Information and Computation 209 (2011) 705–716

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

LCS approximation via embedding into locally non-repetitive strings[☆,☆☆]

G.M. Landau[a,b,1,2], A. Levy[b,c,*], I. Newman[a,2]

[a] Department of Computer Science, University of Haifa, Haifa 31905, Israel
[b] Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel
[c] CRI, University of Haifa, Mount Carmel, Haifa 31905, Israel
[d] Department of Computer Science and Engineering, NYU-Poly, Six MetroTech Center, Brooklyn, NY 11201-3840, USA

ARTICLE INFO

Article history:
Received 13 October 2009
Revised 16 November 2010
Available online 23 December 2010

Keywords:
String algorithms
LCS approximation
Embedding

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 quadratic time (shaved by logarithmic factors) to find large LCS. In this paper, the problem of approximating LCS is studied, while focusing on the hard inputs for this problem, namely, approximating LCS of near-linear size in strings over a relatively large alphabet (of size at least n^{ϵ} for some constant $\epsilon > 0$, where n is the length of the string). We show that, any given string over a relatively large alphabet can be embedded into a *locally non-repetitive string*. This embedding has a negligible additive distortion for strings that are not too dissimilar in terms of the edit distance. We also show that LCS can be efficiently approximated in locally-non-repetitive strings. Our new method (the embedding together with the approximation algorithm) gives a strictly sub-quadratic time algorithm (i.e., of complexity $O(n^{2-\epsilon})$ for some constant ϵ) which can find common subsequences of linear (and near linear) size that cannot be detected efficiently by the existing tools.

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 [22,23] and this complexity matches the lower bound in comparison model [1]. Many other algorithms have been suggested over the years [6,7,11,13,14,17,19,20] (see also [12]). However, the state of the art is still not satisfying. To date, all known algorithms may take near-quadratic time (i.e., quadratic shaved by logarithmic factors) to find large LCS. While sub-quadratic time algorithms are known for sub-linear size LCS (e.g. [13]), none of the known algorithms can find LCS of linear size in time polynomially smaller than quadratic, in the worst case (see discussion below). Analysis of very long strings and large databases cannot settle with such methods.

☆ A preliminary version appeared in the proceedings of CPM 2009.
☆☆ This work was partially supported by the Israel Science Foundation (Grant No. 1011/06).
* Corresponding author at: Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel.
E-mail addresses: landau@cs.haifa.ac.il (G.M. Landau), avivitlevy@shenkar.ac.il, avivitlevy@gmail.com (A. Levy), ilan@cs.haifa.ac.il (I. Newman).
1 Partially supported by the National Science Foundation Award 0904246, Israel Science Foundation Grant 347/09, Yahoo, Grant No. 2008217 from the United States–Israel Binational Science Foundation (BSF) and DFG.
2 Fax: +972 4 824 9331.

0890-5401/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2010.12.006

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.

1.1. Approximating LCS in strings over small alphabet

The LCS can be trivially approximated to a factor of $1/|\Sigma|$, where Σ is the alphabet, by just picking the symbol that has the highest joint frequency, i.e., the symbol that has the majority of appearances in both strings. This means that we take the subsequence of both input strings consisting of the most frequent symbol. If the alphabet size is $o(n^\epsilon)$ for every constant $\epsilon > 0$, which we call in this paper *small alphabet*, this trivial algorithm achieves sub-polynomial approximation ratio, which was until very recently roughly the best known approximation ratio for the closely related edit distance [5,21].[3] However, when the alphabet of the strings gets larger this approximation becomes useless. Therefore, our goal is to design efficient algorithms approximating LCS over strings with relatively large alphabet, i.e., alphabet of size at least n^ϵ.

1.2. Sparse vs. large LCS

There are known algorithms that take advantage of some given bounds or assumptions on the size of the LCS in order to give better algorithms to find the LCS. Such algorithms typically introduce a tradeoff between the given bound on the LCS and the time to find the LCS. We refer to such methods as *Sparse LCS techniques*. A relatively large alphabet may reduce the number of matching symbols between the two given strings. In such cases the technique of Hunt–Szymanski can be used to give efficient exact solutions that depend on the matchings set size [7,14]. However, the input strings may have a quadratic size of matching pairs of symbols even if the alphabet is relatively large. In these cases, the Hunt–Szymanski algorithm takes quadratic time. Other algorithms that assume some given bound on the LCS are known. Specifically, LCS of size $O(n^\alpha)$, where n is the string size and $0 < \alpha < 1$ is a constant, can be found in time $O(n^{1+\alpha})$ [13]. Also, if the input strings are LZ78-compressible then the LCS can be found in time quadratic in the number of codewords in the LZ78 compression [17,19]. However, these algorithms may still take quadratic time for finding LCS. Specifically, LCS of linear size and near-linear size in non-compressible strings cannot be found in polynomially sub-quadratic time by these algorithms. Thus, the focus of this paper is on efficiently approximating large LCS in strings over a relatively large alphabet. In fact, we will also present a tradeoff, but our algorithm would perform better (in terms of the approximation ratio) as the LCS gets larger.

Note that, increasing the alphabet size artificially typically does not enable better results. Introducing new symbols to both strings, increases the LCS and, therefore, may cause a loss of the information on the original LCS. On the other hand, introducing different new symbols to each of the strings increases their edit distance. The increase of the ED badly affects the possibility of using the advantage of a large alphabet, as we show in this paper.[4] In this aspect, this paper clarifies another connection between the LCS and ED measures.

1.3. 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. The ED can also be computed by a quadratic time dynamic programming procedure. In fact, using the methods of Landau and Vishkin [18], 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]. Batu et al. [9] gave a quasi-linear time algorithm that achieves approximation factor $n^{1/3+o(1)}$, where n is the length of the strings. The approximation ratio was improved by Andoni and Onak [5] to $2^{\tilde{O}(\sqrt{\log n})}$ by building on the seminal work of Ostrovsky and Rabani [21]. Only recently a poly-logarithmic approximation algorithm for ED was introduced [4].

1.4. Results

In this paper, it is shown that large LCS can be efficiently approximated in strings with a relatively large alphabet if the ED is not too large. Our algorithm has an additional parameter that depends on the *periodicity* of the input strings. Intuitively, a periodic string is a string where there is a prefix that repeats itself throughout the string, possibly cut in the end of the string. If such a phenomenon does not exist, the string is called *aperiodic* (see formal definitions in Section 2). We show that, in particular, LCS of linear size can be approximated to a **constant** factor, if the edit distance is $o\left(\frac{n|\Sigma|}{t \ln t}\right)$, where $|\Sigma|$ is the alphabet size and t in the worst case is the smaller period size in the given input strings ($t = n$ in aperiodic strings). It is important to note, that our algorithm does not need to verify that the requirement on the ED is indeed fulfilled. If the requirement is not fulfilled the algorithm might fail to detect an existing large common subsequence, however, a large

[3] Ostrovsky and Rabani [21] show an embedding into ℓ_1, which is stronger than an approximation algorithm. However, the time complexity of the embedding is high. It can, therefore, be used for various tasks such as sketching and nearest neighbor search, but not as an edit-distance approximation algorithm.

[4] It increases the distortion of the embedding into locally-non-repetitive strings.

Table 1
Worst case performance of our algorithm compared to sparse LCS techniques.

LCS size	Alphabet size	Period size	ED limit	Approximation ratio	Complexity	Sparse LCS Techniques
$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$o\left(\frac{n}{\ln n}\right)$	$\Theta(1)$	$O(n \log n)$	$\Theta(n^2)$
$\Theta(n)$	$\Theta(n^\epsilon)$	$\Theta(n)$	$o\left(\frac{n^\epsilon}{\ln n}\right)$	$\Theta(1)$	$O\left(n^{2-\epsilon} \log \log n\right)$	$\Theta(n^2)$
$\Theta(n)$	$\Theta(n^\epsilon)$	$\Theta(n^\epsilon)$	$o\left(\frac{n}{\ln n}\right)$	$\Theta(1)$	$O\left(n^{2-\epsilon} \log \log n\right)$	$\Theta(n^2)$
$\Theta\left(\frac{n}{\log^c n}\right)$	$\Theta(n)$	$\Theta(n)$	$o\left(\frac{n}{\log^{c+1} n}\right)$	$\Theta\left(\frac{1}{\log^c n}\right)$	$O(n \log n)$	$\Theta\left(\frac{n^2}{\log^c n}\right)$
$\Theta\left(\frac{n}{\log^c n}\right)$	$\Theta(n^\epsilon)$	$\Theta(n)$	$o\left(\frac{n^\epsilon}{\log^{c+1} n}\right)$	$\Theta\left(\frac{1}{\log^c n}\right)$	$O\left(n^{2-\epsilon} \log \log n\right)$	$\Theta\left(\frac{n^2}{\log^c n}\right)$
$\Theta\left(\frac{n}{\log^c n}\right)$	$\Theta(n^\epsilon)$	$\Theta(n^\epsilon)$	$o\left(\frac{n}{\log^{c+1} n}\right)$	$\Theta\left(\frac{1}{\log^c n}\right)$	$O\left(n^{2-\epsilon} \log \log n\right)$	$\Theta\left(\frac{n^2}{\log^c n}\right)$
$\Theta\left(n^{\frac{3}{4}}\right)$	$\Theta(n)$	$\Theta(n)$	$o\left(\frac{n^{\frac{3}{4}}}{\ln n}\right)$	$\Theta\left(\frac{1}{n^{\frac{1}{4}}}\right)$	$O(n \log n)$	$\Theta\left(n^{\frac{7}{4}}\right)$
$\Theta\left(n^{\frac{3}{4}}\right)$	$\Theta(n^\epsilon)$	$\Theta(n)$	$\boxed{o\left(\frac{n^{\epsilon-\frac{1}{4}}}{\ln n}\right)}$	$\Theta\left(\frac{1}{n^{\frac{1}{4}}}\right)$	$O\left(n^{2-\epsilon} \log \log n\right)$	$\Theta\left(n^{\frac{7}{4}}\right)$
$\Theta\left(n^{\frac{3}{4}}\right)$	$\Theta(n^\epsilon)$	$\Theta(n^\epsilon)$	$o\left(\frac{n^{\frac{3}{4}}}{\ln n}\right)$	$\Theta\left(\frac{1}{n^{\frac{1}{4}}}\right)$	$O\left(n^{2-\epsilon} \log \log n\right)$	$\Theta\left(n^{\frac{7}{4}}\right)$

common subsequence detected by the algorithm is indeed an evidence of similarity. For an alphabet of size at least n^ϵ, our algorithm complexity is always $O(n^{2-\epsilon} \log \log n)$, but can be much better depending on the actual alphabet size and the periodicity parameter of the input strings. For example, if the alphabet size is $\Theta(n)$ and the strings are aperiodic, the time complexity of our algorithm is $O(n \log \log n)$. Our contribution to the computation of large common subsequences is, therefore, a strictly sub-quadratic time algorithm (i.e., of complexity $O(n^{2-\epsilon})$ for some constant ϵ) which can find common subsequences of linear (and near linear) size that cannot be detected efficiently by the existing tools. A comparison of our method to the existing tools is discussed in detail in the next paragraph.

The approximation ratio of our algorithm depends on the size of the LCS. It is better as the LCS is longer. Table 1 demonstrates the worst case performance of our algorithm compared to sparse LCS techniques (specifically, Hirshberg's Algorithm [13]) for various LCS size, alphabet size and periodicity parameters. The complexity guarantees presented in the table are a result of combining the theorems proved in this paper (Theorem 1 and Corollary 3 combined with Theorems 2 and 3 and Theorem 5). We stress that these are worst case performances also in the sense that they demonstrate the worst case parameters for given LCS size,the alphabet size and the period length, but the true parameters for a given pair of strings can be much better. The complexity of our method is superior compared to sparse LCS techniques when LCS of near-linear size is concerned, as the first 6 lines of the table indicate. Moreover, even for strictly sub-quadratic size LCS, our method gives a faster approximation algorithm if the alphabet is large enough. As lines 7 and 9 of the table indicate, for LCS of size $\Theta\left(n^{3/4}\right)$ we get a faster algorithm for every $\epsilon > 1/2$. Line 8 of the table represents a case where our technique should not be used due to the requirement on the edit distance. In such a case, sparse LCS techniques should be preferred.

Our method works well for strings A and B where the ED is $o\left(LCS(A,B) \cdot \frac{|\Sigma|}{t \ln t}\right)$, where $|\Sigma|$ is the alphabet size and t depends on the periodicity of the input strings (can be of size n in aperiodic strings). In the worst case, the parameter t is the size of the smaller period in the input strings. The effect of these parameters is also demonstrated in Table 1. Note that, if the edit distance is $\Theta(n^\epsilon)$, the exact LCS can be found in time $\max\{n^{2\epsilon}, n\}$, by finding the edit positions and taking the complement positions. However, for edit distance that is $\Omega(n/\log^c n)$, for some $c > 1$, our algorithm is strictly sub-polynomial, while computing the ED yields a near-quadratic time algorithm. Moreover, even for edit distance that is $\Theta(n^\epsilon)$, our algorithm complexity is always superior when $\epsilon > 2/3$ and can be superior also for smaller ϵ, depending on the parameters of the strings.

1.5. Techniques

We exploit low distortion embedding of strings over a relatively large alphabet into *locally non-repetitive strings*. Intuitively, a locally non-repetitive string is a string where there exists a $w > 0$ for which in every w-size window in the string all the symbols are distinct (see formal definition in Section 2). Local non-repetitiveness has been used for approximating ED [8] and for embedding ED [10]. In [8] and [10], efficient algorithms for input strings that are non-repetitive or locally-non-repetitive with good parameters are designed. Here, we show that any string over a relatively large alphabet can be embedded into a locally non-repetitive string. We prove that this embedding has an additive negligible (contraction) distortion, if $ED = o\left(LCS(A,B) \cdot \frac{|\Sigma|}{t \ln t}\right)$. We then show that local non-repetitiveness can be used to significantly speed-up LCS approximation. The speed-up in the efficiency of our algorithm depends on the local non-repetitiveness parameters of the given strings. We show that local non-repetitiveness can be efficiently sketched so that the best parameters for any two strings can be found by looking at a poly-logarithmic size sketch.

The contribution of this paper is threefold:

1. Suggesting a new method to deal with the computation of large common subsequences, that is capable of handling in sub-quadratic time hard inputs that were not handled efficiently by all existing tools.
2. Understanding better the complexity of the problem, by revealing new aspects and assumptions that make the problem computationally easier.
3. Introducing the use of embedding techniques to the computation of common subsequences. While these techniques were used for speeding-up the computation of ED in [5,9], we are not aware of any use of such methods in the computation of LCS. We stress that, the use of [5,9] is mainly theoretical because their approximation ratio is super poly-logarithmic. Our method is more suited for practical purpose because we give much better approximation ratios.

The paper is organized as follows. Section 2 presents basic definitions and properties. Section 3 presents the embedding of strings over a relatively large alphabet into locally-non-repetitive strings, namely, (1,n/c)-non-repetitive strings, for some c. In Section 4 we present approximation algorithms for this special case of (1,n/c)-non-repetitive strings, where c is a parameter. Finally, in Section 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.

2. Preliminaries

In this section, we give the basic definitions and properties used in this paper.
Problem definition: Let A and B be two strings of length n over an alphabet Σ. 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 n-long strings, then

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

Definition 1 (*LCS preserving embedding*). Let $\mathbf{X}$ and $\mathbf{Y}$ be two classes of strings of length n. A *LCS preserving embedding* of $\mathbf{X}$ into $\mathbf{Y}$ with *distortion* ρ, is an injective mapping $f : \mathbf{X} \mapsto \mathbf{Y}$, such that for every pair $A, B \in \mathbf{X}$, $\rho \cdot LCS(A,B) \leq LCS(f(A), f(B)) \leq LCS(A,B)$, where $\rho \leq 1$.

Note that we require the embedding to be non-expanding. It is only allowed to have a bounded contraction factor.

2.1. Periodicity and non-repetitiveness

Periodicity and non-repetitiveness are two basic properties of a given string that are closely related, as we formally state below.

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

Definition 3 (*A t-substring*). Let S be a string of length n. The *t-substring of S* starting at position i, $i \leq n - t + 1$, is the string $S[i]S[i+1]\ldots S[i+t-1]$.

Definition 4 (*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. If $t = 1$ then S is simply called *locally-non-repetitive*. S 中的每 w 個連續 t 子串是不同的

In the next definition of non-repetitiveness it is required that t-substrings in the range are not only distinct, but also different enough with respect to an additional parameter d.

Definition 5 (*Locally strong non-repetitiveness*). 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 is defined as to continue the period from the point it is cut in the string S.

則迴繞被定義為從它在字符串 S 中被切割的點開始繼續該週期。

Property 2. Let S be a (t, w)-non-repetitive string, then: S 中的每 w 個連續 t 子串是不同的

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 length 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 $n/2 \le w \le n$.

Lemma 1. *Let S be an n-long string over an alphabet Σ with period length p, then S is a $(p, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string. If S is aperiodic then S is an $(n, |\Sigma|/2, |\Sigma|/2)$-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. First, note[5] that $p \ge |\Sigma|$. Let s_i be any p-substring in S, and let s_{i+j} be any p-substring in S such that $0 < j < |\Sigma|/2$. It is sufficient to show that $\mathcal{H}(s_i, s_{i+j}) \ge |\Sigma|/2$. Let $\Sigma = \{\sigma_1, \ldots, \sigma_{|\Sigma|}\}$ and for $r = 1, \ldots, |\Sigma|$, let x_r be the first appearance of σ_r in s_i. Consider the sorted list of the x_r's, L_x. We claim that $\mathcal{H}(s_i, s_{i+j}) \ge |\Sigma| - j$, because each of the last $|\Sigma| - j$ x_r's in L_x adds at least one mismatch to $\mathcal{H}(s_i, s_{i+j})$. The lemma follows. □

Note that, Lemma 1 gives a guarantee for worst case parameters of locally strong non-repetitiveness. For a given pair of strings, the best parameters, i.e., the larger parameters w and d for which the t-substrings are strongly non-repetitive, can be much better. For example, consider an n-long string over an alphabet of size n^ϵ, with period size $p > n^\epsilon$. The lemma only assures that the string is $(p, n^\epsilon/2, n^\epsilon/2)$-non-repetitive, however, the string can actually be (p, p, d)-non-repetitive, for $d \ge n^\epsilon/2$.

3. Embedding strings over a relatively large alphabet into local non-repetitive strings

By Lemma 1, a relatively large alphabet assures the existence of a large enough parameter w and a parameter t such that the t-substrings are locally strong non-repetitive, for a large enough parameter d. We will exploit this to define an embedding into (1,n/c)-non-repetitive strings, for which the solutions of Section 4 are applicable. This embedding has only an additive negligible distortion, if the ED is asymptotically negligible compared to the LCS size and the ratio between the alphabet size and the periodicity parameter of the string. Thus, it facilitates the approximation of large LCS in general strings over a relatively large 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 an embedding that may have an unbounded distortion is described first. We then define and analyze the embedding used in this paper.

3.1. A naive embedding

The idea of the embedding is to exploit Property 3, from which we know that every n long string S over an alphabet Σ is a (t, w)-non-repetitive string for some $|\Sigma| \le t \le n, |\Sigma| \le w \le n$. Therefore, we can define a new symbol for each t-substring (overall, a linear number of new symbols) and replace the original string by the sequence of new symbols according to the sequence of the t-substrings in S. This embedding yields a (1,n/c)-non-repetitive string where $c \le \frac{2n}{|\Sigma|}$.[6]

We now analyze the distortion of this embedding. Given the original n-long strings A and B, denote by A', B' the strings after employing the embedding. Clearly, $LCS(A', B') \le LCS(A, B)$ because positions with different symbols remain different. Also, each of the $\underline{n - LCS(A, B)}$ symbols that do not participate in $LCS(A, B)$ affects only t substrings, thus,

$$LCS(A', B') \ge n - t(\underline{n - LCS(A, B)}) = LCS(A, B) - (\underline{t - 1})(\underline{n - LCS(A, B)}).$$

By Property 1 we get

Property 1.

$$LCS(A', B') \ge LCS(A, B) - \boxed{\frac{t-1}{2} \cdot ED(A, B)}.$$

Thus, this embedding has an additive distortion affected both by t and $ED(A, B)$, which can both be $\Omega(n)$.

[5] In this paper, we consider the alphabet set to include only symbols that actually appear in the string.
[6] Note that since $|\Sigma|$ is relatively large the algorithms of Section 4 are efficient.

3.2. The low distortion embedding

We, therefore, use the following variation of the naive embedding. We use a randomized algorithm to find a final deterministic embedding function f. Fix a random binary vector v of length $t-1$, where each coordinate is 1 with probability $\frac{2d\ln t}{|\Sigma|}$ for an arbitrarily chosen constant $d>2$, and 0 otherwise. Note that v is well defined for a relatively large alphabet (i.e., of size at least n^{ϵ}), since for $|\Sigma|\geq n^{\epsilon}$ and $t\leq n$, $\frac{2d\ln t}{|\Sigma|}=o(1)$.

Given an n-long string S over an alphabet Σ define $f(S)$ as follows. Each location i contains a symbol $\sigma(i)$ which is the new symbol assigned to the string $S_i, S_{i+i_1}, \ldots, S_{i+i_k}$, where $i_1,\ldots,i_k$ are the locations in the $(t-1)$-substring starting at position $i+1$ in S for which the content of the corresponding coordinates $i_1,\ldots,i_k$ in v are 1. Note, that there is no assumption whatsoever on any property of the original string S. Lemma 2 and Corollary 1 give the local non-repetitiveness guarantee on the string produced by the embedding f. Lemma 3 bounds the distortion of the embedding f.

Lemma 2. *Let S be a n-long string over an alphabet Σ 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 $Prob[\mathcal{H}(f(s_i),f(s_j))=0]=\left(1-\frac{2d\ln t}{|\Sigma|}\right)^{|\Sigma|/2}$, if none 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. □

Note that, we can check if the resulting string $f(S)$ is indeed a locally non-repetitive string in linear time. If it is not, the choice of v can be repeated until the result is a locally non-repetitive string. The expected number of vectors v that should be chosen is less than 2. Corollary 1 follows.

Corollary 1. *Let S be a string over an alphabet Σ then, there exists a deterministic embedding f such that $f(S)$ is a $(1,|\Sigma|/2)$-non-repetitive string.*

Lemma 3. *Let A, B be n-long strings over an alphabet Σ, then*

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

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 embedding f the first symbol of the ith t-substring is always taken and the rest $i+1,\ldots,i+t-1$ locations of the ith t-substring are taken with probability $\frac{2d\ln t}{|\Sigma|}$ for a constant $d>2$, we have:

總是取第 i 個 t 子串的第一個符號，其餘的依照對應的機率取

$$\begin{aligned}LCS(f(A),f(B)) &\geq n-\left(1+\overset{(t-1)*\text{機率}}{\underline{\frac{2(t-1)d\ln t}{|\Sigma|}}}\right)(n-LCS(A,B))\\ &=LCS(A,B)-\frac{2(t-1)d\ln t}{|\Sigma|}\cdot(n-LCS(A,B))\\ &\geq LCS(A,B)-\frac{d(t-1)\ln t}{|\Sigma|}\cdot ED(A,B),\end{aligned}$$

Property 1.

where the last inequality is due to Property 1. □

Let $\mathbf{RL}(n,\Sigma)$ be the class of n-long strings over an alphabet Σ, $|\Sigma|\geq n^{\epsilon}$, for some $\epsilon>0$. Let $\mathbf{LNR}(n)$ be the class of locally-non-repetitive n-long strings. 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 $O(n/c)$. 分割成c塊
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.

Theorem 1. *For every $A, B \in \mathbf{RL}(n, \Sigma)$, there exists a parameter t, $|\Sigma| \le t \le n$, and an embedding $f : \mathbf{RL}(n, \Sigma) \mapsto \mathbf{LNR}(n)$ such that $f(A), f(B) \in \mathbf{LNR}(n)$ and if $ED(A, B) = o\left(LCS(A, B) \cdot \frac{|\Sigma|}{t \ln t}\right)$ then f has distortion $1 - o(1)$.*

4. 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-long permutation strings is actually finding the Longest Increasing Subsequence (LIS) of a string over the alphabet $\{1, \ldots, n\}$, which can be done fast.

4.1. $\Theta(1/c)$-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. Note that, the LIS algorithm commonly gets only a single input, however, we refer to the algorithm as having two input strings A' and B', because the input string B' is used to impose an ordering of the alphabet according to which LIS(A') is computed. A detailed description of the algorithm is given in Fig. 1. Lemma 5 and Corollary 2 assure the approximation ratio of this algorithm. Lemma 4 gives its complexity guarantee. Theorem 2 follows.
因為輸入字符串 B' 用於強加字母表的順序，根據該順序計算 LIS(A')

Lemma 4. *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. Therefore, the total time for steps 2–4 is $O(cn \log\log(n/c))$. Step 5 takes another c^2 steps. The lemma then follows. □

In the proof of Lemma 5 (and Lemma 7 below), we use the term *a match* defined as follows. Let A and B two strings and let $LCS(A, B)$ be the longest common subsequence of A and B. Assume that $LCS(A, B) > 0$ and let i_k (respectively, j_k), $0 < k \le LCS(A, B)$, be the locations in which $LCS(A, B)$ appears in A (respectively, in B). Then, the pair $\langle i_k, j_k \rangle$, $0 < k \le LCS(A, B)$, is called *a match*. ik 是 LCS(A, B) 在 A 中出現的位置

Lemma 5. *Let A and B be two strings of length n, then there exists a pair of blocks A_i, B_j such that $l_{i,j} \ge \Theta(1/c) \cdot LCS(A, B)$.*

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

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

1. Consider the block A_i. If $\boxed{\alpha_i^{(k)} < \frac{Opt_k}{2c}}$ 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,

$$Opt_{k+1} = LCS(A^{(k)}, B^{(k+1)}) \ge \left(1 - \frac{1}{2c}\right) Opt_k.$$

如果 α(k) < Optk，則拋出塊 Ai，讓 i 成為 i + 1。
從 B(k) 中的每個塊中拋出 Optk 與 Ai 的匹配以形成 B(k+1)。

```
ALGORITHM APPROX2LCS
Input: Two strings $A, B$ of length $n$, a parameter $c$
1  divide $A, B$ into $c$ blocks of size $O(n/c)$. 分割成c塊
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
           $W : E \to \mathcal{N}$, where:
           $V1 = \{i \mid A_i \text{ is a block in } A\}$
           $V2 = \{j \mid B_j \text{ is a block in } B\}$
           $E = \{(i,j) \mid i \in V1 \text{ and } j \in V2\}$
           $W(i,j) = \ell_{i,j}$
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.

用 β(k+1) 表示塊 Bj 內 Optk+1 的匹配

Denote the matches of Opt_{k+1} within block B_j by $\beta_j^{(k+1)}$. Let k be $k+1$.

2. Consider the block B_j. If $\boxed{\beta_j^{(k)} < \frac{Opt_k}{2c}}$ 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 has with B_j to form $A^{(k+1)}$. Since only $\beta_j^{(k)}$ matches were thrown,

$$Opt_{k+1} = LCS(A^{(k+1)}, B^{(k)}) \geq \left(1 - \frac{1}{2c}\right) Opt_k.$$

Denote the matches of Opt_{k+1} within block A_i by $\alpha_i^{(k+1)}$. Let k be $k+1$.

Repeatedly, loop on the steps 1–2 above, where each time a block is thrown k is increased by one, until the first pair of blocks A_i, B_j such that each has at least $\frac{Opt_k}{2c}$ matches is encountered or there are no more blocks to throw. In the first case we get:

$$LCS(A_i, B_j) \geq \frac{Opt_k}{2c} \geq \frac{Opt}{2c} \cdot \left(1 - \frac{1}{2c}\right)^k \geq \frac{Opt}{2e \cdot c}$$

In the second case, since the total number of blocks is $2c$, the process stops with $Opt_{2c} \geq Opt \cdot \left(1 - \frac{1}{2c}\right)^{2c} \geq Opt \cdot e^{-1}$, which cannot happen. Therefore, we must have stopped in the first case. The lemma then follows. □

Corollary 2. *The approximation ratio of algorithm Approx1LCS is $\underline{\Theta(1/c)}$.*

Theorem 2. *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.*

4.2. $\Theta(k/c)$-approximation algorithm

The $\Theta(1/c)$ approximation ratio of algorithm *ApproxLCS*1 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 *Approx*1*LCS*, 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 pairs 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. We refer to this procedure by *MaximumWeightLegalSequence*. A detailed description of the algorithm is given in Fig. 2. Lemma 7 assures the approximation ratio of this algorithm. Lemma 6 gives its complexity guarantee. Theorem 3 follows.

Lemma 6. *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 the proof of Lemma 4. The graph construction in Line 5 can be done in time linear in its size. Since the graph has $2c$ vertices and c^2 edges, line 5 can be computed in $O(c^2)$ steps. The computation of the maximum weighted legal sequence 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 (*MaximumWeightLegalSequence*) based on the following dynamic programming formulation:

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

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

Lemma 7. *Algorithm Approx2LCS approximates* $LCS(A,B) \ge kn/c$ *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, $\forall i$, $\alpha_i \le 1/c$. Also, $Opt = \sum \alpha_i \ge k/c$. 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, $\sum k_i \le 2c$. Note that $L_{alg} \ge \sum \alpha_i/k_i$, because the algorithm chooses the maximum weight legal sequence, therefore, for each block A_i at least the average contribution α_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 $k_i > 4c/k$, and the rest of the blocks.

Claim 1. $\sum_{A_i \in X} \alpha_i \le \frac{Opt}{2}$.

Since $|X| \le \sum k_i / \frac{4c}{k} \le \frac{k}{2}$, and therefore, $\sum_{A_i \in X} \alpha_i \le \frac{k}{2} \cdot \frac{1}{c} \le \frac{Opt}{2}$.

Claim 2. $L_{alg} \ge \frac{k}{8c} \cdot Opt$.

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

The lemma then follows. $\square$

Theorem 3. *Let A, B be two (1,n/c)-non-repetitive strings then* $LCS(A,B) \ge kn/c$ *can be approximated to a factor of* $\Theta(k/c)$ *in* $O(c \cdot n\log\log(n/c) + c^2)$ *steps.*

5. Sketching local non-repetitiveness

The performance of our method for approximating LCS relies on the extent of local non-repetitiveness parameters of the given strings. It is natural to ask, how quickly these parameters can be found. The almost linear time algorithms presented in this section do not require any pre-computed information on the strings (e.g. the periodicity), and approximate the best parameters to a factor of 2. For our method of approximating the LCS of two given strings this is sufficient. However, the strength of these algorithms lies in the fact that they are sketching algorithms, i.e., they are only used once for a given string and produce a small (poly-logarithmic) size information from which the best parameters can be deduced. This use is valuable for databases applications, in which a query string is typically compared with many stored strings to find a similar (or the most similar) stored string. Short one-time pre-computed sketches of the stored strings save many repeated linear time scans, and thus speed-up computations.

In this section, we show that the best parameters t and w for a given pair of strings can be found by looking at $O(\log^2 n)$ size independently pre-computed *local non-repetitiveness sketches* (LNR-sketch) of the strings. The LNR-sketch gives the exact parameter w for which the best t parameter is approximated to a factor of 2. The implementation of the embedding f from Section 3 using the construction of *strong local non-repetitiveness sketches* (SLNR-sketch) is then described. We also show that our LNR-sketch size matches a lower bound we give on the LNR-sketch size. Finally, 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 what follows, 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.

5.1.1. 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 an $O(n \log^2 t)$ time deterministic algorithm or by an $O(n)$ time randomized algorithm. The deterministic algorithm uses a renaming process as in the string matching algorithm of Karp–Miller–Rosenberg [15]. 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 [16] 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".

5.1.2. 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 long 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 strong local non-repetitiveness

The embedding from strings over a relatively large alphabet into (1,n/c)-non-repetitive strings described in Section 3 requires local non-repetitiveness under the random choices of the vectors v. The algorithms described in Section 5.1 cannot detect such local non-repetitiveness. Nevertheless, we show that the ideas of the sketching algorithms described in Section 5.1 can be used also for this case. We call it *strong local non-repetitiveness sketch* (SLNR-sketch).[7] By Corollary 1, a constant number of vectors v are enough so that two given strings can be compared using the same vector v. Therefore, below we ignore the fact that the algorithm is repeated for each choice of v and keep each of the resulting sketches.[8]

To this end, the substrings as defined by the binary vector v (defined in Section 3), are considered. Observe that both the deterministic and randomized sketching algorithms described in Section 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 Σ 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 sizes of joint alphabets and different t parameter) we must define a v vector for each possible pair. To cover all possible values of Σ, for each t a power of two, $O(\log^2 n)$ vectors v (for each Σ 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 Section 5.1. This would take $O(n \log t)$ time because here t is a power of 2. Since $O(\log^2 n)$ sketches of size $O(\log n)$ are used, Theorem 5 follows.

Theorem 5 (The embedding implementation). *There exist (almost) linear algorithms that for every n long strings A and B, give SLNR-sketch of size $O(\log^3 n)$ which enables finding the maximum w and approximating to a factor of 2 the minimum t for which $f(A)$ and $f(B)$ are both $(1, w)$-non-repetitive.*

Denote by $\gamma(n)$, the time for computing the embedding f from Section 3. Theorem 5 shows that $\gamma(n) = \tilde{O}(n)$. Corollary 3, which is the algorithmic application of Theorem 1, follows.

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

5.3. Lower bound on LNR-sketch size

Note that the w parameter as a function of t is a non-decreasing monotone function that take values in 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.

[7] Should not be confused with the local strong non-repetitiveness.

[8] A database application requires another logarithmic factor in the size of the database to assure that every pair of strings can be compared using the same vector v.

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

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 $(n/\log n)^{\log n}$. $\square$

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*. Lemma 9 shows that $k = \Omega(n)$. Theorem 7 follows.

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

Proof. We show that there exists an instance of the repeating t-substring problem for which $k = \Omega(n)$. We define the following instance of the repeating t-substring problem: $S1$ is $\pi_1 \in S_n$ and $S2$ is $\pi_2 \in S_n$. Consider the boolean matrix for all possible pairs $\langle \pi_1, \pi_2 \rangle \in S_n \times S_n$, representing whether or not a t-substring in π_1 appears in π_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 entry within each part must depend solely on π_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 $\Omega(n)$). $\square$

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

6. Conclusions

We show how embedding strings over a relatively large alphabet into locally non-repetitive strings can be exploited for approximating LCS in strictly sub-quadratic time. An important contribution of the paper is also conceptual in suggesting a different point of view that make the problem algorithmically easier. Our technique works well provided that the dissimilarity in terms of the edit distance of the given strings is not too large. It is still an open question whether LCS can be well-approximated in strings over a relatively large alphabet with large dissimilarity.

References

[1] A.V. Aho, D.S. Hischberg, J.D. Ulman, Bounds on the complexity on the longest common subsequence problem, Journal of the ACM 23 (1) (1976) 1–12.
[2] A. Amir, Y. Aumann, O. Kapah, A. Levy, E. Porat, Approximate string matching with address bit errors, in: P. Ferragina, G.M. Landau (Eds.), CPM, LNCS, vol. 5029, Springer, 2008, pp. 118–129.
[3] A. Andoni, R. Krauthgamer, The computational hardness of estimating edit distance, in: in: Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (2007) 724–734.
[4] A. Andoni, R. Krauthgamer, K. Onak, Polylogarithmic approximation for edit distance and the asymmetric query complexity, FOCS, 2010, pp. 377–386.
[5] A. Andoni, K. Onak, Approximating edit distance in near-linear time, in: in: Proceeding of the 41st ACM Symposium on Theory of Computing (2009) 199–204.
[6] A. Apostolico, C. Guerra, The longest common subsequence problem revisited, Algorithmica 2 (1987) 315–336.
[7] B.S. Baker, R. Giancarlo, Longest common subsequence from fragments via sparse dynamic programming, in: G. Bilardi, G.F. Italiano, A. Pietracaprina, G. Pucci (Eds.), Proceedings of the 6th European Symposium on Algorithm, LNCS, vol. 1461, Springer-Verlag, August 1998, pp. 79–90.
[8] Z. Bar-Yossef, T.S. Jayram, R. Krauthgamer, R. Kumar, Approximating edit distance efficiently, in: in: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science, 2004, pp. 550–559.
[9] T. Batu, F. Ergün, C. Sahinalp, Oblivious string embeddings and edit distance approximation, in: in: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms, 2006, pp. 792–801.
[10] M. Charikar, R. Krauthgamer, Embedding the Ulam metric into ℓ_1, Theory of Computing 2 (2006) 207–224.
[11] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A sub-quadratic sequence alignment algorithm for unrestricted cost matrices, SIAM Journal on Computing 32 (5) (2003) 1654–1673.
[12] D. Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press, 1997.
[13] D.S. Hirshberg, Algorithms for the longest common subsequence problem, Journal of the ACM 24 (4) (1977) 664–675.
[14] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Communications of the ACM 20 (1977) 350–353.
[15] R. Karp, R. Miller, A. Rosenberg, Rapid identification of repeated patterns in strings, arrays and trees, in: Proceeding of the 4th Annual ACM Symposium on the Theory of Computing vol. 4 (1972) 125–136.
[16] R.M. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development 31 (2) (1987) 249–260.

[17] G.M. Landau, B. Scheiber, M. Ziv-Ukelson, Sparse LCS common substring alignment, Information Processing Letters 88 (6) (2003) 259–270.
[18] G.M. Landau, U. Vishkin, Fast string matching with *k* differences, Journal of Computer and System Sciences 37 (1) (1988) 63–78.
[19] G.M. Landau, M. Ziv-Ukelson, On the common substring alignment problem, Journal of Algorithms 41 (2) (2001) 338–359.
[20] W.J. Masek, M.S. Paterson, A faster algorithm for computing string edit distances, Journal of Computer and System Sciences 20 (1980) 18–31.
[21] R. Ostrovsky, Y. Rabani, Low distortion embeddings for edit distance, in: in: Proceedings of the 37th Annual ACM Symposium on Theory of Computing (2005) 218–224.
[22] D. Sankoff, Matching sequences under deletion/insertion constraints, in: Proceedings of the National Academy of Sciences United States of America 69 (1972) 4–6.
[23] R.A. Wagner, M.J. Fischer, The string to string correction problem, Journal of the ACM 21 (1) (1974) 168–173.