# Efficient Approximation of Large LCS in Strings Over Not Small Alphabet

Gad M. Landau[1], Avivit Levy[2,3], and Ilan Newman[1]

[1] Department of Computer Science, Haifa University, Haifa 31905, Israel. E-mail: {landau,ilan}@cs.haifa.ac.il
[2] Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel.
[3] CRI, Haifa University, Mount Carmel, Haifa 31905, Israel. E-mail: avivitlevy@gmail.com

**Abstract.** A classical measure of similarity between strings is the length of the *longest common subsequence*(LCS) between the two given strings. The search for efficient algorithms for finding the LCS has been going on for more than three decades. To date, all known algorithms may take near-quadratic time to find large LCS. Since approximating LCS is trivial in strings over small alphabet, the focus of this paper is on approximating LCS efficiently in strings over not small alphabet. Also, since sparse LCS can be found in time polynomially smaller than quadratic, we focus on efficiently approximating LCS of near linear size. In this paper it is shown that large LCS can be efficiently approximated in strings with not small alphabet if the ED is not large. Specifically, if the alphabet is not small and the ED is not large, LCS of linear size can be approximated to a constant factor! For alphabet of size at least $n^{\epsilon}$, our algorithm complexity is always $O(n^{2-\epsilon}\log\log n)$ but can be much better (for some parameters it is $O(n\log\log n)$). Thus, a polynomially smaller than quadratic time algorithm which can find common subsequences of linear size is described in this paper for the first time. It is also shown that the best parameters for a given pair of strings can be quickly found by looking at local non-repetitiveness poly-logarithmic size sketches (LNR-sketches) of the strings.

## 1 Introduction

Measuring similarity plays an important role in data analysis. As strings are a common data representation, similarity measures defined on strings are widely used. A classical measure of similarity between strings is the length of the *longest common subsequence* (LCS) between the two given strings. The search for efficient algorithms for finding the LCS has been going on for more than three decades. The classical dynamic programming algorithm takes quadratic time [18, 19] and this complexity matches the lower bound in comparison model [1]. Many other algorithms have been suggested over the years [10, 11, 17, 4, 14, 15, 8] (see also [9]). However, the state of the art is still not satisfying. To date, all known algorithms may take near-quadratic time to find large LCS. None of the known algorithms can find LCS of linear size in time polynomially smaller than quadratic.

Analysis of large data bases storing very long strings cannot settle with such methods.

A possible approach is to trade accuracy for speed and employ faster algorithms that approximate the LCS. In fact, for measuring similarity a sufficiently long common subsequence as an evidence of similarity might be as good as the LCS itself. Thus, a good approximation of the LCS that can be found fast is of great importance. Strings over small alphabet have large LCS. Thus, LCS in strings over small alphabet can be trivially approximated to a factor of $1/|\Sigma|$, where $\Sigma$ is the alphabet, by just picking the letter that has the highest joint frequency. However, when the alphabet of the strings gets larger this approximation becomes useless. Therefore, the goal is to design efficient algorithms approximating LCS over strings with not small alphabet. Furthermore, it is known that sparse LCS can be found quickly. Specifically, if the LCS size is polynomially smaller than the string size, it can be found by algorithms that take time polynomially smaller than quadratic [10, 14, 15]. Thus, the focus of this paper is on efficiently approximating large LCS, typically, LCS of size near linear, in strings over not small alphabet.

*Related Work.* LCS is closely related to the *edit distance* (ED). The edit distance is the number of insertions, deletions, and substitutions needed to transform one string into the other. This distance is of key importance in several fields such as text processing, web search and computational biology, and consequently computational problems involving ED have been extensively studied. The ED is the dissimilarity measure corresponding to the LCS similarity measure. The ED can also be computed by a quadratic time dynamic programming procedure. In fact, using the methods of Landau and Vishkin [16], ED can be computed in time $\max\{k^2, n\}$, where $k$ is the bound on ED and $n$ the length of the strings. Thus, a fast algorithm can find if the ED is small or not. Approximating ED efficiently has proved to be quite challenging [3]. Currently, the best quasi-linear time algorithm due to Batu, Ergün and Sahinalp [6], achieves approximation factor $n^{1/3+o(1)}$, where $n$ is the length of the strings.

*Our Results.* In this paper it is shown that large LCS can be efficiently approximated in strings with not small alphabet if the ED is not large. Specifically, if the alphabet is not small and the ED is not large [4], LCS of linear size can be approximated to a constant factor! For alphabet of size at least $n^\epsilon$, our algorithm complexity is always $O(n^{2-\epsilon}\log\log n)$ but can be much better (for some parameters it is $O(n\log\log n)$). To the best of our knowledge, this is the first time that a polynomially smaller than quadratic time algorithm which can find common subsequences of linear size is described. The approximation ratio of our algorithm depends on the size of the LCS, i.e., it is better as the LCS is longer. The worst case complexity guarantee of our algorithm depends on the alphabet size. Table 1 demonstrates the performance of our algorithm for LCS of different

[4] the exact bound on the ED depends on the parameters of the strings and would be quantified below.

sizes. We stress that these are worst case performances also in the sense that they demonstrate the worst case parameters for given LCS size, alphabet size and period length, but the true parameters for a given pair of strings can be much better[5]. Our method works well for strings $A$ and $B$ where the ED is $o(\min\{LCS(A,B), \frac{n|\Sigma|}{t \ln t}\})$, where $\Sigma$ is the alphabet size and $t$ depends on the periodicity of the input strings (can be of size $n$ in aperiodic strings). The effect of these parameters is also demonstrated in Table 1.

**Table 1.** Worst Case Performance of Our Algorithms: Examples

| LCS | Alphabet Size | Period Length | ED | Approximation Ratio | Complexity |
|---|---|---|---|---|---|
| $\theta(n)$ | $\theta(n)$ | $\theta(n)$ | $o(n/\ln n)$ | $\theta(1)$ | $O(n \log\log n)$ |
| $\theta(n)$ | $\theta(n^\epsilon)$ | $\theta(n)$ | $o(n/\ln n)$ | $\theta(1)$ | $O(n^{2-\epsilon} \log\log n)$ |
| $\theta(n)$ | $\theta(n^\epsilon)$ | $\theta(n^\epsilon)$ | $o(n^\epsilon/\ln n)$ | $\theta(1)$ | $O(n^{2-\epsilon} \log\log n)$ |
| $\theta(n/\log^c n)$ | $\theta(n)$ | $\theta(n)$ | $o(n/\log^c n)$ | $\theta(1/\log^c n)$ | $O(n \log\log n)$ |
| $\theta(n/\log^c n)$ | $\theta(n^\epsilon)$ | $\theta(n)$ | $o(n^\epsilon/\ln n)$ | $\theta(1/\log^c n)$ | $O(n^{2-\epsilon} \log\log n)$ |
| $\theta(n/\log^c n)$ | $\theta(n^\epsilon)$ | $\theta(n^\epsilon)$ | $o(n/\log^c n)$ | $\theta(1/\log^c n)$ | $O(n^{2-\epsilon} \log\log n)$ |
| $\theta(n^{3/4})$ | $\theta(n)$ | $\theta(n)$ | $o(n^{3/4})$ | $\theta(1/n^{1/4})$ | $O(n \log\log n)$ |
| $\theta(n^{3/4})$ | $\theta(n^\epsilon)$ | $\theta(n)$ | $o(n^\epsilon/\ln n)$ | $\theta(1/n^{1/4})$ | $O(n^{2-\epsilon} \log\log n)$ |
| $\theta(n^{3/4})$ | $\theta(n^\epsilon)$ | $\theta(n^\epsilon)$ | $o(n^{3/4})$ | $\theta(1/n^{1/4})$ | $O(n^{2-\epsilon} \log\log n)$ |

Our techniques exploit local non-repetitiveness. We show that strings with not small alphabet have enough local non-repetitiveness that can be used to significantly speed-up approximating LCS. Local non-repetitiveness has been used in a limited context for approximating ED [5] and for embedding the ED [7]. Our use is much stronger because we show that good parameters of local non-repetitiveness always exist where not small alphabet is concerned. We also show that local non-repetitiveness can be efficiently sketched so that the best parameters for given two strings can be found by looking at a poly-logarithmic sketch. Our sketching results may be of independent value.

The paper is organized as follows. Sect. 2 presents basic definitions and properties. Sect. 3 presents approximation algorithms for the special case of (1,n/c)-non-repetitive strings, where $c$ is a parameter. In Sect. 4 we show how to transform strings with not small alphabet into the special case of (1,n/c)-non-repetitive strings. Our transformation has the strong property of having only an additive negligible distortion, if $ED = o(\min\{LCS(A,B), \frac{n|\Sigma|}{t \ln t}\})$. Finally, in Sect. 5 we show that the best parameters for a given pair of strings can be quickly found by looking at local non-repetitiveness sketches (LNR-sketches) of the strings. It is shown that our LNR-sketch size matches the lower bound, and a lower bound on the space needed by a LNR-sketching algorithm in the streaming model is also given.

[5] The true parameters are determined by the strong non-repetitiveness (to be defined below) parameters of the strings.

## 2 Preliminaries

In this section we give basic definitions and properties.

*Problem Definition.* Let $A$ and $B$ two $n$ length strings over alphabet $\Sigma$. The *longest common subsequence problem* is to find the longest subsequence, denoted by $LCS(A, B)$, appearing in both $A$ and $B$. We will abuse notation throughout the paper by letting $LCS(A, B)$ denote both the longest common subsequence and its length. It will be clear from the context which is referred to. The well-known Property 1 specifies the relation between the LCS and ED.

*Property 1.* Let $A$, $B$ be two strings of length $n$, then

$$n - LCS(A, B) \leq ED(A, B) \leq 2 \cdot (n - LCS(A, B)).$$

**Definition 1.** *Let $S$ be a string of length $n$. $S$ is called* periodic *if $S = P^i P'$, where $P$ is a substring of $S$ such that $|P| \leq n/2$, and $P'$ is a prefix of $P$. The smallest such substring $P$ is called the* period *of $S$. If $S$ is not periodic it is called* aperiodic.

**Definition 2. (Locally non-repetitive strings).** *A string $S$ is called $(t, w)$-non-repetitive if every $w$ successive $t$-substrings in $S$ are distinct, i.e. for each interval $\{i, \ldots, i + w - 1\}$, the $w$ substrings of length $t$ that start in this interval are distinct.*

**Definition 3. (Locally strongly non-repetitive strings).** *A string $S$ is called $(t, w, d)$-non-repetitive if for each interval $\{i, \ldots, i + w - 1\}$ every pair of $t$-substrings $s_i$, $s_j$ in $S$ starting in this interval have $\mathcal{H}(s_i, s_j) \geq d$, where $\mathcal{H}(s_i, s_j)$ is the hamming distance between $s_i$ and $s_j$ (i.e. the number of indices in which $s_i$ differ from $s_j$).*

*Remark.* Throughout the paper we refer to a wrap-around of the given string $S$, i.e. indices are taken modulo $n$, the length of the string. Thus, all $t$-substrings are well-defined for every $t$. If $S$ is periodic then the wrap-around while continuing the period from the point it is cut.

### 2.1 Properties of Locally Non-Repetitive Strings

*Property 2.* Let $S$ be a $(t, w)$-non-repetitive string, then:

1. $S$ is a $(t', w)$-non-repetitive string, for every $t' > t$.
2. $S$ is a $(t, w')$-non-repetitive string, for every $w' < w$.

*Property 3.* Let $S$ be a string of length $n$, then:

1. If $S$ is a periodic string with period $p$ then $S$ is a $(p, p)$-non-repetitive string.
2. If $S$ is aperiodic then $S$ is a $(n, w)$-non-repetitive string, where $n/2 \leq w \leq n$.

**Lemma 1.** *Let $S$ be a string of length $n$ over alphabet $\Sigma$ with period length $p$ ($p \geq |\Sigma|$), then $S$ is a $(p, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string. If $S$ is aperiodic then $S$ is a $(n, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string.*

*Proof.* We prove the lemma for a periodic string with period length $p$. The proof for aperiodic string is similar by Property 3. Let $s_i$ be any $p$-substring in $S$, and consider the $p$-substring $s_{i+j}$ for any $0 < j < |\Sigma|/2$. It is sufficient to show that $\mathcal{H}(s_j, s_{i+j}) \geq |\Sigma|/2$. Let $x_1, \ldots, x_{|\Sigma|}$ be the first appearances of the symbols of $\Sigma$ in $s_i$. We claim that $\mathcal{H}(s_j, s_{i+j}) \geq |\Sigma| - j$, because each $x_k$, $j+1 \leq k \leq |\Sigma|$, adds at least one mismatch to $\mathcal{H}(s_j, s_{i+j})$. The lemma follows.

## 3 Approximating LCS in (1,n/c)-Non-Repetitive Strings

In this section we present efficient algorithms to approximate the LCS if both strings are (1,n/c)-non-repetitive strings. The algorithms framework is based on the observation that a (1,n/c)-non-repetitive string for small values of parameter $c$ is sufficiently close to being a permutation string (i.e., a string with distinct characters). Finding the LCS in $n$-length permutation strings is actually finding the Longest Increasing Subsequence (LIS) of a string over the alphabet $\{1, \ldots, n\}$, which can be done fast.

### 3.1 $\theta(1/c)$-Approximation Algorithm

The algorithm first divides both input strings $A$ and $B$ into $c$ blocks of size $O(n/c)$. Since $A$ and $B$ are (1,n/c)-non-repetitive, each of their blocks is a permutation string. Therefore, the LCS between any block of $A$ and any block of $B$ can be found fast using the LIS algorithm. Our algorithm exploits this fact by finding the LIS between all $c^2$ pairs of block of $A$ and block of $B$, and chooses the pair with the best score. A detailed description of the algorithm is given in Fig. 1. Lemma 3 and Corollary 1 assure the approximation ratio of this algorithm. Lemma 2 gives its complexity guarantee. Theorem 1 follows.

```
ALGORITHM APPROX1LCS
Input: Two strings $A$, $B$ of length $n$, a parameter $c$
1  divide $A$, $B$ into $c$ blocks of size $O(n/c)$.
2  for each pair of blocks $A_i$, $B_j$ do
3        transform into blocks $A'_i$, $B'_j$ containing only the joint alphabet symbols.
4        $\ell_{i,j} \leftarrow LIS(A'_i, B'_j)$
5  $L_{alg} \leftarrow \max \ell_{i,j}$
Output:
6  $L_{alg}$
```

**Fig. 1.** $\theta(1/c)$-Approximation Algorithm for LCS in (1,n/c)-Non-Repetitive Strings.

**Lemma 2.** *Algorithm $Approx1LCS$ runs in $O(cn \log\log(n/c) + c^2)$ steps.*

*Proof.* It is a well-known fact that LIS can be computed in $(n \log\log n)$ time for $n$-length strings. Algorithm $Approx1LCS$ computes $c^2$ times LIS on strings of size $n/c$, for a total time of $O(cn \log\log(n/c))$ for steps 2-4. Step 5 takes another $c^2$ steps.

**Lemma 3.** *Let $A$ and $B$ be two strings of length $n$, then there exists a pair of blocks $A_i$, $B_j$ such that $l_{i,j} \geq \theta(1/c) \cdot LCS(A, B)$.*

*Proof.* Denote $LCS(A, B) = Opt$. For every $i$, $j$ denote by $LCS(A_j, B_j)$ the number of matches $Opt$ has between blocks $A_i$ and $B_j$. Clearly, $\ell_{i,j} \geq LCS(A_j, B_j)$. We now claim that there exists a pair $i$, $j$ such that $LCS(A_i, B_j) \geq \frac{Opt}{2e \cdot c}$.

Let $\alpha_i$ denote the number of matches $Opt = Opt_0$ has in block $A_i$. Let $k = 0$, $i = 1$, $j = 1$, $A^{(0)} = A$, $B^{(0)} = B$ and $\alpha_i^{(0)} = \alpha_i$. Consider the following process:

1. Consider the block $A_i$. If $\alpha_i^{(k)} < \frac{Opt_k}{2c}$ throw the block $A_i$ and let $i$ be $i + 1$. Throw from each of the blocks in $B^{(k)}$ the matches $Opt_k$ has with $A_i$ to form $B^{(k+1)}$. Since only $\alpha_i^{(k)}$ matches were thrown, $Opt_{k+1} = LCS(A^{(k)}, B^{(k+1)}) \geq (1 - \frac{1}{2c})Opt_k$.
2. Denote the matches of $Opt_{k+1}$ within block $B_j$ by $\beta_j^{(k+1)}$. If $\beta_j^{(k+1)} < \frac{Opt_{k+1}}{2c}$ throw the block $B_j$ and let $j$ be $j + 1$. Now throw from each of the blocks in $A^{(k)}$ the matches $Opt_{k+1}$ has with $B_j$ to form $A^{(k+1)}$. Since only $\beta_j^{(k)}$ matches were thrown, $Opt_{k+2} = LCS(A^{(k+1)}, B^{(k+1)}) \geq (1 - \frac{1}{2c})Opt_{k+1}$. Denote the matches of $Opt_{k+2}$ within block $A_i$ by $\alpha_i^{(k+2)}$.

Repeat the process $k$ times where in each time exactly one block is thrown until the first blocks $A_i$, $B_j$ such that each has $\geq \frac{Opt_k}{2c}$ matches, and therefore $LCS(A_i, B_j) \geq \frac{Opt_k}{2c} \geq \frac{Opt}{2c} \cdot (1 - \frac{1}{2c})^k \geq \frac{Opt}{2e \cdot c}$, or there are no more blocks to throw. Since the total number of blocks is $2c$, in the second case, the process stops with $Opt_{2c} \geq Opt \cdot (1 - \frac{1}{2c})^{2c} \geq Opt \cdot e^{-1}$, which cannot happen. Therefore, we must have stopped in the first case. The lemma then follows.

**Corollary 1.** *The approximation ratio of algorithm $Approx1LCS$ is $\theta(1/c)$.*

**Theorem 1.** *Let A,B be two (1,n/c)-non-repetitive strings then $LCS(A, B)$ can be approximated to a factor of $\theta(1/c)$ in $O(c \cdot n \log\log(n/c) + c^2)$ steps.*

### 3.2 $\theta(k/c)$-Approximation Algorithm

The $\theta(1/c)$ approximation ratio of algorithm $ApproxLCS1$ is quite well if $c$ is constant. However, as $c$ grows it gets worse. In fact, for $c = \sqrt{n}$ it gives nothing but a trivial approximation. We thus give another algorithm with the same framework as algorithm $ApproxLCS1$, in which additional work is done (but asymptotically takes the same time) in order to improve the approximation ratio. This new algorithm does not choose only one pair of blocks with best score, but rather gather a legal sequence of pair of blocks with total best score. A legal sequence does not contain crossing pairs. Clearly, any legal sequence defines a common subsequence of $A$ and $B$. Fortunately, such a legal sequence of pairs can be found by a dynamic programming procedure in $O(c^2)$ time. A detailed description of the algorithm is given in Fig. 2. Lemma 5 assures the approximation ratio of this algorithm. Lemma 4 gives its complexity guarantee. Theorem 2 follows.

```
Algorithm Approx2LCS
Input: Two strings $A$, $B$ of length $n$, a parameter $c$
1  divide $A$, $B$ into $c$ blocks of size $O(n/c)$.
2  for each pair of blocks $A_i$, $B_j$ do
3          transform into blocks $A'_i$, $B'_j$ containing only the joint alphabet symbols.
4          $\ell_{i,j} \leftarrow LIS(A'_i, B'_j)$
5  construct a weighted bipartite graph $G =< V1 \cup V2, E >$ with weight function
           $W : E \rightarrow \mathcal{N}$, where:
           $V1 = \{i \mid A_i \textit{ is a block in } A\}$
           $V2 = \{j \mid B_j \textit{ is a block in } B\}$
           $E = \{(i,j) \mid i \in V1 \textit{ and } j \in V2\}$
           $W(i,j) = \ell_{i,j}$
6  $L_{alg} \leftarrow MaximumWeightLegalSequence(G, W)$
Output:
7  $L_{alg}$
```

**Fig. 2.** $\theta(k/c)$-Approximation Algorithm for $LCS \geq kn/c$ in (1,n/c)-Non-Repetitive Strings.

**Lemma 4.** *Algorithm Approx2LCS runs in $O(cn \log\log(n/c) + c^2)$ steps.*

*Proof.* Lines 1-4 of the algorithm are identical to algorithm *Approx*1*LCS* and therefore cost $O(cn \log\log(n/c))$ steps as computed in Lemma 2's proof. The graph construction in Line 5 can be done in time linear with its size. Since the graph has $2c$ vertices and $c^2$ edges, line 5 can be computed in $O(c^2)$ steps. The maximum weighted legal sequence computation in line 6 can be done in $O(c^2)$ steps (linear in the size of the graph) by using a simple dynamic programming procedure based on the following recursion relation:

$$OPT(i,j) = \min\{w(i,j) + OPT(i-1, j-1), OPT(i, j-1), OPT(i-1, j)\},$$

where $OPT(i,j)$ is the maximum weighted legal sequence defined on the subgraph containing only vertices $\{i' \in V1 \mid \imath' \leq i\}$ and $\{j' \in V2 \mid j' \leq j\}$. The dynamic programming table is computed row, column, alternately. Since, by the recursion relation each cell can be computed in $O(1)$ time, the overall computation takes $O(c^2)$. The lemma follows.

**Lemma 5.** *Algorithm Approx2LCS approximates $LCS(A,B) \geq kn/c$ to a factor of $\theta(k/c)$.*

*Proof.* Let $A_1, \ldots, A_r$ be the blocks in $A$ that participate in $LCS(A,B)$ and let $\alpha_1, \ldots, \alpha_r$ be the fraction (of $n$) that each of them contributes to $LCS(A,B)$, respectively. Since each block is of size $n/c$, $\forall i$, $\alpha_i \leq 1/c$. Also, $Opt = \sum \alpha_i \geq k/c$. For each block $A_i$ let $k_i$ be the number of blocks in $B$ that participate in the matches of $LCS(A,B)$. Since there are $c$ blocks in $B$ and the matches do not cross, $\sum k_i \leq c$. Note that $L_{alg} \geq \sum \alpha_i/k_i$, because the algorithm chooses the longest path, therefore, for each block $A_i$ at least the average contribution $\alpha_i/k_i$ is taken by the algorithm.

Split the set of blocks in $A$ into two sets, the set $X$ of blocks for which $k_i > 2c/k$, and the rest of the blocks.

**Claim 1.** $\sum_{A_i \in X} \alpha_i \leq \frac{Opt}{2}$.
Since $|X| \leq \sum k_i / \frac{2c}{k} \leq \frac{k}{2}$, and therefore, $\sum_{A_i \in X} \alpha_i \leq \frac{k}{2} \cdot \frac{1}{c} \leq \frac{Opt}{2}$.

**Claim 2.** $L_{alg} \geq \frac{k}{4c} \cdot Opt$.
Since $L_{alg} \geq \sum \alpha_i/k_i \geq \sum_{A_i \notin X} \alpha_i/k_i \leq \frac{k}{2c} \sum_{A_i \notin X} \alpha_i$, and by Claim 1 this is at least $\frac{k}{2c} \cdot \frac{Opt}{2}$.

The lemma then follows.

**Theorem 2.** *Let A,B be two (1,n/c)-non-repetitive strings then $LCS(A,B) \geq kn/c$ can be approximated to a factor of $\theta(k/c)$ in $O(c \cdot n \log\log(n/c) + c^2)$ steps.*

## 4 Approximating Large LCS in Strings with Not Small Alphabet

By Lemma 1, not small alphabet assures a large enough parameter $w$ of local strong non-repetitiveness. We will exploit this to define a transformation to (1,n/c)-non-repetitive strings, for which the solutions of Sect. 3 are applicable. This transformation has the strong property of having only an additive negligible distortion, if the LCS is large and the ED is not large (the term "large" would be quantified below). Thus, it enables approximating large LCS in general strings having not small alphabet with effectively the same approximation ratio as the algorithms for (1,n/c)-non-repetitive strings, provided that the ED is not large. For clarity of exposition, a simple idea of a transformation that may have an unbearable distortion is described first. After analyzing its weaknesses it is shown how these can be overcome by defining an efficient randomized transformation.

*A Naive Transformation.* The idea is to exploit the property that every $n$ length string $S$ over alphabet $\Sigma$ is a $(t,w)$-non-repetitive string for some $|\Sigma| \leq t \leq n$, $|\Sigma| \leq w \leq n$. Each new $t$-substring defines a new symbol (overall, a linear number of new symbols). This transformation yields a (1,n/c)-non-repetitive string where $c \leq \frac{2n}{|\Sigma|}$, and since $|\Sigma|$ is not small the algorithms of Sect. 3 are efficient.

We now analyze the distortion of this transformation. Given the original $n$-length strings $A$ and $B$, denote by $A'$, $B'$ the strings after the transformation. Clearly, $LCS(A', B') \leq LCS(A, B)$ because positions with different symbols remain different. Also, each of the $n - LCS(A, B)$ symbols that do not participate in $LCS(A, B)$ affects only $t$ substrings, thus, $LCS(A', B') \geq n - t(n - LCS(A, B)) = LCS(A, B) - (t-1)(n - LCS(A, B))$. By Property 1 we get $LCS(A', B') \geq LCS(A, B) - \frac{t-1}{2} \cdot ED(A, B)$. Thus, this transformation has an additive distortion affected both by $t$ and $ED(A, B)$, which can both be $\Omega(n)$!

*The Randomized Transformation.* Fix a random binary vector $v$ of length $t-1$, where each coordinate is 1 with probability $\frac{2d \ln t}{|\Sigma|}$ for a constant $d > 2$, and 0 otherwise. Note that $v$ is well defined for not small alphabet, since $\Sigma \geq n^{\epsilon}$ and $t \leq n/2$, thus, $\frac{2d \ln t}{|\Sigma|} = o(1)$. Given an $n$-length string $S$ over alphabet $\Sigma$ define $f(S)$ as follows. Each location $i$ is given a symbol $\sigma(i)$ which identifies the string $S_i, S_{i_1}, \ldots, S_{i_k}$, where $S_{i_1}, \ldots, S_{i_k}$ are the locations in the $(t-1)$-substring starting at position $i+1$ in $S$ for which the corresponding coordinates in $v$ are 1.

**Lemma 6.** *Let $S$ be a string over alphabet $\Sigma$ then, there exists a parameter $t$, $|\Sigma| \leq t \leq n$ such that $f(S)$ is $(1, |\Sigma|/2)$-non-repetitive string with probability at least $1 - 1/t^{d-2}$.*

*Proof.* By Lemma 1, there exists a $t$, $|\Sigma| \leq t \leq n$, such that $S$ is a $(t, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string. Let $i, j$ be any indices in $S$ such that $|i - j| < |\Sigma|/2$, and let $s_i$ be the $t$-substring starting at position $i$ in $S$. By Lemma 1 we have $\mathcal{H}(s_i, s_j) \geq |\Sigma|/2$. We first claim that

$$Prob[\mathcal{H}(f(s_i), f(s_j)) = 0] \leq 1/t^d.$$

This is because $Prob[\mathcal{H}(f(s_i), f(s_j)) = 0] = (1 - \frac{2d \ln t}{|\Sigma|})^{|\Sigma|/2}$, if non of the $|\Sigma|/2$ coordinates in which $s_i$ and $s_j$ differ are chosen. Thus, by the union bound

$$Prob[\exists i, j | \mathcal{H}(f(s_i), f(s_j)) = 0] \leq 1/t^{d-2}.$$

The lemma follows.

**Lemma 7.** *Let $A$, $B$ be $n$-length strings over alphabet $\Sigma$, then*

$$LCS(f(A), f(B)) \geq LCS(A, B) - \frac{d(t-1) \ln t}{|\Sigma|} \cdot ED(A, B)$$

*Proof.* First note that $LCS(A, B) \geq LCS(f(A), f(B))$, because positions with different symbols in $A$ and $B$ remain different in $f(A)$ and $f(B)$. We now bound the contraction factor of $f$. Since by the definition of the randomized transformation $f$ the first symbol of the $i$-th $t$-substring is always taken and the rest $i+1,\dots,i+t-1$ locations of the $i$-th $t$-substring are taken with probability $\frac{2d\ln t}{|\Sigma|}$ for a constant $d > 2$, we have: $LCS(f(A),f(B)) \geq n - (1 + \frac{2(t-1)d\ln t}{|\Sigma|})(n - LCS(A,B)) = LCS(A,B) - \frac{2(t-1)d\ln t}{|\Sigma|} \cdot (n - LCS(A,B)) \geq LCS(A,B) - \frac{d(t-1)\ln t}{|\Sigma|} \cdot ED(A,B)$, where the last inequality is due to Property 1.

Thus, if $ED = o(\min\{LCS(A,B), \frac{n|\Sigma|}{t\ln t}\})$, this transformation has an additive negligible distortion. Theorem 3 follows.

**Theorem 3.** *Let A,B be two strings over alphabet $\Sigma$. Then, there exists a parameter t, $|\Sigma| \leq t \leq n$, such that if $ED(A,B) = o(\min\{LCS(A,B), \frac{n|\Sigma|}{t\ln t}\})$, any algorithm approximating $LCS(f(A),f(B))$ to a factor of $\alpha$ in $O(\beta(n))$ steps, can be used to approximate $LCS(A,B)$ to a factor of $\alpha + o(1)$ in $O(\beta(n))$ steps.*

## 5 Sketching Local Non-Repetitiveness

In this section we show that the best $w$ and $t$ parameters for a given pair of strings can be quickly found by looking at local non-repetitiveness sketches (LNR-sketches) of the strings. We construct a local non-repetitiveness sketch of size $O(\log^2 n)$ which gives the exact parameter $w$ for which the best $t$ parameter is approximated to a factor of 2. A construction of local strong non-repetitiveness (LSNR-sketch) is also described. We also show that our LNR-sketch size matches the lower bound. A lower bound on the space needed by a LNR-sketching algorithm in the streaming model is also given.

### 5.1 The LNR-Sketching Algorithms

If both $t$ and $w$ are given in advance, a trivial sketch of one bit can be built. Simply, keep the one bit answer of the check if $S$ is a $(t,w)$-non-repetitive string. This check can obviously be done in time $O(tn)$, and therefore the sketching algorithm is efficient (i.e., has a polynomial time complexity). In the sequel, we assume that the $t$ and $w$ parameters are unknown when the sketching is done, which is the interesting case. We explain the algorithms for a given $t$ parameter, and then use them for the case that $t$ is not given.

*Sketching with a Given t.* The sketching algorithms are based on finding the minimum distance between any repeating $t$-substrings. This distance is returned as the $w$ parameter. The correctness of this returned value is ensured by Property 2. The number of bits needed to store this value is $O(\log n)$. Finding the minimum distance between any repeating $t$-substrings can be found either by a $O(n\log^2 t)$ time deterministic algorithm or by a $O(n)$ time randomized algorithm. The deterministic algorithm uses a renaming process as in the string matching algorithm of Karp-Miller-Rosenberg [12]. It is usually assumed, for convenience, that $t$ is a power of 2. This assumption can be removed by using standard splitting techniques, while adding only a $O(\log t)$ factor to the $O(n\log t)$

complexity. The randomized algorithm uses the Rabin-Karp string matching algorithm [13] to produce a distinct polynomial representing each $t$-substring with high probability. In both the deterministic and the randomized algorithm after the "names" representing the $n$ $t$-substrings are determined all is needed is a linear scan to find the minimum distance between repeating "names".

*Sketching with Unknown t.* In order to have the $w$ for every $t$, we find the exact parameter $w$ for every $t = 2^i$, $0 \leq i \leq \log n$. For each such $t$ we use the algorithms described above for a given $t$. Since we only do that for $O(\log n)$ values of $t$, and for each the sketch size is $O(\log n)$ we get a total $O(\log^2 n)$ sketch size. For each value $t$ , the $w$ parameter is the one stored for the closest power of two that is less than or equal to $t$. The correctness of this value is ensured by Property 2.

**Theorem 4.** *Let $A$, $B$ be $n$ length strings, then, there exist (almost) linear algorithms giving LNR-sketch of size $O(\log^2 n)$ enabling finding the maximum $w$ and approximating to a factor of 2 the minimum $t$ for which $A$ and $B$ are both $(t, w)$-non-repetitive.*

### 5.2 Sketching Locally Strong Non-Repetitiveness

The transformation from general strings to (1,n/c)-non-repetitive strings described in Sect. 4 requires a locally strong non-repetitiveness, which is not detected by the algorithms described in Sect. 5.1. Nevertheless, in this section we show that the ideas of the sketching algorithms described in Sect. 5.1 can be used also for this case. To this end, the substrings as defined by the binary vector $v$, are considered. Observe that both the deterministic and randomized sketching algorithms described in Sect. 5.1 work as well for non-contiguous strings. Such non-standard use of the KMR algorithm also appears in [2]. Note that the binary vector $v$ depends only on $\Sigma$ and $t$ and is independent of $S$. Thus, the definition of the vector can be done in the sketching time. Also, note that in order to be able to compare any two strings (with possibly different size of joint alphabet and different $t$ parameter) we must define a $v$ vector for each possible pair. To cover all possible values of $\Sigma$, for each $t$ a power of two, $O(\log^2 n)$ vectors $v$ (for each $\Sigma$ a power of two and $t$ a power of two) are computed. Once a specific vector $v$ is defined, the sketch for non-repetitiveness can be done as explained in Sect. 5.1. Since $O(\log^2 n)$ sketches of size $O(\log n)$ are used Theorem 5 follows.

**Theorem 5.** *Let $A$, $B$ be $n$ length strings over alphabet $\Sigma$, then, there exist (almost) linear algorithms giving LSNR-sketch of size $O(\log^3 n)$ enabling finding the maximum $w$ and approximating to a factor of 2 the minimum $t$ for which $A$ and $B$ are both $(t, w, d)$-non-repetitive, where $d \geq \Sigma/2$.*

### 5.3 The Lower Bound on LNR-Sketch size

Note that the $w$ parameter as a function of $t$ is a nondecreasing monotone function that take values on the range $\{1, \ldots, n\}$. We show a feasible set of monotone sequences, i.e., monotone sequences that represent $w$ as a function of $t$ for some string. The size of this set gives a lower bound on the number of bits needed to represent a LNR-sketch.

**Lemma 8.** *The size of the feasible set is at least* $(\frac{n}{\log n})^{\log n}$.

*Proof.* First, observe that the following is a feasible set of sequences. Divide the range $\{1, \ldots, n\}$ into $n/\log n$ blocks. In each block choose one point to be the value of $w$ for all $t$ values in the block range. The number of different sequences in this set is $(n/\log n)^{\log n}$.

The next theorem is an immediate corollary of Lemma 8.

**Theorem 6.** *Any LNR-sketch of $n$-length string requires $\Omega(\log^2 n)$ bits.*

### 5.4 A $\Omega(n/\log n)$ Space Lower Bound of LNR-Sketching Algorithms in Streaming Model

We now show that LNR-sketch cannot be done in streaming model. Consider the following one-round two-party communication setting for the problem. Alice has a string $S1$ of length $n$ and Bob has a string $S2$ of length $n$. Alice and Bob should decide whether there exists a $t$-substring in $S1$ repeating in $S2$ while Alice may pass at most $k$ bits to Bob. We call this setting the *repeating $t$-substring problem.* The next lemma shows that $k = \Omega(n)$. Theorem 7 follows.

**Lemma 9.** *The repeating $t$-substring problem requires passing $\Omega(n)$ bits.*

*Proof.* We show that the following instance of the repeating $t$-substring problem: $S1$ is $\pi_1 \in S_n$ and $S2$ is $\pi_2 \in S_n$, requires passing $\Omega(n)$ bits. Consider the boolean matrix for all possible pairs $<\pi_1, \pi_2> \in S_n \times S_n$, representing wether or not a $t$-substring in $\pi_1$ appears in $\pi_2$. The $k$ bits that are passed from Alice to Bob divide this matrix into $2^k$ parts that can be separated using the $k$ passed bits. However, within each part the passed bits give no separating information. Thus, the matrix entrance within each part must depend solely on $\pi_2$. Therefore, in each part the matrix rows within each column must be all zeroes or all ones. The number of permutations for which there exists a repeating $t$-substring, i.e. the number of columns for which all rows are 1, is: $n \cdot n \cdot (n-t)!$. Since the matrix is divided into $2^k$ parts there exists a part with $1/2^k$-fraction of the total number of permutations. Thus, $2^k \geq \frac{n!}{n \cdot n \cdot (n-t)!} \approx \frac{(n/e)^n}{((n-t)/e)^{(n-t)}}$. Therefore, $k \geq n \log n - (n-t)\log(n-t) + t \log e = \Omega(n)$ (because if $t \leq n/2$ then the $n \log n$ term is dominant, otherwise the $O(t)$ term is $\Omega(n)$).

**Theorem 7.** *Any LNR-sketching deterministic algorithm in streaming model requires $\Omega(n/\log n)$ space.*

## References

1. A. V. Aho, D. S. Hischberg, and J. D. Ulman, *Bounds on the complexity on the longest common subsequence problem*, JACM **23** (1976), no. 1, 1–12.
2. A. Amir, Y. Aumann, O. Kapah, A. Levy, and E. Porat, *Approximate string matching with address bit errors*, CPM (P. Ferragina and G. M. Landau, eds.), 2008, pp. 118–129.
3. A. Andoni and R. Krauthgamer, *The computational hardness of estimating edit distance*, FOCS, 2007, pp. 724–734.
4. A. Apostolico and C. Guerra, *The longest common subsequence problem revisited*, Algorithmica **2** (1987), 315–336.
5. Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, and R. Kumar, *Approximating edit distance efficiently*, FOCS, 2004, pp. 550–559.
6. T. Batu, F. Ergün, and C. Sahinalp, *Oblivious string embeddings and edit distance approximation*, SODA, 2006, pp. 792–801.
7. M. Charikar and R. Krauthgamer, *Embedding the ulam metric into $\ell_1$*, Theory of Computing **2** (2006), 207–224.
8. M. Crochemore, G. M. Landau, and M. Ziv-Ukelson, *A sub-quadratic sequence alignment algorithm for unrestricted cost matrices*, SIAM J. Comput. **32** (2003), no. 5, 1654–1673.
9. D. Gusfield, *Algorithms on strings, trees and sequences*, Cambridge University Press, 1997.
10. D. S. Hirshberg, *Algorithms for the longest common subsequence problem*, JACM **24** (1977), no. 4, 664–675.
11. J. W. Hunt and T. G. Szymanski, *A fast algorithm for computing longest common subsequences*, CACM **20** (1977), 350–353.
12. R. Karp, R. Miller, and A. Rosenberg, *Rapid identification of repeated patterns in strings, arrays and trees*, Symposium on the Theory of Computing **4** (1972), 125–136.
13. R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development **31** (1987), no. 2, 249–260.
14. G. M. Landau, B. Scheiber, and M. Ziv-Ukelson, *Sparse lcs commom substring alignment*, Information Processing Letters **88** (2003), no. 6, 259–270.
15. G.M. Landau and M. Ziv-Ukelson, *On the common substring alignment problem*, Journal of Algorithms **41** (2001), no. 2, 338–359.
16. G. M. Landua and U. Vishkin, *Fast string matching with k differences*, Journal of Computer and System Sciences **37** (1988), no. 1, 63–78.
17. W. J. Masek and M. S. Paterson, *A faster algorithm for computing string edit distances*, JCSS **20** (1980), 18–31.
18. D. Sankoff, *Matching sequences under deletion/insertion constraints*, Pro. Nat. Acad. Sct. USA, vol. 69, January 1972, pp. 4–6.
19. R. A. Wagner and M. J. Fischer, *The string to string correction problem*, JACM **21** (1974), no. 1, 168–173.