LCS Approximation via Embedding into Local Non-Repetitive Strings

Gad M. Landau[1,5,⋆], Avivit Levy[2,3], and Ilan Newman[1]

[1] Department of Computer Science, University of Haifa, Haifa 31905, Israel. E-mail: {landau,ilan}@cs.haifa.ac.il
[2] Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel. E-mail: avivitlevy@shenkar.ac.il
[3] CRI, University of Haifa, Mount Carmel, Haifa 31905, Israel.
[4] Department of Computer and Information Science, Polytechnic Institute of NYU, Six MetroTech Center, Brooklyn, NY 11201-3840

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 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 relatively large alphabet can be embedded into a local 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.

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 [21, 22] and this complexity matches the lower bound in comparison model [1]. Many other algorithms have been suggested over the years [12, 13, 19, 4, 5, 16, 18, 9] (see also [11]). 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

⋆ partially supported by the Israel Science Foundation grant 35/05 and the Israel-Korea Scientific Research Cooperation.

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.

Approximating LCS in Strings Over Small Alphabet. 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 Σ is the alphabet, by just picking the letter that has the highest frequency. If the alphabet size is $o(n^\epsilon)$ for every constant $\epsilon > 0$, this trivial algorithm achieves sub-polynomial approximation ratio, which is roughly the best known approximation ratio for the closely related edit distance [20][5]. 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.

Sparse vs. Large LCS. Relatively large alphabet may reduce the number of matching symbols between the two given strings. In such cases the sparse LCS techniques of Hunt-Szymanski can be used to give efficient exact solutions that depend on the matchings set size [13, 5]. However, the input strings may have a quadratic size of matching pairs of symbols even if the alphabet is relatively large. In these cases, these sparse LCS algorithms take quadratic time. Other methods for finding sparse LCS quickly 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 by algorithms that take time $O(n^{1+\alpha})$ [12, 16, 18]. However, these algorithms take quadratic time for finding LCS of linear size. Thus, the focus of this paper is on efficiently approximating large LCS, typically, LCS of near linear size, in strings over relatively large 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 [17], ED can be computed in time $\max\{k^2, n\}$, where k is the bound on ED and n the length

[5] [20] 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.

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 [7], achieves approximation factor $n^{1/3+o(1)}$, where n is the length of the strings.

Results. In this paper it is shown that large LCS can be efficiently approximated in strings with relatively large alphabet if the ED is not too large. In particular, LCS of linear size can be approximated to a constant factor, if the edit distance is $o(\frac{n|\Sigma|}{t \ln t})$, where $|\Sigma|$ is the alphabet size and t is the period size ($t = n$ 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. A large LCS detected by the algorithm is an evidence of similarity. For alphabet of size at least n^ϵ, our algorithm complexity is always $O(n^{2-\epsilon} \log\log n)$ but can be much better (for some parameters it is $O(n \log\log n)$). 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.

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 for LCS of different sizes. The complexity guarantees presented in the table are a result of combining the theorems proved in this paper (Theorem 1 and Corollary 2 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, alphabet size and 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 $\Theta(n^{3/4})$ 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 $o(LCS(A,B) \cdot \frac{|\Sigma|}{t \ln t})$, where Σ 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. 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.

Table 1. Worst Case Performance of Our Algorithm: Examples

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

Techniques. We exploit low distortion embedding of strings over relatively large alphabet into local non-repetitive strings. Local non-repetitiveness has been used for approximating ED [6] and for embedding ED [8]. In [6] and [8], 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 relatively large alphabet can be embedded into a locally non-repetitive string. We prove that this embedding has an additive negligible (contraction) distortion, if $ED = o(LCS(A,B)\cdot\frac{|\Sigma|}{t\ln t})$. 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 sketch.

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

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 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 n-long strings. 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.

Periodicity and Non-Repetitiveness. Periodicity and non-repetitiveness are two basic properties of a given string that, as we formally state in the sequel, are closely related.

Definition 2. *Let S be a string of length n. S is called* periodic *if $S = P^i P'$, 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.

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.

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 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 $n/2 \leq w \leq n$.

Lemma 1. *Let S be a n-long string over 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 a $(n, |\Sigma|/2, |\Sigma|/2)$-non-repetitive string.*

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 a n-long string over alphabet n^ϵ, with period $p > n^\epsilon$. The lemma only assures that it is $(p, n^\epsilon/2, n^\epsilon/2)$-non-repetitive, however, it can actually be (p, p, d)-non-repetitive, for $d \geq n^\epsilon/2$.

3 Embedding Strings Over Relatively Large Alphabet into Local Non-Repetitive Strings

By Lemma 1, 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 Sect. 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 enables approximating large LCS in general strings over 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 unbearable distortion is described first. After analyzing its weaknesses it is shown how these can be overcome by defining our embedding. Finally, we discuss the algorithmic applications of this embedding.

A Naive Embedding. The idea is to exploit Property 3, namely, that every n long string S over alphabet Σ 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 embedding yields a (1,n/c)-non-repetitive string where $c \leq \frac{2n}{|\Sigma|}$, and since $|\Sigma|$ is relatively large the algorithms of Sect. 4 are efficient.

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, $LCS(A', B') \leq LCS(A, B)$ 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 embedding has an additive distortion affected both by t and $ED(A, B)$, which can both be $\Omega(n)$.

The Embedding 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 relatively large alphabet, since for $|\Sigma| \geq n^{\epsilon}$ and $t \leq n$, $\frac{2d \ln t}{|\Sigma|} = o(1)$. Given an n-long string S over alphabet Σ 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. 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 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 $Prob[\mathcal{H}(f(s_i), f(s_j)) = 0] = (1 - \frac{2d \ln t}{|\Sigma|})^{|\Sigma|/2}$, 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.

The resulting string $f(S)$ can be checked if it 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 alphabet Σ then, there exists a deterministic embedding f such that $f(S)$ is $(1, |\Sigma|/2)$-non-repetitive string.*

Lemma 3. *Let A, B be n-long strings over 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 i-th t-substring is always taken and the rest $i+1, \ldots, i+t-1$ locations of the i-th t-substring are taken with probability $\frac{2d\ln t}{|\Sigma|}$ for a constant $d > 2$, we have:

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

where the last inequality is due to Property 1.

Let $\mathbf{RL}(n, \Sigma)$ be the class of n-long strings over 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.

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

Implementation and Algorithmic Application. The discussion of efficient algorithms for computing the embedding f is postponed to Sect. 5, since the algorithms we present are also sketching algorithms, and therefore, require the relevant study from this point of view. Denote by $\gamma(n)$, the time for computing f. In Sect. 5 it is shown that $\gamma(n) = \tilde{O}(n)$. Corollary 2 is the algorithmic application of the embedding f.

Corollary 2. *Let A,B be two n-long strings over alphabet Σ. Then, there exists a parameter t, $|\Sigma| \leq t \leq n$, such that if $ED(A,B) = o(LCS(A,B) \cdot \frac{|\Sigma|}{t\ln t})$, 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.*

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. A detailed description of the algorithm is given in Fig. 1. Lemma 5 and Corollary 3 assure the approximation ratio of this algorithm. Lemma 4 gives its complexity guarantee. Theorem 2 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 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.

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 $l_{i,j} \geq \Theta(1/c) \cdot LCS(A, B)$.*

Corollary 3. *The approximation ratio of algorithm Approx*$1LCS$ *is* $\Theta(1/c)$.

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 *ApproxLCS*1, 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.

```
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:
           $V1 = \{i \mid A_i \ is\ a\ block\ in\ A\}$
           $V2 = \{j \mid B_j \ is\ a\ block\ in\ B\}$
           $E = \{(i,j) \mid i \in V1 \ and\ j \in V2\}$
           $W(i,j) = \ell_{i,j}$
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 6. *Algorithm Approx*$2LCS$ *runs in* $O(cn \log\log(n/c) + c^2)$ *steps.*

Lemma 7. *Algorithm Approx*$2LCS$ *approximates* $LCS(A, B) \geq kn/c$ *to a factor of* $\Theta(k/c)$.

Theorem 3. *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.*

5 Sketching Local Non-Repetitiveness

Since the performance of our method for approximating LCS rely on the extent of local non-repetitiveness parameters of the given strings, it is natural to ask how quickly can these parameters 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 data-bases 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 Sect. 3 using the construction of *strong local non-repetitiveness sketches* (SLNR-sketch) is then described. We also show that our LNR-sketch size matches the lower bound. 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 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 [14]. 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 [15] 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 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 relatively large alphabet into (1,n/c)-non-repetitive strings described in Sect. 3 requires local non-repetitiveness under the choices of the randomized vectors v, which is not detected by the algorithms described in Sect. 5.1. Nevertheless, we show that the ideas of the sketching algorithms described in Sect. 5.1 can be used also for this case. We call it *strong local non-repetitiveness sketch* (SLNR-sketch)[6]. 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, in the sequel we ignore the fact that the algorithm is repeated for each choice of v and keep each of the resulting sketches[7].

To this end, the substrings as defined by the binary vector v (defined in Sect. 3), 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 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 size of joint alphabet 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

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

[7] A data-base 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.

a specific vector v is defined, the sketch for non-repetitiveness can be done as explained in Sect. 5.1. This would take $O(n \log t)$ 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) *Let A, B be n long strings, then, there exist (almost) linear algorithms giving SLNR-sketch of size $O(\log^3 n)$ enabling 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.*

5.3 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 $(\frac{n}{\log n})^{\log n}$.*

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.*

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

6 Conclusions

We show how embedding strings over relatively large alphabet into local 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 wether LCS can be well-approximated in strings over relatively large alphabet with large dissimilarity.

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. B. S. Baker and R. Giancarlo, *Longest common subsequence from fragments via sparse dynamic programming*, Proc. 6th Eur. Symp. Algorithm (ESA) (G. Bilardi, G. F. Italiano, A. Pietracaprina, and G. Pucci, eds.), LNCS 1461, Springer-Verlag, Aug. 1998, pp. 79–90.
6. Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, and R. Kumar, *Approximating edit distance efficiently*, FOCS, 2004, pp. 550–559.
7. T. Batu, F. Ergün, and C. Sahinalp, *Oblivious string embeddings and edit distance approximation*, SODA, 2006, pp. 792–801.
8. M. Charikar and R. Krauthgamer, *Embedding the* ULAM *metric into* ℓ_1, Theory of Computing **2** (2006), 207–224.
9. 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.
10. M. Crochemore and E. Porat, *Computing a longest increasing subsequence of length k in time $o(n \log\log k)$*, Visions of computer science (Swindon, UK) (E. Gelenbe, S. Abramsky, and V. Sassone, eds.), The British Computer Society, 2008, pp. 69–74.
11. D. Gusfield, *Algorithms on strings, trees and sequences*, Cambridge University Press, 1997.
12. D. S. Hirshberg, *Algorithms for the longest common subsequence problem*, JACM **24** (1977), no. 4, 664–675.
13. J. W. Hunt and T. G. Szymanski, *A fast algorithm for computing longest common subsequences*, CACM **20** (1977), 350–353.
14. 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.
15. R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development **31** (1987), no. 2, 249–260.
16. G. M. Landau, B. Scheiber, and M. Ziv-Ukelson, *Sparse* LCS *commom substring alignment*, Information Processing Letters **88** (2003), no. 6, 259–270.
17. G. M. Landau and U. Vishkin, *Fast string matching with k differences*, Journal of Computer and System Sciences **37** (1988), no. 1, 63–78.
18. G.M. Landau and M. Ziv-Ukelson, *On the common substring alignment problem*, Journal of Algorithms **41** (2001), no. 2, 338–359.
19. W. J. Masek and M. S. Paterson, *A faster algorithm for computing string edit distances*, JCSS **20** (1980), 18–31.
20. R. Ostrovsky and Y. Rabani, *Low distortion embeddings for edit distance*, Proceedings of the 37th annual ACM symposium on Theory of computing (STOC), 2005, pp. 218–224.

21. D. Sankoff, *Matching sequences under deletion/insertion constraints*, Pro. Nat. Acad. Sct. USA, vol. 69, January 1972, pp. 4–6.
22. R. A. Wagner and M. J. Fischer, *The string to string correction problem*, JACM **21** (1974), no. 1, 168–173.