Pattern Matching with Address Errors: Rearrangement Distances

Amihood Amir [*] [†] Yonatan Aumann [*] Gary Benson [‡] Avivit Levy [*§]
Ohad Lipsky [*] Ely Porat [*] Steven Skiena [¶] Uzi Vishne [‖]

Abstract

Historically, approximate pattern matching has mainly focused at coping with errors in the data, while the order of the text/pattern was assumed to be more or less correct. In this paper we consider a class of pattern matching problems where the content is assumed to be correct, while the *locations* may have shifted/changed. We formally define a broad class of problems of this type, capturing situations in which the pattern is obtained from the text by a sequence of *rearrangements*. We consider several natural rearrangement schemes, including the analogues of the ℓ_1 and ℓ_2 distances, as well as two distances based on interchanges. For these, we present efficient algorithms to solve the resulting string matching problems.

1 Introduction

Historically, approximate pattern matching grappled with the challenge of coping with errors in the data. The traditional *Hamming distance* problem assumes that some elements in the pattern are erroneous, and one seeks the text locations where the number of errors is sufficiently small [19, 16, 5], or efficiently calculating the Hamming distance at every text location [1, 18, 5]. The *edit distance* problem adds to the possibility that some elements of the text are deleted, or that noise is added at some text locations [20, 12]. Indexing and dictionary matching under these errors has also been considered [17, 14, 23, 11].

Implicit in all these problems is the assumption that there may indeed be errors in the *content* of the data, but the *order* of the data is inviolate. Data may be lost or noise may appear, but the relative position of the symbols is unchanged. Data *does not move around*. Even when *don't cares* were added [15], when non-standard models were considered[7, 22, 2] the order of the data was assumed to be ironclad.

Nevertheless, some non-conforming problems have been gnawing at the walls of this assumption. The *swap* error, motivated by the common typing error where two adjacent symbols are exchanged [21, 4], does not assume error in the content of the data, but rather, in the order. However, here too the general order was assumed accurate, with the difference being at most one location away. Recently, the advent of computational biology has added several problems wherein the "error" is in the order, rather than the content. During the course of evolution, whole areas of genome may translocate, shifting from one location in genome to another. Alternatively, two pieces of genome may exchange places. Considering the genome as a string over the four letter alphabet A,C,G,T, these cases represent a situation where the content of the individual entries does not change, but rather the difference between the original string and resulting one is in the locations of the different elements. Several works have considered specific versions of this biological setting, primarily focusing on the sorting problem (*sorting by reversals* [8, 9], *sorting by transpositions* [6], and *sorting by block interchanges* [10]).

Motivated by these questions, we propose a new pattern matching paradigm, which we call *pattern matching with address errors*. In this paradigm we study pattern matching problems where the content is unaltered, and only the locations of the different entries may change. We believe that the advantages in formalizing this as a general new paradigm are three-fold:

1. By providing a unified general framework, the relationships between the different problems can be better understood.

[*]Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel {amir,aumann,levyav2,lipsky,porately}@cs.biu.ac.il

[†]College of Computing, Georgia Tech, Atlanta, GA 30332-0280. Partially supported by NSF grant CCR-01-04494 and ISF grant 35/05

[‡]Departments of Biology and Computer Science, Program in Bioinformatics, Boston University, Rm 207, 44 Cummington St., Boston, MA 02215, gbenson@bu.edu.

[§]Partly supported by a BIU President Fellowship. This work is part of A. Levy's Ph.D. thesis.

[¶]Department of Computer Science, State University of New York at Stony Brook Stony Brook, NY 11794-4400, USA, skiena@cs.sunysb.edu.

[‖]Department of Mathematics, Bar-Ilan University, Ramat-Gan 52900, ISRAEL, vishne@math.biu.ac.il.

SODA '06, January 22-26, Miami, FL
©2006 SIAM ISBN 0-89871-605-5/06/01

2. General techniques can be developed, rather than ad-hoc solutions.

3. Future problems can be more readily analyzed.

In this paper we consider a broad class of problems in this new paradigm, namely - the class of *rearrangement errors*. In this type of errors the pattern is transformed through a sequence of *rearrangement operations*, each with an associated *cost*. The cost induces a distance measure between the strings, defined as the total cost to convert one string to the other. Given a pattern and a text, we seek the subsequence of the text closest to the pattern. We consider several natural distance measures, including the analogues of the ℓ_1 and ℓ_2 distances, as well as two *interchange measures*. For these, we provide efficient algorithms for different variants of the associated string matching problems.

In a separate paper [3] we consider another, different, broad class of location errors - that of *bit address errors*.

It is exciting to point out that many of the techniques we found useful in this new paradigm of pattern matching are not generally used in the classical paradigm. This reinforces our belief that their is room for this new model, as well as gives hope to new research directions in the field of pattern matching.

1.1 Rearrangement Distances. Consider a set A and let x and y be two m-tuples over A. We wish to formally define the process of converting x to y through a sequence of *rearrangement* operations. A *rearrangement operator* π is a function $\pi : [0..m-1] \rightarrow [0..m-1]$, with the intuitive meaning being that for each i, π moves the element currently at location i to location $\pi(i)$. Let $s = (\pi_1, \pi_2, \ldots, \pi_k)$ be a sequence of rearrangement operators, and let $\pi_s = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_k$ be the composition of the π_j's. We say that *s converts x into y* if for any $i \in [0..n-1]$, $x_i = y_{\pi_s(i)}$. That is, y is obtained from x by moving elements according to the designated sequence of rearrangement operations.

Let Π be a set of rearrangement operators, we say that Π *can convert x to y*, if there exists a sequence s of operators from Π that converts x to y. Given a set Π of rearrangement operators, we associate a non-negative *cost* with each sequence from Π, $w : \Pi^* \rightarrow R^+$. We call the pair (Π, w) a *rearrangement system*. Consider two vectors $x, y \in A^n$ and a rearrangement system $\mathcal{R} = (\Pi, w)$, we define the distance from x to y under $\mathcal{R}$ to be:

$$d_{\mathcal{R}}(x,y) = \min\{\mathit{cost}(s) | s \text{ from } \mathcal{R} \text{ converts } x \text{ to } y\ \}$$

If there is no sequence that converts x to y then the distance is ∞.

The String Matching Problem. Let $\mathcal{R}$ be a rearrangement system and let $d_{\mathcal{R}}$ be the induced distance function. Consider a text $T = T[0], \ldots, T[n-1]$ and pattern $P = P[0], \ldots, P[m-1]$ $(m \leq n)$. For $0 \leq i \leq n-m$ denote by $T^{(i)}$ the m-long substring of T starting at location i. Given a text T and pattern P, we wish to find the i such that $d_{\mathcal{R}}(P, T^{(i)})$ is minimal.

1.2 Our Results. We consider several natural rearrangement systems and the resulting distances. For these, we provide efficient algorithms for computing the distances.

1.2.1 The ℓ_1 and ℓ_2 Rearrangement Distances. The simplest set of rearrangement operations allows any element to be inserted at any other location. Under the *ℓ_1 Rearrangement System*, the cost of such a rearrangement is the sum of the distances the individual elements have been moved. We call the resulting distance the *ℓ_1Rearrangement Distance.* In the *ℓ_2 Rearrangement System* we use the same set of operators, with the cost being the sum of squares of the distances the individual elements have moved.[1] We call the resulting distance the *ℓ_2Rearrangement Distance.* We prove:

THEOREM 1.1. *For T and P of sizes n and m respectively ($m \leq n$), the ℓ_1 Rearrangement Distance can be computed in time $O(m(n-m+1))$. If all entries of P are distinct, then the distance can be computed in time $O(n)$.*

Interestingly, the ℓ_2 distance can be computed much more efficiently:

THEOREM 1.2. *For T and P of sizes n and m respectively ($m \leq n$) the ℓ_2 Rearrangement Distance can be computed in time $O(n \log m)$.*

1.2.2 The Interchange Distances. Consider the set of rearrangement operators were in each operation the location of exactly two entries can be interchanged. The cost of a sequence is the total number of interchanges. We call the resulting distance the *interchanges distance.* We prove:

THEOREM 1.3. *For T and P of sizes n and m, respectively ($m \leq n$), if all entries of P are distinct, then the interchanges distance can be computed in time $O(m(n-m+1))$.*

[1]For simplicity of exposition we omit the square root usually used in the ℓ_2 distance. This does not change the complexity, since the square root operation is monotone, and can be computed at the end.

Next we consider the case were multiple pairs can be interchanged in parallel, i.e. in any given step an element can participate in at most one interchange. The cost of a sequence is the number of parallel steps. We call the resulting distance the *parallel interchanges distance*, denoted by $d_{p\text{-}interchange}(\cdot,\cdot)$. We prove:

THEOREM 1.4. *For any two tuples x and y, either $d_{p\text{-}interchange}(x,y)=\infty$ or $d_{p\text{-}interchange}(x,y)\le 2$.*

This means that if it is altogether possible to convert x to y, then it is possible to do so in at most two parallel steps of interchange operations!

With regards to computing the distance we prove:

THEOREM 1.5. *For T and P of sizes n and m respectively ($m\le n$), if there are k distinct entries in P, then the parallel interchanges distance can be computed deterministically in time $O(k^2 n\log m)$.*

THEOREM 1.6. *For T and P of sizes m and n respectively ($m\le n$), the parallel interchanges distance can be computed randomly in expected time $O(n\log m)$.*

2 The ℓ_1 Rearrangement Distance

Let x and y be strings of length m. Note that any rearrangement under the ℓ_1 operators can simply be viewed as a permutation $\pi:[0..m-1]\to[0..m-1]$, where the cost is $cost(\pi)=\sum_{j=0}^{m-1}|j-\pi(j)|$. We call a permutation π that can convert x to y, an *element preserving permutation*. The following lemma characterizes the minimal cost permutation converting x to y.

LEMMA 2.1. *Let $x,y\in A^m$ be two strings such that $d_{\ell_1}(x,y)<\infty$. Let π_o be the permutation that for all a and k, moves the k-th a in x to the location of the k-th a in y. Then,*

$$d_{\ell_1}(x,y)=cost(\pi_o).$$

I.e. π_o is the permutation of the least cost.

Proof. For a permutation π, and $i<j$ such that $x[i]=x[j]$, say that π *reverses i and j* if $\pi(j)>\pi(k)$. Note that π_o has no *reversals*. Now we show that it has the least cost. Let τ be a permutation converting x to y of minimal cost that has the minimal number of reversals. If there are no reversals in τ, then there is nothing to prove, since it is exactly the permutation π_0. Otherwise, suppose τ reverses j and k ($j<k$). Let τ' be the permutation which is identical to τ, except that $\tau'(j)=\tau(k)$ and $\tau'(k)=\tau(j)$. Then, clearly τ' also converts x to y. We show that $cost(\tau')\le cost(\tau)$. Consider two cases:

Case 1: $\tau(j)\ge k$ or $\tau(k)\le j$. Consider the case $\tau(j)\ge k$. Then clearly $\tau(j)>j$, hence:

$$\begin{aligned}
&cost(\tau)-cost(\tau')=\\
&=|\tau(j)-j|+|\tau(k)-k|-|\tau'(j)-j|-|\tau'(k)-k|\\
&=|\tau(j)-j|+|\tau(k)-k|-|\tau(k)-j|-|\tau(j)-k|\\
&=(\tau(j)-j)+|\tau(k)-k|-|\tau(k)-j|-(\tau(j)-k)\\
&=\tau(j)-k+k-j+|\tau(k)-k|-|\tau(k)-j|-(\tau(j)-k)\\
&=(k-j)+|\tau(k)-k|-|\tau(k)-j|\\
&\ge|(k-j)+(\tau(k)-k)|-|\tau(k)-j|=0
\end{aligned}$$

The argument for $\tau(k)\le j$ is symmetrical.

Case 2: $j<\tau(k)<\tau(j)<k$. Then,

$$\begin{aligned}
&cost(\tau)-cost(\tau')=\\
&=|\tau(j)-j|+|\tau(k)-k|-|\tau'(j)-j|-|\tau'(k)-k|\\
&=|\tau(j)-j|+|\tau(k)-k|-|\tau(k)-j|-|\tau(j)-k|\\
&=(\tau(j)-j)+(k-\tau(k))-(\tau(k)-j)-(k-\tau(j))\\
&=2(\tau(j)-\tau(k))>0
\end{aligned}$$

Thus, the cost of τ' is at most that of τ, and there is one less reversal in τ', in contradiction.

Thus, the minimal cost permutation is the one in which for each symbol a, the first a in x is moved to the place of the first a in y, the second a in x to the second a is y, and so forth for all symbols. Thus, in order to compute the ℓ_1 distance of x and y, we create for each symbol a two lists, $\psi_a(x)$ and $\psi_a(y)$, the first being the list of locations of a in x, and the other - the locations of a in y. Both lists are sorted. These lists can be created in linear time. Clearly, if there exists an a for which the lists are of different lengths then $d_{\ell_1}(x,y)=\infty$. Otherwise, for each a, compute the ℓ_1 distance between the corresponding lists, and sum over all a's. This provides a linear time algorithm for strings of identical lengths, and an $O(m(n-m+1))$ algorithm for the general case.

We now show that if all entries of P are distinct, then the problem can be solved in $O(n)$. In this case, w.l.o.g. we may assume that the pattern is simply the string $0,1,\ldots,m-1$. The basic idea is first to compute the distance for the first text location, as described above. Then inductively compute the distance for one text location, based on the previous location, making the proper adjustments. Consider a text location i such that $d_{\ell_1}(P,T^{(i)})<\infty$. Then, since all entries of P are distinct, for each $j\in P$ there is exactly one *matching* entry in $T^{(i)}$. As we move from one text location to the next, the matching symbols all move one location to the left - relative to the pattern (except for the leftmost - which falls out, and the rightmost - which is added). For symbols that are further to the right in the text than in the pattern, this movement decreases the ℓ_1 distance by 1. For symbols that are further to the left in the text than in the pattern, this movement increases the distance by 1. Thus, given the distance at location i, in order to compute the distance at location $i+1$, we only need to know how many there are of each type (the new symbol and the one removed are easily handled). To this end we keep track for each symbol j if it is currently to the left or to the right (this is stored in the array *location*$[\cdot]$), and the current number in each type (stored in *L-count* and *R-count*). In addition, we store for each symbol the point at which it moves from being at the right to being at the left (this is stored in the array *Trans-point*$[\cdot]$). Since P is simply the sequence $0, 1, \ldots, m-1$, this *Trans-point*$[\cdot]$ can be easily computed. In this way we are able to compute the distances for each location in $O(1)$ steps per location, for a total of $O(n)$. A full description of the algorithm is provided in Figure 1. Note that each symbol in the text participates in line 16 at most once, so the amortized cost of this line is $O(1)$. Also, note that the main part of the algorithm (lines 1–18) computes the distance correctly only for those locations which have bounded distance. However, by simple counting it is easy to eliminate (in $O(n)$ steps), all the locations of infinite distance. Thus, in line 19 we find the minimum among those which have bounded distance.

3 The ℓ_2 Rearrangement Distance

3.1 Equal length sequences. Let x and y be strings of length m. Any conversion from x to y can be viewed as a permutation $\pi : [0..m-1] \to [0..m-1]$. For the ℓ_2 distance, the cost is $cost(\pi) = \sum_{j=0}^{m-1} |j - \pi(j)|^2$. The following lemma characterizes the minimal cost permutation converting x to y, as in the ℓ_1 distance. Note that it may not hold for other distances.

LEMMA 3.1. *Let $x, y \in A^m$ be two strings such that $d_{\ell_2}(x,y) < \infty$. Let π_o be the permutation that for all a and k, moves the k-th a in x to the location of the k-th a in y. Then,*

$$d_{\ell_2}(x,y) = cost(\pi_o).$$

I.e. π_o is the permutation of the least cost.

Proof. An important property of π_o is that it has no *reversals.* Recall that a reversal is a triple $a \in A$, $i, j \in [0..m-1]$, where $x_i = x_j = a$ and $i < j$ but $\pi(i) > \pi(j)$. We can transform any element preserving permutation π to π_o by a series of steps. Each such step transforms an element preserving permutation to another element preserving permutation that has less reversals, by exchanging the images that cause the reversal. This operation transforms π to π' by defining $\pi'(i) \leftarrow \pi(j)$ and $\pi'(j) \leftarrow \pi(i)$. All other values of π' are the same as π. It is easy to see that the total number of reversals is reduced.

We will show that $cost(\pi') \leq cost(\pi)$. The consequence is that for every element preserving permutation π, $cost(\pi) \geq cost(\pi_o)$, and that will conclude the proof of the lemma. The only remaining thing to prove, then, is that $cost(\pi') \leq cost(\pi)$.

Without loss of generality there are three cases to consider.

1. $\pi(j) \leq i < j \leq \pi(i)$. Let $c = |i - \pi(j)|$, $d = |j - i|$, and $e = |\pi(i) - j|$. Then the cost contributed by this pair to $cost(\pi)$ is $(i-\pi(i))^2 + (j-\pi(j))^2 = (d+e)^2 + (c+d)^2 \leq e^2 + d^2 = (j-\pi'(j))^2 + (i-\pi'(i))^2$.
2. $i \leq \pi(j) \leq j \leq \pi(i)$. Let $c = |i - \pi(j)|$, $d = |\pi(j) - \pi(i)|$, and $e = |\pi(i) - j|$. Then the cost contribution of this pair to $cost(\pi)$ is $(i-\pi(i))^2 + (j-\pi(j))^2 = (c+d)^2 + (d+e)^2 \leq c^2 + d^2 = (i-\pi'(i))^2 + (j-\pi'(j))^2$
3. $\pi(j) \leq i \leq \pi(i) \leq j$. Let $c = |i - \pi(j)|$, $d = |i - \pi(i)|$, and $e = |\pi(i) - j|$. The cost contributed by this pair to $cost(\pi)$ is $(i-\pi(i))^2 + (j-\pi(j))^2 = d^2 + (c+d+e)^2 \leq c^2 + e^2 = (i-\pi'(i))^2 + (j-\pi'(j))^2$

Now that we are guaranteed that π_o provides the minimum distance, we need to compute $cost(\pi_o)$. In our framework, where $x, y \in A^m$ it can be computed in the following manner. Consider an element $a \in A$, and let $\text{occ}_a(x)$ be the number of occurrences of a in x. Note that if x can be converted to y then necessarily $\text{occ}_a(x) = \text{occ}_a(y)$. Let $\psi_x(a)$ be the sorted sequence (of length $\text{occ}_a(x)$) of locations of a in x. Similarly $\psi_a(y)$ is this sequence for y. Then,

$$cost(\pi_o) = \sum_{a \in x} \sum_{j=0}^{\text{occ}_a(x)-1} (\psi_a(x)[j] - \psi_a(y)[j])^2. \quad (3.1)$$

Since $\sum_{a \in x} \text{occ}_a(x) = m$, the above sum can be computed in linear time.

3.2 The Case of Text and Pattern of Different Sizes. Consider a text T of length n and a pattern P of length m. We wish to compute the ℓ_2 distance of P to each text substring $T^{(i)}$ ($T^{(i)}$ is the m-long substring of T starting at position i). First note that by simple counting we can easily find all locations for which the

```
COMPUTING THE $\ell_1$DISTANCE (all variables initialized to 0)
1  For $j = 0$ to $m-1$ do
2        if $T[j] \le j$ then $location[T[j]] \leftarrow Left$
3        else $location[T[j]] \leftarrow Right$
4        set $Trans\text{-}point[T[j]] \leftarrow j - T[j]$
5  For $j = 0$ to $m-1$ do
6        add $T[j]$ to the list $Trans\text{-}symbols[Trans\text{-}point[T[j]]]$
7  $R\text{-}count \leftarrow |\{j | location[j] = Right\}|$
8  $L\text{-}count \leftarrow |\{j | location[j] = Left\}|$
9  $d[0] \leftarrow \sum_{j=0}^{m-1} |j - T[j]|$
10 For $i = 1$ to $n-m$ do
11       set $t \leftarrow T[i+m]$
12       if $location[t] = Left$ then $L\text{-}count - -$
13       else remove $t$ from $Trans\text{-}symbols[Trans\text{-}point[t]]$
14       add $t$ to the list $Trans\text{-}symbols[i+m-t]$ and set $Trans\text{-}point[t] \leftarrow i+m-t$
15       $d[i] \leftarrow d[i-1] + L\text{-}count - R\text{-}count + m - t$
16       for each $t' \in Trans\text{-}symbols[i]$ do $location[t'] \leftarrow Left$
17       $L\text{-}count \leftarrow L\text{-}count + |\{Trans\text{-}symbols[i]\}|$
18       $R\text{-}count \leftarrow R\text{-}count - |\{Trans\text{-}symbols[i]\}|$
19 $d_{\min} \leftarrow \min\{d[i] \mid T^{(i)}$ is a permutation of $[0..m-1]\}$
20 return $d_{\min}$
```

Figure 1: Computing the ℓ_1 rearrangement distance for $P = (0, 1, \ldots, m-1)$

distance is ∞, i.e. the locations for which there is no way to convert the one string to the another. Thus, we need only regard the substrings $T^{(i)}$ which are a permutation of P. For these substrings, $\text{occ}_a(P) = \text{occ}_a(T^{(i)})$ for all $a \in P$.

We can certainly compute the distances using the algorithm presented above. The total time would be $O(nm)$. However, we can obtain a much faster algorithm, as follows. Consider a symbol a, and let $\psi_a(P)$ and $\psi_a(T)$ be the lists of locations of a in P and T, respectively. Note that these two lists need not be of the same length. Similarly, let $\psi_a(T^{(i)})$ be the list of locations of a in $T^{(i)}$. Then, by equation (3.1), for any $T^{(i)}$ (which is a permutation of P):

$$d_{\ell_2}(P, T^{(i)}) = \sum_{a \in P} \sum_{j=0}^{\text{occ}_a(P)-1} (\psi_a(P)[j] - \psi_a(T^{(i)})[j])^2 \tag{3.2}$$

We now wish to express the above sum using $\psi_a(T)$ instead of the individual $\psi_a(T^{(i)})$'s. Note that all the a's referred to in $\psi_a(T^{(i)})$ are also referred to in $\psi_a(T)$. However, $\psi_a(T)$ gives the locations with regards to the beginning of T, whereas $\psi_a(T^{(i)})$ gives the locations with regards to the beginning of $T^{(i)}$ - which is i positions ahead.

For each i and a, let $match_a(i)$ be the index of the smallest entry in $\psi_a(T)$ with value at least i. Then, $match_a(i)$ is the first entry in $\psi_a(T)$ also referenced by $\psi_a(T^{(i)})$. Then, for any a, i and $j \le \text{occ}_a(P)$:

$$\psi_a(T^{(i)})[j] = \psi_a(T)[match_a(i) + j] - i.$$

Thus, Equation (3.2) can now be rewritten as:

$$d_{\ell_2}(P, T^{(i)}) = \tag{3.3}$$
$$\sum_{a \in P} \sum_{j=0}^{\text{occ}_a(P)-1} (\psi_a(P)[j] - (\psi_a(T)[match_a(i) + j] - i))^2$$

We thus want to compute this sum for all i. We do so by a combination of convolution and polynomial interpolation, as follows.

The values of $match_a(i)$. We first show how to efficiently compute $match_a(i)$ for all a and i. Consider two consecutive locations i and $i+1$. Let $T[i]$, the symbol at the i-th location in T. Then,

$$match_a(i+1) = \begin{cases} match_a(i) + 1 & a = T[i] \\ match_a(i) & a \ne T[i] \end{cases} \tag{3.4}$$

This is because $T[i]$ is the only symbol no longer available when considering $T^{(i)}$. Equation (3.4) allows us to incrementally compute $match_a(i)$ for all i. That is, if we know $match_a(i)$ for all a, then we can also know $match_a(i+1)$, for all a, in $O(1)$ steps.

The functions G_x and F_x. Fix a number x, and suppose that instead of the computing the sum in Equation (3.3), we want to compute the sum:

$$G_x(i) = \sum_{a\in P} \sum_{j=0}^{\mathrm{occ}_a(P)-1} (\psi_a(P)[j] - (\psi_a(T)[match_a(i)+j] - x))^2$$

This is the same sum as in Equation (3.3), but instead of subtracting i in the parenthesis, we subtract the fixed x. The important difference is that now x is independent of i. Note that by Equation (3.3) $d_{\ell_2}(P, T^{(i)}) = G_i(i)$.

For a, k let

$$F_x(a,k) = \sum_{j=0}^{\mathrm{occ}_a(P)-1} (\psi_a(P)[j] - (\psi_a(T)[k+j] - x))^2$$

Then,

$$G_x(i) = \sum_{a\in P} F_x(a, match_a(i))$$

Suppose that we have pre-computed $F_x(a,k)$ for all a and k. We show how to compute $G_x(i)$ for all i (for the fixed x). We do so by induction. For $i = 0$ we compute $G_x(i)$ using the algorithm presented above, in $O(m)$ steps. Suppose that we have computed $G_x(i)$ and we now wish to compute $G_x(i+1)$. Then,

$$G_x(i) = \sum_{a\in P} F_x(a, match_a(i))$$

while

$$G_x(i+1) = \sum_{a\in P} F_x(a, match_a(i+1))$$

However, by Equation (3.4), for most of the a's $match_a(i+1) = match_a(i)$ and for $a = T[i]$, $match_a(i+1) = match_a(i) + 1$. Thus,

$$G_x(i+1) - G_x(i) = -F_x(T[i], match_{T[i]}(i)) \quad + F_x(T[i], match_{T[i]}(i)+1)$$

Thus, assuming that $G_x(i)$ is known, and that all $F_x(a,k)$ have been pre-computed, $G_x(i+1)$ can be computed in $O(1)$ steps. (The values of $match_a(i)$ are incrementally computed as we advance from i to $i+1$.)

Computing $F_x(a,k)$. We now show how to compute $F_x(a,k)$ for all a and k. We do so using the following general lemma:

Lemma 3.2. *Let Q and W be two sequences of real numbers, with lengths $|Q|$ and $|W|$, respectively ($|Q| \le |W|$). Let $p(q,w)$ be a polynomial in two variables, and t an integer ($t \le |Q|$). For $i = 0, \ldots, |W| - |Q|$, let $P_{Q,W}(i) = \sum_{j=0}^{t-1} p(Q[j], W[i+j])$. Then, $P_{Q,W}(i)$ can be computed for all i's together, in $O(|W| \log |Q|)$ steps.*

Proof. It is sufficient to prove for p having only a single addend. For more addends, simply compute each separately and add the results. Thus, $p = cq^\alpha w^\beta$, for some constants c, α, and β. Create two new sequences $Q' = cQ[1]^\alpha, cQ[2]^\alpha, \ldots$, and $W' = W[1]^\beta, W[2]^\beta, \ldots$. Let Z be the convolution of Q' and W'. Then, $P_{Q,W}(i) = Z[i]$. The convolution can be computed in $O(|Q| \log |W|)$ steps.

Applying the lemma to our setting let $p(q,w) = (q - w + x)^2$, $t = \mathrm{occ}_a(P)$, $Q = \psi_a(P)$ and $W = \psi_a(T)$. Then, $F_x(a,k) = \sum_{j=0}^{t-1} p(Q[j], W[k+j])$. Thus, the lemma holds, and $F_x(a,k)$ can be computed for all k's together in $O(\mathrm{occ}_a(T)\log(\mathrm{occ}_a(P)))$ steps. Combining for all a's, the computation takes:

$$\sum_{a\in P} O(\mathrm{occ}_a(T)\log(\mathrm{occ}_a(P))) = O(n \log m)$$

(since $\sum_{a\in P} \mathrm{occ}_a(T) \le n$ and $\sum_{a\in P} \mathrm{occ}_a(P) = m$).

From $G_x(i)$ to $d_{\ell_2}(P, T^{(i)})$. We have so far seen that for any fixed x, we can compute $G_x(i)$ for all i in $O(n \log m)$ steps. Recall that $d_{\ell_2}(P, T^{(i)}) = G_i(i)$. Thus, we wish to compute $G_i(i)$ for all i. For any fixed i, considering x as a variable, $G_x(i)$ is a polynomial in x of degree ≤ 2. Thus, if we know the value of $G_x(i)$ for three different values of x, we can then compute its value for any other x in a constant number of steps using polynomial interpolation. Thus, in order to compute $G_i(i)$ we need only know the value of $G_x(i)$ for three arbitrary values of x, say 0, 1 and 2. Accordingly, we first compute $G_0(i), G_1(i)$ and $G_2(i)$, for all i in $O(n \log m)$ time, as explained above. Then, using interpolation, we compute $G_i(i)$ for each i separately, in $O(1)$ steps per i. The total complexity is thus $O(n \log m)$. This completes the proof of Theorem 1.2.

4 The Interchanges Distance

In this section we show how to compute the interchanges distance in the case that all entries in the pattern are different (i.e. P is a permutation). We begin with a known definition and a fact.

Definition 1. A *permutation cycle* is a subsequence of a permutation whose elements trade places cyclically with one another.

Fact 4.1. *The representation of a permutation as a product of disjoint permutation cycles is unique (up to the ordering of the cycles).*

The next lemma is needed for the characterization of the interchange distance.

Lemma 4.1. *Sorting a k-length permutation cycle requires exactly $k-1$ interchanges.*

Proof. By induction on k. A cycle of length 1 is already sorted, so the lemma hold. In a cycle of length 2, one interchange sorts the two elements, so again the lemma holds. Now, for a cycle of length $k > 2$ any interchange sorts only one element. Choosing any pair in the cycle for a single interchange sorts one element and we are left with a cycle of length $k-1$, for which the induction hypothesis holds.

The next theorem characterizes the interchange distance, i.e. the minimum number of interchanges needed to sort a permutation.

THEOREM 4.1. *The interchange distance of an m-length permutation π is $m - c(\pi)$, where $c(\pi)$ is the number of permutation cycles in π.*

Proof. From fact 4.1 we know that there is a unique decomposition of π into cycles. Since interchanges of elements in different cycles do not sort any element we clearly get a smaller distance by interchanging elements only within cycles. Now, from lemma 4.1 we get that each cycle 'saves' exactly one interchange. Therefore, the theorem follows.

The theorem leads to the following $O(nm)$ algorithm for the interchange distance problem. By a linear scan of the pattern and text find all locations in the text which have a bounded distance. These locations are exactly the ones in which all pattern symbols appear exactly once. For each such text position i, construct all pairs (j, k), where k is the position of $T[i+j]$ in the pattern. There are exactly m such pairs. Sort the pairs by a linear sorting method. Now count the number of cycles by actually tracing them one by one. Finally, use the theorem to get the distance result.

5 The Parallel Interchanges Distance

5.1 Bounding the Parallel Interchanges Distance. Previously we saw that a cycle of length ℓ can be sorted by $\ell - 1$ interchanges. The natural question is what is the minimal number of parallel interchange steps required for this sorting. The next lemma surprisingly shows that with a careful choice of the interchanges, we can always do it with at most two parallel steps.

LEMMA 5.1. *Let σ be a cycle of length $\ell > 2$. It is possible to sort σ in two parallel interchanges steps.*

Proof. The given string is $(1, 2, 3, ..., \ell-2, \ell-1, 0)$. In the first parallel step we invert the segment $(1, 2, ..., \ell-1)$, namely perform the $(\ell-1)/2$ interchanges $(1, \ell-1)$, $(2, \ell-2)$, etc. The resulting string is $(\ell-1, \ell-2, \ell-3, ..., 3, 2, 1, 0)$, from which the sorted string can be obtained in an additional $\ell/2$ interchanges: $(0, \ell-1)$, $(1, \ell-2)$ etc.

Which proves Theorem 1.4.

Remark. Sorting a permutation by a given set Ω of allowed operations can be also viewed as asking how many permutations from a given set are required to express the permutation as their product. In other words, the maximal distance between two permutations is equal to the minimal number of elements of Ω that need to be multiplied in order to cover the whole symmetric group S_n (where Ω is usually a conjugacy class), see [13]. This problem was dealt in the mathematical literature under the name of 'covering'. In [25] the problem of covering S_{2n} with permutations which are the product of exactly n interchanges is studied. In the current terminology, this is like asking for the parallel interchanges distance under the requirement that exactly n interchanges are performed in each step (for a space of size $2n$). It is shown that 3 operations suffice almost always, but for a (given) set of permutation, 4 operations are required.

5.2 Computing the Parallel Interchanges Distance. By theorem 1.4 there are only four different possibilities for the parallel-interchanges distance between a pattern and a text. Thus, in order to compute the distance, we need only check which of the four is the correct one. Distance 0 signifies an exact match, and can be found in time $O(n)$ using standard techniques. Distance ∞ means that at any text location i, the strings P and $T^{(i)}$ either contain different symbols, or with different multiplicity. This can again be computed in time $O(n)$ by simple counting methods. Thus, it remains to be able distinguish between distances 1 and 2. We show how to check for distance 1.

We start by describing the deterministic algorithm. If two strings have distance 1, then we say that one is a *parallel interchange* of the other. For each i and pair of alphabet symbols (a, b), we count the number of times that a appears in the pattern and b appears in the corresponding location in the text $T^{(i)}$. Then, P is a parallel interchange of $T^{(i)}$ iff for any a, b, the count for (a, b) equals that for (b, a). This count can be implemented by convolutions in the following manner.

Let S be a string over alphabet Σ and let $a \in \Sigma$. Denote by $\chi_a(S)$ the binary string of length $|S|$ where every occurrence of a is replaced by 1 and every other symbol occurrence is replaced by 0. The dot product of $\chi_a(T^{(i)})$ with $\chi_b(P)$ gives precisely the number of times an a in $T^{(i)}$ is aligned with a b in P. This number is computed for all alignments of the pattern with the text in time $O(n \log m)$ using convolutions [15]. Clearly, it is sufficient to consider only symbols from Σ_P. We thus obtain that the parallel interchanges distance can be computed deterministically in time $O(|\Sigma_P|^2 n \log m)$

(Theorem 1.5).

For unbounded alphabets this is not very helpful. So, we seek a further speedup via randomization. The idea is to view the symbols of the alphabet as symbolic variables, and use the Schwartz-Zippel Lemma [24, 27], as follows. For variables a, b, let $h(a, b) = a^2b - b^2a$. Note that $h(a, a) = 0$ and $h(a, b) = -h(b, a)$. Given two strings $x, y \in A^m$, define the polynomial:

$$H_{x,y} = \sum_{j=0}^{m-1} h(x_j, y_j)$$

Then, $H_{x,y} \equiv 0$ (i.e. $H_{x,y}$ is the all zeros polynomial) iff x is a parallel interchange of y. Thus, for each text location i, we wish to check if $H_{P,T^{(i)}} \equiv 0$. We do so by randomly assigning numeric values to the symbolic variables, and using the Schwartz-Zippel Lemma. Specifically, each variable is assigned a random value chosen uniformly and independently from the set $\{1, \ldots, 3m\}$. Let r be the random assignment. Then by the Schwartz-Zippel lemma, for any x and y, $\Pr[H_{x,y}(r) = 0 | H_{x,y} \not\equiv 0] \le \frac{deg(H_{x,y})}{3m} = \frac{1}{m}$. Clearly, if $H_{x,y} \equiv 0$ then $H_{x,y}(r) = 0$ for all r. Accordingly, given the random assignment r, we compute the value of $H_{P,T^{(i)}}(r)$, for all i. If the value is different from 0 for all i, then clearly there is no parallel interchange of the pattern in the text, and the distance cannot be 1. Otherwise, we check one by one each locations i for which $H_{P,T^{(i)}}(r) = 0$. For each such i, we check if $H_{P,T^{(i)}} \equiv 0$ (as a symbolic polynomial). For each specific location i, this can be performed in time $O(m)$. Once the first location for which $H_{P,T^{(i)}} \equiv 0$ is found, we conclude that the distance is 1, and no further locations are checked.

It remains to explain how to compute $H_{P,T^{(i)}}$, for all i. We do so using convolutions. Specifically, from the string P, we create a string P' of length $2m$, by replacing each entry a, by the pair $r(a)^2, r(a)$ (where $r(a)$ is the value given to the symbolic variable a under the random assignment r). Similarly, from T we create a string T' of length $2n$, by replacing each b with the pair $-r(b), r(b)^2$. Then, if C is the convolution of T' and P', then for all i, $C(2i) = H_{P,T^{(i)}}(r)$. We obtain:

Lemma 5.2. *The above algorithm determines if there is a parallel interchange of P in T in expected time $O(n \log m)$.*

Proof. The convolution takes $O(n \log m)$. For each location i, consider two cases. First consider the case where $T^{(i)}$ is a not a parallel interchange of P. In this case, if $C(2i) = H_{P,T^{(i)}}(r) \neq 0$ then there is no additional to do for this location. Otherwise ($C(2i) = 0$), there is $O(m)$ work to check the symbolic polynomial. However, this happens with probability $\le m^{-1}$. Thus, the expected work for each such location is $O(1)$. Next, consider the locations $T^{(i)}$ that *are* parallel interchanges of P. For the first of these locations, the symbolic polynomial is going to be checked, in $O(m)$ steps. However, once this first location is checked, it is found to be a parallel interchange of P, and no further work will be performed. Thus, these locations add only $O(m)$ work. Hence, the total work is $O(n \log m)$.

We thus a randomized algorithm that computes the parallel interchanges distance in expected $O(n \log m)$ steps.

We may now be tempted to try and extend this method to obtain a more efficient deterministic algorithm, in the following method. Suppose that we could find a small number of polynomials, $H^{(1)}, H^{(2)}, \ldots, H^{(k)}$, such that for a *given* assignment, computing their values at each text location i, would provide a deterministic indication of a parallel interchange. For example, suppose we could find a "good" set of polynomials such that for any assignment they vanish iff there is a parallel interchange. Then, if we could compute their values using convolutions, we could hope for an efficient algorithm. The next lemma, which is based on communication complexity arguments, proves that such an approach cannot provide better performance than $\tilde{\Omega}(nm)$. To this end we use the *convolution model*, as defined in [2].

Lemma 5.3. *Any algorithm in the convolution model for determining if there is a parallel interchanges requires $(m(n - m + 1))$ bit operations.*

Proof. The proof is by reduction from the communication complexity of the *word equality problem.* The word equality problem is the following:
INPUT: Two m bit words, $W_1 = W_1[1], ..., W_1[m]$ and $W_2 = W_2[1], ..., W_2[m]$.
DECIDE: Whether $W_1 = W_2$ (i.e. $W_1[i] = W_2[i]$, $i = 1, ..., m$) or not.

The reduction is done in the following way. Given two words W_1 and W_2, construct the strings: $T = W_1, 1, 2, ...m$ (the concatenation of W_1 with $1, ..., m$), and $P = 1, 2, ..., m, W_2$. Then, T is a parallel interchange of P iff $W_1 = W_2$.

Suppose that the parallel interchange problem can be solved using $c(m)$ convolutions, $C_1, ..., C_{c(m)}$. Then, specifically for T and P as above, it is possible to determine if P is a parallel interchange of T based on the results of these $c(m)$ convolutions at location 1. The convolution values can be computed for the first part of the strings and the second part separately, and then added. Furthermore, since in each part one of the strings is fixed, each player can compute his/her part separately. Thus, only the partial convolution results need to be communicated. Thus, the total communication is bounded by the sum of the values of the convolution results. However, a known result in communication complexity is that the word equality problem takes $\Omega(m)$ bits [26]. Thus, the total bit complexity of the convolutions is $O(m)$ *for each text location* i. Thus, the total bit complexity is $O(m(n-m+1))$.

References

[1] K. Abrahamson, *Generalized string matching*, SIAM J. Comp. **16** (1987), no. 6, 1039–1051.

[2] A. Amir, A. Aumann, R. Cole, M. Lewenstein, and E. Porat, *Function matching: Algorithms, applications, and a lower bound*, Proc. 30*th* ICALP, 2003, pp. 929–942.

[3] A. Amir, Y. Aumann, and A. Levy, *Pattern matching with address errors: Renaming schemes*, Manuscript.

[4] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat, *Overlap matching*, Information and Computation **181** (2003), no. 1, 57–74.

[5] A. Amir, M. Lewenstein, and E. Porat, *Faster algorithms for string matching with k mismatches.*, J. Algorithms **50** (2004), no. 2, 257–275.

[6] V. Bafna and P.A. Pevzner, *Sorting by transpositions*, SIAM J. on Discrete Mathematics **11** (1998), 221–240.

[7] B. S. Baker, *A theory of parameterized pattern matching: algorithms and applications*, Proc. 25th Annual ACM Symposium on the Theory of Computation, 1993, pp. 71–80.

[8] P. Berman and S. Hannenhalli, *Fast sorting by reversal*, Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM) (D.S. Hirschberg and E.W. Myers, eds.), LNCS, vol. 1075, Springer, 1996, pp. 168–185.

[9] A. Carpara, *Sorting by reversals is difficult*, Proc. 1st Annual Intl. Conf. on Research in Computational Biology (RECOMB), ACM Press, 1997, pp. 75–83.

[10] D. A. Christie, *Sorting by block-interchanges*, Information Processing Letters **60** (1996), 165–169.

[11] R. Cole, L.A. Gottlieb, and M. Lewenstein, *Dictionary matching and indexing with errors and don't cares*, STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, 2004, pp. 91–100.

[12] R. Cole and R. Hariharan, *Approximate string matching: A faster simpler algorithm*, Proc. 9th ACM-SIAM Symposium on Discrete Algorithms (SODA), 1998, pp. 463–472.

[13] Y. Dvir, *Covering properties of permutation groups*, Products of Conjugacy Classes in Groups (Z. Arad and M. Herzog, eds.), Lecture Notes in Mathematics 1112, 1985.

[14] P. Ferragina and R. Grossi, *Fast incremental text editing*, Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (1995), 531–540.

[15] M.J. Fischer and M.S. Paterson, *String matching and other products*, Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings **7** (1974), 113–125.

[16] Z. Galil and R. Giancarlo, *Improved string matching with k mismatches*, SIGACT News **17** (1986), no. 4, 52–54.

[17] M. Gu, M. Farach, and R. Beigel, *An efficient algorithm for dynamic text indexing*, Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms (1994), 697–704.

[18] H. Karloff, *Fast algorithms for approximately counting mismatches*, Information Processing Letters **48** (1993), no. 2, 53–60.

[19] G. M. Landau and U. Vishkin, *Efficient string matching with k mismatches*, Theoretical Computer Science **43** (1986), 239–249.

[20] V. I. Levenshtein, *Binary codes capable of correcting, deletions, insertions and reversals*, Soviet Phys. Dokl. **10** (1966), 707–710.

[21] R. Lowrance and R. A. Wagner, *An extension of the string-to-string correction problem*, J. of the ACM (1975), 177–183.

[22] S. Muthukrishnan and H. Ramesh, *String matching under a general matching relation*, Information and Computation **122** (1995), no. 1, 140–148.

[23] S. C. Sahinalp and U. Vishkin, *Efficient approximate and dynamic matching of patterns using a labeling paradigm*, Proc. 37th FOCS (1996), 320–328.

[24] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, J. of the ACM **27** (1980), 701–717.

[25] U. Vishne, *Mixing and covering in the symmetric group*, Journal of Algebra **205** (1998), no. 1, 119–140.

[26] A. C. C. Yao, *Some complexity questions related to distributed computing*, Proc. 11th Annual Symposium on the Theory of Computing (STOC), 1979, pp. 209–213.

[27] R. Zippel, *Probabilistic algorithms for sparse polynomials*, Proc. EUROSAM, LNCS, vol. 72, Springer, 1979, pp. 216–226.