Approximate nearest neighbors and sequence comparison with block operations

S. Muthukrishnan * Süleyman Cenk Şahinalp †

Abstract

We study sequence nearest neighbors (SNN). Let D be a database of n sequences; we would like to preprocess D so that given any on-line query sequence Q we can quickly find a sequence S in D for which $d(S,Q) \leq d(S,T)$ for any other sequence T in D. Here $d(S,Q)$ denotes the distance between sequences S and Q, defined to be the minimum number of edit operations needed to transform one to another (all edit operations will be reversible so that $d(S,T) = d(T,S)$ for any two sequences T and S). These operations correspond to the notion of similarity between sequences that we wish to capture in a given application. Natural edit operations include character edits (inserts, replacements, deletes etc), block edits (moves, copies, deletes, reversals) and block numerical transformations (scaling by an additive or a multiplicative constant). The SNN problem arises in many applications.

We present the first known efficient algorithm for "approximate" nearest neighbor search for sequences with preprocessing time and space polynomial in size of D and query time near-linear in size of Q. We assume the distance $d(S,T)$ between two sequences S and T is the minimum number of character edits and block operations needed to transform one to the other. The approximation factor we achieve is $O(\log \ell (\log^* \ell)^2)$, where ℓ is the size of the longest sequence in D. In addition, we also give an algorithm for exactly computing the distance between two sequences when edit operations of the type character replacements and block reversals are allowed. The time and space requirements of the algorithm is near linear; previously known approaches take at least quadratic time.

1 Introduction

The *sequence nearest neighbor (SNN) problem* is as follows. We are given a database D of sequences for preprocessing; given an on-line query sequence Q, our goal is to return a sequence S in D whose distance $d(S,Q)$ to Q is no more than that of any other sequence T to Q. Distance between two sequences is defined to be the minimum number of edit operations needed to transform one to another (all edit operations of interest are reversible so that $d(S,T) = d(T,S)$ for any two sequences T and S). The nature of SNN problem depends on the edit operation that is permitted which corresponds to the notion of similarity between sequences that we wish to capture for an application. Natural edit operations include character edits (inserts, replacements, deletes etc), block edits (moves, copies, deletes, reversals) and block numerical transformations (scaling by an additive or a multiplicative constant). The SNN problem arises in many applications and it is a "grand problem" of Combinatorial Pattern Matching.

Let the number of sequences in D be n and the maximum length of the sequences in D be ℓ. Under nearly any set of edit operations, all known algorithms for the SNN problem face the "dimensionality bottleneck" in the worst case, that is, they cannot use subexponential ($o(2^\ell)$) preprocessing cost and yet obtain query time bounds better than that obtained by comparing the query sequence to each in the database (say, sublinear – $o(n\ell)$ – time). Overcoming this dimensionality bottleneck under nontrivial edit operations is an important open issue in Combinatorial Pattern Matching.

*AT& T Labs – Research, 180 Park Avenue, Florham Park, NJ 07932; email:muthu@research.att.com.

†Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106; email:cenk@eecs.cwru.edu.

STOC 2000 Portland Oregon USA

In this paper, we make some limited progress towards solving this problem – we present the first known efficient algorithm for "approximate" nearest neighbor search with preprocessing time and space polynomial in size of D and query time near-linear in size of Q. Our result holds for the block edit distance $d(S,T)$ between two sequences S and T defined to be the minimum number of character edits and block operations needed to transform one to the other. Another contribution of the paper is an algorithm to exactly compute the distance between two sequences under a block operation, namely, reversals. We now describe our results in more detail.

Context. Significant progress has been made on the nearest neighbor problem (NN) recently; however, most of those results are for vector spaces.[1] That is, the objects are vectors and the distance between two vectors is defined by a norm (such as Euclidean, or L_∞, etc) of their difference. We refer to these as *vector nearest neighbor* (VNN) problems. While a sequence of length ℓ may be thought of as a ℓ dimensional vector, SNN problems offer distinct challenges. Edit operations allow nontrivial alignments between two sequences (position i in one sequence may align with position $j \neq i$ in the other for distance computation) while the distance functions for vectors allow only the trivial alignment of ith position in one vector with that in the other. Nontrivial alignments pose major technical difficulties. In presence of such alignments, even computing the distance between two sequences may be expensive (quadratic or even NP-hard, depending on the edit operations being considered); in contrast, the vector norms can be computed in time linear in the number of dimensions.

Distance between two sequences. The distance between two sequences is the minimum number of edit operations needed to transform one to another. An important technical condition is that the operations must be non-overlapping, i.e., no character can be involved in more than one edit operation. Some edit operations are as follows:

1. *Character edits* which includes inserting or deleting a single character or replacing a single character by another. These are motivated by transcription errors in text, bit transmission errors in communication channels, or experimental errors in reading a genomic sequence, etc. These are of classical interest, having been studied at least as early as 60's [Lev66].

2. *Block arrangements* which involves moving a block (any consecutive set of characters) from one place to another; this is a rather natural notion in defining similarity of objects (such as in moving a paragraph of a text to another location [T84] or moving objects around in pen computing [LT96] or intrasequence rearrangements in genomic data [GD91]). It also involves copying blocks from one place to another within a sequence, or deleting a copy of a block that exists elsewhere. These operations are motivated by data compression.

3. *Block reversals* which involves reversing an entire block; this is a common operation in multimedia data such as music scales [SK83], or in assembling genomic sequences [G].

4. *Block linear-transformation* which involves changing each position $Q[i]$ in some block of Q to $\phi Q[i]+\kappa$ where ϕ and κ are multiplicative and additive scaling constants respectively. This is of interest in financial data analysis where one allows scaling effects in tracking similar trends [AL+95].

Although these edit operations are motivated from many applications, we do not consider the specifics of any application. Rather, we focus on the complexity of nearest neighbor search while allowing various edit operations above.

Our Contributions. Our first set of results are Monte Carlo type algorithms for *approximately* solving the SNN problem. Say $S_1, \ldots, S_n$ is the database D of sequences, $\ell = \max_i |S_i|$ and $|D| = \sum_i |S_i|$. Our algorithms use preprocessing time and space polynomial in $|D|$ and ℓ. A query with sequence Q takes time $O(|Q|\, polylog(n\ell))$ and returns a sequence S in D such that $d(S,Q) = O(\log \ell (\log^* \ell)^2) d(T,Q)$ for any other sequence T in D, with high probability (at least $1-1/n$).[2] Here, $d(S,Q)$ is what we call the *block edit distance* defined to be the minimum number of character edits (insertions, deletions and replacements) and block operations (moves, copies, deletes, reversals) needed to transform S to Q (or vice versa). In contrast, for VNN problems, $(1+\epsilon)$ [KOR98, IM98] or $O(\log\log\ell)$ [I98] approximations are known. However, ours is the first result for the SNN problem with any non-trivial edit operation (if only character replacements are allowed in which case the alignment is trivial, the known results for VNN problem under the Hamming distance apply to the SNN problem).

Our second result is on the basic problem of computing the distance between two sequences with block

[1] An exception is the recent result in [FI99] for the Hausdorff measure which is not related to SNN problems we study.

[2] Given an integer k, $\log^* k$ denotes, $\min\{i \geq 0 : \log^{(i)} k \leq 1\}$, where $\log^{(i)} k = \log(\log^{(i-1)} k)$, and $\log^{(0)} x = 0$.

edit operations. If the distance $d(S,T)$ between two sequences S and T is the minimum number of arbitrary block moves needed to transform S to T, the problem of computing $d(S,T)$ become NP-hard [LT96]. Thus, we focus on block reversals only. Since it is not possible to transform a given sequence S (e.g., all 0's) to every other sequence T (e.g., all 1's) by block reversals alone, $d(S,T)$ is not well defined if we allow only block reversal operations. The simplest well-defined distance with block reversals additionally allows character replacements. In this paper, we consider this distance between two sequences, and present an $O(|S|\ polylog(|S|))$ time deterministic algorithm for exactly computing it; the naive solution employs dynamic programming and it takes time $\Theta(|S|^2)$.

Technical Overview. Algorithms for VNN problems work by dimensionality reduction, space partitioning, bucketing etc., none of which seems to work directly with distance functions involving nontrivial alignments such as ours. Our approach for SNN problems transforms sequences into sparse vectors in very large dimensions. The transformation is inspired by recent results on the communication complexity of exchanging documents [CP+00] which used the idea of parsing sequences into "core" subsequences as per [SV96] (relying in turn on the "symmetry breaking" technique from [CV86]). We develop more general parsing strategies here which capture properties of cores involving not only equality of subsequences, but also equality under reversals.

Map. We describe the notation in Section 2 and our general parsing strategy for sequences in Section 3. In Section 4 we show how to solve the SNN problem approximately using our parsing. In Section 5, we show how to compute the distance between two sequences under edit operations specified received. In Section 6 we discuss open problems.

2 Preliminaries

In the rest of the paper we denote sequences by $P, Q, R, S..$, integers and binary numbers by $i, j, k..$ and constants by $\alpha, \beta, \gamma...$ All sequences have characters drawn from alphabet σ. Given a sequence S, let $S[i]$ denote its ith character and $S[i:j]$ denote the block extending from $S[i]$ to $S[j]$. We denote by $S||Q$ the concatenation of two sequences S and Q. The reverse of S is the sequence $S[|S|], \ldots, S[2], S[1]$, and is denoted by $S^{\leftarrow}$. A sequence in which all characters are identical is called a *repeating sequence*. We denote by a^l, a repeating sequence composed of l repetitions of character a. Similarly, for a given sequence Q, we denote by Q^l, a sequence obtained by concatenating l copies of Q. Here are a few standard definitions. A sequence S is called *periodic* with $l > 1$ repetitions if there exists a block Q such that $S = Q^l||Q^-$ where Q^- is a prefix of Q; Q is called a period of S. The shortest such Q is called *the* period of S; the period of S is the period of all periods of S. A sequence S in which $S[i] \neq S[i+1]$ for all i is called a *varying sequence*. Given sequence S, a *max-repeating* block of S is a repeating sequence of at least two characters which can not be further extended as a repeating sequence. Following the removal of all max-repeating blocks of S, we get a number of sequences which are called the *remaining* blocks of S. We remind readers again that in defining the edit distance between two sequences, the edit operations applied to one sequence to transform it to the other can not overlap.

3 Generalized Parsing

In this section, we partition any given sequence into blocks (called cores) using a parsing technique that generalizes the Locally Consistent Parsing (LCP) technique of Sahinalp and Vishkin [SV96]. Many of the properties of LCP will also hold for our generalized parsing, but our parsing will have additional properties not only in terms of the equality of cores, but also with their reversals. We call the resultant parsing *Symmetric LCP* and obtain *Symmetric Cores*.

3.1 Symmetric LCP

Given a varying sequence S (i.e., $S[i] \neq S[i+1]$ for $i = 1 \ldots |S|$), symmetric LCP designates at most $|S|/2$ and at least $|S|/3$ (possibly overlapping) blocks of S as *core-blocks*. The concatenation of these core-blocks (after removing their overlaps) gives S with the exception of a constant number of characters on its left and right margins.

Step 1. Mark the leftmost character of S and every second (i.e., odd indexed) character to its right, till the $(\log^* |\sigma| + \alpha)$th character for some constant α to be determined later. Similarly mark the rightmost character of S and every second character to its left up to the $(\log^* |\sigma| + \alpha)$ character.

Step 2. For each character x between the leftmost and the rightmost marked characters compute a *tag*, as follows. Let v be the left neighbor of x and y be the right neighbor. Initially assign $tag_0(x) = x$. Compute $tag_j(x)$ for $1 \leq j \leq \log^* |\sigma|$, in $\log^* |\sigma|$ iterations as follows.

Denote by l the index of the least significant bit in which $tag_{j+1}(x)$ is different from $tag_{j+1}(v)$, and r that between $tag_{j+1}(x)$ and $tag_{j+1}(y)$. Also let l' be the bit

value of $tag_{j+1}(x)$ at position l and likewise with r' (it is crucial that we consider the bit value of x). Define $L = l||l'$ and $R = r||r'$. Let $max(x) = \max\{R, L\}$ and $min(x) = \min\{R, L\}$ Assign

$$tag_{j+1}(x) = max(x)||min(x).$$

The tag obtained after the last iteration is simply denoted as *tag* henceforth.

Step 3. Mark each character whose tag is a local maxima, i.e., is greater than the tags of both its left and its right neighbors. Also, mark each character whose tag is a local minima, (i.e., is smaller than the tags of both its left and its right neighbors), and which has no neighbors that have previously been marked. The characters marked so far are called the *primary marked characters.*

Step 4. Mark characters between any two successive marked characters as equally spaced as possible so that no two successive marked characters are more than 3 apart. This marking is done in a symmetric manner walking between the two consecutive marked characters from both left and right simultaneously. The characters marked here are called the *secondary marked characters.* ∎

For each marked character $S[i]$, we designate the block $S[i-\log^* |\sigma|-\alpha : i+\log^* |\sigma|+\alpha]$ as a *core-block.*

Lemma 1 *For any j and i, $tag_j(S[i]) \neq tag_j(S[i+1])$.*

Proof: Consider any 4 consecutive characters $abcd$ where $b \neq c$. Let $L(b)$, $R(b)$, $L(c)$ and $R(c)$, be as defined earlier. If $min(b) = min(c)$ (otherwise, we are done), then the bit value at the least significant bit location implied by $min(b)$ is identical in both b and c. This means that $max(b)$ must be $R(b)||r(b)$ and $max(c)$ must be $L(c)||l(c)$ and hence $max(b) \neq max(c)$ since $R(b)$ and $L(c)$ must differ in the least significant bit. ∎

Therefore, the "sequence" of tag values after each iteration remains a varying one and iteration proceeds as usual. Also, at the end of the iterations, the following holds.

Lemma 2 *The maximum value of a tag is a small constant α which does not depend on S and σ.*

Proof: For any j, if the number of bits to represent $tag_j(x)$ is k, then the number of bits to represent $tag_{j+1}(x)$ is $2(\log k)+2$. After $\log^* |\sigma|$ iterations k will be a small constant (denoted by α henceforth) which can not be decreased further. ∎

Lemma 3 *If a block Q is designated as a core-block, then all blocks identical to Q or its reverse are designated as core-blocks.*

Proof: Consider the marked location that led to $Q[0 : |Q|-1]$ being a core-block, and let that be $Q[i]$. In the first iteration of the tag computation, $tag_1(Q[i])$, determined by character $Q[i]$, its left neighbor and its right neighbor. In each of the following iterations, $tag_j(Q[i])$ depends on one additional character to the left and one additional character to the right. Thus its *tag* is determined only by at most $\log^* |\sigma|$ characters to its left and $\log^* |\sigma|$ characters to its right. Consider any other block R that is a reversal of Q. Then $R[|Q|-i-1]$ has the same tag values as $Q[i]$ in each iteration until the end, and therefore R would be declared a core-block as well. (If R were identical to Q, $R[i]$ would play the role of $R[|Q|-i-1]$.) ∎

Following observations have easy proofs. (1) The number of core-blocks is at most $|S|/2$. (Proof follows from the fact that no two adjacent characters may be marked owing to the local maxima and local minima requirement in Step 4). (2) For any given sequence S, every character except possibly the rightmost $\log^* |\sigma| + \alpha$ and the leftmost $\log^* |\sigma| + \alpha$ characters is included in at least one core-block. (Proof follows since no two primary marked characters are separated by more than $\alpha + 1$, again due to Step 4).

Remark. One of the main difference between the standard LCP [SV96] and our symmetric LCP lies in Step 2 (there are other technical differences we will bring out explicitly in the final version). In standard LCP, tags are computed only from the left neighbor. In doing so, they adopted the symmetry-breaking technique of Cole and Vishkin [CV86]. Modifying it as we do to consider both the left and the right neighbor would mean that we exploit the symmetry, as we need to, in order to identify subsequences and their reversals by local tag computations. ∎

It is easy to see that for a sequence S of alphabet set σ, symmetric LCP can be executed in time $O(|S| \log^* |\sigma|)$.

3.2 Symmetric Core Computation

Given a sequence S, we show how to designate and label some of its blocks as *symmetric cores* in $O(\log |S|)$ iterations. Symmetric LCP is used repeatedly in this process. Essentially, we would like to use symmetric LCP recursively; however, there are repeating blocks of a sequence that complicate the issues. Recall the constant α from the previous section.

Initially, we designate each character of S as a core of level-0. In iteration i, we compute the level-i cores of

S by using cores of level-$(i-1)$; although some level-i cores overlap, no level-i core completely covers another. We denote by $S(0)$ the sequence S itself, and by $S(i)$ the sequence of level-i cores (in the order they appear in S). We also denote by $\sigma(0)$ the alphabet σ itself, and by $\sigma(i)$ the set of all core labels at level-i. The cores of level-i are computed as follows.

Replicative cores of level-i. We designate as a core-block every max-repeating block $R = a^r$ of $S(i-1)$ for which $r > \log^* |\sigma(i-1)| + \alpha$. The core of such a core-block is the concatenation of the r level-$(i-1)$ cores represented by label a in $S(i-1)$. We label this core with the three-tuple $(a,\ i-1,\ r)$.

We then consider each label x in $S(i-1)$ which represents a level-$(i-1)$ replicative core. Let the label of such a core C be the three-tuple (a, l, r) (where $l \leq i-2$ is the level entry and r the length entry). If $r \geq (2^{i-l}-1)(\log^* |\sigma(i-1)| + \alpha) + 2^{i-1}$, we designate this single label as singleton core-block and abuse our terminology to call it a max-repeating block. We identify its level-$(i-1)$ core as a level-i (replicative) core with label (a, l, r) once again.

Varying cores of level-i. Given a max-repeating or remaining block R for which $|R| \leq \log^* |\sigma(i-1)|+\alpha$, we mark every second label in a symmetric manner walking from left end and right end simultaneously. In each left-over block Q for which $|Q| > \log^* |\sigma(i-1)| + \alpha$, we mark the label implied by symmetric LCP from the previous section.

For each marked label x, let y be the leftmost label to the right of x which is designated as a singleton core-block. Similarly, let z be the rightmost label to the left of x which is designated as a singleton core-block. Also, let k denote the number of labels between x and y and l denote the number of labels between x and z.

We append the leftmost $\min(k+1, \log^* |\sigma(i-1)| + \alpha)$ labels to the right of x, and the rightmost $\min(l+1, \log^* |\sigma(i-1)|+\alpha)$ labels to the left of x. Such a block is called the *core-block* of label x. However, we avoid designating a core-block for x if to the left (respectively, right) of x, there are fewer than $\log^* |\sigma(i-1)|+\alpha$ labels and no singleton core-block.

For each core-block obtained, we compute its core C as follows: (1) If the leftmost label is a singleton core-block R and its level entry is m, then C includes the rightmost $2^{i-m} \times (k+1-\log^* |\sigma(m)| - \alpha)$ level-m cores of R and the cores of the labels in the core-block of x. (2) If the leftmost label is not singleton core-block we simply concatenate all the cores in the core-block of x.

Finally we consider each label x which represents a level-$(i-1)$ replicative core, but is not designated as a singleton core-block. We treat x as if it is a max-repeating block of $S(i-1)$ of size $|C|/2^{i-l-1}$, where $|C|$ is the length entry of x. Therefore, we obtain $|C|/2^{i-l}$ core-blocks that include x.

We label each varying level-i core by an integer in a way that: (i) distinct cores get distinct labels (ii) two cores which are reversals of each other have identical "primary labels", but also have a bit flag to distinguish one from the other. We use primary labels of level-i for LCP (i.e., for computing the tags of level-$(i+1)$) so that reverses of sequences are treated identically during the computation of cores. However we concatenate the primary level-i labels their bit flags (and distinguish cores from their reversals) when labeling level-$(i+1)$ cores. ∎

Recall that the set of cores at level-i is denoted by $\sigma(i)$; the following can be deduced.

Lemma 4 *We have* $|S|/(3^i - 2(\log^* |\sigma(i)| + \alpha)) \leq |S(i)| \leq |S|/2^i$.

The following property can be derived using Lemma 3.

Lemma 5 *Let R be a level-i core and let Q be a block identical to R or its reverse. If R is a non-replicative core, then Q is a non-replicative core of level-i. If R is a replicative core, then there exists a replicative core U which covers Q and has the same level and (primary) label entries as R.*

Careful accounting for repeating cores and using the geometric decreasing size of varying sequences will let us conclude that,

Theorem 6 *Given a sequence S, its symmetric cores can be computed in time $O(|S| \log^* |S|)$. The number of symmetric cores is $O(|S|)$.*

4 Approximate nearest neighbors

We only focus on binary sequences; extension to larger alphabets is trivial. Let $d(X, Y)$ be the edit distance between two sequences X and Y under the following operations: (i) character edits, in the form of inserting, deleting or replacing a character, and (ii) block edits, in the form of moving or copying a block to another location, reversing a block or deleting a block which has another copy in the string. No two operations are allowed to overlap. Notice that $d(X, Y)$ provides a metric as each of the edit operations described above have a corresponding operation which reverses it.

Definition 1 *Given a sequence S, its* level-i binary histogram, *denoted $T_i(S)$, is the binary vector of size $O((2(\log^* |S| + \alpha + 1))^i)$, whose jth entry, denoted $T_i(S)[j]$, has value 1 if the binary representation of j is a level-i-core of S. If j is not a core of S then $T_i(S)[j] = 0$. The* edit transformation of S *is the concatenation of all $T_j(S)$ for $j = 0, \ldots, \log n$, and we denote it by $T(S)$.*

The edit transformation of S lists all possible cores of all levels of S, a $O(2^{|S|})$ dimensional vector of flags each of which is set to 1 if the corresponding core is present in S. In what follows, we will not use this explicit representation but rather just a list of cores (as pointers into the sequence) and the number of their occurrences which is an $O(|S|)$ representation. Under this alternative representation, it takes $O(|S| \log^* |S|)$ time to generate this transform.

Theorem 7 *$\Omega(d(X,Y))/\log^*(|X| + |Y|) = ||(T(X), T(Y)||_1 = O(d(X,Y) \log(|X| + |Y|) \log^*(|X| + |Y|))$ where $||U, V||_1 = \sum_i |U[i] - V[i]|$.*

Proof.

(1) $||T(X), T(Y)||_1 = O(\log(|X| + |Y|) \cdot \log^*(|X| + |Y|) \cdot d(X,Y))$. Let $\mathcal{S}_\mathcal{X}$ denote the set of all distinct cores of X.

We make two observations both of which requires a proof that we have omitted here. Each character edit operation on X, i.e., insertion, deletion and replacement of any single character, can add or remove at most $\alpha + 1 + \log^*(|\sigma(i)|)$ cores from $\mathcal{S}_\mathcal{X}$, for $0 \leq i \leq \log(|X| + |Y|)$. Each block edit operation, i.e. changing the location of a block, reversing it, copying it to another location, or deleting it provided that another copy exists in X, can add or remove at most $2(\log^*(|\sigma(i)|) + \alpha + 1)$ cores from $\mathcal{S}_\mathcal{X}$ for $0 \leq i \leq \log(|X| + |Y|)$. Thus a sequence of $d(X,Y)$ edit operations applied on X to obtain Y can change $O(d(X,Y) \cdot \log(|X| + |Y|) \cdot \log^*(|X| + |Y|))$ entries of $T(X)$.

(2) $\Omega(d(X,Y)) = ||T(X), T(Y)||_1 \cdot \log^*(|X| + |Y|)$. We describe a procedure that obtains Y by applying at most $O(||T(X), T(Y)||_1) \log^*(|X|+|Y|)$ operations on X. We will describe a procedure to obtain $X||Y$ by applying $O(||T(X), T(Y)||_1) \log^*(|X| + |Y|)$ block or character copy operations to X (or Y). Because each edit operation is reversible, this implies that we can obtain Y from $X||Y$ by $O(||T(X), T(Y)||_1) \log^*(|X| + |Y|)$ block or character delete operations.

We now show how to obtain $X||Y$ from X doing $O(||T(X), T(Y)||_1 \cdot \log^*(|X| + |Y|)$ iterations. Let $Y_0(X)$ denote the empty string. In a given iteration i, we copy a substring of X or add a single character from σ to the right end of $Y_{i-1}(X)$ to obtain $Y_i(X)$. In the final iteration $Y_i(X)$ will be identical to Y, which we concatenate with X to obtain $X||Y$.

Iteration i works as follows. Consider the cores of Y that are also present in X or $Y_{i-1}(X)$ whose starting positions are within $Y_{i-1}(X)$ in Y. Among them let C be the core whose ending position is the rightmost one (if more than one such core exists, pick up the one which is of the highest level). In this case we copy to the end of $Y_{i-1}(X)$ the suffix of C which starts in Y at the right end of $Y_{i-1}(X)$.

We claim that each core of Y which does not exist in X can result in the execution of $O(\log^*(|X| + |Y|))$ iterations described above. Observe that at the end of iteration $i - 1$, no core C' in Y that includes core C can exist in X or $Y_{i-1}(X)$; otherwise C' would be used instead of C. Charge the copying operation of the suffix of C to the rightmost core C'' in Y which includes C and is one level above it. The total number of cores C that can charge to a given core C'' of Y which does not exist in X is $O(\log^*(|X| + |Y|))$; hence the claim follows. ∎

Remark. A somewhat different transformation was used in [CP+00] for what is called LZ-distances. A result similar to above is proven there to obtain an $O(\log^2 \ell)$ bound. Our bound is tighter, our distance function is allowed to be more general, and both upper and lower bound arguments are different from [CP+00].

SNN data structure. In order to solve the SNN problem, it now suffices to solve the VNN problem under L_1 distance with query $T(Q)$ and database of vectors $T(X)$ for each sequence X in D. For simplicity, assume that all sequences have length ℓ; it is a simple detail otherwise. Recall that each $T(.)$ is an $O(2^\ell)$ vector in which there are at most $O(\ell)$ nonzero entries. Furthermore, we can not only deduce that each entry is an integer at most ℓ, but also that the sum of all entires in a vector is at most ℓ. This VNN problem is equivalent to a VNN problem on the Hamming metric by replacing each position by ℓ long unary encoding to obtain binary vectors. We can solve this equivalent "sparse" VNN problem efficiently to an approximation as follows.

We sketch our first approach. (There are many technical details which we have omitted here, so this is just a sketch.) The approach is to take $O(\log \ell)$ samples of sizes $2, 2^2, \ldots$ respectively. Each sample picks each location in the binary vector uniformly, randomly, with appropriate probability. All sampled positions are XORed to get a $O(\log \ell)$ sized signature. We do this for each vector $T(X)$ in D and store it in a trie. For any

query vector $T(Q)$, we find its signature and retrieve the longest matching prefix of a signature in the database using the trie. This gives us a *candidate*. We repeat this procedure with $O(\log n)$ different signatures. We finally compare $T(Q)$ with $T(X)$ for each candidate X, and output the one with smallest L_1 distance with $T(Q)$. We can now prove that with high probability (at least $1 - 1/n$), the sequence thus determined is an $1+\epsilon$ approximation to the VNN of $T(Q)$ and hence an $O(\log \ell \log^* \ell)$ approximation to the SNN using previous theorem. There is an important technical detail to be understood. We need to ensure that the sampling is done independently for each location in the vectors so that it can be accomplished in time proportional to the *number of nonzeros* rather than the *length* of the vectors. We will show in the final version that this procedure works (with care, especially when query processing).

The result described thus far constructs a data structure in which most queries are successful (i.e., they find the approximate SNN). We can get a stronger result – that if the algorithm succeeds in constructing the data structure, it is good for every query – by modifying the data structure of [KOR98] so that we need to only sample from non-zero items. With this modification (details omitted), the result is

Theorem 8 *There is an algorithm for the SNN problem that takes $O((n\ell)^{O(1)})$ time and space. Every query runs in time $O(\ell \ polylog(n\ell))$.*

5 Sequence comparison with block reversals and character replacements

Let X and Y be two size ℓ sequences. In the rest of the section we will denote by $d(X,Y)$ the minimum number of character replacements and block reversals to convert X into Y; since edit operations do not overlap, no reversal is allowed on blocks wherein there is a character replacement. Observe that $d(X,Y)$ is the "simplest" nontrivial distance measure between sequences allowing block operations, hence it is a natural measure. One can use standard dynamic programing together with block labeling to compute $d(X,Y)$ in $O(\ell^2)$ time. We know of no algorithm in the literature that has a better running time. Below we describe an algorithm for computing $d(X,Y)$ in time $O(\ell \log^3 \ell)$. The procedure relies on combinatorial properties of reversals in sequences.

Definition 2 *The substring $X[i:j]$ is said to be reversible (with respect to Y) if $X[i:j] = Y[i:j]^{\leftarrow}$.*

The algorithm. We compute $d(X,Y)$ in ℓ iterations as follows. In iteration i, we compute $d(i) = d(X[1:i], Y[1:i])$. Let $X[i_1 : i]$ be the longest suffix of $X[1:i]$ which is reversible and let p_1 the length of $X[i_1 : i]$'s period. For $h \geq 1$, we inductively define $\#_h = \lceil (i - i_h)/p_h \rceil - 1$, and $i_{h+1} = i_h + \#_h \cdot p_h$. We also define p_{h+1} to be the period of $X[i_{h+1} : i]$ and define H the minimum h such that $\#_h = 0$; notice that $H \leq \log i$.

The algorithm computes $d(i)$ as follows. For $1 \leq h \leq H$, we set $d_h(i) = d(i_h - 1) + 1$ if $\#_h \leq 1$ and set $d_h(i) = d_h(i - p_h)$ if $\#_h > 1$. We also set $d_0(i) = d(i-1) + d(X[i], Y[i])$. Then we compute $d(i) = \min\{d_h(i)\}$ which completes the description of the algorithm. ∎

We prove the correctness of the algorithm through the following lemmas.

Lemma 9 *If $X[j:i]$ is reversible and p is the length of the period of $X[j:i]$, then for any $k \leq i - p$, the substring $X[k:i]$ is reversible if and only if $k = p \cdot h$ for $0 \leq h \leq \lceil (k - j + 1)/p \rceil - 1$.*

Proof. If $X[j:i]$ is reversible, then $X[j:i] = Y[j:i]^{\leftarrow}$, and hence $X[j+k] = Y[i-k]$ for all $k \leq j - i + 1$. Because p is the length of the period of $X[j:i]$, $X[j+k] = X[j+p \cdot h+k] = Y[i-k]$ for all $k < i-j-p \cdot h+1$, which implies $X[j + p \cdot h : i] = Y[j + p \cdot h : i]^{\leftarrow}$; therefore $X[j + p \cdot h : i]$ is reversible. If $X[j + k : i]$ is reversible, then $X[j + h] = Y[i - h] = X[j + k + h]$ for all $h \leq i - j - k$. Therefore k must be the length of a period of $X[i - p - 1 : i]$ and hence of $X[j:i]$, which implies that k must be a multiple of p. ∎

The statements below follow.

Corollary 10 *The only reversible suffixes of $X[1:i]$ are of the form $X[i_h + p_h j : i]$ for all $0 \leq j \leq \#_h$ and $1 \leq h \leq H$. If $X[j:i]$ is reversible and p is the length of the period of $X[j:i]$, then for any $k \geq j + p$, the substring $X[j:k]$ is reversible if and only if $k = p \cdot h$ for $0 \leq h \leq \lceil (k - i + 1)/p \rceil - 1$. Let $X[j:i]$ be a reversible string whose period is of length p and let $X[h : i - kp]$ be a substring of $X[j:i]$ such that $h + p < i - kp$. If $X[h : i - kp]$ is reversible then so is $X[h:i]$.*

Now the proof of the lemma below follows, immediately giving the correctness of the entire algorithm.

Lemma 11 *Consider any i and define $p_0 = i - i_1$ where i_1 is as defined in the algorithm. For $1 \leq h \leq H$, $d_h(i)$ is equal to the distance between $X[1:i]$ and $Y[1:i]$ provided a suffix of length k for which $p_h \geq k > p_{h+1}$ is reversed.*

It remains for us to determine the running time of the algorithm. We first focus on computing i_1 for all

$1 \leq i \leq \ell$; this will take time $O(\ell)$. The algorithm for computing i_1 for all $1 \leq i \leq \ell$ runs in three steps.

1. First we set $X' = X[1], \$, X[2], \$, \ldots, \$, X[\ell]$ and $Y' = Y[1], \$, Y[2], \$ \ldots, \$, Y[\ell]$ where \$ is a special character which is not in the original alphabet. Notice that for $X[i:j]$ is reversible with respect to Y if and only if $X'[2i-1 : 2j-1]$ is reversible with respect to Y'.

2. For all $1 \leq j \leq 2\ell - 1$ we compute the largest k for which $X'[j-k : j+k]$ is reversible with respect to Y'; let this be k_j. This can be done by finding the longest common prefix between $X'[j : 2\ell-1]$ and $Y'[1:j]^{\leftarrow}$, the longest common suffix of $X'[1 : j]$ and $Y'[j : 2\ell-1]^{\leftarrow}$, and picking the shortest of the two. For finding either of the two longest common items, it suffices to build a suffix tree of X and $Y'^{\leftarrow}$ which takes $O(\ell)$ time and to preprocess it to answer the lowest common ancestor queries which takes $O(1)$ time per query after $O(\ell)$ preprocessing.

3. For $i = 1, \ldots, 2\ell-1$ we set i_1 to the smallest j for which $j + k_j \geq i$. Such a j can be computed for all i by pushing each j, k_j pair to a queue while going through all j from 1 to $2\ell - 1$.

Next we focus on computing the period of any substring $X[j,i]$. We will show that one can preprocess X in $O(\ell \log^2 \ell)$ time to be able to compute the period of substring $X[j : i]$ in $O(\log^2(i-j))$ time. As the preprocessing step we simply give labels to all substrings of X of size 2^k $0 \leq k \leq \lfloor \log \ell \rfloor$ such that each distinct substring gets a distinct label. This can be done in $O(\ell \log \ell)$ time using [KMR72]. As the second preprocessing step, we sort all substrings that are given the same label according to their starting location in X. This can be done using stable sorting in time $O(\ell \log^2 \ell)$ for all labels of all lengths. Using these labels one can compute the period of a substring S of size $i-j$ through the following observation. If a substring S of X has a period of length p_S then p_X, the length of the period of X, satisfies $p_S \geq p_X$ or $p_X \geq |S|$. Therefore if two identical substrings $X[j - 2^{\lfloor \log i \rfloor - h} : k]$ and $X[k - 2^{\lfloor \log i \rfloor - h} : k']$ (which have identical labels) overlap, then the period of X can not be within the range $\{2^{\lfloor \log i \rfloor - h - 1} + 1, \ldots, 2^{\lfloor \log i \rfloor - h}\}$.

Thus we can check for all $\alpha = \lfloor \log i - j \rfloor - 1, \ldots, 1, 0$ whether S has a period of length in the range $2^\alpha, \ldots 2^{\alpha-1} + 1$ by finding the largest $k < i - 2^{\alpha-1}$ for which $X[k - 2^\alpha : k] = X[i - 2^\alpha : i]$. This is easily done in $O(\log \ell)$ time by binary search on the sorted list for substrings with label identical to that of $X[i - 2^\alpha : i]$. Each query for the period of a substring therefore takes time $O(\log^2 \ell)$ in the worst case.

This combined with the iteration for computing $d(i)$'s gives,

Theorem 12 *Given two sequences X and Y of length ℓ, there exists an $O(\ell \log^3 \ell)$ time algorithm to compute $d(X,Y)$ if only character replacements and block reversals are allowed.*

Remark. Our original version of this algorithm used symmetric LCP; here, we provide a simpler algorithm without using symmetric LCP. By a more sophisticated algorithm, we can improve the running time to $O(\ell \log^2 \ell)$.

6 Discussion

We can extend our results to take into account linear transform of blocks as well. Consider LCP parsing in Section 3.1. In order to take care of block linear-transforms in addition to block reversals, we replace each symbol $S[i]$ as follows. Say $\frac{S[i]-S[i-1]}{S[i-1]-S[i-2]} = x/y$ when common factors are removed. We use $x||y$ in place of $S[i]$ at each iteration of symmetric LCP. We call this *scaled symmetric LCP*. The core-blocks we obtain will have the property that *if there is a block R that is a linear transform of Q for* any *constant κ and ϕ, and Q is a core-block, then R will also be a core-block.* Other properties of symmetric LCP and cores can be proved for scaled symmetric LCP and cores with appropriate modifications. If we wanted to allow neither block reversals or transforms, we use the vanilla LCP and cores from [SV96]. If we wanted to allow only linear transforms and no block reversals, we use vanilla LCP using the character transformation above. Full technical description will be given in the final version of this paper.

The basic problem still remains open: can we solve the SNN problem without any block operations while allowing only character edits? Our LCP based transformation does not seem to provide any insight into the edit distance between two sequences when block operations are disallowed: while the upper bound in Theorem 7 still holds, the lower bound is no longer valid. Hence, a new approach may have to be developed. Another interesting direction is to design efficient algorithms for simple sequence comparison problems with other block operations [LT96].

References

[AL+95] R. Agarwal, K. Lin, H. Sawhney and K. Shim. Fast similarity search in the presence of noise,

scaling and translation in time-series databases. *Proc. 21st VLDB conf*, 1995.

[CL90] W. Chang and E. Lawler, Approximate String Matching in Sublinear Expected Time, Proceedings of *IEEE Symposium on Foundations of Computer Science*, (1990).

[CP+00] G. Cormode, M. Paterson, S. C. Sahinalp and U. Vishkin. Communication Complexity of Document Exchange. *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 2000.

[CV86] R. Cole and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades, Micro and Macro Techniques for Designing Parallel Algorithms, Proceedings of *ACM Symposium on Theory of Computing*, (1986).

[FI99] M. Farach-Colton and P. Indyk. Approximate Nearest Neighbor Algorithms for Hausdorff Metrics via Embeddings. *Proc. IEEE Symp. Foundations of Computer Science*, 1999.

[G] *GelAssemble*. `http:..staffa.wi.mit.edu/gcg/gelassemble.html`.

[GD91] M. Gribskov and J. Devereux *Sequence Analysis Primer*, Stockton Press, 1991.

[I98] P. Indyk. On Approximate Nearest Neighbors in Non-Euclidean Spaces. *Proc IEEE Symp on Foundations of Computer Science*, 1998, 148–155.

[IM98] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Remving the Curse of Dimensionality. *Proc. ACM Symp. on Theory of Computing*, 1998, 604–613.

[J97] J. Kleinberg. Two Algorithms for Nearest Neighbor Search in High Dimensions. *Proc. ACM Symp. on Theory of Computing*, 1997, 599–608.

[KMR72] R. Karp, R. Miller and A. Rosenberg, Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays, Proceedings of *ACM Symposium on Theory of Computing*, (1972).

[KOR98] E. Kushilevitz, R. Ostrovsky and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proc. ACM Symposium on Theory of Computing*, 1998, 614–623.

[LT96] D. Lopresti and A. Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 1996.

[SK83] D. Sankoff and J. Kruskal, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, *Addison-Wesley*, Reading, Mass., 1983.

[LV89] G. Landau and U. Vishkin, Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms*, 10, (1989):158-169.

[Lev66] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Cybernetics and Control Theory*, 10(8):707-710, 1966.

[SV96] S. C. Sahinalp and U. Vishkin, Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm, Proceedings of *IEEE Symposium on Foundations of Computer Science*, (1996).

[Se80] P. Sellers, The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, 1, (1980):359-373.

[T84] W. F. Tichy, The string-to-string correction problem with block moves. *ACM Trans. on Computer Systems*, 2(4): 309-321, 1984.

[Uk83] E. Ukkonen, On Approximate String Matching. Proceedings of *Conference on Foundations of Computation Theory*, (1983).