DIMACS Technical Report 2001-26
November 2001

The String Edit Distance Matching Problem with Moves

by

Graham Cormode[1]
Dept of Computer Science
University of Warwick
Coventry CV4 7AL, UK
grahamc@dcs.warwick.ac.uk

S. Muthukrishnan[2]
AT&T Research
Florham Park
New Jersey 07932 USA
muthu@research.att.com

[1]DIMACS Visitor
[2]Permanent Member

ABSTRACT

The edit distance between two strings S and R is defined to be the minimum number of character inserts, deletes and changes needed to convert R to S. Given a text string t of length n, and a pattern string p of length m, informally, the string edit distance matching problem is to compute the smallest edit distance between p and substrings of t. A well known dynamic programming algorithm takes time $O(nm)$ to solve this problem, and it is an important open problem in Combinatorial Pattern Matching to significantly improve this bound.

We relax the problem so that (a) we allow an additional operation, namely, *substring moves*, and (b) we approximate the string edit distance upto a factor of $O(\log n \log^* n)$.[1] Our result is a near linear time deterministic algorithm for this version of the problem. This is the first known significantly subquadratic algorithm for a string edit distance problem in which the distance involves nontrivial alignments. Our results are obtained by embedding strings into L_1 vector space using a simplified parsing technique we call *Edit Sensitive Parsing* (ESP). This embedding is approximately distance preserving, and we show many applications of this embedding to string proximity problems including nearest neighbors, outliers, and streaming computations with strings.

[1] $\log^* n$ is the number of times log function is applied to n to produce a constant.

1 Introduction

String matching has a long history in computer science, dating back to the first compilers in the sixties and before. Text comparison now appears in all areas of the discipline, from compression and pattern matching to computational biology and web searching. The basic notion of string similarity used in such comparisons is that of edit distance between pairs of strings. If R and S are strings then the *edit distance* between R and S, $e(R, S)$, is defined as the minimum number of character insertions, deletions or changes necessary to turn R into S. This distance measure is motivated from a number of scenarios: insertions and deletions on lossy communication channels [Lev66], typing errors in documents [CR94], evolutionary changes in biological sequences [SK83], etc.

In the string edit distance matching problem studied in Combinatorial Pattern Matching area, we are given a text string t of length n and a pattern string p of length $m < n$. The *string edit distance matching problem* is to compute the minimum string edit distance between p and any prefix of $t[i \dots n]$ for each i; we denote this distance by $D[i]$. It is well known that this problem can be solved in $O(mn)$ time using dynamic programming [Gus97]. The open problem is whether this quadratic bound can be improved substantially in the worst case.

There has been some progress on this open problem. Masek and Paterson [MP80] used the Four Russians method to improve the bound to $O(mn/\log n)$, which remains the best known bound in general to this date. Progress since then has been obtained by relaxing the problem in a number of ways.

- Restrict $D[i]$'s of interest.
  Specifically, the restricted goal is to only determine i's for which $D[i] < k$ for a given parameter k. By again adapting the dynamic programming approach a solution can be found in $O(kn)$ time and space in this case [LV86, Mye86]. An exciting improvement was presented in [ȘV96] (since improved by [CH98]) with an $O(n\,\text{poly}(k)/m + n)$ time algorithm which is significantly better. These algorithms still have running time of $\Omega(nm)$ in the worst case.
- Consider simpler string distances.
  If we restrict the string distances to exclude insertions and deletions, we obtain Hamming distance measure. In a simple yet significant development, [Abr87] gave a $\tilde{O}(n\sqrt{m})$ time solution breaking the quadratic bound[2]; since then, it has been improved to $\tilde{O}(n\sqrt{k})$ [ALP00]. Karloff improved this to $\tilde{O}(n)$ by approximating the Hamming distances to $1+\epsilon$ factor [Kar93]. Hamming distance results however sidestep the fundamental difficulty in string edit distance problem, namely, the need to consider nontrivial *alignment* of the pattern against text when characters are inserted or deleted.

In this paper, we present a near linear time algorithm for the string edit distance matching problem. However, there is a caveat: our result relies strongly on relaxing the problem as described below. Since our result is the first to consider nontrivial alignment between the

[2]The notation $\tilde{O}$ hides polylog factors.

text and the pattern and still obtain significantly subquadratic algorithm in the worst case, we believe it is of interest. Specifically, we relax the string edit distance matching problem in two ways:
(1) We allow approximating $D[i]$'s.
(2) We allow an extra operation, *substring move*, which moves a substring from one position in a string to another.

This modified edit distance between strings S and R is called string edit distance with moves and is denoted $d(S, R)$. There are many applications where substring moves are taken as a primitive: in certain computation biology settings a large subsequence being moved is just as likely as an insertion or deletion; in a text processing environment, moving a large block intact may be considered a similar level rearrangement to inserting or deleting characters. Note that string edit distance with moves still faces the challenge of dealing with nontrivial alignments. Formally, then, $d(S, R)$ is length of the shortest sequence of edit operations to transform S into R, where the permitted operations affect a string S are defined as follows:

- A character *deletion* at position i transforms S into $S[1] \dots S[i-1], S[i+1] \dots S[n]$.
- An *insertion* of character a at position i results in $S[1] \dots S[i-1], a, S[i] \dots S[n]$.
- A *replacement* at position i with character a gives $S[1] \dots S[i-1], a, S[i+1] \dots S[n]$.
- A *substring move* with parameters $1 \leq i \leq j \leq k \leq n$ transforms $S[1] \dots S[n]$ into $S[1] \dots S[i-1], S[j] \dots S[k-1], S[i] \dots S[j-1], S[k] \dots S[n]$.

Note that there is no restriction on the interaction of edit operations so, for example, it is quite possible for a substring move to take a substring to a new location and then for a subsequent move to operate on a substring which overlaps the moved substring and its surrounding characters.

Our main result is a deterministic algorithm for the string edit distance matching problem *with moves* which runs in time $O(n \log n)$. The output is the matching array D where each $D[i]$ is approximated to within a $O(\log n \log^* n)$ factor. Our approach relies on an embedding of strings into vectors in the L_1 space. The L_1 distance between two such vectors is an $O(\log n \log^* n)$ approximation of the string edit distance with moves between the two original strings. This is a general approach, and can be used to solve many other questions of interest beyond the core string edit distance matching problem. These include string similarity search problems such as indexing for string nearest neighbors, outliers, clustering etc. under the metric of edit distance with moves.

All of our results rely at the core on a few components. First, we parse strings into a hierarchy of substrings. This relies on deterministic coin tossing (aka local symmetry breaking) that is a well known technique in parallel algorithms [CV86, GPS87, AM91] with applications to string algorithms [ŞV94, ŞV96, MSU97, CPŞV00, ABR00, MŞ00]. In its application to string matching, precise methods for obtaining hierarchical substrings differ from one application instance to another, and are fairly sophisticated: in some cases, they produce non-trees, in other cases trees of degree 4 etc. Inspired by these techniques we present a simple hierarchical parsing procedure called *Edit Sensitive Parsing* (ESP) that produces a tree of degree 3. ESP should not be perceived to be a novel parsing technique; however, it is an attempt to simplify the technical description of applying deterministic coin tossing to obtain hierarchical decomposition of strings. We hope that simplicity of ESP helps reveal further applications of hierarchical string decompositions.

The second component of our work is the approximately distance preserving embedding of strings into vector spaces based on the hierarchical parsing. This general style of solution was taken earlier in [CPṢV00, MṢ00]. However, the key difference between our work and previous work is that we embed into L_1 space (in contrast, these prior works embedded into the Hamming space), allowing us to approximate edit distance with moves and obtain a variety of string proximity results based on this distance for the first time (the Hamming embeddings cannot be used to obtain our result).

Previously attempts to relax string edit distance involved additional substring operations such as copying and deleting substrings at unit cost. Although this appears to be a further relaxation of our distance measure, a crucial lemma we use to solve the string edit distance problem with substring moves (Lemma 3.1) no longer holds under this distance, and so the string edit distance problem under this metric is open. Our algorithm for the string edit distance problem with moves remains the only significantly subquadratic algorithm known for non-trivial alignments.

Layout. We present our Edit Sensitive Parsing (ESP) and show embedding of strings into the L_1 space in Section 2. We go on to solve problems of approximate string distance and approximate pattern alignment in Section 3. In Section 4 we give our results for other related problems based around approximating the string distance, and concluding remarks are in Section 5.

2 String Embedding

In this section, we describe how to embed strings into a vector space so that $d()$, the string edit distance with substring moves, will be approximated by vector distances. Consider any string S over an alphabet set σ. We will embed it as $V(S)$, a vector with an exponential number of dimensions, $O(|\sigma|^{O(|S|)})$; however, the number of dimensions in which the vector is nonzero will be quite small, in fact, $O(|S|)$. This embedding V will be computable in near linear time, and it will have the approximation preserving property we seek.

At the high level, our approach will *parse* S into special substrings, and consider the multiset $T(S)$ of all such substrings. We will ensure that the size of $T(S)$ is at most $2|S|$. Then, $V(S)$ will be the "characteristic" vector for the multiset $T(S)$, and will be defined precisely later.

The technical crux is the parsing of S into its special substrings to generate $T(S)$. We call this procedure as *Edit Sensitive Parsing*, or ESP for short. In what follows, we will first describe ESP, and then describe our vector embedding and prove its approximation preserving properties.

i	text	c	a	b	a	g	e	h	e	a	d	b	a	g
ii	in binary	010	000	001	000	110	100	111	100	000	011	001	000	110
iii	labels	-	010	001	000	011	010	001	000	100	001	010	000	011
iv	labels as integers	-	2	1	0	3	2	1	0	4	1	2	0	3
v	final labels	-	$\boxed{2}$	1	$\boxed{0}$	1	$\boxed{2}$	1	0	$\boxed{2}$	1	$\boxed{2}$	0	$\boxed{1}$

The original text, drawn from an alphabet of size 8 (i), is written out as binary integers (ii). Following one round of alphabet reduction, the new alphabet is size 6 (iii), and the new text is rewritten as integers (iv). A final stage of alphabet reduction brings the alphabet size to 3 (v) and local maxima and some local minima are used as landmarks (denoted by boxes)

Figure 1: The process of alphabet reduction and landmark finding

2.1 Edit Sensitive Parsing.

We will build a parse tree, called the *ESP tree* (denoted $ET(S)$), for string S: S will be parsed into hierarchical substrings corresponding to the nodes of $ET(S)$. The goal is that string edit operations only have a localized effect on the ET. An obvious parse tree will have all dyadic strings of length 2^i, that is, $S[k2^i \dots ((k+1)2^i - 1)]$ for all integers k and i; this will yield a fully binary parse tree. But if S is edited by a single character insertion or deletion to obtain S', S and S' will get parsed by this approach into two very different multisets of hierarchical substrings and the resulting embedding will no longer be approximation preserving.

Given a string S, we now show how to hierarchically build its ESP tree in $O(\log |S|)$ iterations. Each iteration generates a new level of the tree, where each level contains between a half and a third of the number of nodes in the level from which it is derived. At each iteration i, we start with a string S_i and *partition* it into *blocks* of length 2 or 3. We replace each such block $S_i[j \dots k]$ by its *name*, $h(S_i[j \dots k])$, where h is a one-to-one hash function on strings of length at most 3. Then S_{i+1} consists of the $h()$ values for the blocks in the order in which they appear in S_i. So $|S_{i+1}| \leq |S_i|/2$. We assume $S_0 = S$, and the iterations continue until we are left with a string of length 1. The ESP tree of S consists of levels such that there is a node at level i for each of the blocks of S_{i-1}; their children are the nodes in level $i-1$ that correspond to the symbols in the block. Each character of $S_0 = S$ is a leaf node. We also denote by σ_0 the alphabet σ itself, and the set of names in S_i as σ_i, the alphabet at level-i.

It remains for us to specify how to partition the string S_i at iteration i. This will be based on designating some local features as "landmarks" of the string. A landmark (say $S_i[j]$) has the property that if S_i is transformed into S_i' by an edit operation (say character insertion at k) far away from j i.e., $|k-j| >> 1$), our partitioning strategy will ensure that $S_i'[j]$ will still be designated a landmark. In other words, an edit operation on $S_i[k]$ will only affect j being a landmark if j were close to k. This will have the effect that each edit operation will only change $k^* = O(\max_j |k_j - j|)$ nodes of the ESP tree at every level, where k_j is the closest unaffected landmark to j. In order to inflict the minimal number of changes to the ESP tree, we would like k^* as small as possible, but still require S_i's to be geometrically decreasing in size.

In what follows, we will describe our method for marking landmarks and partitioning S_i into blocks more precisely. We canonically parse any string into maximal non-overlapping substrings of three types:

1. Maximal contiguous substrings of S_i that consist of a repeated symbol (so they are of the form a^l for $a \in \sigma_j$ where $l > 1$ and $j \leq i$),
2. "Long" substrings of length at least $\log^* |\sigma_{i-1}|$ not of type 1 above.
3. "Short" substrings of length less than $\log^* |\sigma_{i-1}|$ not of type 1.

Each such substring is called a *metablock*. We process each metablock as described below to generate the next level in the parsing.

2.1.1 Type 2: Long strings without repeats.

The discussion here is similar to those in [GPS87] and [MSU97]. Suppose we are given a string A in which no two adjacent symbols are identical and is counted as a metablock of type 2. We will carry out a procedure on it which will enable it to be parsed into nodes of two or three symbols.

Given a sequence S with no repeats (i.e., $S[i] \neq S[i+1]$ for $i = 1 \ldots |S| - 1$), we will designate at most $|S|/2$ and at least $|S|/3$ substrings of S as *nodes*. The concatenation of these nodes gives S. The first stage consists of iterating an alphabet reduction technique known as "tagging". This is effectively the same procedure as the Deterministic Coin Tossing in [CV86], but applied to strings.

Alphabet reduction. For each symbol $A[i]$ compute a new label, as follows. $A[i-1]$ is the left neighbor of $A[i]$, and consider $A[i]$ and $A[i-1]$ represented as binary integers. Denote by l the index of the least significant bit in which $A[i]$ differs from $A[i-1]$, and let $bit(l, A[i])$ be the value of $A[i]$ at that bit location. Form $label(A[i])$ as $2l + bit(l, A[i])$ — in other words, as the index l followed by the value at that index.

Lemma 2.1 *For any i, if $A[i] \neq A[i+1]$ then $label(A[i]) \neq label(A[i+1])$.*

Proof. Suppose that the least significant bit position at which $A[i]$ differs from $A[i+1]$ is the same as that at which $A[i]$ differs from $A[i-1]$ (otherwise, $label(A[i]) \neq label(A[i+1])$). But, on our assumption that $A[i] \neq A[i+1]$, then the bit values at this location in each character must differ, and hence $label(A[i]) \neq label(A[i+1])$. ∎

Following this procedure, we generate a new sequence. If the original alphabet was size τ, then the new alphabet is sized $2 \log |\tau|$. We now iterate (note this iteration is orthogonal to the iteration that constructs the ESP tree of S; we are iterating on A which is a sequence with no identical adjacent symbols) and perform the alphabet reduction until the size of the alphabet no longer shrinks. This takes $\log^* |\tau|$ iterations. Note that there will be no labels for the first $\log^* |\tau|$ characters.

Lemma 2.2 *After the final iteration of alphabet reduction, the alphabet size is 6.*

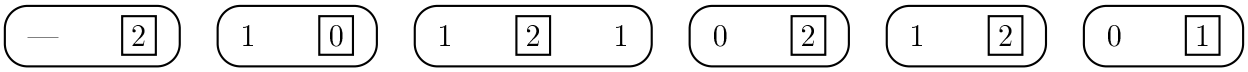

Figure 2: Given the landmark characters, the nodes are formed.

Proof. At each iteration of tagging, the alphabet size goes from $|\sigma|$ to $2\lceil \log |\sigma| \rceil$. If $|\sigma| > 6$, then $2 \log |\sigma|$ is strictly less than this quantity. ∎

Since A did not have identical adjacent symbols, neither does the final sequence of labels on A using Lemma 2.1 repeatedly.

Finally, we perform three passes over the sequence of symbols to reduce the alphabet from $\{0 \ldots 5\}$ to $\{0, 1, 2\}$: first we replace each 3 with the least element from $\{0, 1, 2\}$ that does not neighbor the 3, then do the same for each 4 and 5. This generates a sequence of labels drawn from the alphabet {0,1,2} where no adjacent characters are identical. Denote this sequence as A'.

Finding landmarks. We can now pick out special locations, known as *landmarks*, from this sequence that are sufficiently close together. We first select any position i which is a *local maximum*, that is, $A'[i-1] < A'[i] > A'[i+1]$, as a landmark. Two maxima could still have four intervening labels, so in addition we select as a landmark any i which is a local minimum that is, $A'[i-1] > A'[i] < A'[i+1]$, and is not adjacent to an already chosen landmark. An example of the whole process is given in Figure 1.

Lemma 2.3 *For any two landmark positions i and j, we have $2 \leq |i - j| \leq 3$.*

Proof. By our marking procedure, we insist that no adjacent pair of tags are marked – since we cannot have two adjacent maxima, and we specifically forbid marking local minima which are next to minima. Simple case analysis shows that the separation of landmark positions is at most two intervening symbols. ∎

Lemma 2.4 *Determining the closest landmark to position i depends on only $\log^* |\tau| + 5$ contiguous positions to the left and 5 to the right.*

Proof. After one iteration of alphabet reduction, each label depends only on the symbol to its left. We repeat this $\log^* |\tau|$ times, hence the label at position i depends on $\log^* |\tau|$ symbols to its left. When we perform the final step of alphabet reduction from an alphabet of size six to one of size three, the final symbol at position i depends on at most three additional symbols to its left and to its right. We must mark any position that is a local maximum, and then any that is a local minimum not adjacent to a local maximum; hence we must examine at most two labels to the left of i and two labels to the right, which in turn each depend on $\log^* |\tau| + 3$ symbols to the left and 3 to the right. The total dependency is therefore as stated. ∎

Now we show how to partition A into blocks of length 2 or 3 around the landmarks. We treat the leftmost $\log^* |\sigma_{i-1}|$ symbols of the substring as if they were a short metablock (type 3, the procedure for which is described below). The other positions are treated as follows. We make each position part of the block generated by its closest landmark, breaking ties to the right (see Figure 2). Consequent of Lemma 2.3 each block is now of length two or three.

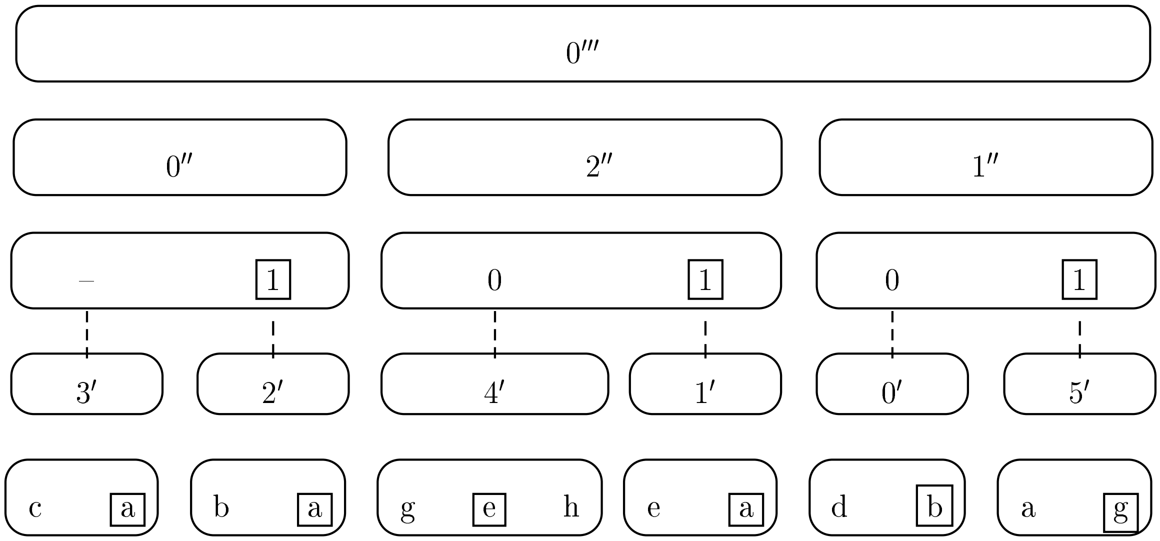

Figure 3: Following the generation of the first level nodes, we relabel from the alphabet $\{0', 1' \ldots\}$. We parse this string, generating three second level nodes which we relabel from $\{0'', 1'' \ldots\}$, which in turn become a single top level node.

2.1.2 Type 1 (Repeating metablocks) and Type 3 (Short metablocks)

Recall that we seek "landmarks" which can be identified easily based only on a local neighborhood. Then we can treat repeating metablocks as large landmarks. Type 1 and Type 3 blocks can each be parsed in a regular fashion, the details we give for completeness. Metablocks of length one would be attached to the repeating metablock to the left or the right, with preference to the left when both are possible, and parsed as described below. Metablocks of length two or three are retained as blocks without further partitioning, while a metablock of length four is divided into two blocks of length two. In any metablock of length five or more, we parse the leftmost three symbols as a block and iterate on the remainder.

2.1.3 Constructing $ET(S)$.

Having partitioned S_i into blocks of 2 or 3 symbols, we construct S_{i+1} by replacing each block b by $h(b)$ where h is a one-to-one (hash) function. Note that blocks of different levels can use different hash functions for computing names, so we focus on any given level i. If we use randomization, $h()$ can be computed for any block (recall they are of length 2 or 3) in $O(1)$ time using Karp-Rabin fingerprints [KR87]; they are one-to-one with high probability. For deterministic solutions, we can use the algorithm in [KMR72]. Using bucket sorting, hashing can be implemented in $O(1)$ time and linear space. Each block is a node in the

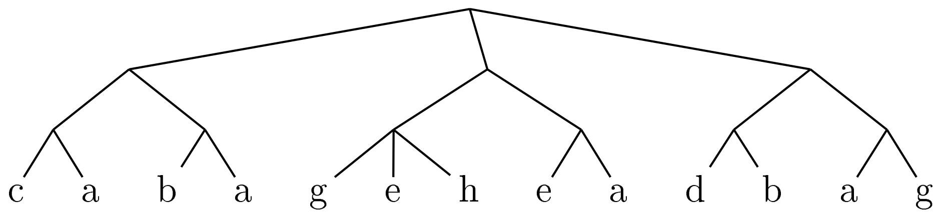

Figure 4: The hierarchical structure of nodes is represented as a parse tree on the string S. $T(S) = \{c, a, b, a, g, e, h, e, a, d, b, a, g, ca, ba, geh, ea, db, ag, caba, gehea, dbag, cabageheadbag\}$

parse tree, and its children are the 2 or 3 nodes from which it was formed.

This generates the sequence S_{i+1}; we then iterate this procedure until the sequence is of length 1: this is then the root of the tree. Let $N_i(S)$ be the number of nodes in $ET(S)$ at level i. Since the first (leaf) level is formed from the characters of the original string, $N_0(S) = |S|$. We have $N_i(S)/3 \leq N_{i+1}(S) \leq \lfloor N_i(S)/2 \rfloor$. Therefore, $\frac{3}{2}|S| \leq \sum_i N_i(S) \leq 2|S|$. Hence for any i, $|\sigma_i| \leq |S|$ (recall that $h()$ is one-to-one) and so $\log^* |\sigma_i| \leq \log^* |S|$. The example is completed in figure 3.

Theorem 1 *Given a string S, its ESP tree $ET(S)$ can be computed in time $O(|S| \log^* |S|)$.*

2.1.4 Properties of ESP.

We can compute $ET(S)$ for any string S as described above. (see Figure 4). Each node x in $ET(S)$ represents a substring of S given by the concatenation of the leaf nodes in the subtree rooted at x.

Definition 2.1 *Define the multiset $T(S)$ as all substrings of S that are represented by the nodes of $ET(S)$ (over all levels). We define $V(S)$ to be the "characteristic vector" of $T(S)$, that is, $V(S)[x]$ is the number of times a substring x appears in $T(S)$. Finally, we define $V_i(S)$ the characteristic vector restricted to only nodes which occur at level i in $ET(S)$.*

Note that $T(S)$ comprises at most $2|S|$ strings of length at most $|S|$. $V(S)$ is a $|\sigma|^{|S|}$ dimensional vector since its domain is any string that may be present in $T(S)$; however, it is a (highly) sparse vector since at most $2|S|$ components are nonzero.

We denote the standard L_1 distance between two vectors u and v by $||u - v||_1$. By defnition, $||V(S) - V(R)||_1 = \sum_{x \in T(S) \cup T(R)} |V(S)[x] - V(R)[x]|$. Recall that $d(R, S)$ denotes the edit distance with moves between strings R and S. Our main theorem shows that $V()$ is an approximation preserving embedding of string edit distance with moves.

Theorem 2 *For strings R and S, let n be* $\max(|R|, |S|)$. *Then*

$$d(R, S) \leq 2||V(R) - V(S)||_1 = O(\log n \log^* n) d(R, S)$$

2.2 Upper Bound Proof.

$$||V(R) - V(S)||_1 = O(\log n \cdot \log^* n) \cdot d(R,S)$$

Proof. To show this bound on the L_1 distance, we consider the effect of the editing operations, and demonstrate that each one causes a contribution to the L_1 distance that is bounded by $O(\log n \log^* n)$. Note again that, as mentioned in the introduction, edit operations are allowed to "overlap" on blocks that were previously the subject of moves etc. We give a Lemma which is similar to Lemma 2.4 but which applies to any string, not just those with no adjacent repeated characters.

Lemma 2.5 *The closest landmark to any symbol of S_i is determined by at most $\log^* |\sigma_i| + 5$ consecutive symbols of S_i to the left, and at most 5 consecutive symbols of S_i to the right.*

Proof. Given a symbol of S_i, say $S_i[j]$, we show how to find the closest landmark.
Type 1 Repeating metablock Recall that a long repeat of a symbol a is treated as a single, large landmark. $S_i[j]$ is included in such a meta-block if $S_i[j] = S_i[j+1]$ or if $S_i[j] = S_i[j-1]$. We also consider $S_i[j]$ to be part of a repeating substring if $S_i[j-1] = S_i[j-2]$; $S_i[j+1] = S_i[j+2]$; and $S_i[j] \neq S_i[j+1]$ and $S_i[j] \neq S_i[j-1]$ — this is the special case of a metablock of length one. In total, only 2 consecutive symbols to the left and right need to be examined.
Types 2 and 3 Non-repeating metablocks If it is determined that $S_i[j]$ is not part of a repeating metablock, then we have to decide whether it is in a short or long metablock. We examine the substring $S_i[j - \log^* |\sigma_i| - 3 \ldots j-1]$. If there is any k such that $S_i[k] = S_i[k-1]$ then there is a repeating metablock terminating at position k. This is a landmark, and so we parse $S_i[j]$ as part of a short metablock, starting from $S[k+1]$ (recall that the first $\log^* |\sigma_i|$ symbols of a long metablock get parsed as if they were in a short metablock). Examining the substring $S[j+1 \ldots j+5]$ allows us to determine if there is another repeating metablock this close to position j, and hence we can determine what node to form containing $S_i[j]$. If there is no repeating metablock evident in $S_i[j - \log^* |\sigma_i| - 3 \ldots j-1]$ then it is possible to apply the alphabet reduction technique to find a landmark. From Lemma 2.4, we know that this can be done by examining $\log^* |\sigma_i| + 5$ consecutive symbols to the left and 5 to the right. ∎

This ability to find the nearest landmark to a symbol by examining only a bounded number of consecutive neighboring symbols means that if an editing operation occurs outside of this region, the same landmark will be found, and so the same node will be formed containing that symbol. This allows us to prove the following lemma.

Lemma 2.6 *Inserting $k \leq \log^* n + 10$ consecutive characters into S to get S' means $||V_i(S) - V_i(S')||_1 \leq 2(\log^* n + 10)$ for all levels i.*

Proof. We shall make use of Lemma 2.5 to show this. We have a contribution to the L_1 distance from the insertion itself, plus its effect on the surrounding locality. Consider the total number of symbols at level i that are parsed into different nodes after the insertion compared to the nodes beforehand. Let the number of symbols at level i which are parsed differently as a consequence of the insertion be M_i. Lemma 2.5 means that in a non-repeating metablock, any symbol more than 5 positions to the left, or $\log^* |\sigma_i| + 5$ positions to the right of any symbols which have changed, will find the same closest landmark as it did before, and so will be formed into the same node. Therefore it will not contribute to M_i. Similarly, for a repeating metablock, any symbol inside the block will be parsed into the same node (that is, into a triple of that symbol), except for the last 4 symbols, which depend on the length of the block. So for a repeating metablock, $M_i \leq 4$. The number of symbols from the level below which are parsed differently into nodes as a consequence of the insertion is at most $M_{i-1}/2$, and there is a region of at most 5 symbols to the left and $\log^* |\sigma_i| + 5$ symbols to the right which will be parsed differently at level i. Because $|\sigma_i| \leq |S| \leq n$ as previously observed, we can therefore form the recurrence, $M_i \leq M_{i-1}/2 + \log^* n + 10$. If $M_{i-1} \leq 2(\log^* n + 10)$ then $M_i \leq 2(\log^* n) + 10$. From the insertion itself, $M_0 \leq \log^* n + 10$. Finally $||V_i(S) - V_i(S')||_1 \leq 2(M_{i-1}/2)$, since we could lose $M_{i-1}/2$ old nodes, and gain this many new nodes. ∎

Lemma 2.7 *Deleting* $k < \log^* n + 10$ *consecutive symbols from* S *to get* S' *means* $||V_i(S) - V_i(S')||_1 \leq 2(\log^* n + 10)$.

Proof. Observe that a deletion of a sequence of labels is precisely the dual to an insertion of that sequence at the same location. If we imagine that a sequence of characters is inserted, then deleted, the resultant string is identical to the original string. Therefore, the number of affected nodes must be bounded by the same amount as for an insertion, as described in Lemma 2.6. ∎

We combine these two lemmas to show that editing operations have only a bounded effect on the parse tree.

Lemma 2.8 *If a single permitted edit operation transforms a string* S *into* S' *then* $||V(S) - V(S')||_1 \leq 8 \log n(\log^* n + 10)$.

Proof. We consider each allowable operation in turn.
Character edit operations. The case for insertion follows immediately from Lemma 2.6 since the effect of the character insertion affects the parsing of at most $2(\log^* n + 10)$ symbols at each level and there are at most $\log_2 n$ levels. In total then $||V(S) - V(S')||_1 \leq 2 \log n(\log^* n + 10)$. Similarly, the case for deletion follows immediately from Lemma 2.7. Finally, the case for a replacement is shown by noting that a character replacement can be considered to be a deletion immediately adjacent to an insertion.
Substring Moves. If the substring being moved is at most $\log^* n + 10$ in length, then a move can be thought of as a deletion of the substring followed by its re-insertion elsewhere. From Lemma 2.6 and Lemma 2.7, then $||V(S) - V(S')||_1 \leq 4 \log n(\log^* n + 10)$. Otherwise, we consider the parsing of the substring using ESP. Consider a character in a non-repeating metablock which is more than $\log^* n + 5$ characters from the start of the substring and more than 5 characters from the end. Then according to Lemma 2.5, only characters within the substring being moved determine how that character is parsed. Hence the parsing of all such characters, and so the contribution to $V(S)$, is independent of the location of this

substring in the string. Only the first $\log^* n + 5$ and last 5 characters of the substring will affect the parsing of the string. We can treat these as the deletion of two substrings of length $k \leq \log^* n + 10$ and their re-insertion elsewhere. For a repeating metablock, if this extends to the boundary of the substring being moved then still only 4 symbols of the block can be parsed into different nodes. So by appealing to Lemmas 2.6 and 2.7 then $||V(S) - V(S')|| \leq 8 \log n(\log^* n + 10)$. ∎

Lemma 2.8 shows that each allowable operation affects the L_1 distance of a transform by at most $8 \log n(\log^* n + 10)$. Suppose we begin with R, and perform a series of d editing operations, generating $R_1, R_2, \ldots R_d$. At the conclusion, $R_d = S$, so $||V(R_d) - V(S)||_1 = 0$. We begin with a quantity $||V(R) - V(S)||_1$, and we also know that at each step from the above argument that $||V(R_j) - V(R_{j+1})|| \leq 8 \log n(\log^* n + 10)$. Hence, if $d(R, S)$ is the minimum number of operations to transform R into S, then $||V(R) - V(S)||_1 / 8 \log n(\log^* n + 10)$ must be at least $d(R, S)$, giving a bound of $d(R, S) \cdot 8 \log n(\log^* n + 10)$. ∎

2.3 Lower Bound Proof.

$$d(R, S) \leq 2||V(R) - V(S)||_1$$

Here, we shall prove a slightly more general statement, since we do not need to take account of any of the special properties of the parsing; instead, we need only assume that the parse structure built on the strings has bounded degree (in this case three), and forms a tree whose leaves are the characters of the string. Our technique is to show a particular way we can use the 'credit' from $||V(R) - V(S)||_1$ to transform R into S. We give a constructive proof, although the computational efficiency of the construction is not important. For the purpose of this proof, we treat the parse trees as if they were static tree structures, so following an editing operation, we do not need to consider the effect this has on the parse structure.

Lemma 2.9 *If trees which represent the transforms have degree at most k, then the tree $ET(S)$ can be made from the tree $ET(R)$ using no more than $(k-1)||V(S) - V(R)||_1$ move, insert and delete operations.*

Proof. We first ensure that any good features of R are preserved. In a top-down, left to right pass over the tree of R, we 'protect' certain nodes — we place a mark on any node x that occurs in the parse tree of both R and S, provided that the total number of nodes marked as protected does not exceed $V_i(S)[x]$. If a node is protected, then all its descendents become protected. The number of marked copies of any node x is $\min(V(R)[x], V(S)[x])$. Once this has been done, the actual editing commences, with the restriction that we do not allow any edit operation to split a protected node.

We shall proceed bottom-up in $\log n$ rounds ensuring that after round i when we have created R_i that $||V_i(S) - V_i(R_i)||_1 = 0$. The base case to create R_0 deals with individual symbols, and is trivial: for any symbol a, if $V_0(R)[a] > V_0(S)[a]$ then we delete the $(V_0(R)[a] - V_0(S)[a])$ unmarked copies of a from R; else if $V_0(R)[a] < V_0(S)[a]$ then at the end of R we insert $(V_0(S)[a] - V_0(R)[a])$ copies of a. In each case we perform exactly $|V_0(R)[a] - V_0(S)[a]|$

operations, which is the contribution to $||V_0(R) - V_0(S)||_1$ from symbol a. R_0 then has the property that $||V_0(R_0) - V_0(S)||_1 = 0$.

Each subsequent case follows an inductive argument: assuming we have enough nodes of level $i-1$ (so $||V_{i-1}(S) - V_{i-1}(R_{i-1})||_1 = 0$), we show how to make R_i using just $(k-1)||V_i(S) - V_i(R)||_1$ move operations. Consider each node x at level i in the tree $ET(S)$. If $V_i(R)[x] \geq V_i(S)[x]$, then we would have protected $V_i(S)[x]$ copies of x and not altered these. The remaining copies of x will be split to form other nodes. Else $V_i(S)[x] > V_i(R)[x]$ and we would have protected $V_i(R)[x]$ copies of x. Hence we need to build $V_i(S)[x] - V_i(R)[x]$ new copies of x, and the contribution from x to $||V_i(S) - V_i(R)||_1$ is exactly $V_i(S)[x] - V_i(R)[x]$: this gives us the credit to build each copy of x. To make each of the copies of x, we need to bring together at most k nodes from level $i-1$. So pick one of these, and move the other $k-1$ into place around it (note that we can move any node from level $i-1$ so long as its *parent* is not protected). We do not care *where* the node is made – this will be taken care of at higher levels. Because $||V_{i-1}(S) - V_{i-1}(R_{i-1})||_1$ we know that there are enough nodes from level $i-1$ to build every level i node in S. We then require at most $k-1$ move operations to form each copy of x by moving unprotected nodes. ∎

Since this inductive argument holds, and we use at most $k-1=2$ moves for each contribution to the L_1 distance, the claim follows.

3 Solving the String Edit Distance Matching Problem

In this section, we present an algorithm to solve the string edit distance problem with moves. For any string S, we will assume that $V(S)$ can be stored in $O(|S|)$ space by listing only the nonzero components of $|S|$. More formally, we store $V(S)[x]$ if it is nonzero in a table indexed by $h(x)$, and we store x as a pointer into S together with $|x|$.

The result below on pairwise string comparison follows immediately from Theorems 1 and 2 together with the observation that given $V(R)$ and $V(S)$, $||V(R) - V(S)||_1$ can be found in $O(|R| + |S|)$ time.

Theorem 3 *Given strings S and R with $n = \max(|S|, |R|)$, there exists a deterministic algorithm to approximate $d(R, S)$ accurate upto $O(\log n \log^* n)$ factor in $O(n \log^* n)$ time with $O(n)$ space.*

3.1 Pruning Lemma.

In order to go on to solve the string edit distance problem, we need to "compare" pattern p of length m against $t[i \ldots n]$ for each i, and there are $O(n)$ such "comparisons" to be made. Further, we need to compute the distance between p and $t[i \ldots k]$ for all possible $k \geq i$ in order to compute the best alignment starting at position i, which presents $O(mn)$ subproblems in general. The classical dynamic programming algorithm performs all these comparisons in a total of $O(mn)$ time in the worst case by using the dependence amongst the subproblems. Our algorithm will take a different approach. First, we make the following crucial observation:

Lemma 3.1 (Pruning Lemma) *Given a pattern p and text t, $\forall l, r : l \le r \le n$,*

$$d(p, t[l \dots l+m-1]) \le 2\, d(p, t[l \dots r]).$$

Proof. Observe that for all r in the lemma, $d(p, t[l \dots r]) \ge |(r-l+1) - m|$ since this many characters must be inserted or deleted. Using triangle inequality of edit distance with moves, we have for all r, $d(p, t[l \dots l+m-1])$

$$\begin{aligned} &\le d(p, t[l \dots r]) + d(t[l \dots r],\ t[l \dots l+m-1) \\ &= d(p, t[l \dots r]) + |(r-l+1) \ - \ m| \\ &\le 2d(p, t[l \dots r]) \end{aligned}$$

which follows by considering the longest common prefix of $t[l \dots r]$ and $t[l \dots l+m-1]$. ∎

The significance of the Pruning Lemma is that it suffices to approximate only $O(n)$ distances, namely, $d(p, t[l \dots l+m-1)$ for all l, in order to solve the string edit distance problem with moves, correct upto factor 2 approximation.[3] Hence, it prunes candidates away from the "quadratic" number of distance computations that a straightforward procedure would entail.

Still, we cannot directly apply Theorem 3 to compute $d(p, t[l \dots l+m-1])$ for all l, because that will be expensive. It will be desirable to use the answer for $d(p, t[l \dots l+m-1])$ to compute $d(p, t[l+1 \dots l+m])$ more efficiently. In what follows, we will give a more general procedure that will help compute $d(p, t[l \dots l+m-1])$ very fast for every l, by using further properties of ESP.

3.2 ESP subtrees.

Given a string S and its corresponding ESP tree, $ET(S)$, we show that the subtree of $ET(S)$ induced by the substring $S[l \dots r]$ has the same edit-sensitive properties as the whole tree.

Definition 3.1 *Let $ET_i(S)_j$ be the jth node in level i of the parsing of R.*
Define $range(ET_i(S)_j)$ as the set of values $[a \dots b]$ so that the leaf labels of the subtree rooted at $ET_i(S)_j$ correspond to the substring $S[a \dots b]$.
We define an ESP Subtree *of S, $EST(S, l, r)$ as the subtree of $ET(S)$ containing nodes which correspond to substrings which overlap or are contained in $S[l \dots r]$. Formally, we find all nodes of $ET_i(S)_j$ where $[l \dots r] \cap range(ET_i(S)_j) \ne \emptyset$. The name of a node derived from $ET_i(S)_j$ is $h(S[range(ET_i(S)_j) \cap [a \dots b]])$.*

[3] The Pruning Lemma also holds for the classical edit distance where substring moves are not allowed since we have only used the triangle inequality and unit cost to insert or delete characters in its proof; hence, it may be of independent interest. However, it does not hold when substrings may be copied or deleted, such as the "LZ distance" [CPṢV00].

This yields a proper subtree of $ET(R)$, since a node is included in the subtree if and only if at least one of its children is included (as the ranges of the children partition the range of the parent). As before, we can define a vector representation of this tree.

Definition 3.2 *Define $VS(S, l, r)$ as the characteristic vector of $EST(S)$ by analogy with $V(S)$, that is, $VS(S, l, r)[x]$ is the number of times the substring x is represented as a node in $EST(S, l, r)$.*

Note that $EST(S, 1, |S|) = ET(S)$, but in general it is not the case that $EST(S, l, r) = ET(S[l \dots r])$. However, $EST(S, l, r)$ shares the properties of the edit sensitive parsing. We can now state a theorem that is analogous to Theorem 2.

Theorem 4 *Let d be $d(R[l_p \dots r_p], S[l_q \dots r_q])$. Then*

$$d \leq 2||VS(R, l_p, r_p]) - VS(S, l_q, r_q])||_1 = O(\log n \log^* n)d$$

$$d \leq 2||VS(R, l_p, r_p]) - VS(S, l_q, r_q])||_1 = O(\log n \log^* n)d.$$

We need one final lemma before proceeding to build an algorithm to solve the String Edit Distance problem with moves.

Lemma 3.2 *$VS(S, l+1, r+1)$ can be computed from $VS(S, l, r))$ in time $O(\log |S|)$.*

Proof. Recall that a node is included in $EST(S, l, r)$ if and only if one of its children is. A leaf node corresponding to $S[i]$ is included if and only if $i \in [l \dots r]$. This gives a simple procedure for finding $EST(S, l+1, r+1)$ from $EST(S, l, r)$, and so for finding $VS(S, l+1, r+1)$: (1) At the left hand end, let x be the node corresponding to $S[l-1]$ in $EST(S, l, r)$. We must remove x from $EST(S, l, r)$. We must also adjust every ancestor of x to ensure that their name is correct, and remove any ancestors which do not contain $S[l-1]$. (2) At the right hand end let y be the node corresponding to $S[r]$ in $ET(S)$. We must add y to $EST(S, l, r)$, and set the parent of y to be its parent in $ET(S)$, adding any ancestor if it is not present. We then adjust every ancestor of y to ensure that their name is correct. Since in both cases we only consider ancestors of a leaf nodes, and the depth of the tree is $O(\log |S|)$, it follows that this procedure takes time $O(\log |S|)$. ∎

3.3 String Edit Distance Matching Algorithm.

Combining these results allows us to solve the main problem we study in this paper.

Theorem 5 *Given text t and pattern p, we can solve the string edit distance problem with moves, that is, compute an $O(\log n \log^* n)$ approximation to $D[i] = \min_{i \leq k \leq n} d(p, t[i \dots k])$ for each i, in time $O(n \log n)$.*

Proof. Our algorithm is as follows: given pattern p of length m and text t of length n, we compute $ET(p)$ and $ET(t)$ in time $O(n \log^* n)$ as per Theorem 1. We then compute $EST(t, 1, m)$. This can be carried out in time at worst $O(n)$ since we have to perform a pre-order traversal of $ET(t)$ to discover which nodes are in $EST(t, 1, m)$. From this we can compute $\hat{D}[1] = ||VS(t, 1, m) - VS(p, 1, m)||_1$. We then iteratively compute $||VS(t, i, i + m - 1) - VS(p, 1, m)||_1$ from $||VS(t, i - 1, i + m - 2) - VS(p, 1, m)||_1$ by using Lemma 3.2 to find which nodes to add or remove to $EST(t, i, i + m - 1)$ and adjusting the count of the difference appropriately. This takes n comparisons, each of which takes $O(\log n)$ time. By Theorem 4 and Lemma 3.1, $D[i] \leq \hat{D}[i] \leq O(\log n \log^* n) D[i]$. ∎

If $\log^* n$ is $O(\log m)$ as one would expect for any reasonable sized pattern and text, then a tighter analysis can show the running time to be $O(n \log m)$. This is because we only need to consider the lower $\log m$ levels of the parse trees; above this $EST(t, i, i + m - 1)$ has only a single node in each level.

4 Applications and Extensions of Our Techniques

Our embedding of strings into vector spaces in an approximately distance preserving manner has many other applications as such, and with extensions. In this section, we will describe some of the important results we obtain. In contrast to our previous results, which have all been deterministic, many of these applications make use of randomized techniques.

4.1 String Indexing.

A fundamental open problem in string matching is that of approximate string indexing (Open problem 10 of [Gal85]). Specifically, we are given a collection C of strings that may be preprocessed. Given a query string q, in the nearest neighbors problem, the goal is to find the string $c \in C$ closest to q under string edit distance, that is, $\forall x \in C. d(q, c) \leq d(q, x)$. In this paper, we focus on edit distance with moves, and let d denote this function. The approximate version of the nearest neighbors problem is to find $c \in C$ such that $\forall x \in C. d(q, c) \leq f \cdot d(q, x)$ where f is the factor of approximation. Let $k = |C|$ be the number of strings in the collection C, and n the length of the longest string in the collection. The challenge here is to break the "curse" of high-dimensionality, and provide schemes with polynomial preprocessing whose query cost is $o(kn)$. That is, which takes less time to respond to queries than it takes to examine the whole collection. For vectors, randomized answers to these questions have been given in [IM98] and [KOR98]. By modifying the scheme in [IM98] for our large but sparse vectors $V(S)$, we obtain (in the spirit of [MṢ00]):

Theorem 6 *With $O(kn \log n)$ preprocessing of a collection C, approximate nearest neighbors queries can be answered in time $O(n \log k + k^{1/2} \log n)$ finding a string from C that is an $O(\log n \log^* n)$ approximation of the nearest neighbor with constant probability.*

4.2 String Outliers.

A problem of interest in string data mining is to find "outlier" strings, that is, those that differ substantially from the rest. More formally, we are given a set D of k strings each of length at most n that may be preprocessed. Given a query string q, the goal is to find a string $s \in D$ such that $d(q, s) \geq \epsilon n$, for some constant fraction ϵ. We can strengthen our analysis of the embedding to show an improved result for this problem.

Lemma 4.1 *For strings R and S, let $n = \max(|R|, |S|)$. Then*

$$d(R, S) \leq 2||V(R) - V(S)||_1 = O(\log(n/d(R, S)) \log^* n) d(R, S)$$

Proof. We improve the upper bound by observing that in the top levels of the tree, there are only a limited number of nodes, so although these might change many times during a sequence of editing operations we have the bound $||V_i(S) - V_i(R)||_1 \leq N_i(S) + N_i(R)$. A worst case argument says that the greatest number of nodes that could be affected is when one level in the tree is completely altered. The size of this level is $N_i(S) + N_i(R) = 8d(\log^* n + 10)$, and the number of nodes above it in the tree (which may all be affected) is $\sum_{j \geq i} N_i(S) + N_i(R) \leq 16d(\log^* n + 10)$. Below this, we may assume that the worst case is when each edit operation contributes $8(\log^* n + 10)$ to $||V_j(S) - V_j(R)||$ for $j < i$. Thus $||V(S) - V(R)||_1 \leq d(\log n - \log(d \log^* n))8(\log^* n + 10) + 16d(\log^* n + 10) = O(d \log^* n \log(n/d \log^* n)) = O(d \log(n/d) \log^* n)$. ∎

We note that since the approximation depends on $\log n/d$, the quality of the approximation actually increases the less alike the strings are. This improved approximation helps in the outliers problem. To solve this problem, we adapt the methodology of [GIV01] for solving approximate furthest neighbors problems with constant probability and constant approximation factor. By analogy with Approximate Nearest Neighbors, approximate furthest neighbors asks us to find $c \in C$ satisfying $\forall x \in C. d(q, c) \geq f \ d(q, z)$

Theorem 7 *We preprocess a set C of strings in time $O(kn \,\text{poly-log}(kn))$. For a query q, we either return an approximate outlier s or a null value. If returned, s will satisfy the property that $d(q, s) \geq \epsilon n/O(\log^* n)$ with constant probability and if t is an outlier for q in C, then $d(q, s) \geq d(q, t)/O(\log^* n)$; hence it is a $O(\log^* n)$ approximation. This requires time $O(n \log k + \log^3 k)$ per query. If no outlier is reported, then there is no outlier for q.*

4.3 Sketches in the Streaming model.

We consider the embedding of a string S into a vector space as before, but now suppose S is truly massive, too large to be contained in main memory. Instead, the string arrives as a stream of characters in order: $(s_1, s_2 \dots s_n)$. This is the *data streams* model [HRR98], and we must perform computations with $o(n)$ space, preferably polylogarithmic space, without backtracking on the data. We are able to perform our embedding in this restricted model, the first such result known in string matching, as shown below. The result of our computations is a sketch vector for the string S. The idea of sketches is that they can be used as much smaller surrogates for the actual objects in distance computations.

Theorem 8 *A sketch $sk(V(S))$ can be computed in the streaming model to allow approximation of the string edit distance with moves using $O(\log n \log^* n)$ space. For a combining function f, then $|f(sk(V(R)), sk(V(S))) - d(R,S)| \leq O(\log n \log^* n) d(R,S)$ with probability $1-\delta$. Each sketch is a vector of length $O(\log 1/\delta)$ that can be manipulated in time linear in its size. Sketch creation takes total time $O(n \log^* n \log 1/\delta)$.*

Proof. It is precisely the properties of ESP that ensures edit operations have local effect on the parse structure that also allow us to process the stream with very little space requirements. Since Lemma 2.5 tells us that the parsing of any symbol at any level S_j depends only on at most $O(\log^* n)$ other symbols, we only need to have these symbols in memory to make the parsing. This is true at every level in the parsing: only $O(\log^* n)$ nodes at each level need to be held in memory to make the parsing. When we group nodes of level i together to make a node of level $i+1$ we can conceptually "pass up" this new node to the next level in the process. Each node corresponds to addition of one to an entry in $V(S)$. We cannot store all the entries of the vector $V(S)$ without using linear space. Instead, we can store a short summary of $V(S)$ which can be used as a surrogate for distance computations. Existing techniques can be adapted to achieve this, in particular the streaming techniques described in Theorem 2 of [Ind00]. For vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, we can compute *sketches*, $sk(\boldsymbol{x})$ and $sk(\boldsymbol{y})$ so with probability $1-\delta$ for a combining function f:

$$|f(sk(\boldsymbol{x}), sk(\boldsymbol{y})) - ||\boldsymbol{x} - \boldsymbol{y}||_1| \leq \epsilon ||\boldsymbol{x} - \boldsymbol{y}||_1$$

Such methods are necessarily approximate and randomized. The requirement for a naming function $h()$ is solved by using Karp-Rabin signatures for the substrings [KR87]. These have the useful property that the signature for a long substring can be computed from the signatures of its two or three component substrings. Thus, we can compute entries of $V(S)$ and so build a sketch of $V(S)$ using poly-logarithmic space. Since $V(S)$ can be used to approximate the string edit distance with moves up to a $O(\log n \log^* n)$ factor, it follows that these sketches achieve the same order of approximation. Overall, the total working space needed to create the parsing is $\log n$ levels each keeping $\log^* n$ nodes, totalling $O(\log n \log^* n)$ space. ∎

This type of computation on the data stream is tremendously useful in the case where the string is too large to be stored in memory, and so is held on secondary storage, or is communicated over a network. Sketches allow rapid comparison of strings: hence they can be used in many situations to allow approximate comparisons to be carried out probabilistically in time $O(\log 1/\delta)$ instead of the $O(n)$ time necessary to even inspect both strings.

4.4 Other String Edit Distances.

The ESP approach can be adapted to handle similar string distance measures, which allow additional operations such as substring reversals, linear scaling and copying, amongst others.

We describe one particular distance, which we call the Compression distance because of its connection to lossless text compression methods. This distance permits substring moves, character edits, copying an arbitrary substring, and uncopying a substring[4]. It was considered previously in [CPṢV00, MṢ00], where it is defined formally. We denote it as $c(R, S)$.

Theorem 9 *For strings R and S with $|R| + |S| = n$,*

$$c(R, S) \leq 3|T(R) \;\Delta\; T(S)| = O(\log n \log^* n) \cdot c(R, S)$$

where $T(R)\Delta T(S)$ indicates the symmetric difference of the supporting sets of the multisets $T(R)$, $T(S)$.

Proof. We need to give a proof analogous to that for Theorem 2. Again, we show an upper and lower bound.

Upper bound. As in the proof of Theorem 2, we will show that each permitted edit operation has a limited effect. In particular, we show that if an edit operation transforms S into S', then $|T(S)\Delta T(S')| = O(\log n \log^* n)$. We already know that every move, insert, delete or change operation affects the L_1 distance by at most $O(\log n \log^* n)$. Since $|T(R)\Delta T(S)| \leq ||V(R) - V(S)||_1$ (ie the Hamming distance of two integer valued vectors is less than the L_1 distance), then we only need to consider the extra operations, of copying and uncopying a block.

For copy operations, we consider a copy of a (long) block of S generating S'. Note that any element within the block that is more than $\log^* |\sigma| + 5$ from the left end of the block, or 5 from the right end, will be oblivious of whatever goes on outside the block, and so will be parsed in the same way in both copies. Therefore, the copy operation will have an effect no greater than that of inserting of the first $\log^* |\sigma| + 5$ and last 5 elements, since we can neglect the centre of the block from after the first $\log^* |\sigma| + 5$ characters to 5 from the end. We have seen that such an insertion affects $O(\log n \log^* n)$ existing nodes, and it will add only $O(\log^* n)$ nodes for the inserted blocks themselves, so $|T(S)\Delta T(S')| = O(\log n \log^* n)$. For a block deletion, note that we can only delete a block that has a copy elsewhere in the sequence, so that deletion is the inverse operation of copying. For this case also, $|T(S)\Delta T(S')| = O(\log n \log^* n)$. Consequently, each operation can contribute $O(\log n \log^* n)$ to the Hamming distance, and so the total Hamming distance between $T(S)$ and $T(R)$ can be at most $c(R, S)O(\log n \log^* n)$.

Lower bound. To show $c(R, S) \leq 3|T(R) \;\Delta\; T(S)|$, we first show how given R we can build the compound string RS (R followed by S) using a number of operations linear in the difference between the transforms.

Lemma 4.2 *The compound string RS can be built frmo R using no more than $3|T(R)/T(S)|$ copy and insert operations.*

[4]For technical reasons, deleting a substring is permitted only if a copy of it exists in the remainder of the string.

Proof. We begin with the highest level node of $ET(S)$ (note that there will be precisely one, which represents the whole sequence S). We shall proceed inductively on the nodes of $ET(S)$. We assume that S is not a substring of R, and vice-versa,[5] so this node is not present in R, in which case it contributes to $T(S)/T(R)$. We will charge three units of credit to this element of the difference, and allocate these by giving one unit to each of its children (recall that each level-i node is formed from at most 3 level-$(i-1)$ nodes). The induction considers each node that has been given a credit. The base case is if a node under consideration is a leaf node. This character can be inserted at unit cost. Otherwise, the node represents a substring of S. If a child node is present in R, or already present in the partially built string S, then the unit credit can be used to copy the string that the node represents. Otherwise, we can apply this argument recursively (since this node is present in S but not in R and hence contributes to the set difference) to build this node. ∎

We can use exactly the same argument symmetrically to argue that starting from S we can build the compound string RS using $c(S, RS) \leq 3|T(S)/T(R)|$ operations. Since c is a metric, this means that $c(RS, S) \leq 3|T(S)/T(R)|$. Using the triangle inequality we have that $c(R, S) \leq c(R, RS) + c(RS, S) \leq 3|T(S)/T(R)| + 3|T(R)/T(S)| = 3|T(R) \ \Delta \ T(S)|$. ∎ This is currently the best approximation known for this string edit distance. [CPṢV00] showed an $O(\log^2 n \log^* n)$ approximation which was improved to $O(\log n (\log^* n)^2)$ in [MṢ00]; here, we improve it modestly to $O(\log n \log^* n)$. This approach also allows us to apply many of the applications in this paper to this compression distance as well as the transposition distance although we do not discuss this further.

4.5 Dynamic Indexing.

Thus far we have considered strings to be static immutable objects. The *Dynamic Indexing* problem is to maintain a data structure on a set of strings so that, under certain permitted editing operations on individual strings, we can rapidly compute the (approximate) distance between any pair of strings. A similar scenario was adopted in [MSU97] where the problem is to maintain strings under editing operations to rapidly answer equality queries: this is a special case of the general dynamic indexing problem we address. The technique was developed in [ABR00] for dynamic pattern matching: finding exact occurrences of one string in another.

Our situation is that we are given a collection of strings to preprocess. We are then given a sequence of requests to perform on the collection on-line. The requests are of the following types: (1) perform a string edit operation (inserts, deletes, changes, substring moves) on one of the strings (2) perform a split operation on one of the strings — split a string S into two new strings $S[1 \ldots i]$ and $S[i+1 \ldots |S|]$ for a parameter i. (3) perform a join operation on two strings — create a new string R from S_1 and S_2 as $S_1 S_2$. (4) return an approximation to $d(R, S)$ for any two strings R and S in the collection. We consider a set of strings whose total length is n. For simplicity, we assume here that the operations of split and join are non-persistent — their input strings are lost following the operation.

[5] This special case can be dealt with using a largely similar argument.

Theorem 10 *Following $O(n \log n \log^* n)$ preprocessing time using $O(n \log n)$ storage, approximating the distance between two strings from the collection takes $O(\log n)$ time. This gives an $O(\log n \log^* n)$ approximation with constant probability. Edit operations upon strings of type (1),(2) or (3) above take $O(\log^2 n \log^* n)$ time each.*

Proof. For each string S in the collection, we shall maintain $ET(S)$, and for each node in $ET(S)$ we shall store a sketch of size $O(\log n)$ corresponding to the characteristic vector of the ESP subtree rooted at that node, and a Karp-Rabin signature for the substring corresponding to the concatenation of the leaves of that subtree. We show that each update operation (insertion, deletion, replacement, move, split, join) can be performed efficiently to generate S' from S while ensuring that the parse tree $ET(S)$ is correctly maintained, as well as the ESP subtrees at every node. We again make use of Theorem 2 and its proof, to show that any edit operation will require the reparsing of no more than $O(\log^* n)$ symbols for each level i in $ET(S)$.

For character inserts, deletes and changes, we only need to reparse an $O(\log^* n)$ region at each level in $ET(S)$. This follows from Lemma 2.5. The nodes can be changed, and the stored sketches and signatures updated using arithmetic operations on them. As noted in the proof of Theorem 2, moves can be handled by retaining the parsing of all but the fringes of the substring that is being moved. Using pointer manipulations, the move of a large substring can be dealt with using only $O(\log n)$ pointer changes. The fringes must be reparsed in the same way as with character operations, at a cost of $O(\log^* n)$ operations per level. Similarly, split and join operations can be handled by just reparsing the $O(\log^* n)$ symbols which are affected by the split or join at every level. We can update the sketches and signatures for each node as this is going on, since these can be manipulated arithmetically [Ind00].

Requests of type (4) are handled by taking using the sketches of the strings in question, and using these to generate the approximation of their distance. Since we can maintain a sketch for every string in a collection, it follows from Theorem 8 that we can approximate the distance between any pair accurate up to a factor of $O(\log n \log^* n)$. ∎

5 Conclusion

We have provided a deterministic near linear time algorithm that is an $O(\log n \log^* n)$ approximation to the string edit distance matching problem with moves. This is the first substantially subquadratic algorithm known for any string edit distance matching problem with nontrivial alignment. Our result was obtained by embedding this string distance into the L_1 vector distance; this embedding is of independent interest since we use it to solve a variety of string proximity problems such as nearest neighbors, outlier detection, and stream-based approximation of string distances, etc. We can further apply this embedding to other problems such as string furthest neighbors, string clustering etc; details will be in the final version of this paper. All of these results are the first known for this string edit distance. It is open whether the $O(\log n \log^* n)$ factor in our approximations can be improved.

With only minor modifications, the techniques in this paper can allow the distances being approximated to incorporate additional operations such as linear scalings, reversals, etc. However, the outstanding open problem is to understand the standard string edit distance matching problem (or quite simply computing the standard edit distance between two strings) where substring moves are not allowed. We have yet to make progress on this problem.

Additional Note. We have recently learned of [ȘV95] where strings are parsed into 2-3 trees similar to our ESP method. That paper proposes a novel approach for data compression by parsing strings hierarchically online; their methods do not involve embeddings, and do not yield results in this paper.

Acknowledgements We gratefully thank Cenk Șahinalp for introducing us to string parsings, useful comments and pointing us to [ȘV95]. We also thank Sariel Har-Peled, Piotr Indyk and Mike Paterson for comments on early versions.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[ABR00] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, pages 819–828, 2000.

[ALP00] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k-mismatches. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*, pages 794–803, 2000.

[AM91] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991.

[CH98] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. In *Procedings of the 9th Annual Symposium on Discrete Algorithms*, pages 463–472, 1998.

[CPȘV00] G. Cormode, M. Paterson, S. C. Șahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proceedings of the 11th Symposium on Discrete Algorithms*, pages 197–206, 2000.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms.* OUP, 1994.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th Symposium on Theory of Computing*, pages 206–219, 1986.

[Gal85] Z. Galil. Open problems in stringology. In *Combinatorial Algorithms on Words*, pages 1–8. Springer, 1985.

[GIV01] A. Goel, P. Indyk, and K. Varadarajan. Reductions among high dimensional proximity problems. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms*, 2001.

[GPS87] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *19th Symposium on Theory of Computing*, pages 315–324, 1987.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* CUP, 1997.

[HRR98] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report SRC 1998-011, DEC Systems Research Centre, 1998.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *30th Symposium on Theory of Computing*, pages 604–613, 1998.

[Ind00] P. Indyk. Stable distributions, psuedorandom generators, embeddings and data stream computation. In *40th Symposium on Foundations of Computer Science*, 2000.

[Kar93] H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, November 1993.

[KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *4th Symposium on Theory of Computing*, pages 125–136, 1972.

[KOR98] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *30th Symposium on Theory of Computing*, pages 614–623, 1998.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710, 1966.

[LV86] G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *18th Symposium on Theory of Computing*, pages 220–230, 1986.

[MP80] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[MṢ00] S. Muthukrishnan and S. C. Ṣahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *32nd Symposium on Theory of Computing*, 2000.

[MSU97] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, February 1997.

[Mye86] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

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

[ṢV94] S. C. Ṣahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *26th Symposium on Theory of Computing*, pages 300–309, 1994.

[ṢV95] S. C. Ṣahinalp and U. Vishkin. Data compression using locally consistent parsing, 1995.

[ṢV96] S. C. Ṣahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *37th Symposium on Foundations of Computer Science*, pages 320–328, 1996.