Locally Consistent Parsing and Applications to Approximate String Comparisons

Tuğkan Batu and S. Cenk Sahinalp

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

Abstract. Locally consistent parsing (LCP) is a context sensitive partitioning method which achieves partition consistency in (almost) linear time. When iteratively applied, LCP followed by consistent block labeling provides a powerful tool for processing strings for a multitude of problems. In this paper we summarize applications of LCP in approximating well known distance measures between pairs of strings in (almost) linear time.

1 Introduction

Locally consistent parsing (LCP) is a method to partition a string S from an alphabet σ to short substrings in a "consistent" fashion. Consistency is achieved in the following manner: identical substrings are partitioned identically with the possible exception of their margins. We give a more precise definition of consistency later in the paper.

LCP has been introduced more than 10 years ago. Since then, it has been applied to a multitude of problems involving strings such as data structures problems, pattern matching applications, data compression, and string embeddings. Below, we give a brief history of the development of LCP and some of its key applications.

LCP is based on the Deterministic Coin Tossing (DCT) procedure of Cole and Vishkin [5], which was introduced to deterministically partition a ring of n processors, each with a unique ID, to blocks of size 2 or 3 in a distributed fashion. The procedure was iteratively applied to the representative processors (say, the leftmost one) in each block to obtain a hierarchical partitioning. This was in turn used to perform list ranking of the processors in $O(\log n)$ rounds and a total of $O(n)$ operations.

Two surprising applications of the DCT appeared (almost simultaneously) in 1994, which generalized DCT to strings with character repetitions [13, 15, 16]. LCP is this generalization of DCT to string partitioning. In [13], an efficient data structure for maintaining a dynamic collection of strings that allow equality tests, concatenation and split operations is described. In [15, 16], a novel algorithm for building the suffix tree of a given string in polylogarithmic parallel time while performing a total of $O(n)$ operations is given. Both applications are based iterative application of LCP followed by a consistent labeling of these blocks or their extensions.

Later in [17], a single-pass data compression algorithm based on LCP is introduced. The number of codewords output by this algorithm is at least as many as that output by the ever popular Lempel-Ziv'77 algorithm and at most $O(\log n)$ factor higher. However, the size of the dictionary and, thus, the size of each codeword can be substantially smaller for low entropy texts, potentially resulting in a better compression. A recent paper [7] presents (among other results) an extension of this algorithm that achieves the following: after preprocessing a given string in time $O(n \log^2 n)$ compute the Lempel-Ziv'77 compressibility of any of its substrings in $O(1)$ time within an approximation factor of $O(\log n \log^* n)$.

In [18], LCP is applied to pattern matching under edit distance; here, the goal is to find all substrings of a given text string T whose edit distance to a pattern string P is no more than k. The algorithm presented here is the first one achieving $o(|P| \cdot |T|)$ time for small values of k. This result was later improved in [4] by the use of other techniques. Also in [18], an efficient data structure for dynamic text indexing based on LCP is described. Here the goal is to maintain a dynamic string, subject to single character insertions, deletions and replacements, so that substring membership queries can be answered efficiently. An improvement of this data structure, which is again based on LCP, was later described in [1]. Finally, [18] shows how to maintain a dictionary data structure under insertion and deletion of dictionary entries so that given a text string, all occurrences of each dictionary entry can be efficiently computed.

In this paper we review applications of LCP to approximately computing several variants of the edit distance between a pair of strings in almost linear time. The standard (character) edit distance between two strings can be computed exactly in quadratic time for general alphabets and slightly under quadratic time for a constant-size alphabet [11]. On the other hand, the block edit distance and edit distance with (block) moves are known to be NP-hard to compute. The task of efficiently computing these edit distance variants, even approximately, is of significance, especially in computational biology, where the data is very large and thus fast algorithms are highly desired. LCP has been successfully used to achieve this task for all three measures of string similarity.

One fast method for quickly approximating certain variants of edit distance is based on embedding strings to metric spaces with simpler-to-compute distance measures. The first such embedding is described in [14] for strings under block edit distance (the minimum number of character edits and block moves, copies and "uncopies" to transform one string into the other) into the Hamming space. The embedding, which results in a distortion of $O(\log n(\log^* n)^2)$, is computed in almost linear time, implying a fast approximation algorithm for the block edit distance. A followup result [6] gives a similar embedding of strings under edit distance with (block) moves into L_1 space with distortion $O(\log n(\log^* n)^2)$.

All the results described so far are obtained by a specific version of LCP which partitions an input string into blocks of size 2 or 3. A generalization of LCP so that the input strings are partitioned into blocks of size at least c and at most $2c-1$ for any user defined c is recently described in [3]. This particular version of LCP is applied to obtain a dimensionality reduction technique on

strings. More specifically [3] shows how to embed any string S of length n into a string of length at most n/r for any value of the *contraction parameter* r. The embedding preserves the edit distance between two strings within a factor of $\tilde{O}(r^{1+\mu})$ for some small μ. This embedding together with some annotations are later used to compute the edit distance $\mathcal{D}(S, R)$ between two strings S and R within an approximation factor of $\min\{n^{\frac{1}{3}+o(1)}, \mathcal{D}(S, R)^{\frac{1}{2}+o(1)}\}$ in almost linear time.

Overview of the paper. We start in Section 2, by giving some definitions and notation. In Section 3, we describe the (generalized) Locally Consistent Parsing method for any value of (minimum block length) c. We summarize applications of LCP to approximately computing edit distances in Section 4.1.

2 Preliminaries

Let S, R be two strings over some alphabet σ. 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 $S[i, j]$. For two strings S and R, $S \circ R$ denotes the string obtained by concatenating S and R. We denote by $\mathcal{D}(S, R)$ the edit distance between S and R, i.e., the minimum number of character insertions, deletions, and substitutions needed to obtain R from S.

An alignment between S and R associates each character of S (and R) with at most one character from R (correspondingly S) such that given $i < j$, if $S[i]$ is aligned with $R[i']$ and $S[j]$ with $R[j']$ then $i' < j'$. An optimal alignment between S and R minimizes the sum of the number of unaligned characters and misalignments (alignments between non-identical characters) in S (and R). The sum of the number of unaligned characters in S and R and misalignments between S and R in an optimal alignment is equal to $\mathcal{D}(S, R)$.

Another measure of similarity between the strings S and R is the block edit distance, denoted $\mathcal{BED}(S, R)$, which is defined to be the minimum number of character edits and *block edits* to transform one string into the other. Character edits are insertion, deletion and replacement of a single character. Block (substring) edits are relocating an entire substring, copying a substring from one location to another and "uncopying" a block; i.e. deleting one of the two copies of a substring.[1]

One final measure of similarity between strings S and R is edit distance with moves, denoted by $\mathcal{MV}(S, R)$. Here, all single character edit operations as well as substring relocation operation is allowed.

Metric Embeddings. Many of the applications of LCP to "approximately" compare strings are based on embedding strings under various edit distances to other

[1] Other versions of block edit distance that allow substring reversals and linear transformations on substrings have also been described. Here we only focus on the "vanilla" block edit distance.

metric spaces. 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, $\phi : X_1 \rightarrow X_2$ is an embedding with distortion $d = d_1 \cdot d_2$ for $d_1, d_2 \geq 0$, if, for any $y, z \in X_1$,

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

3 A Generalized Description of Locally Consistent Parsing

Locally Consistent Parsing is 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 $2c - 1$. For full generality, 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 $O((\gamma(c) + \gamma'(c))/c)$.

LCP was originally described for $c = 2$. This basic case, which we denote by LCP(2), relies on the Deterministic Coin Tossing technique [5]; it is quite simple but is sufficiently powerful to achieve the desired performance in many applications, some of which are described here. The more general LCP(c)was introduced in [3] to solve a number of problems where the choice of c turns out to be of crucial importance. Here we describe the general version while illustrating the main ideas through examples based on LCP(2).

3.1 Description of LCP(c) for Small Alphabets

Given input string S, LCP(c) treats repetitive and nonrepetitive substrings of S differently. Repetitive substrings are partitioned in a straightforward way; for partitioning non-repetitive substrings, a generalized version of the Deterministic Coin Tossing technique is used to guarantee block sizes of c to $2c - 1$.

Here we describe LCP(c) for small alphabets; in Section 3.2, we show how to generalize it to integer alphabets. We start off by describing how to identify the repetitive substrings of the input string.

Definition 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 $T = ababa$, as well as $T' = babab$, the only substring that is maximally 2-repetitive is $baba$. This information is helpful since it implies that T and T' 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 $\ell < c$ 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 $S = aabaaaababa$ and $c = 2$, then LCP(c) will first identify $S[1,2] = aa$ and $S[4,7] = aaaa$ as maximally 1-repetitive substrings.

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.

In the above example, $S[1,2] = aa$ will be identified as a single block of size 2 and $S[4,7] = aaaa$ will be partitioned into two blocks $S[4,5]$ and $S[6,7]$. The non-repetitive block $S[3,3] = b$ is then merged to the block to its right to form the block $S[3,5] = baa$.

For non-repetitive substrings of length at least c LCP(c) performs a more sophisticated partitioning scheme that ensures 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 defined below facilitate the comparison of a character to other characters in its neighborhood.

Definition 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 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, LCP(c) generates 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* as long as $k' < k$. Since Q' will only be used to determine block boundaries, the information loss resulting from this alphabet reduction is not problematic.

Lemma 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 $Q'[3c-5, |Q|-c+2]$ is non-repetitive.*

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 the 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.

Lemma 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 give the proof for 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 give the proof for the upper bound. It is by induction on the size t of the alphabet for Q'. The bound holds for $t = 3$. By Lemma 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), the one to the left of the leftmost primary marker, and the one 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 claim then follows.

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

The consistency of the above partitioning is established in the following manner. 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 3. *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 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 3. 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 3) and is only a constant in a repetitive substring.

3.2 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 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.)

Theorem 1. *A sequence of h single character 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 in the sequence of 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.

Claim. 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 1. □

Lemma 4. *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., [12]). 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., [8, 19]). 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. □

4 Applications of LCP to Approximate String Comparisons

Among the applications of LCP our focus will be embeddings of strings under character and block edit distances to other metric spaces such as the Hamming Space or L_1, as well as to "shorter" strings (such an embedding is called a dimensionality reduction). These embeddings (sometimes together with additional information) give fast algorithms to approximately compute the edit distance variants of interest. Here we give a brief description of the techniques underlying each application as well as theorem statements without proofs.

4.1 Dimensionality Reduction in Strings Under Edit Distance

An embedding ϕ from $M_1 = (X_1, D_1)$ to $M_2 = (X_2, D_2)$, is a dimensionality reduction if $D_1 = D_2$ and each element in X_1 is mapped to an element with shorter representation in X_2. A string embedding with *contraction* $r > 1$ 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, which contracts the length of the string to be embedded by a factor of at least r. Thus, such an embedding is a dimensionality reduction. A proof for the following basic lemma can be found in [3].

Lemma 5. *A string embedding with contraction $r > 1$ cannot have a distortion d less than r.*

Here we present a dimensionality reduction technique for strings which follows from an iterative application of LCP(c) followed by consistent labeling of blocks [3].[2] The number of iterations, ℓ, is determined by the contraction parameter r as follows.

Given input strings S and R and $|\sigma| = 2^k$, denote by S_1 and R_1 the strings that are obtained after a single application of LCP(c), respectively. Now, denote by S_ℓ 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 $[c^\ell, (2c-1)^\ell]$.

Theorem 1 implies the following lemma.

[2] 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.

Lemma 6. *A single edit operation on S results in $O((3c^2 \log c)^c/c + \log^* kc)$ labels to change in S_ℓ. Thus,*

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

Now let $c = O((\log\log n)/\log\log\log n)$; then, $\phi(S) = S_{\lceil \log_c r \rceil}$ provides a string embedding with distortion (roughly) r, which is almost optimal.

Theorem 2. *Given string S and $r > 1$, the embedding $\phi(S) = S_{\lceil \log_c r \rceil}$ has contraction r and distortion $\tilde{O}(r^{1+\mu})$, where $\mu = \Omega(2/\log\log\log n)$. The embedding can be computed in time $\tilde{O}(2^{1/\mu} \cdot n)$.*

4.2 Embedding Strings into L_1 and Hamming Space

In this section we describe two very similar results: (i) embedding strings under block edit distance to the Hamming space [14] and (ii) embedding strings under edit distance with moves to L_1 [6].

The embedding of a string S of length n under block edit distance to a Hamming vector is based on iterative application of LCP(2). The consistent labeling following LCP(2), however, is not performed on blocks obtained by LCP(2). Rather, it is applied to "extended blocks" that are called *core* blocks, which are defined as follows. For each block divider between $S[i-1]$ and $S[i]$, there is a corresponding core block, which is, for some $b = O(\log^* n)$, the string $S[i-b, i+b]$.[3] Accordingly, let S'_1 be the sequence of labels of the core blocks implied by LCP(2) applied on S. One can define, for $\ell > 1$, the string S'_ℓ as the sequence of labels of the core blocks implied by LCP(2) applied on $S'_{\ell-1}$. Notice that each core block at level ℓ corresponds to a substring of S; this substring is called a *core substring* of S. Appropriate choice of b ensures the following property of the core substrings [14].

Lemma 7. *If Q is a non-repetitive core substring of S and Q' is another substring of S identical to Q, then Q' must be a core substring of S as well.*

We now describe the embedding of S into a binary vector; without loss of generality, we assume that $\sigma = \{0, 1\}$.

Definition 4. *Given string S, its* level-i binary histogram*, denoted $T_i(S)$, is the binary vector of size c^i where $c = O(\log^* n)$; the jth entry of $T_i(S)$, denoted $T_i(S)[j]$, is 1 if the binary representation of j is a core substring of S whose corresponding core block is in S'_i. If j is not a core of S then $T_i(S)[j] = 0$. The embedding $\phi'(S)$ is then the concatenation of all $T_i(S)$ for $i = 0, \ldots, \log n$.*

The embedding $\phi'(S)$ is an $O(2^{|S|})$ dimensional binary vector whose j^{th} entry is set to 1 if the corresponding core is present in S. Although the number of dimensions in $\phi'(S)$ is very large, it is possible to represent it in $O(n)$ space by

[3] The LCP(2) description in [14] is a slightly different from the one provided here for the purpose of tolerating substring reversal as a block edit operation.

simply listing the $O(n)$ core strings of S by the use of pointers. This alternative representation can be computed in time $O(n \cdot \log^* n)$.

The following lemma is implied by the proof of Theorem 1.

Lemma 8. *Let $C(S)$ denote the set of core blocks obtained by LCP(2) on a string S. A sequence of h single character edits, block copy, block move and block uncopy operations to S can add or remove most $O(h \log^* n)$ and at least $\Omega(h)$ core blocks from $C(S)$.*

As a result, given two strings S and R s.t. $|S|, |R| \leq n$, their embeddings $\phi'(S)$ and $\phi'(R)$ satisfy the following.

Theorem 3. *$\Omega(\mathcal{BED}(S,R))/\log^*(n) = h(\phi'(S), \phi'(R)) = O(\mathcal{BED}(S,R) \log n \log^* n)$; here $h(,)$ denotes the Hamming distance.*

The embedding of strings under block edit distance to Hamming vectors, together with efficient data structures for $(1+\epsilon)$-factor approximate nearest neighbor search in the Hamming space [10, 9], can be used to obtain an $O(\log n \log^* n)$ factor approximate nearest neighbor search data structure for strings under block edit distance. The preprocessing time and space for the data structure is $O((nm)^{O(1)})$ where m is the number of strings in the data structure; the query time is $O(n \cdot \text{polylog}(nm))$. This is the first and yet the only known approximate nearest neighbor search data structure for strings under non-trivial edit operations that achieves almost optimal preprocessing time, space and query time, while guaranteeing an approximation factor polylogarithmic in the query size.

A very similar embedding $\phi''(S)$ from strings under edit distance with moves to L_1 space is described in [6]. Without loss of generality, we describe this embedding for $\sigma = \{0, 1\}$.

Definition 5. *Given string S, its* level-i histogram*, denoted $H_i(S)$, is the integer vector of size c^i ($c = O(\log^* n)$) where the jth entry of $H_i(S)$, denoted $H_i(S)[j]$, is the number of occurrences of the binary representation of j as a core substring in S, whose corresponding core block is in S''_i. The embedding $\phi''(S)$ is the concatenation of all $H_i(S)$ for $i = 0, \ldots, \log n$.*

The embedding ϕ'' can be shown to satisfy the following property based on a variation of Lemma 8.

Theorem 4. *$\Omega(\mathcal{MV}(S,R)) = ||\phi'(S) - \phi'(R)||_1 = O(\mathcal{MV}(S,R) \log n \log^* n)$.*

4.3 Approximately Computing the Edit Distance in (Near) Linear Time

Given $\gamma > 1$, a γ-factor approximation algorithm for $\mathcal{D}(S,R)$ outputs a value d such that $\mathcal{D}(S,R) \leq d \leq \gamma \cdot \mathcal{D}(S,R)$. Recently Bar-Yossef et al. developed an algorithm that computes $\mathcal{D}(S,R)$ within an approximation factor of $n^{3/7}$ in $\tilde{O}(n)$ time [2]. It was recently shown in [3] a method to apply LCP to compute

the edit distance between two strings within an approximation factor of $n^{\frac{1}{3}+o(1)}$ again in $\tilde{O}(n)$ time. Here we summarize this result.

Let S_ℓ and R_ℓ be as described in Section 4.1 for $c = (\log n)/\log\log n$. We have already demonstrated that $\mathcal{D}(S_\ell, R_\ell)$ approximates $\mathcal{D}(S, R)$ within a factor of $\tilde{O}((2c-1)^\ell)$. By the use of standard dynamic programming, $\mathcal{D}(S_\ell, R_\ell)$ can be computed in time $O((n/c^\ell)^2)$. However, if an upper bound t on $\mathcal{D}(S, R)$ is given, it is possible to approximate $\mathcal{D}(S, R)$ by computing $\mathcal{D}(S_\ell, R_\ell)$ only along a band of width $2t/c^\ell$ around the main diagonal of the dynamic programming table.

By applying this strategy for potential upper bounds on $\mathcal{D}(S, R)$, $t = 2^i$, for $i = 1, \ldots, O(\log n)$, followed by checking t indeed provides an upper bound, it is possible to obtain an approximation to $\mathcal{D}(S, R)$ as per [3].

Theorem 5. *One can compute $\mathcal{D}(S, R)$ 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. Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 819–828, 2000.
2. Ziv Bar-Yossef, T.S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2004.
3. Tugkan Batu, Funda Ergun, and S. Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. Technical Report TR2005-11, School of Computing Science, Simon Fraser University, 2005.
4. Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. In *SODA*, pages 463–472, 1998.
5. Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *ACM Symposium on Theory of Computing (STOC)*, 1986.
6. Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *SODA*, pages 667–676, 2002.
7. Graham Cormode and S. Muthukrishnan. Substring compression problems. In *SODA*, 2005.
8. Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.
9. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.
10. Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *STOC*, pages 614–623, 1998.
11. William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, February 1980.

12. Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
13. Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *SODA*, pages 213–222, 1994.
14. S. Muthukrishnan and S. Cenk Sahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *STOC*, pages 416–424, 2000.
15. S. Cenk Sahinalp and Uzi Vishkin. On a parallel-algorithms method for string matching problems. In *CIAC*, pages 22–32, 1994.
16. S. Cenk Sahinalp and Uzi Vishkin. Symmetry breaking for suffix tree construction. In *STOC*, pages 300–309, 1994.
17. S. Cenk Sahinalp and Uzi Vishkin. Data compression using locally consistent parsing. *UMIACS Technical Report*, 1995.
18. S. Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.
19. Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, December 1988.