Oblivious String Embeddings and Edit Distance Approximations

Tuğkan Batu* Funda Ergun* Cenk Sahinalp*

Abstract

We introduce an oblivious embedding that maps strings of length n under edit distance to strings of length at most n/r under edit distance for any value of parameter r. For any given r, our embedding provides a distortion of $\tilde{O}(r^{1+\mu})$ for some $\mu = o(1)$, which we prove to be (almost) optimal. The embedding can be computed in $\tilde{O}(2^{1/\mu}n)$ time.

We also show how to use the main ideas behind the construction of our embedding to obtain an efficient algorithm for approximating the edit distance between two strings. More specifically, for any $1 > \epsilon \geq 0$, we describe an algorithm to compute the edit distance $\mathcal{D}(S, R)$ between two strings S and R of length n in time $\tilde{O}(n^{1+\epsilon})$, within an approximation factor of $\min\{n^{\frac{1-\epsilon}{3}+o(1)}, (\mathcal{D}(S,R)/n^{\epsilon})^{\frac{1}{2}+o(1)}\}$. For the case of $\epsilon = 0$, we get a $\tilde{O}(n)$-time algorithm that approximates the edit distance within a factor of $\min\{n^{\frac{1}{3}+o(1)}, \mathcal{D}(S,R)^{\frac{1}{2}+o(1)}\}$, improving the recent result of Bar-Yossef et al. [2].

1 Introduction

Let S and R be two strings over an alphabet σ. The *edit distance*, denoted here by $\mathcal{D}(S, R)$, between S and R is defined as the minimum number of insertions, deletions, and substitutions required to transform S into R, or vice versa. This measure of string similarity (along with its variants) is widely used in areas such as computational biology, text processing, and web searching. It is known that the edit distance between two strings can be computed in quadratic time for general alphabets and slightly under quadratic time for a constant-size alphabet [8]. However, until recently, no faster algorithm for computing $\mathcal{D}(S, R)$, even approximately, was known. On the other hand, the task of efficiently computing the edit distance, even if approximately, has gained a lot of attention, especially in the computational biology community, where the data is very large and thus fast algorithms are highly desired.

Another interesting class of string problems concerns (metric) embeddings of strings into strings under the edit distance. An embedding as a mapping of strings into shorter strings provides a mechanism to "summarize" the input string, thus reducing its dimensionality. The shorter string obtained from the original can later be used to make inferences about the original string. For example, if an embedding has low distortion under edit distance, then the embeddings of any two given strings can be compared to infer the similarity of the original strings under edit distance. We can measure the quality of such a mapping under edit distance by the tradeoff it offers between the reduction in the dimensionality and the "precision" of the mapping and desire one that will map a string S to as short a string S' as possible while maintaining a reasonable amount of correspondence between S and S'.

1.1 Our Results

We introduce an oblivious embedding (which indeed is a dimensionality-reduction technique) that maps any string S of length n to a string of length at most n/r for any value of the *reduction parameter* r. The embedding is oblivious in the sense that it is applied to *any* string independently of any other string the embedding may be applied to. For any value of the reduction parameter r, our embedding provides a distortion of $\tilde{O}(r^{1+\mu})$ for $\mu = O(1/\log\log\log n)$ under edit distance. The embedding can be computed in time $\tilde{O}(2^{1/\mu}n)$, which is (almost) optimal. We prove that no oblivious string to string embedding under edit distance can have a distortion better than $\Omega(r)$ thus our distortion is (almost) optimal for any value of r.

Note that an embedding with the above properties provides a straightforward approximation algorithm for the edit distance between two strings: namely, applying the embedding with an appropriate reduction parameter and computing the edit distance between the embeddings exactly. Unfortunately, for any non-trivial approximation factor $d = \tilde{o}(\sqrt{n})$, this straightforward approach requires a running time of $\tilde{\omega}(n)$, because the reduction parameter has to be $O(d)$ and the lengths of the embeddings will be $\Omega(n/d)$. In this paper we show how to use the ideas behind the construction of our embedding to obtain a more efficient algorithm for approximating the edit distance.

In particular, for any $0 \leq \epsilon < 1$, we describe an algorithm to compute the edit distance between two

*School of Computing Science, Simon Fraser University. {batu,funda,cenk}@cs.sfu.ca.

strings S and R in time $\tilde{O}(n^{1+\epsilon})$, within an approximation factor of $\min\{n^{\frac{1-\epsilon}{3}+o(1)}, (\mathcal{D}(S,R)/n^{\epsilon})^{\frac{1}{2}+o(1)}\}$. For the case of $\epsilon = 0$, we get a $\tilde{O}(n)$-time algorithm that approximates the edit distance within a factor of $\min\{n^{\frac{1}{3}+o(1)}, \mathcal{D}(S,R)^{\frac{1}{2}+o(1)}\}$. Note that for $\mathcal{D}(S,R) \leq n^{2/3}$, the approximation factor depends only on the distance $\mathcal{D}(S,R)$ and not on the string length n.

1.2 Our Techniques Our oblivious string embedding partitions an input string S over some alphabet σ into non-overlapping blocks and labels them in a consistent manner (i.e., identical blocks get identical labels). Clearly, a content-independent partitioning (one that does not depend on the specific composition of S) cannot have small distortion: a single character insertion to the beginning of S would change the contents of the blocks substantially. A content-dependent partitioning called the Locally Consistent Parsing (LCP) was introduced for addressing this problem in [12]. A variant of LCP [11] partitions a given string S into non-overlapping blocks of length 2 or 3. LCP guarantees that a single edit operation on S can change the content of only $O(\log^* |\sigma|)$ blocks. Thus, LCP leads to a string embedding with reduction parameter $2 \leq r \leq 3$ with distortion of $O(\log^* |\sigma|)$.

It is possible to obtain a larger reduction parameter r by iteratively applying the above embedding $O(\log_2 r)$ times. Unfortunately, each iteration magnifies the distortion by a factor of 3, resulting in an distortion of $O(\log^* |\sigma| \cdot 3^{\log_2 r}) = O(\log^* |\sigma| \cdot r^{\log 3})$. This is much larger than r, which, as we later show, is a lower bound on the distortion of a string embedding with reduction parameter r.

The first part of our paper is devoted to improving LCP to partition any input string into non-overlapping blocks of length c to $2c-1$ for any value of c. This poses significant technical difficulties in handling various types of periodicities in S and developing appropriate block partition rules. Through a number of combinatorial lemmas that generalize the Deterministic Coin Tossing technique of Cole and Vishkin [4][1] and methods to handle various types of local periodicities in S, we obtain a string embedding ϕ, that, for $c = (\log\log n)/\log\log\log n$, and any reduction parameter r, guarantees a distortion of $\tilde{O}(r^{1+\frac{2}{\log\log\log n}})$. The embedding can be computed for S in time $\tilde{O}(n)$.

Given the above embedding of two strings S, R, one can approximate $\mathcal{D}(S,R)$ within a factor of $\tilde{O}(r^{1+\frac{2}{\log\log\log n}})$ by computing $\mathcal{D}(\phi(S),\phi(R))$ in time $\Theta(\frac{n}{r}\cdot\mathcal{D}(\phi(S),\phi(R)))$ trivially by dynamic programming. Since $\mathcal{D}(\phi(S),\phi(R))$ could be larger than $\mathcal{D}(S,R)$, the running time of this approach is $\Omega(\frac{n}{r}\cdot\mathcal{D}(S,R))$. In the second part of the paper we show how to annotate the block labels implied by our string embedding ϕ to compute the edit distance between $\phi(S)$ and $\phi(R)$ in time $O(\frac{n}{r}\cdot\frac{\mathcal{D}(S,R)}{r})$.

1.3 Related Work The most relevant work to ours is that of Bar-Yossef et al., which presents two types of algorithms to approximate the edit distance between two strings of length n [2]. The first is a sketching algorithm, where a sketch of each given string is produced independently. The sketches for the two strings are then used to distinguish between close and far pairs of strings. In particular, using $O(1)$-size sketches and $\tilde{O}(n)$ time, they distinguish between string pairs that have edit distance at most k and pairs that have edit distance $\Omega((kn)^{2/3})$ for any $k \leq \sqrt{n}$. The second algorithm they present also runs in $\tilde{O}(n)$ time and directly computes $\mathcal{D}(S,R)$ within an approximation factor of $n^{3/7}$. Our methods give a $\tilde{O}(n)$-time sketching algorithm that uses sketches of size $O(n^{2/3})$ to obtain an approximation to edit distance within a factor of $n^{\frac{1}{3}+o(1)}$.

A sublinear algorithm that provides a weak approximation to edit distance is proposed by Batu et al. [3]. For a given parameter $\alpha < 1$, this algorithm runs in time $\tilde{O}(n^{\alpha/2} + n^{2\alpha-1})$ and distinguishes between pairs of strings that have edit distance at most n^{α} and pairs of strings that have edit distance $\Omega(n)$. This algorithm can also be viewed as a sketching algorithm with sketch size $\tilde{O}(n^{\alpha/2} + n^{2\alpha-1})$. However, it cannot provide any multiplicative-factor approximation to the edit distance.

Embedding strings into other metric spaces, in particular into normed spaces such as ℓ_1, with low distortion has also attracted much attention. Andoni et al. [1] give a lower bound of 3/2 for the distortion of any embedding of strings under edit distance to vectors (of arbitrary dimension) under the ℓ_1 metric. Khot and Naor [6] recently obtained a lower bound of $\Omega(\sqrt{(\log n)/\log\log n})$ for the distortion when embedding strings of length n into ℓ_1. Krauthgamer [7] later improved this result to $\Omega(\log n)$. A recent result by Ostrovsky and Rabani [10] obtains an embedding of strings of length n into vectors under ℓ_1 with distortion at most $2^{O(\sqrt{(\log n)\log\log n})}$, which is better than n^{ϵ} for any constant $\epsilon > 0$. The dimension of these vectors, however, is at least quadratic in n, making the embedding impractical for approximating edit distance in subquadratic time.

1.4 Organization of the Paper The rest of the paper is organized as follows. In Section 2, we provide

[1] This could be of independent interest in the context of distributed computing where deterministically partitioning a ring in a balanced manner could be useful.

some notation and describe in detail the problem of edit distance approximation. In Section 3, we describe the problem of embedding/dimensionality reduction for strings under edit distance and provide a lower bound on the distortion. In Section 4, we describe our generalized Locally Consistent Parsing (LCP(c)) method for any value of minimum block length c. Using this method iteratively, we obtain our string embedding result in Section 5. Finally, in Section 6, we present our approximation algorithm to the edit distance, which is based on our embedding techniques.

2 Preliminaries

Let S, R be two strings over some alphabet σ. We let $\mathcal{D}(S, R)$ denote the edit distance between S and R, i.e., the minimum number of character insertions, deletions, and substitutions needed to obtain R from S. We use $S[i]$ to denote the i^{th} character of the string S and $S[i, j]$ to denote the substring of S between positions i and j (inclusive). $|S|$ denotes the length of S and $S^r[i, j]$ denotes the reverse of $S[i, j]$.

An *alignment* $f : \{1, \ldots, |S|\} \to \{1, \ldots, |R|\} \cup \{\varepsilon\}$ between S and R associates each character of S (respectively, R) with at most one character from R (respectively, S) such that given $i < j$, if $f(i) = i'$ (i.e., $S[i]$ is aligned with $R[i']$) and $f(j) = j'$ (i.e., $S[j]$ is aligned with $R[j']$), then $i' < j'$. An optimal alignment f between S and R minimizes the sum of the number of unaligned characters (i.e., $\{i|f(i) = \varepsilon\} \cup \{j|\neg\exists i\, f(i) = j\}$) and misalignments (i.e., $\{i|f(i) = j$ and $S[i] \neq R[j]\}$). The sum of the number of unaligned characters and misalignments in an optimal alignment between S and R is equal to $\mathcal{D}(S, R)$.

In this paper we build efficient algorithms to approximate the edit distance between two strings. Since the edit distance problem is a minimization problem, we seek a one-sided approximation as follows.

Definition 2.1. *For $\gamma > 1$, a γ-approximation algorithm for the edit distance takes two strings S and R as inputs and outputs a value d such that $\mathcal{D}(S, R) \leq d \leq \gamma \cdot \mathcal{D}(S, R)$.*

We particularly investigate what approximation factor is obtainable in nearly linear (more precisely, in quasi-linear[2]) time for the edit distance. The approximation factors we obtain are functions of either the string length or $\mathcal{D}(S, R)$.

[2] $f(n)$ is in quasi-linear time, denoted $\tilde{O}(n)$, if $f(n) = O(n \cdot (\log n)^c)$ for some constant c.

3 Dimensionality Reduction in Strings Under Edit Distance

Given two metric spaces $M_1 = (X_1, D_1)$ and $M_2 = (X_2, D_2)$, where X_i is the universe and D_i is the distance measure for metric M_i, $\phi : X_1 \to X_2$ is an embedding with distortion $d = d_1 \cdot d_2$ for $d_1, d_2 \geq 0$, if, for any $y, z \in X_1$,

$$D_1(y, z)/d_1 \leq D_2(\phi(y), \phi(z)) \leq d_2 \cdot D_1(y, z).$$

Embedding $\phi : \sigma_1^* \to \sigma_2^*$, where σ_1^* and σ_2^* are string spaces under edit distance $\mathcal{D}(\cdot, \cdot)$, is a dimensionality reduction if ϕ maps each string to a shorter string.

A string embedding with *reduction* $r > 1$ is an embedding from strings of length at most n over an alphabet σ_1 under edit distance, to strings of length at most n/r over another alphabet σ_2, again under edit distance, that reduces the length of the string to be embedded by a factor of at least r. Thus, such an embedding is a dimensionality reduction. The following basic lemma will demonstrate that for any reduction parameter r, our string embedding has (almost) optimal distortion.

Lemma 3.1. *A string embedding with reduction $r > 1$ cannot have a distortion d less than r.*

Proof. Let $\phi : \sigma_1^* \to \sigma_2^*$ be a string embedding with reduction $r > 1$. Let $S, R \in \sigma_1^*$ be two strings such that $\mathcal{D}(S, R) = 1$. Clearly $\mathcal{D}(\phi(S), \phi(R)) > 0$; otherwise d_1 would be unbounded. Because the edit distance between two distinct strings is always a positive integer, d_2 is at least 1.

Now let $Z = a^n$ and $W = b^n$ where $a, b \in \sigma_1$. Observe that $\mathcal{D}(Z, W) = n$. Because ϕ has reduction r, we have that both $|\phi(Z)| \leq n/r$ and $|\phi(W)| \leq n/r$; thus, by the definition of edit distance, $\mathcal{D}(\phi(Z), \phi(W)) \leq n/r$. This implies that $d_1 \geq r$. Therefore $d = d_1 \cdot d_2 \geq r$. □

We note that Lemma 3.1 still holds true when the normalized edit distance, that is, $\mathcal{D}(S, R)/\max(|S|, |R|)$, is used. In that case, a single edit operation on the original string of length n creates a normalized edit distance of $1/n$, whereas after the embedding, the normalized edit distance will be at least r/n.

4 Generalized Locally Consistent Parsing

In this section we develop a *consistent* way of partitioning any string S into non-overlapping blocks such that the minimum block length is c and the maximum block length is $2c - 1$. In order to partition every string, the maximum block length cannot be less than $2c-1$; for instance, if $|S| = 2c-1$, then S cannot be partitioned into

blocks with length in the range $[c, 2c-2]$. The blocks obtained will be consistent in the following sense: if two identical substrings $S[i, i+b]$ and $S[j, j+b]$ appear in long enough identical "contexts" $S[i-\gamma(c), i+b+\gamma'(c)]$ and $S[j-\gamma(c), j+b+\gamma'(c)]$ for increasing functions $\gamma(c), \gamma'(c)$, and if $S[i, i+b]$ is identified as a block then $S[j, j+b]$ must be identified as a block. Note that $c \leq b+1 \leq 2c-1$.

Observe that a single edit operation on S will only have a local effect on such a partitioning of S. A single edit can change 1) the block in which it lies, and 2) the blocks whose neighborhoods as defined by $\gamma()$ and $\gamma'()$ contain the edit operation. Hence the number of blocks that can change as a result of a single edit operation is $O((\gamma(c)+\gamma'(c))/c)$.

A procedure that achieves consistent block partitioning as defined above, called Locally Consistent Parsing (LCP), was described in [12] for $c = 2$. LCP is based on the Deterministic Coin Tossing (DCT) procedure of Cole and Vishkin [4]. The block partitioning procedure we describe here generalizes DCT, and, as a result, LCP, for any value of c. We call the generalized procedure LCP(c).

The appropriate choice of c turns out to be crucial for our purposes. The reader may notice that the original LCP procedure, followed by labeling of the blocks, can be used recursively to ultimately partition the input string into blocks of size at least c (for any value of c). However, this approach would yield a ratio of $c^{\log_2 3/2}$ between the sizes of the longest and shortest blocks. By our generalized LCP, we improve this ratio to 2 for any c, thus obtaining more precise control over the partitioning of S.

While partitioning S, LCP(c) treats repetitive and nonrepetitive substrings of S differently. Repetitive substrings are partitioned in a straightforward way; for partitioning non-repetitive substrings we develop a novel algorithm that generalizes the Deterministic Coin Tossing technique and guarantees block sizes of c to $2c-1$. In this section we describe this new algorithm for small alphabets and in Section 4.1 we generalize it to integer alphabets. The importance of this generalization to our embedding will be seen in Section 5.

We start off by describing how to identify the repetitive substrings of the input string.

DEFINITION 4.1. *A string R is called* r-repetitive *if it is of the form Q^ℓ where $\ell \geq 2$ and Q, the* repetition, *is a string of length r. Given a string S, a substring R of S is called* maximally r-repetitive *if (1) it is r-repetitive with repetition T, where T is the substring that is the lexicographically greatest substring among all length-r substrings of R, and (2) the length-r substring following or preceding R (in S) is not T.*

For example, for $S = ababa$, as well as $S' = babab$, the only substring that is maximally 2-repetitive is $baba$. This information is helpful since it implies that S and S' have a long common substring. Note that every maximally repetitive string is periodic but not all periodic strings are maximally repetitive, e.g., $abab$ and $ababa$ are both periodic with period 2 but are not maximally repetitive since ab, a substring of both, is lexicographically smaller than ba.

LCP(c) performs the partitioning of S in two phases. Phase 1 partitions S into substrings that are maximally ℓ-repetitive for every $\ell < c$ and maximally non-repetitive as follows. For $r = c-1, \ldots, 1$, LCP(c) extracts all maximally r-repetitive substrings of S of length at least c that so far remain unextracted. All the remaining substrings (of maximal length) are identified as maximally non-repetitive substrings.

For example, if $S = aababaabd$ and $c = 3$, then LCP(c) will first identify $S[3,6] = baba$ as a maximally 2-repetitive substring; it will identify no maximally 1-repetitive substring and then will identify $S[1,2] = aa$ and $S[7,9] = abd$ as non-repetitive substrings (since $c = 3$, aa will not be identified as 1-repetitive).

Phase 2 further partitions the substrings extracted in Phase 1 to obtain blocks of length c to $2c-1$.

For partitioning repetitive substrings, each maximally r-repetitive substring is partitioned into blocks of length t where t is the smallest multiple of r greater than c. If the substring length is not divisible by t, the two leftmost blocks can be arranged so that the leftmost one is of size c. (This choice is arbitrary.)

For partitioning maximally non-repetitive substrings, first, any non-repetitive substring Q of length less than c is merged with the (necessarily repetitive) block to its left. If Q is a prefix of S, it is merged with the (again necessarily repetitive) block to its right. If the resulting block is of length greater than $2c$, it is partitioned (arbitrarily) into two blocks such that the left one is of length c. For non-repetitive substrings of length at least c, we perform a more sophisticated partitioning scheme that will ensure partition consistency as stated earlier. To achieve this, whether a character is selected to be a block boundary depends on the character's immediate neighborhood. The operations below facilitate the comparison of a character to other characters in its neigborhood.

DEFINITION 4.2. *Given two distinct binary words w and y of length k each, let $f_y(w)$ be a binary word of length $k' = \lceil \log k \rceil + 1$, defined as the concatenation of (i) the position of the rightmost bit b of w where w differs from y, represented as a binary number (counted starting from 1 at the right end), and (ii) the value of w at bit b.*

We define $f_w(w) = 0^{k'}$.

For example, $f_{1111}(1101) = 0100$ as the position of the rightmost bit of 1101 that differs from 1111 is 2 (010 in binary) and its value is 0.

DEFINITION 4.3. *For a character* $S[i]$ *and positive integers* c *and* ℓ*, we define*

$$g_{c,\ell}(S[i]) \stackrel{\text{def}}{=} f_{S[i-c-\ell+2,i+c-\ell-2]}(S[i-c+2, i+c-2]).$$

If $S[i]$ *is represented by a* k*-bit word, then* $g_{c,\ell}(S[i])$ *is a* k'*-bit word where* $k' = \lceil \log((2c-3)k) \rceil + 2$.

Intuitively, $g_{c,\ell}(S[i])$ relates the substring of size $2c-3$ around $S[i]$ to that of size $2c-3$ around $S[i-\ell]$.

Given a maximally non-repetitive substring Q, we generate an auxiliary substring Q' to help identify some of the block boundaries. Let $\lceil \log_2 |\sigma| \rceil = k$, where σ is the alphabet. For each $Q[i]$ (represented as a k-bit word), let

$$Q'[i] \stackrel{\text{def}}{=} g_{c,c-1}(Q[i]) \circ g_{c,c-2}(Q[i]) \circ \ldots \circ g_{c,1}(Q[i]).$$

The characters of Q' are represented as words of length $k' = (c-1) \cdot (\lceil \log((2c-3)k) \rceil + 2) = O(c \log(ck))$ bits. Thus the construction of Q' from Q constitutes an *alphabet reduction.* Since Q' will only be used to determine block boundaries, the information loss resulting from this alphabet reduction is not problematic.

We now prove that if Q is non-repetitive then Q' satisfies $Q'[i] \neq Q'[i-t]$ for $i = 4c-6, \ldots, |Q'| - c + 2$ and $t = 1, \ldots, c-1$, and is also non-repetitive.

LEMMA 4.1. *Let* Q *be a non-repetitive substring and let* $Q'[3c-5, |Q| - c + 2]$ *be the string obtained from* $Q[3c-5, |Q|-c+2]$ *after the alphabet reduction. Then, we have* $Q'[i] \neq Q'[i-\ell]$ *for* $\ell = 1, \ldots, c-1$.

Proof. Observe that given binary words x, y, z, such that $x \neq y$ and $y \neq z$, if the position of the rightmost bit b of x that differs from y is identical to the position of the rightmost bit b' of y that differs from z, then the bit values of b and b' must be different; i.e., $f_x(y) \neq f_y(z)$.

Fix $i \in [4c-6, |Q|-c+2]$ and $\ell \in [1, c-1]$. Consider

$$g_{c,\ell}(Q[i]) = f_{Q[i-c-\ell+2,i+c-\ell-2]}(Q[i-c+2, i+c-2])$$

and

$$g_{c,\ell}(Q[i-\ell]) = f_{Q[i-c-2\ell+2,i+c-2\ell-2]}(Q[i-c-\ell+2, i+c-\ell-2]).$$

Now, let $x = Q[i-c-2\ell+2, i+c-2\ell-2]$, $y = Q[i-c-\ell+2, i+c-\ell-2]$, and $z = Q[i-c+2, i+c-2]$.

Note that $x \neq y$; otherwise $Q[i-c-2\ell+2, i+c-\ell-2]$ includes an ℓ-repetitive substring of length more than c (which is impossible for a non-repetitive substring extracted by the algorithm). Similarly, $y \neq z$. Using the opening observation of the proof, we have $g_{c,\ell}(Q[i-\ell]) = f_x(y) \neq f_y(z) = g_{c,\ell}(Q[i])$. Hence, $Q'[i-\ell] \neq Q'[i]$. □

Now we are ready to identify our block boundaries on the nonrepetitive substring Q, using information from Q'. A character $Q[i]$ is set to be a *primary marker* if $Q'[i]$ is lexicographically greater than each character in its immediate neighborhood of length $2c-1$, namely, in $Q'[i-c+1, i+c-1]$. Note that primary markers are set in Q; Q' is solely used in helping determine their locations. The next lemma states that the distance between two consecutive primary markers is at least c and at most $2^{k'+2} = O((kc)^c)$ due to the construction of Q'.

LEMMA 4.2. *Let* $Q[i]$ *and* $Q[j]$ *be two consecutive primary markers in* Q *such that* $i < j$. *Then,* $c \leq j - i \leq 4(2^{k'}) = O((kc)^c)$.

Proof. We first prove the lower bound. Assume for contradiction that both $Q[i]$ and $Q[i+\ell]$ are primary markers for $\ell < c$. Let Q' be the string obtained from Q by the alphabet reduction. Then, by definition of a primary marker, $Q'[i]$ must be lexicographically greater than $Q'[i+\ell]$ and $Q'[i+\ell]$ must be lexicographically greater than $Q'[i]$, a contradiction. Thus, both $Q[i]$ and $Q[i+\ell]$ cannot be primary markers for $\ell < c$. .

We now prove the upper bound by induction on the size t of the alphabet for Q'. The bound holds for $t = 3$. By Lemma 4.1, we know that $Q'[i] \notin \{Q'[i-c+1], Q'[i-1], Q'[i+1], Q'[i+c-1]\}$. Also, recall that the characters of Q' are represented by binary strings of length k' (i.e., $t = 2^{k'}$). Without loss of generality, assume that both $Q'[i]$ and $Q'[j]$ are the lexicographically maximal alphabet character in σ' (this is the worst case). Then, since there are no primary markers between i and j, no character in $Q'[i+1, j-1]$ is the lexicographically maximal character. Moreover, in $Q'[i, j]$, the character just below the lexicographically maximal character can only be in positions $i+1, i+2, j-2, j-1$; otherwise, another primary marker would have been set. Without loss of generality, assume $Q'[i+2]$ and $Q'[j-2]$ are at most this lexicographically second largest character. Then, by the induction hypothesis, $j - 2 - (i+2) \leq 4(t-1)$. Thus, we get $j - i \leq 4t$. □

Having established the primary markers, LCP(c) now partitions Q into blocks as follows. Q is partitioned into the substrings that are between two consecutive primary markers (inclusive of the left primary marker),

to the left of the leftmost primary marker, or to the right of the rightmost primary marker. Each of these substrings is further partitioned into blocks of length c; if the substring length is not divisible by c, the leftmost block will be of length between $c+1$ and $2c-1$. The next lemma then follows.

LEMMA 4.3. *If $S[i,j]$ is a block obtained by LCP(c) then $c \leq j-i+1 \leq 2c-1$.*

Proof. By the rule that we use to place the secondary markers between two consecutive primary markers $Q[\ell_1]$ and $Q[\ell_2]$ ($\ell_1 < \ell_2$), markers are c positions apart, except the last secondary marker before the primary marker $Q[\ell_2]$, which might be on $Q[\ell_2 - 2c + 1], Q[\ell_2 - 2c + 2], \ldots,$ or $Q[\ell_2 - c]$ depending on the value of $(\ell_2 - \ell_1) \bmod c$. □

We now prove the consistency property of the above partitioning. We show that, if $S[i,j]$ and $S[i',j']$ are two identical non-repetitive substrings of S of sufficient length, then the blocks within $S[i,j]$ and $S[i',j']$, except at the left and right ends, are identical, regardless of the locations of $S[i,j]$ and $S[i',j']$ in S.

LEMMA 4.4. *Suppose that for some $b \in [c-1, 2c-2]$, $S[i-2^{k'+2}-4c+7, i+b+4c-3] = S[i'-2^{k'+2}-4c+7, i'+b+4c-3]$, and furthermore, both substrings are parts of substrings identified as being maximally nonrepetitive in S. Then, if $S[i,i+b]$ is set as a block by LCP(c), then so is $S[i',i'+b]$.*

Proof. By definition of primary markers and LCP(c), whether a character $S[\ell]$ (within a non-repetitive substring) is set as a marker depends only on $S[\ell - 4c + 7, \ell + 2c - 3]$. Since the decision to set $S[i,i+b]$ as a block depends only on the primary marker immediately to the left of $S[i]$ and whether there is a primary maker before $S[i+2c]$, we can conclude that this decision depends only on $S[i - 2^{k'+2} - 4c + 7, i + b + 4c - 3]$ by Lemma 4.2. As a result, $S[i,i+b]$ is set as a block only if $S[i',i'+b]$ is set as a block as well. □

In the preceding discussion, we described how to partition a nonrepetitive string Q, where Q is over alphabet σ such that $\lceil \log |\sigma| \rceil = k$, into blocks of size between c and $2c-1$ while maintaining a consistency property formalized in Lemma 4.4. This lemma guarantees that identical substrings are partitioned identically except in the margins. This implies thus that a single edit operation cannot change the partitioning of a string by "too much." More specifically, the number of blocks that can change as a result of an edit operation is $O((ck)^c)$ in a non-repetitive substring (by Lemma 4.4) and is only a constant in a repetitive substring.

4.1 Iterative Reduction of the Alphabet Size If $c \cdot k = O(1)$, by the above discussion, each edit operation results in only $O(1)$ changes in the partitioning of the input string and, thus, one application of the alphabet reduction suffices to obtain the desired partition. For $ck = \omega(1)$, there is a need to reduce the alphabet size further before setting the primary markers in order to guarantee that an edit operation will have limited effect on the partitioning. In this case, LCP(c) performs the alphabet reduction on each non-repetitive substring $S[i,j]$ of S, for $\log^* kc + O(1)$ iterations before setting the primary markers. Let $S^*[i,j]$ be the output of this process. Due to Lemma 4.1, since $S^*[i,j]$ is non-repetitive, so is $S^*[i,j]$. In the first iteration the alphabet size will be reduced to $O((ck)^c)$; in the second it will be $O((c^2 \log ck)^c)$ and so on. After $\log^* kc + O(1)$ iterations, the alphabet size will be $O((3c^2 \log c)^c)$, which is independent of k. The primary markers of $S[i,j]$ are then chosen as the local maxima in $S^*[i,j]$; this will assure that the maximum distance between two primary markers will be $O((3c^2 \log c)^c)$ as well. (Recall that the alphabet reduction is only for the purpose of obtaining primary markers. Afterwards, the partitioning is performed on the original string.)

Now we prove that the blocks on S obtained by LCP(c) have the property that h edit operations can change only $O(h \cdot [(\log^* kc) + (3c^2 \log c)^c/c])$ blocks.

THEOREM 4.1. *A sequence of h edit operations to a string S can change at most $O(h \cdot [(3c^2 \log c)^c/c + \log^* kc])$ and at least $h/(2c-1)$ blocks obtained by LCP(c).*

Proof. The lower bound follows from the fact that the maximum block size is $2c-1$ and, thus, the minimum possible number of blocks that can contain all h edit operations is $h/(2c-1)$.

The upper bound follows from repeated application of the next lemma.

LEMMA 4.5. *An edit operation on S can change only $O((3c^2 \log c)^c/c + \log^* kc)$ blocks obtained by LCP(c).*

Proof. LCP(c) initially partitions S into non-repetitive and r-repetitive substrings for $1 \leq r < c$.

Suppose the edit operation is performed on a non-repetitive substring, which remains non-repetitive after the operation. The first alphabet reduction on any $S[i]$ depends only on $S[i-3c+6, i+c-2]$. In general, the j^{th} application of the alphabet reduction on $S[i]$ depends on the substring $S[i-(3c-6)j, i+(c-2)j]$. Thus, for $j = \log^* kc + O(1)$, the output of the j^{th} alphabet reduction on $S[i]$ will be of size $O((3c^2 \log c)^c)$ and depend only on a substring of size $4c(\log^* kc + O(1))$ that contains

$S[i]$. This further implies that the decision of whether to choose $S[i]$ as a primary marker also depends only on a size $4c(\log^* kc + O(1)) + O((3c^2 \log c)^c)$ substring that contains $S[i]$. All blocks within this substring can change as a result of an edit operation on $S[i]$, implying a change of $4(\log^* kc+O(1))+O((3c^2 \log c)^c/c)$ blocks. As the distance between the first changed primary marker and its preceding primary marker is $O((3c^2 \log c)^c)$, a further $O((3c^2 \log c)^c/c)$ blocks can change as a result.

If the edit operation is performed on a non-repetitive substring that becomes repetitive then the same argument applies: The new repetitive substring splits the non-repetitive substring into two. This can change $4(\log^* kc + O(1)) + O((3c^2 \log c)^c/c)$ blocks on the two sides of the new repetitive substring.

If the edit operation is performed on a repetitive substring then the exact locations of the blocks may change; however only $O(1)$ of these blocks will change content. That is, one has to edit only $O(1)$ blocks in the original string in order to obtain the partitioning of the modified string. □

This completes the proof of Theorem 4.1. □

The next lemma states the running time of the LCP(c) procedure.

LEMMA 4.6. *LCP(c) runs in time* $O(n[c \log c + (k + c) \log^* kc])$.

Proof. Clearly the partitioning of a repetitive substring into blocks can be done in linear time in the size of the substring. We now show that the partitioning of all non-repetitive substrings of S takes $O(n[c \log c + (k + c) \log^* kc])$ time.

We first focus on the time for the first application of the alphabet reduction on a given $S[i]$ to obtain $S'[i]$. Consider the compact trie T_S that comprises the bitwise representations of $S^r[j-c+2, j+c-2]$ for all j. T_S can be obtained in $O(nk)$ time using any linear time suffix tree construction algorithm (e.g. [9]). After preprocessing T_S in $O(n)$ time, the lowest common ancestor (LCA) of two leaves representing $S^r[i - c + 2, i + c - 2]$ and $S^r[i'-c+2, i'+c-2]$ for any $i-c+1 \leq i' < i$ can be found in $O(1)$ time (c.f., [5, 13]). The LCA of these leaves gives $g_{c,i-i'}(S[i])$. To obtain $S'[i]$ one only needs to compute $g_{c,i-i'}(S[i])$ for all i' such that $i-c+1 \leq i' < i$; this can be done in time $O(c)$. Thus the overall running time for performing the alphabet reduction for all characters of S is $O(nk + nc)$.

Subsequent $O(\log^* kc)$ applications of the alphabet reduction work on smaller size alphabets; thus the overall running time is $O(n(k + c) \log^* kc)$.

After the alphabet reduction, determining whether each $S[i]$ is a primary marker can be done as follows. The number of bits needed to represent $S^*[i]$ is $O(c \log c)$; because $c \leq n$ this requires $O(c)$ machine words. One can use a priority queue that includes each one of the $O(c)$ characters in the substring $S^*[i - c + 2, i + c - 2]$ to determine whether $S^*[i]$ is the local maxima. This can be done, for all i, in time $O(nc \log c)$.

Once the primary markers are obtained, the final partitioning can be obtained in $O(n)$ time. □

5 String Embeddings via Iterative Applications of LCP(c)

LCP(c) partitions a string S into blocks of size c to $2c - 1$. These blocks can be labeled consistently; i.e., identical blocks get identical labels. (The label of a block could either be set to the block's contents, implying a large alphabet, or be computed via a hash function, introducing a small probability of error.) We denote the string of the labels obtained from S by S_1. This gives an oblivious embedding of strings that reduces their size by a factor of at least c and at most $2c - 1$.

Theorem 4.1 implies that given input strings S and R, the corresponding strings S_1 and R_1 satisfy

$$\begin{aligned} \mathcal{D}(S, R)/(2c - 1) &\leq \mathcal{D}(S_1, R_1) \\ &= \mathcal{D}(S, R) \cdot O((3c^2 \log c)^c/c + \log^* kc). \end{aligned}$$

For constant alphabet size k, $\mathcal{D}(S_1, R_1)$ provides a $\Theta((3c^2 \log c)^c)$ factor approximation to $\mathcal{D}(S, R)$. The tradeoff between the approximation factor and the running time is far from ideal. Because S_1 and R_1 are a factor c smaller than S and R, when one uses dynamic programming to compute $\mathcal{D}(S_1, R_1)$, as c grows, the time for the dynamic program decreases with $1/c^2$. On the other hand, the time for computing the embedding increases with c, and the approximation factor increases exponentially with c. In what follows we maintain a running time of $\tilde{O}(n)$ for computing the embedding, while improving the tradeoff between the approximation factor and the running time of the dynamic program by iteratively applying LCP(c) followed by a consistent labeling of blocks.

Define S_1 as above; let S_ℓ denote the string obtained by applying LCP(c) on $S_{\ell-1}$ followed by consistent block labeling. Each label in S_ℓ corresponds to a substring of S with size in the range $[c^\ell, (2c - 1)^\ell]$.

LEMMA 5.1. *A single edit operation on S results in* $O((3c^2 \log c)^c/c + \log^* kc)$ *labels to change in* S_ℓ.

Proof. By Lemma 4.5, the number of labels that can change in S_1 as a result of a single edit operation on S

is at most $\lambda \cdot ((3c^2 \log c)^c/c + \log^* kc)$ for some constant λ. The number of labels that can similarly change in S_2 is at most

$$\lambda \cdot (\frac{(3c^2 \log c)^c}{c} + \log^*(2c-1)kc) + \frac{\lambda}{c} \cdot (\frac{(3c^2 \log c)^c}{c} + \log^* kc).$$

This is due to the fact that the $\lambda \cdot ((3c^2 \log c)^c/c + \log^* kc)$ labels in S_1 that have changed as a result of the original edit operation on S will be grouped in at most $\frac{1}{c}\lambda \cdot ((3c^2 \log c)^c/c + \log^* kc)$ blocks, and, thus, will have an effect on an equal number of labels in S_2. As these labels are consecutive, there will only be an additional $\lambda \cdot ((3c^2 \log c)^c/c + \log^*(2c-1)kc)$ label changes on S_2. Note that

$$\begin{aligned} & \lambda \cdot (\frac{(3c^2 \log c)^c}{c} + \log^*(2c-1)kc) \\ & + \frac{\lambda}{c} \cdot (\frac{(3c^2 \log c)^c}{c} + \log^* kc) \\ & \leq \quad \frac{3}{2}\lambda \cdot ((3c^2 \log c)^c/c + \log^* kc) \end{aligned}$$

for $c \geq 3$. Thus at most $2\lambda \cdot ((3c^2 \log c)^c/c + \log^* kc)$ labels can change in S_ℓ for any $\ell \geq 2$. □

Thus, we reach the following corollary that is crucial to our edit distance approximation.

COROLLARY 5.1. $\mathcal{D}(S,R)/(2c-1)^\ell \leq \mathcal{D}(S_\ell, R_\ell) \leq \mathcal{D}(S,R) \cdot O((3c^2 \log c)^c/c + \log^* kc)$.

LEMMA 5.2. *We can compute S_ℓ, for all $1 \leq \ell \leq \log_c n$, in time $O(n[c \log c + (k+c)\log^* kc])$.*

Proof. By Lemma 4.6, we can compute S_1 from S in $O(n[c \log c + (k+c)\log^* kc])$ time. Since $|S_1| \leq |S|/c$ and the alphabet of S_1 is at most 4^c times that of S, we can compute S_2 from S_1 in $O(\frac{n}{c}[c \log c + (k+c)\log^*(2c-1)kc]) = \frac{1}{c} \cdot O(n[c \log c + (k+c)\log^* kc])$ time. The running time for computing S_ℓ drops by at least a constant fraction from that of $S_{\ell-1}$. Summing over all ℓ, the lemma follows. □

The above observations result in a better tradeoff between the approximation factor and the running time of approximating $\mathcal{D}(S,R)$ via $\mathcal{D}(S_\ell, R_\ell)$. As ℓ increases, the approximation factor increases proportionally with $(2c-1)^\ell$ due to Corollary 5.1. However, as S_ℓ is of length at most n/c^ℓ, the running time decreases proportionally with $1/c^{2\ell}$.

THEOREM 5.1. *There is a string embedding ϕ with reduction r and distortion $\tilde{O}(r^{1+\mu})$ for $\mu > 2/\log\log\log n$. For any string S, its embedding $\phi(S)$ can be computed in time $\tilde{O}(2^{1/\mu} \cdot n)$.*

Proof. We set $c = 2^{1/\mu} \leq (\log\log n)/\log\log\log n$. Let $\phi(S) = S_{\lceil \log_c r \rceil}$ for this value of c. The distortion follows from Corollary 5.1 and the running time follows from Lemma 5.2 as $O(c \log c + (k+c)\log^* kc)$ is, by choice of c, easily bounded by polylog(n). □

The gain from using LCP(c) instead of LCP(2) in our embedding arises when we use LCP(c) recursively. The ℓ^{th}-level recursive partitioning produced by LCP(2) yields block sizes between 2^ℓ and 3^ℓ, whereas LCP(c) yields block sizes between c^ℓ and $(2c-1)^\ell$. While using LCP(c) in order to reduce the string length to n/r, we have to choose $\ell \geq \log_c r$. This, however, only guarantees a distortion of $(2c-1)^\ell = (2c-1)^{\log_c r} \sim r^{1+\log_c 2}$. Hence, for a fixed r, a larger value for c yields a smaller distortion.

6 Edit Distance Approximation Using our String Embedding

In this section we describe our edit distance approximation algorithm. Let S_ℓ and R_ℓ be strings obtained by the application of the oblivious embedding ϕ (from the previous section) on the input strings S and R.

By Corollary 5.1, $\mathcal{D}(S_\ell, R_\ell)$ provides an approximation to $\mathcal{D}(S,R)$ within a factor of $O((2c-1)^\ell \cdot [(3c^2 \log c)^c/c + \log^* kc])$. Because $|S_\ell|, |R_\ell| \leq n/c^\ell$, one can trivially compute $\mathcal{D}(S_\ell, R_\ell)$ in time $O((n/c^\ell)^2)$ by dynamic programming. Hence, for $\epsilon \geq 0$, one can get an $(n^{\frac{1-\epsilon}{2}+o(1)})$-approximation to edit distance in time $\tilde{O}(n^{1+\epsilon})$ by setting $(n/c^\ell)^2 = n^{1+\epsilon}$. Below we improve this by using observations about the embedding ϕ.

Given an upper bound t on $\mathcal{D}(S,R)$, one can determine the actual value of $\mathcal{D}(S,R)$ in $O(nt)$ time by computing the dynamic programming table along a band of width $2t$ around the main diagonal, since any entries corresponding to positions in S and R that are more than t locations apart do not need to be filled in. This technique can be tailored to compute, given a parameter t, an approximation $\mathcal{D}_t(S_\ell, R_\ell)$ to $\mathcal{D}(S_\ell, R_\ell)$ in time $O(nt/c^{2\ell})$ with the property that, if $\mathcal{D}(S,R) \leq t$, then $\mathcal{D}_t(S_\ell, R_\ell)$ will be close to $\mathcal{D}(S_\ell, R_\ell)$. The cost-cutting measure in the computation of $\mathcal{D}_t(S_\ell, R_\ell)$ is that entries for label pairs corresponding to substrings of S and R whose starting locations (in S and R respectively) are more than t positions apart are not filled in in the dynamic programming table. Consequently, only $O(nt/c^{2\ell})$ entries are filled in since each of the n/c^ℓ labels in S_ℓ is compared to at most $O(t/c^\ell)$ labels in R_ℓ.

The next lemma shows an analog of Corollary 5.1 for $\mathcal{D}_t(S_\ell, R_\ell)$.

LEMMA 6.1. *If $\mathcal{D}(S,R) \leq t$, then $\mathcal{D}(S,R)/(2c-1)^\ell \leq \mathcal{D}_t(S_\ell, R_\ell) \leq \mathcal{D}(S,R) \cdot O((3c^2 \log c)^c/c + \log^* kc)$.*

Proof. The left-hand side holds since at most $(2c-1)^\ell$ edits to S can be grouped into one block and, thus, appear as a single edit to S_ℓ. To prove the upper bound, consider the optimal alignment between S and R, and the corresponding alignment between S_ℓ and R_ℓ. That is, the latter alignment matches a label in S_ℓ to one in R_ℓ only if all the characters represented by these labels are matched to each other between R and S in the former (optimal) alignment. Consider a sequence of edit operations on S to obtain R (respecting the optimal alignment between S and R). Applying Lemma 5.1 for each such edit shows that the edit distance implied by the alignment between S_ℓ and R_ℓ mentioned above satisfies the bounds mentioned in Corollary 5.1. Since this alignment clearly never matches two labels which represent substrings of S and T whose starting points are more than t locations apart, it will be considered by the dynamic program computing $\mathcal{D}_t(S_\ell, R_\ell)$, and, thus, $\mathcal{D}_t(S_\ell, R_\ell)$, will respect the upper bound. □

We now state our main theorem regarding the edit distance approximation.

THEOREM 6.1. *For any $\epsilon \geq 0$, it is possible to approximate $\mathcal{D}(S,R)$ in time $\tilde{O}(n^{1+\epsilon})$ within an approximation factor*

$$\tilde{O}(\min\{n^{\frac{1-\epsilon+(1-\epsilon)\delta}{3+\delta}}, (\mathcal{D}(S,R)/n^\epsilon)^{\frac{1}{2}(1+\delta)}\}),$$

where $\delta = \frac{2}{\log\log\log n}$.

Proof. We produce $O(\log n)$ estimates for $\mathcal{D}(S,R)$. In iteration h, we set $t = 2^h$ and $\ell = \frac{1}{2}\log_c \frac{t}{n^\epsilon}$. We then estimate $\mathcal{D}(S,R)$ to be $\mathcal{D}_t(S_\ell, R_\ell) \cdot (2c-1)^\ell$. Note that the estimate of any iteration provides an upper bound to $\mathcal{D}(S,R)$. By the preceding discussion, the running time of iteration h is $O(nt/c^{2\ell}) = O(n^{1+\epsilon})$. If $\mathcal{D}(S,R) \leq t$, then the estimate of iteration h provides an approximation factor of

$$\begin{aligned} & O((2c-1)^\ell \cdot [(3c^2\log c)^c/c + \log^* kc]) \\ = \;& O((t/n^\epsilon)^{\frac{1}{2}(1+\log_c 2)} \cdot [(3c^2\log c)^c/c + \log^* kc]). \end{aligned}$$

Let c be set to $(\log\log n)/\log\log\log n$. The minimum of all estimates guarantees an approximation factor of $\tilde{O}((\mathcal{D}(S,R)/n^\epsilon)^{\frac{1}{2}(1+\delta)})$, where $\delta = 2/\log\log\log n$. If this estimate exceeds n, then the algorithm estimates $\mathcal{D}(S,R)$ to be n.

Now we bound the above approximation factor as a function of n. If

$$\mathcal{D}(S,R) \cdot \left(\frac{\mathcal{D}(S,R)}{n^\epsilon}\right)^{(1+\delta)/2} > n,$$

that is,

$$\mathcal{D}(S,R) > n^{\frac{2+\epsilon(1+\delta)}{3+\delta}},$$

then the algorithm's estimate, which is at most n, will provide an approximation to $\mathcal{D}(S,R)$ within a factor of

$$n/n^{\frac{2+\epsilon(1+\delta)}{3+\delta}} = n^{\frac{1+(1-\epsilon)\delta-\epsilon}{3+\delta}}.$$

On the other hand, if $\mathcal{D}(S,R) \leq n^{\frac{2+\epsilon(1+\delta)}{3+\delta}}$, then the algorithm's approximation factor becomes

$$\tilde{O}((\mathcal{D}(S,R)/n^\epsilon)^{\frac{1}{2}(1+\delta)}) = \tilde{O}(n^{\frac{1+(1-\epsilon)\delta-\epsilon}{3+\delta}}).$$

Thus, the overall approximation factor of the algorithm is

$$\tilde{O}(\min\{n^{\frac{1-\epsilon+(1-\epsilon)\delta}{3+\delta}}, (\mathcal{D}(S,R)/n^\epsilon)^{\frac{1}{2}(1+\delta)}\}).$$

□

Setting ϵ to be 0, we obtain the following corollary.

COROLLARY 6.1. *There is an algorithm that computes $\mathcal{D}(S,R)$ for two strings S and R of length n within an approximation factor of*

$$\min\{n^{\frac{1}{3}+o(1)}, \mathcal{D}(S,R)^{\frac{1}{2}+o(1)}\}$$

in time $\tilde{O}(n)$.

References

[1] A. Andoni, M. Deza, A. Gupta, P. Indyk, and S. Raskhodnikova. Lower bounds for embedding edit distance into normed spaces. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[2] Z. Bar-Yossef, T. Jayram, R. Krauthgamer, and R. Kumar. Approximating edit distance efficiently. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2004.

[3] T. Batu, F. Ergün, J. Kilian, A. Magen, S. Raskhodnikova, R. Rubinfeld, and R. Sami. A sublinear algorithm for weakly approximating edit distance. In *ACM Symposium on Theory of Computing, (STOC)*, 2003.

[4] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *ACM Symposium on Theory of Computing (STOC)*, 1986.

[5] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[6] S. Khot and A. Naor. Nonembeddability theorems via Fourier analysis. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2005.

[7] R. Krauthgamer. On L_1-embeddings of the edit distance. Unpublished manuscript, 2005.

[8] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, February 1980.

[9] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[10] R. Ostrovsky and Y. Rabani. Low distortion embeddings for edit distance. In *ACM Symposium on Theory of Computing (STOC)*, 2005.

[11] S. C. Sahinalp and U. Vishkin. Data compression using locally consistent parsing. *UMIACS Technical Report*, 1995.

[12] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.

[13] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, December 1988.