_{similar}^{(n)})]}{n} > \frac{E[\text{LCSk++}(X_{unrelated}^{(n)}, Y_{unrelated}^{(n)})]}{n} \quad (4)$$

Computing the asymptotic behavior in Eq. 4 is still an open problem [8, 9], though the limit does exist - it can be shown by following the argument for the expectation of LCS for random strings by Chvátal and Sankoff in [10]. We compute them by a Monte Carlo simulation.
Unpublished work of Rabinovitch[11] experimentally demonstrates that the value of $\frac{E[LCS(X^{(n)},Y^{(n)})]}{n}$ increases with n, eventually hitting the limit. The value

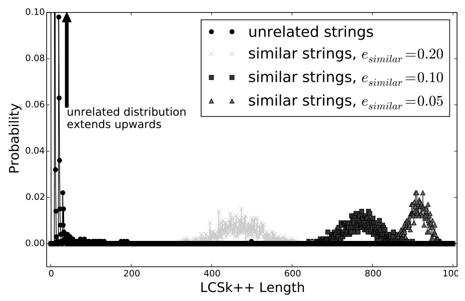
of $\frac{StdDev[LCS(X^{(n)},Y^{(n)})]}{n}$ decreases with n. Experiments presented in Table 1 confirm this behavior for $LCSk$++. Effectively this means that the separability gets better as n increases.



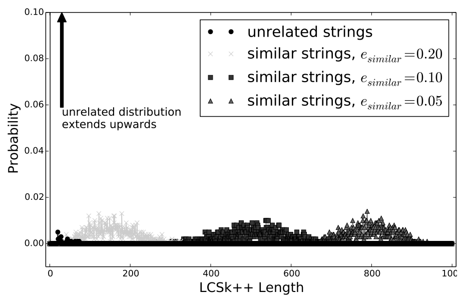
(a) $k = 1$



(b) $k = 5$



(c) $k = 10$



(d) $k = 20$

Fig. 2: $LCSk$++ **distributions** for strings of length 1000 and several values of k. It is clear that increasing k lowers the average score that string pairs achieve and increases the deviation. For our similarity model, the boundary of good separability is reached by setting k to around 20. DNA substrings of length 20 can be perfectly hashed in a 64-bit integer. That is useful for implementation because the step of finding all the match pairs can be implemented with a simple hash table.

4.2 Expected value of match pairs r

Let X, Y be either unrelated or similar pair of strings of length m, n constructed over the alphabet {A, C, G, T} with a priori distributions p_A, p_C, p_G, p_T. Then $S := E[X_i = Y_j] = (p_A^2 + p_C^2 + p_G^2 + p_T^2)$ for all $i \neq j$ (this restriction is needed to cover both models with this proof). Expected number of match pairs then equals:

$$
\begin{aligned}
E[r] &= E[\#\text{ of pairs (i,j) such that } X_{i..i+k-1} = Y_{j..j+k-1}] \\
&= E\left[\sum_{i=0}^{n-k}\sum_{j=0}^{m-k} \mathbb{1}[X_{i..i+k-1} = Y_{j..j+k-1}]\right] \\
&= E\left[\sum_{i=0}^{n-k}\sum_{j=0}^{m-k} \mathbb{1}[i=j]\mathbb{1}[X_{i..i+k-1} = Y_{j..j+k-1}]\right] + \\
&\quad E\left[\sum_{i=0}^{n-k}\sum_{j=0}^{m-k} \mathbb{1}[i\neq j]\mathbb{1}[X_{i..i+k-1} = Y_{j..j+k-1}]\right] \\
&= O(n+m) + \sum_{i=0}^{n-k}\sum_{j=0}^{m-k} E\left[\mathbb{1}[i\neq j]\mathbb{1}[X_{i..i+k-1} = Y_{j..j+k-1}]\right] \\
&= O(n+m+nmS^k) \qquad (5)
\end{aligned}
$$

Corollary 1. *By choosing* $k_{fast} = \log_{1/S}\frac{nm}{n+m}$ *it follows that* $E[r] = O(n+m)$*, so the expected complexity of the whole algorithm is* $O((n+m)\log(n+m))$*.*

Corollary 2. *For uniformly distributed alphabets, the expected number of match pairs drops as the size of the alphabet increases. That implies that the bigger the alphabet is, the smaller k is needed for the LCSk++ computation to run efficiently.*

Table 1: **Expected values and standard deviations of the** $LCSk$**++ distribution** for strings of length 100, 1000 and 10000, k of 10 and 20, and for unrelated and similar pairs (with various $e_{similar}$).

k	string length	error	$\frac{E[LCSk++]}{n}$	$\frac{Std.Dev.}{n}$	k	string length	error	$\frac{E[LCSk++]}{n}$	$\frac{Std.Dev.}{n}$
10	1000	unrelated	0.015	0.024	20	1000	unrelated	0.001	0.006
		0.20	0.470	0.051			0.20	0.154	0.057
		0.10	0.770	0.041			0.10	0.512	0.075
		0.05	0.911	0.025			0.05	0.793	0.058
	10000	unrelated	0.032	0.015		10000	unrelated	0.006	0.009
		0.20	0.471	0.017			0.20	0.154	0.018
		0.10	0.772	0.014			0.10	0.516	0.025
		0.05	0.914	0.008			0.05	0.801	0.018
	100000	unrelated	0.041	0.015		100000	unrelated	0.011	0.008
		0.20	0.471	0.006			0.20	0.154	0.006
		0.10	0.772	0.005			0.10	0.516	0.008
		0.05	0.914	0.003			0.05	0.801	0.006

5 Conclusions

In this paper we defined *LCSk*++, a similarity metric for long strings and we proposed an efficient algorithm for its computation. *LCSk*++ is a natural extension over the previous *LCSk* metric [1], with improved sensitivity. The only parameter k gives a flexible tradeoff between computational efficiency and sensitivity. Assuming strings X and Y follow a realistic random model, we presented a $O((|X| + |Y|) \log(|X| + |Y|))$ algorithm for computing the metric. Because of dependence on only a simple Fenwick tree data structure, the implementation is almost straightforward, making *LCSk*++ attractive for usage in practice.

Acknowledgments. The authors would like to thank Mario Lučić for valuable comments on the manuscript. This work was supported by the University of Zagreb under grant "Efficient algorithms for data processing in bioinformatics" and the Croatian academy of sciences and arts.

References

1. G. Benson, A. Levy, and B. R. Shalom, "Longest common subsequence in k-length substrings," *CoRR*, vol. abs/1402.2097, 2014.
2. C. Y. Chen, J. Y. Yeh, and H. R. Ke, "Plagiarism detection using rouge and wordnet," *arXiv preprint arXiv:1003.4065*, 2010.
3. J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, pp. 350–353, May 1977.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd ed., 2009.
5. B. S. Baker and R. Giancarlo, "Sparse dynamic programming for longest common subsequence from fragments," *J. Algorithms*, vol. 42, no. 2, pp. 231–254, 2002.
6. S. Deorowicz and S. Grabowski, "Efficient algorithms for the longest common subsequence in k-length substrings," *CoRR*, vol. abs/1311.4552, 2013.
7. P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.
8. R. Bundschuh, "High precision simulations of the longest common subsequence problem," *The European Physical Journal B - Condensed Matter and Complex Systems*, vol. 22, no. 4, pp. 533–541, 2001.
9. M. Kiwi and J. Soto, "On a speculated relation between Chvátal-Sankoff constants of several sequences," *Combinatorics, Probability & Computing*, vol. 18, no. 4, pp. 517–532, 2009.
10. V. Chvátal and D. Sankoff, "Longest common subsequences of two random sequences," *Journal of Applied Probability*, no. 12, pp. 306–315, 1975.
11. P. Rabinovitch, "Expected length of the longest common subsequence." December 2007.