Pattern Matching with Swaps*

Amihood Amir[†‡] — Bar-Ilan University & Georgia Tech
Yonatan Aumann[†] — Bar-Ilan University
Gad M. Landau[§] — Polytechnic University & Haifa University
Moshe Lewenstein[†] — Bar-Ilan University
Noa Lewenstein[† ¶] — Bar-Ilan University

Abstract

Let a text string T of n symbols and a pattern string P of m symbols from alphabet Σ be given. A *swapped version* T' of T is a length n string derived from T by a series of *local swaps*, (i.e. $t'_\ell \leftarrow t_{\ell+1}$ and $t'_{\ell+1} \leftarrow t_\ell$) where each element can participate in *no more than one swap*.

The *Pattern Matching with Swaps* problem is that of finding all locations i for which there exists a swapped version T' of T with an exact matching of P in location i of T'. It has been an open problem whether swapped matching can be done in less than $O(nm)$ time.

In this paper we show the first algorithm that solves the pattern matching with swaps problem in time $o(nm)$. We present an algorithm whose time complexity is $O(nm^{1/3} \log m \log \sigma)$ for a general alphabet Σ, where $\sigma = min(m, |\Sigma|)$.

Key Words: Design and analysis of algorithms, combinatorial algorithms on words, pattern matching, pattern matching with swaps, non-standard pattern matching.

1 Introduction

1.1 Background

String Matching is one of the most widely studied problems in computer science [9]. Part of its appeal is in its direct applicability to "real world" problems. The Boyer-Moore [5] algorithm is directly implemented in the *emacs* "`s`" and *UNIX* "`grep`" commands. The longest common subsequence dynamic programming algorithm [6] is implemented in the *UNIX* "`diff`" command.

*A preliminary version of this paper appeared in FOCS 97.

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; {amir,aumann,moshe,noa}@cs.biu.ac.il

‡Partially supported by NSF grant CCR-96-10170, BSF grant 96-00509, and a BIU internal research grant.

§Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, phone: (718) 260-3154, FAX: (718) 260-3906; Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; email: landau@poly.edu; partially supported by NSF grants CCR-9305873 and CCR-9610238; and by the Israel Science Foundation founded by the Israeli Academy of Sciences and Humanities.

¶Partially supported by the Israel Ministry of Science and the Arts grant 8560.

The largest overlap heuristic for finding the shortest common superstring [28] has been used in DNA sequencing, and there are many other examples.

Advances in Multimedia, Digital libraries and Computational Biology have shown that a much more generalized theoretical basis of pattern matching could be of tremendous benefit [26]. To this end, pattern matching has to adapt itself to increasingly broader definitions of "matching". In computational biology one may be interested in finding a "close" mutation, in communications one may want to adjust for transmission noise, in texts it may be desirable to allow common typing errors. In multimedia one may want to adjust for lossy compressions, occlusions, scaling, affine transformations or dimension loss.

All above applications motivated the *Generalized Pattern Matching* problem. In generalized matching the input is still a text and pattern but the "matching" relation is defined differently. The output is all locations in the text where the pattern "matches" under the new definition of match. The different applications define the matching relation. An early generalized matching was defined by Fischer and Paterson [8]. They allow a special "don't care" character that serves as a wildcard. It matches every other alphabet symbol. A text location where the pattern matches under these conditions is an exact match.

Even under the new matching relation there is still a distinction between *exact matching* and *approximate matching*. In the latter case, a metric is defined on the text. A text location is considered a match if the distance between it and the pattern, under the given metric, is within the tolerated bounds.

The fundamental question is what type of approximations are inherently hard computationally, and what types are faster to compute. This question motivated much of the pattern matching research in the last twenty years.

The earliest, and by now classic, metrics for approximate matching are the following. Levenshtein [18] identified three types of errors, mismatches, insertions, and deletions. These operations are traditionally used to define the *edit distance* between two strings. Lowrance and Wagner [20, 29] added the *swap* operation to the set of operations defining the distance metric. The swap, where the order of two consecutive symbols is reversed, is one of the most typical typing errors. More complex versions of the swap occur in nature. The phenomenon of swaps occurs in gene mutations and duplications (e.g. the region of human chromosome 5 that is implicated in the disease called Spinal Muscular Atrophy, a common recessive form of muscular dystrophy [19]). While the Biological swaps occur at a gene level, and have several additional constraints and characteristics, which make the problem much more difficult, they do serve as a convincing pointer to the theoretical study of swaps as a natural edit operation for the approximation metric.

Using the edit distance or extended edit distance as a metric, one may then define a "match" to be a text location where the pattern matches with no more than a given number of allowed operations.

The lower bound story is rather short. Muthukrishnan and Ramesh [24] prove that practically all general matching relations, where the generalization is in the definition of single symbol matches, are equivalent to the boolean convolutions, i.e. it is unlikely that they could be solved in time faster than $O(n \log m)$, where n is the text length and m is the pattern length. Furthermore, Muthukrishnan and Palem [23] proved that in the *boolean convolutions model* there are non trivial lower bounds on the number of necessary convolutions for several non-standard string matching problems.

Most research success in generalized and approximate pattern matching lies with faster upper bounds. Let n be the text length and m the pattern length. Lowrance and Wagner proposed an $O(nm)$ dynamic programming algorithm for the extended edit distance problem. In [11, 16, 17] $O(kn)$ algorithms are given for the edit distance with only k allowed edit operations. Sahinalp and Vishkin [27] presented an $O(nk^{8+\frac{1}{\log 3}}(\frac{\log^* n}{n})^{\frac{1}{\log 3}})$ algorithm for this problem. A faster algorithm was presented by Cole and Hariharan [7] with $O(nk^4/m + n)$ running time.

Since the upper bound for the edit distance seems very tough to break, attempts were made to consider the edit operations separately. If only mismatches are counted for the distance metric, we get the *hamming distance*, which defines the *string matching with mismatches* problem. A great amount of work was done on finding efficient algorithms for string matching with mismatches [1, 10, 13, 14, 15, 4].

1.2 Isolating the Swap Operation

In this paper we attempt to isolate the *swap* operation from the rest of the extended edit operations, in the manner that the *mismatch* operation had been isolated in the past, to enable careful analysis of the inherent difficulty of the swap operation alone. We present the first algorithm whose complexity beats the naive $O(nm)$ bound.

The existing lower bounds in non-standard stringology do not apply in the case of the *String Matching with Swaps* problem, because of its contextual nature. The only exception is a new result reducing boolean convolutions to string matching with swaps [22]. However, there were no known upper bounds better than the naive $O(nm)$ algorithm. Indeed, this problem was formulated and described as one of the open problems in non-standard string matching [21].

The way we tackle the problem is by first finding a solution to the generalized matching problem with swaps (see [21]). In this problem one seeks all text locations where the pattern matches allowing swaps in the text, regardless of the number of swaps. We present an algorithm whose time complexity is $O(nm^{1/3}\log m \log \sigma)$ for a general alphabet Σ, where $\sigma = min(m, |\Sigma|)$.

Our method is as follows. We first reduce the general alphabet swapped matching problem to a two letter alphabet problem with a complexity penalty of only an $O(\log \sigma)$, multiplicative factor, where $\sigma = min(m, |\Sigma|)$. We then concentrate our efforts on the swapped matching problem over alphabet $\Sigma = \{a, b\}$. We prove a structural concatenation lemma that defines the conditions under which it is permissible to "put together" swapped matches of consecutive pattern substrings to get a swapped match of the entire pattern. We show that the less-than matching problem ([2]) can be used to efficiently handle the concatenation conditions. Finally, we solve our special case of less-than matching with "don't cares" in time $O(nm^{1/3}\log m)$.

Now that we have an efficient solution to the swapped matching problem as a generalized matching problem, we consider the number of swaps as a distance metric. We show how our algorithm can be used to provide a $O(n\sqrt{m}\ \text{polylog}(m))$ algorithm for the approximate string matching problem where the distance metric is the number of swaps.

Paper organization. This paper is organized in the following way. In section 2 we give basic definitions and prove some crucial traits of the problem. In section 3 we prove that reduction to a two-symbol alphabet merely causes a multiplicative $\log \sigma$ degradation in time complexity. In section

4 we concentrate on solving the swapped matching problem over alphabet $\Sigma = \{a, b\}$ and achieve a $O(nm^{1/3} \log m)$ algorithm. Coupled with the result of section 3 this gives a solution to the swapped matching problem over general alphabets in time $O(nm^{1/3} \log m \log \sigma))$, where $\sigma = min(m, |\Sigma|)$.

2 Problem Definition and Preliminaries

Definition: Let $T = t_1, \ldots, t_n$ be a *text* string over alphabet Σ. A *swap permutation* for T is a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that

1. if $\pi(i) = j$ then $\pi(j) = i$ (characters are swapped).
2. for all i, $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters are swapped).
3. if $\pi(i) \neq i$ then $t_{\pi(i)} \neq t_i$ (identical characters are not swapped).

For a given swap permutation π and text T we denote $\pi(T) = t_{\pi(1)}, t_{\pi(2)}, \ldots, t_{\pi(n)}$. We call $\pi(T)$ a *swapped version* of T. For pattern $P = p_1, \ldots, p_m$ and text T, we say that P has a *swapped match (SM) at location* i if there exists a swapped version T' of T such that P has an exact match with T' starting at location i, i.e. $p_j = t'_{i+j-1}$ for $j = 1, \ldots, m$.

The *Pattern Matching with Swaps Problem* is the following:
INPUT: Text string $T = t_1, \ldots, t_n$ and pattern string $P = p_1, \ldots, p_m$ over alphabet Σ.
OUTPUT: All locations i where P has a swapped match in T.

For ease of the exposition we consider the following restriction on the matching.

Restriction: We say that pattern P has a *swapped match with internal starting symbol (SM_{sin})* at location i, if P exactly matches a swapped version $\pi(T)$ starting at location i, and $\pi(i) \geq i$ (i.e. symbol i was not swapped to "outside" the pattern).

Note that any swapped match either has a swapped match with internal starting symbol or $p_2 \ldots p_m$ does. Therefore, it follows that:

Lemma 1 *If the Swapped Pattern Matching with Internal Starting Symbol Problem can be solved in time $O(f(n,m))$ then the Pattern Matching with Swaps problem can also be solved in time $O(f(n,m))$.*

Further Restriction: We further add the restriction that the *ending symbol* is not allowed to swap externally, i.e. $\pi(i+m-1) \leq i+m-1$. We call a match for which both the starting symbol and the ending symbol do not swap externally a *swapped match with internal endpoints (SM_{in}).*

In a similar manner to Lemma 1, one can show that any solution of time complexity $O(f(n,m))$ for the Swapped Pattern Matching with Internal Endpoints problem, can be used to derive a solution for the general Swapped Matching problem in time $O(f(n,m))$. From now on, in all our algorithms we use any of the internal endpoints versions of the problem as convenience dictates.

Uniqueness of the swap permutation. For a string $S = s_1 \ldots s_t$ we denote $S[i,j] = s_i \ldots s_j$. The following lemma proves the uniqueness of the swap permutation in the case of internal starting point swapped match. The uniqueness is needed in section 3 where we show the reduction of general alphabets to the binary alphabet case.

Lemma 2 *Consider an alphabet Σ, text T and pattern P, $|P| = m$. If there exists a swapped match with internal starting symbol of P in T at location i then the swap permutation providing this match is unique on the set $\{i, \ldots, i+m\}$, i.e. there exists a unique swap permutation $\pi : \{i, \ldots, i+m\} \to \{i, \ldots, i+m\}$ such that $t_{\pi(\ell)} = p_\ell$ for $\ell = i, \ldots, i+m-1$*

Proof:

Since the match is with an internal starting point we may assume w.l.o.g. that $i = 1$. We prove by contradiction.

Say m is the smallest length of a pattern P for which this does not hold. It is easy to verify that $m > 1$. Let π and τ be two different swap permutations producing this match. Consider the restriction, π' and τ', of π and τ (respectively), to $\{1, \ldots, m\}$. Similarly, let π'' and τ'' be the restriction of π and τ (respectively), to $\{1, \ldots, m-1\}$. Note that π' and τ' both produce a match for $P[1, m-1]$, and π'', τ'' both produce a match for $P[1, m-2]$. Also note that, in general, the restrictions $\pi', \tau', \pi'', \tau''$ are not necessarily permutations. There are three possible cases:

1. $\pi(m+1) = \tau(m+1) = m+1$: then both π' and τ' are permutations. Hence, by minimality of m, $\pi' = \tau'$. Thus, $\pi = \tau$, a contradiction.

2. $\pi(m+1) = \tau(m+1) = m$: then $\pi(m) = \tau(m) = m+1$ and both π'' and τ'' are permutations. Thus, by minimality of m, $\pi'' = \tau''$ and $\pi = \tau$, a contradiction.

3. $\pi(m+1) = m+1$, $\tau(m+1) = m$: then π' is a permutation. Moreover, since we do not swap identical symbols and $\tau(m) \neq m$, it must be that $p_m \neq t_m$. Hence $\pi'(m) = m-1$. Define the following swap permutation $\hat{\tau}$ on $\{1, \ldots, m\}$: $\hat{\tau}(i) = \tau(i)$, for $i < m$ and $\hat{\tau}(m) = m$ (this is well defined since $\tau(m) = m+1$). Note that $\hat{\tau}(T[1, m-1]) = \tau(T[1, m-1]) = P[1, m-1]$. Thus, both π' and $\hat{\tau}$ are swap permutation for $P[1, m-1]$. However $\pi'(m) \neq \hat{\tau}(m)$. Thus, $\pi' \neq \hat{\tau}$, in contradiction to the minimality of m. □

Applying lemma 2 on P and T and then on P^R and T^R yields the following.

Corollary 1 *For any alphabet Σ, text T and pattern P, if there exists a swapped match with internal endpoints of P in T at location i then the swap permutation providing this match is unique on the set $\{i, \ldots, i+m-1\}$.*

3 Reducing General Alphabets to a Two Letter Alphabet

In this section we show how to reduce the swapped matching problem over unbounded alphabets to the problem over a two letter alphabet. The reduction entails an $O(\log |\Sigma|)$ multiplicative overhead, where $|\Sigma|$ is the size of the alphabet. Here, we are assuming that $|\Sigma| \leq m+1$. Otherwise, change all text symbols that are not in the pattern to a single symbol that does not appear in the pattern.

Definition: Let Σ be an alphabet, and let $F = \{\chi_1, \dots, \chi_k\}$ be a family of characteristic functions, $\chi_j : \Sigma \to \{0,1\}$. We say that F is a *separating family* if for each ordered triplet of characters $(a,b,c) \in \Sigma^3$, there exists a χ_j such that $\chi_j(a) \neq \chi_j(b) = \chi_j(c)$.

A natural separating family is the set of *projections* $\{\chi_\sigma\}_{\sigma\in\Sigma}$, where

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma \\ 0 & \text{if } x \neq \sigma \end{cases}$$

We extend the definition of the functions χ_j to a strings in the usual manner, i.e. for $S = s_1 s_2 \dots s_n$, $\chi_j(S) = \chi_j(s_1)\chi_j(s_2)\dots\chi_j(s_n)$.

Theorem 1 *Let P be a pattern, T a text, both over an arbitrary alphabet Σ, and let $F = \{\chi_1, \dots, \chi_k\}$ be a separating family for Σ. There is a SM_{sin} for P in T at location i iff for all j there is a SM_{sin} of $\chi_j(P)$ in $\chi_j(T)$ at location i.*

Proof: W.l.o.g. $i = 1$.

Only if: The swap permutation providing the match for P in T yields a match for $\chi_j(P)$ in $\chi_j(T)$ for all j (technically, we eliminate from the permutation the swaps which swap identical symbols in $\chi_j(T)$).

If: We show that if there is no SM_{sin} for P in T (at location 1) then there exists a j for which there is no SM_{sin} . The proof is by induction on ℓ, the length of the largest prefix $P[1,\ell]$ of P, for which there exists a SM_{sin} of $P[1,\ell]$ in T (at location 1). For this ℓ we show that there exists a j for which there is no SM_{sin} of $\chi_j(P[1,\ell+1])$ in $\chi_j(T)$ at location 1. Thus, there is also no SM_{sin} of $\chi_j(P)$ in $\chi_j(T)$ at location 1.

Base Case $\ell = 0$: This happens in case $P[1,1]$ does not have a SM_{sin} at location 1. Thus, $p_1 = a \neq b = t_1$, and $t_2 = c \neq a$. By definition of separating family there exists a j such that $\chi_j(a) \neq \chi_j(b) = \chi_j(c)$. For this j there is no SM_{sin} of $\chi_j(P[1,1])$ in $\chi_j(T)$ at location 1.

Inductive Step: Assume that $P[1,\ell]$ is the longest prefix of P that has a SM_{sin} at location 1. Let π be the permutation that provides the match for $P[1,\ell]$. The following cases need to be considered:

1. π does not swap t_ℓ to the right (externally). Let $p_{\ell+1} = a$, $t_{\ell+1} = b$ and $t_{\ell+2} = c$.
   (a) $a = b$ or $a = c$: in this case $P[1,\ell+1]$ has a SM_{sin} , violating the maximality of ℓ.
   (b) $a \neq b$ and $a \neq c$: by assumption there exists a j such that $\chi_j(a) \neq \chi_j(b) = \chi_j(c)$. For this j there is no SM_{in} for $\chi_j(P[1,\ell+1])$ in $\chi_j(T)$ at location 1. This is because if there were a match, then, by the uniqueness of the swap permutations (Lemma 2) the swap permutation, τ, providing such a match must agree with π of $T[1,\ell]$. Thus, $\chi_j(p_{\ell+1})$ must match $\chi_j(t_{\ell+1})$ or $\chi_j(t_{\ell+2})$, which it does not.
2. π swaps t_ℓ to the right. Let $p_\ell = t_{\ell+1} = a$, $t_\ell = b$ and $p_{\ell+1} = c$.
   (a) $b = c$: then $P[1,\ell+1]$ has a SM_{sin} , violating the maximality of ℓ.
   (b) $b \neq c$: let j be such that $\chi_j(b) \neq \chi_j(a) = \chi_j(c)$. For this j there is no SM_{in} for $\chi_j(P)$ in $\chi_j(T)$ at location 1 (the reasoning is similar to that of 1.(b)). □

We now show how to construct a small separating family for an alphabet.

Definition: An (n,k)-*universal set* is a set S of n-digit binary vectors such that for any set of positions $L \subset \{1,...,n\}$, $|L| = k$, the projection of S on L gives all possible 2^k combinations of $0-1$s.

In [25] it is shown how to construct (n,k)-universal set of cardinality $2^{O(k)} \log n$ with the same time bounds. Note that a $(|\Sigma|, 3)$-universal set is a separating family for Σ. This yields the following corollary.

Corollary 2 *If the Swapped Pattern Matching (with Internal Endpoints) problem can be solved over alphabet $\{a,b\}$ in time $O(f(n,m))$ then the Swapped Pattern Matching (with Internal Endpoints) problem can be solved in time $O(\log|\Sigma| f(n,m))$ over a general alphabet Σ.*

Proof. This follows from Theorem 1 and the existence of an $O(\log|\Sigma|)$ size separating family. □

4 Swapped Matching over a Two Letter Alphabet

Following Corollary 2, from here on we only consider a two letter alphabet $\Sigma = \{a,b\}$.

Definition: Let $S = s_1 \dots s_m$ be a string. We say that S is *alternating* if $\forall i > 1, s_{i-1} \neq s_i$. For a given pattern P we say that $P[i,j]$ is a *maximal alternating substring* of P if $P[i,j]$ is alternating and no other alternating substring of P strictly includes locations i through j in P.

Since P is a pattern over $\{a,b\}$, it is obvious that there is a unique partition $P = P^1 P^2 \cdots P^t$, such that each P^j is a maximal alternating substring of P. This partition can easily be constructed in $O(n)$ time.

To find the match of P in T our algorithm checks for a match of each maximal alternating substring separately. The following lemma shows that this is sufficient.

Lemma 3 (Concatenation Property) *Let P be a pattern and $P = P^1 \cdots P^t$ be the partition of P into maximal alternating substrings. For text T, P has a swapped match with internal endpoints at location i iff for each $j = 1, \dots, t$, P^j has a swapped match with internal endpoints in T at the corresponding locations (i.e. at location $i + \sum_{\ell=1}^{j-1} |P^\ell|$).*

Proof: Denote $m_j = |P^j|$ and $i_j = i + \sum_{\ell=1}^{j-1} m_\ell$.

If: Let π_j be the swap permutation providing the SM_{in} of P^j in T at location i_j. We construct a joint swap permutation π for P, by pasting the segments of the π_j's on the P^j's. Formally, $\pi(k) = \pi_j(k)$ for $i_j \leq k < i_{j+1}$ and $\pi(k) = k$ for all other k. Since the π_j are internal endpoints swap permutations, π is also an internal endpoints swap permutation for P.

Only if: Let π be the swap permutation providing the SM_{in} of P in T at location i. We show that for each j, π provides a SM_{in} for P^j at location i_j. In contradiction assume that π does not provide a SM_{in} for P^j. Clearly π provides a regular swapped matching for P^j in T at location i_j. Thus, it must be that π either swaps i_j with $i_j - 1$ or $i_j + m_j - 1$ with $i_j + m_j$. The first case would only happen if $p_{i_j} \neq p_{i_j - 1}$ in which case P^j could be extended one symbol to the left for a longer alternating substring in contradiction to the maximality of P^j. The second case is similar. □

It thus remains to show how to check that all maximal substrings have an SM_{in} . In the next section we present the notion of *streams* and their construction. The streams are the fundamental building block in the algorithm for finding the matches of alternating substrings.

4.1 Streams

Given a text T and a pattern P, we are interested in finding all SM_{in} of maximal alternating substrings of P in T. As a first step we ignore P, and for each location i in T find the longest alternating string that has a SM_{sin} at this location. To this end we define the notion of *streams.*

Definition: For text T a *stream* S starting at location i is an alternating string S with a SM_{sin} at location i. A stream S starting at location i is *maximal* if there is no other stream S' at location $i' \leq i$ such that S is a proper substring of S', and S and S' agree on the overlapping locations, i.e. $s_j = s'_{i-i'+j}$, $j = 1, \ldots, |S|$. Following is an example of a maximal stream starting at location 1. The arrows show the swap permutation providing the match.

Example:

$$\begin{array}{rcccccccccccc}
S_1 = & a & b & a & b & a & b & a & & & & & \\
T = & a & \vec{a} & \overset{\leftarrow}{b} & \vec{a} & \overset{\leftarrow}{b} & \vec{a} & \overset{\leftarrow}{b} & a & a & b & a & b
\end{array}$$

We distinguish between two types of streams. In the *odd-a type* stream, the a's of the stream match the odd locations of the text, and in the *even-a type* streams, the a's match the even text locations. The above example is of an odd-a type stream. Generally, there may be more than one maximal stream of the same type at any text location. This can be seen in the following example.

Example:

$$\begin{array}{rcccccccccccc}
S_1 = & a & b & a & b & a & & & & a & b & = & S_3 \\
T = & a & \vec{a} & \overset{\leftarrow}{b} & \vec{a} & \overset{\leftarrow}{b} & a & a & a & \vec{b} & \overset{\leftarrow}{a} & b & b \\
S_2 = & & & \vec{a} & \overset{\leftarrow}{b} & \vec{a} & \overset{\leftarrow}{b} & a & \vec{b} & \overset{\leftarrow}{a} & \vec{b} & \overset{\leftarrow}{a} & b
\end{array}$$

In this example the text appears in the middle line. S_1, S_2 and S_3 are maximal streams of type odd-a. The arrows provide the corresponding swap permutation (no arrow indicates an identity). Note that in locations 3,4 and 5 of the text there is an overlap of the two different streams S_1 and S_2 and in locations 9 and 10 there is an overlap between S_2 and S_3.

Fortunately, it turns out that there can be no more than two different overlapping maximal streams of the same type at any text location.

Lemma 4 *For any location i in the text and a given stream type, there are at most two distinct maximal streams of that type that cover location i.*

Proof: In contradiction, assume that S_1, S_2 and S_3 are distinct maximal streams of the same type all covering the same location. Let i_1, i_2, i_3 be the starting points, and j_1, j_2, j_3 the ending points of S_1, S_2 and S_3, respectively. Note that since all three streams are of the same type they correspond on overlapping locations. Thus, since they are all maximal, we may assume w.l.o.g. that $i_1 < i_2 < i_3$ and $j_1 < j_2 < j_3$ (otherwise one string would subsume the other). Let π_1, π_2, π_3 be the swap permutations for S_1, S_2, S_3, respectively. Consider the following two cases:

1. Both π_1 and π_2 swap i_3 with $i_3 - 1$. Then we can combine π_1 up to i_3 with π_2 from $i_3 + 1$ and on, and obtain a swap permutation for a stream from i_1 to j_2, violating the maximality of S_1, S_2.

2. Either π_1 or π_2 do not swap i_3 to the left. W.l.o.g. assume it is π_2. Note that since the match of S_3 is with an internal starting symbol, by definition π_3 does not swap i_3 to the left. Thus, we may combine π_2 up to $i_3 - 1$ with π_3 from i_3 and on, and obtain a swap permutation for a stream from i_2 to j_3, violating the maximality of S_2, S_3. □

Thus, we have the following algorithm for finding all streams.

Algorithm for Finding All Streams

Run through the text from left to right. Keep a list of "active" streams. At each location location i.

1. Extend each of the active streams if possible.
2. Start all new possible streams.

End Algorithm

Note that Lemma 2 guarantees that at each location there is only one possible extension of a given stream. Thus, by Lemma 4 the time complexity of the algorithm is $O(n)$.

Once we have the maximal streams of T, we can easily determine for each location i in T the longest stream of a given type that has a SM_{sin} starting at this location. Thus, for each i we define
odd-a(T,i)=(maximal length of odd-a stream starting at location i) and
even-a(T,i)=(maximal length of even-a stream starting at location i).

For example, in the above example *odd-a*$(T, 2) = 4$, *odd-a*$(T, 3) = 10$ and *odd-a*$(T, 9) = 2$. Given the set of all maximal streams we can determine *odd-a*(T, i) and *even-a*(T, i) for all i in $O(n)$ steps. Next we produce two new strings

OddEven(T)=*odd-a*(T,1),*even-a*(T,2),...,*odd-a*(T,2i-1), *even-a*(T,2i),...

EvenOdd(T)=*even-a*(T,1),*odd-a*(T,2),...,*even-a*(T,2i-1), *odd-a*(T,2i),...

OddEven(T) and *EvenOdd*(T) are the basis for the algorithm for finding the partitioned SM_{sin} match, as described in the next section.

Note that *odd-a*(T, i) is only an *upper bound* on the length of a odd-a stream of that matches at location i. In fact, for any $k \leq$ *odd-a*(T, i) there is a SM_{sin} for an odd-a stream of length k at location i. Similarly for even-a type streams.

4.2 Determining SM_{sin} for all Maximal Alternating Substrings of a Pattern

Consider a text T and pattern P. Let $P = P^1 \cdots P^t$ be the partition of P into maximal alternating substrings. We now show how to find all locations i for which, if P is placed over T at location i, then each P^j has a SM_{sin} starting at its corresponding location. (Formally, all locations i, for which for each j, P^j has a SM_{sin} starting at location $i + \sum_{\ell=1}^{j-1} |P^\ell|$.) We call such a "match" a *partitioned*

SM_{sin} match. Note that this type of match is "almost" sufficient for applying the concatenation of Lemma 3. There we need an SM_{in} for all P^j's rather than a SM_{sin} . In the next section we show how to make this final step.

Let *OddEven*(T) and *EvenOdd*(T) be as defined in the previous section. Set $m_j = |P^j|$. For the pattern P and substring P^j set $\alpha_j = m_j$ if P^j starts with an a and $\alpha_j = \phi$ otherwise. Set $\beta_j = m_j$ if P^j starts with a b and $\beta_j = \phi$ otherwise. For the pattern P define $P^{start\text{-}a} = \alpha_1\phi^{m_i-1}\alpha_2\phi^{m_2-1}\cdots\alpha_t\phi^{m_t-1}$, $P^{start\text{-}b} = \beta_1\phi^{m_i-1}\beta_2\phi^{m_2-1}\cdots\beta_t\phi^{m_t-1}$, where ϕ^ℓ is the string of ℓ ϕ's, and ϕ is a "don't care" symbol.

Lemma 5 *Let P be a pattern and T a text. Consider location i. There is a* partitioned SM_{sin} match *for P in T at location i, iff for all $k = 1,\ldots,m$:*

1. *$(P^{start\text{-}a})_k = \phi$ or $(P^{start\text{-}a})_k \leq (OddEven(T))_{i+k-1}$, and*
2. *$(P^{start\text{-}b})_k = \phi$ or $(P^{start\text{-}b})_k \leq (EvenOdd(T))_{i+k-1}$,*

Proof: The pattern locations which are ϕ always have a match. Consider the partition P^j, and suppose that P^j starts with an a. Let $i_j = i + \sum_{\ell=1}^{j-1} m_\ell$. If i_j is odd then a SM_{sin} match for P^j at location i_j corresponds to odd-a stream starting at location i_j. Thus, there is a SM_{sin} for P^j at the location i_j iff $(P^{start\text{-}a})_{i_j} = m_j \leq odd\text{-}a(T, i_j) = (OddEven(T))_{i_j}$. Similarly, if i_j is even then there is a match iff $(P^{start\text{-}a})_{i_j} \leq even\text{-}a(T, i_j) = (OddEven(T))_{i_j}$. If P^j starts with a b then the same hold with regards to *EvenOdd*(T). □

Thus, we have reduced the problem to that of two *less-than matching* problem. The *less-than matching* problem is defined as follows [2]:
INPUT: Text $T = t_1,\ldots,t_n$, pattern $P = p_1,\ldots,p_m$, $t_i, p_i \in N \cup \{\phi\}$.
OUTPUT: All locations i for which $p_k = \phi$ or $p_k \leq t_{i+k-1}$, for all $k = 1,\ldots,m$.

We have obtained the following algorithm for finding all location for which there is a partitioned SM_{sin} match of P in T.

Begin Algorithm

1. Compute all the less-than matches of:
   (a) $P^{start\text{-}a}$ and *OddEven*(T)
   (b) $P^{start\text{-}b}$ and *EvenOdd*(T)
2. For each $i = 1,\ldots,n-m+1$ do

   If at location i both $P^{start\text{-}a}$ is less-than *OddEven*(T) and $P^{start\text{-}b}$ less-than *EvenOdd*(T), then output i.

End algorithm

In section 5 we show how to take advantage of our special case, which has a large number of ϕ's in the pattern, in order to compute the necessary less-than matchings in $O(nm^{1/3}\log m)$ steps.

4.3 From Partitioned SM_{sin} Match to a Full SM_{in} Match

In the previous section we showed how to find all locations for which there is a partitioned SM_{sin} of P in T. This is *almost* what we need for applying Lemma 3, but not quite. Lemma 3 requires that each partition have a SM_{in} , rather than a SM_{sin} . We now show how to isolate the locations for which the partitioned SM_{sin} match is actually a SM_{in} match for each partition. The following immediate lemma provides the basic tool:

Lemma 6 *Let T be a text and S a string, with $|S| = m$, and suppose that S ends with an a. If there is a SM_{sin} for S in T at location i then*

$$\textit{(number of a's in S)} \geq \textit{(number of a's in } T[i, i+m-1]\textit{)}$$

and equality holds iff the match is a SM_{in} . The same hold also if S ends with b, with the weak inequality reversed.

Thus, in order to check that the partitioned SM_{sin} match is actually a SM_{in} match we only have to check that the number of a's (b's) in the corresponding substrings are identical. Note that since the inequality is always in one direction, an excess in one substring cannot be canceled by another. Thus, it is sufficient to count the total number of a's (b's) in the corresponding sections. This we can easily do with two convolutions, as follows.

Given $P = P^1 \cdots P^t$, we define a-end(P) to be the string obtained from P by replacing all symbols of a-ending P^j's with 1's and of b-ending symbols with 0's. Let b-end(P) be the bitwise complement of a-end(P). Now, we perform the convolution of T (with a's replaced by 1's and b's by 0's) with a-end(P) and b-end(P). Let $A(a)$ be the total number of a's in the a-ending P^j's, and $B(a)$ be the total number of A's in the b-ending P^j's. At any location i, the convolution of T and a-end(P) is exactly $A(a)$ iff for each a-ending P^j the number of a's in P^j is equal to that in the corresponding locations of T. Similarly, the convolution of T and b-end(P) is exactly $B(a)$ iff for each b-ending P^j the number of a's in P^j is equal to that in the corresponding locations of T. Combining Lemma 6 with Lemma 3, we obtain:

Corollary 3 *For text T pattern P and location i, P has a SM_{in} in T at location i iff: (i) P has a partitioned SM_{sin} in T at location i, and (ii) the convolution of a-end(P) and T at location i is $A(a)$, and (iii) the convolution of b-end(P) and T at location i is $B(a)$.*

4.4 Putting It All Together

We summarize the steps of the algorithm:

Input: Text T, pattern P.
Output: All locations i for which P has a SM_{in} in T.

Algorithm

1. Partition P into maximal alternating substrings . Time: $O(n)$.
2. Find streams and create $EvenOdd(T)$ and $OddEven(T)$ (Section 4.1). Time: $O(n)$.

3. Using $EvenOdd(T)$ and $OddEven(T)$ find all locations i for which P has a partitioned SM_{sin} match in T (Section 4.2). Time: $O(nm^{1/3}\log m)$.
4. Using convolutions isolate from locations found in step 3, the locations for which there is a SM_{in} (Section 4.3). Time: $O(n \log m)$.

End algorithm.

Thus we have obtained:

Theorem 2 *The swapped match problem can be solved in* $O(nm^{1/3}\log m)$ *steps over a constant size alphabet and* $O(nm^{1/3}\log m \log \sigma))$ *steps over a general alphabet* Σ*, where* $\sigma = min(m, |\Sigma|)$.

5 Less-Than Matching

The *less-than matching with don't cares problem* is defined as follows: *Input:* Text string $T = t_0, .., .t_{n-1}$ and pattern string $P = p_0, ..., p_{m-1}$ where $t_i, p_i \in N \cup \{\phi\}$. *Output:* All locations i in T where for all $k = 1, \ldots, m$ either $t_{i+k-1} = \phi$, $p_k = \phi$, or $t_{i+k-1} \geq p_k$. In words, every matched element of the pattern is not greater than the corresponding text element, unless either of them is a *don't care* element.

Assume that there are g elements in the pattern that are *not* ϕ (i.e. $m - g$ elements are ϕ). A minor adjustment to the algorithm of [2] allows us to solve the less-than matching with don't cares problem in time $O(\sqrt{g}\ n \log m)$.

Notation: For $\sigma \in N, x \in N \cup \{\phi\}$ let

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma \\ 0 & \text{if } x \neq \sigma \end{cases}$$

$$\chi_{<\sigma}(x) = \begin{cases} 1 & \text{if } x < \sigma \\ 0 & \text{if } x \geq \sigma \text{ or } x = \phi \end{cases}$$

If $X = x_1, \ldots, x_n$ then $\chi_\sigma(X) = \chi_\sigma(x_1), \ldots, \chi_\sigma(x_n)$. Similarly define $\chi_{<\sigma}(X)$.

We would like to know for each non-ϕ element of the pattern, where it is lined up with something less than it. We can achieve this by computing, for each σ in P, $\chi_{<\sigma}(T) \otimes \chi_\sigma(P^R)$ (where $\otimes$ is polynomial multiplication), and considering all non-zero locations.

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_g\}$ be the set of all different numbers appearing in P. Let $M_i = \chi_{<\sigma_i}(T) \otimes \chi_{\sigma_i}(P^R)$ (where $\otimes$ is polynomial multiplication). Then M_i is non-zero at position t iff there is a σ_i in the pattern matched with something smaller than σ_i when the pattern is lined up at t. These cases are exactly when we get a mismatch. If we let M be the sum of all the M_i's we get a non-zero if there was a mismatch caused by any $\sigma \in \Sigma$. By using FFT we can calculate each of the polynomial multiplications in time $O(n \log m)$ (assuming a computer word of $O(\log m)$ bits), for a total of $O(g\ n \log m)$.

Following [1, 14], Amir and Farach [2] use the multiplication technique on a limited alphabet of size $\sqrt{m}$ and then "fine tune" by another method. We point out that in reality the limited alphabet is of size $\sqrt{g}$ and the "fine tuning" algorithm works on blocks of size $O(\sqrt{g})$.

Algorithm

Input $T = t_0, \ldots, t_{n-1}$; $P = p_0, \ldots, p_{m-1}$. Assume that the text alphabet is the same as the pattern alphabet (this can be easily achieved by replacing every text number by the largest pattern number that does not exceed it).

1. For every $a \in \Sigma$ that appears more than $\sqrt{g}$ times in P (a *frequent element*), use FFT to disqualify all text locations where there is a text element that is smaller than its corresponding pattern element.

2. Let $p_{j_0}, \ldots, p_{j_{g_1}}$ be the non-frequent, non-ϕ elements of P. Consider $L = \langle p_{j_0}, j_0 \rangle$, $\langle p_{j_1}, j_1 \rangle$, ..., $\langle p_{j_{g_1}}, j_{g_1} \rangle$. [Every non-$\phi$ pattern element is considered a pair $\langle s, d \rangle$ where s is a number and d is the location of the number in the array P.]

3. Sort L lexicographically. [There are no more than g elements in L.] Call the sorted array L'.

4. Divide L' into $g_2 \leq \sqrt{g}$ blocks, each containing no more than $2\sqrt{g}$ elements, in a manner that no number appears in more than one block. [We are assured that such a division is possible because all remaining numbers are non-frequent.]

5. For each block B_i, $i = 0, \ldots, g_2 \leq \sqrt{g}$ let b_i be the smallest (leftmost) element in the block; call b_i the *representative* of block B_i.

6. Let T' and P' be T and P such that every t_i and p_i is replaced by the representative of the block it is in. [Implemented by a sequential scan of L'.]

7. Find all less-than matches of P' in T'.

   [P' and T' can be considered "flattened out" versions of P an T. When we seek all less-than matches of P' in T' we only detect the "large" mismatches, i.e., those between elements that are so different that they are in different blocks. However, mismatches between elements of the same block are undetected. At this stage we must "fine tune" our approximate solution. We scan T and for every element t_i of T we only compare it to the $O(\sqrt{g})$) elements of P that are in t_i's block.]

8. For $i = 0$ to $n - 1$ if $t_i \neq \phi$ do
   Let B_{t_i} be the block of L' that t_i is in,
   Let $P^{B_{t_i}} \leftarrow \{\langle s, d \rangle | \langle s, d \rangle \in B_{t_i}\}$
   For every element $\langle s, d \rangle$ in $P^{B_{t_i}}$
   (at most $2\sqrt{g}$ elements)
   if $t_i < s$ then $M[i - d] \leftarrow M[i - d] + 1$
   end
   end

   [The vector M is now correct since the first part of the algorithm included all the errors between blocks and the last part found all the errors within a block.]

end Algorithm

Time: $O(g \log g)$ for sorting
$O(n\sqrt{g} \log m)$ for less-than matching of the representatives
$O(n\sqrt{g})$ for correcting mismatches within blocks
Total: $O(n\sqrt{g} \log m)$

5.1 The Case of Lengths

In our case the numbers represent lengths of streams, therefore we are guaranteed that following each number i in the pattern there are at least i don't cares.

In order to solve the less-than matching problem under these circumstances we first eliminate all the cases where a match of P in T causes one of the numbers $1, 2, \ldots, m^{1/3}$ in the pattern to mismatch with the corresponding text position. This can be done by $m^{1/3}$ convolutions for a total time of $O(nm^{1/3} \log m)$.

We now have to eliminate the cases where there are mismatches in the remaining numbers. Exchange all numbers in the text and pattern that are not greater than $m^{1/3}$ by ϕ's. The smallest remaining number is $m^{1/3} + 1$, and it is followed by $m^{1/3} + 1$ ϕ's. Thus there are at most $m^{2/3}$ non-ϕ elements in the pattern ($g = m^{2/3}$).

By the previous section we can find all locations where the new pattern is less than the new text, in time $O(n\sqrt{m^{2/3}} \log m) = O(nm^{1/3} \log m)$.

The intersection of all these locations and the ones where there is no mismatch of the numbers $1, \ldots, m^{1/3}$ gives the required result.

6 The Approximation Problem

We have seen an $o(nm)$ algorithm for solving the swapped matching problem as a generalized matching problem. This algorithm can easily be adapted for solving the approximation problem with the number of swaps as the distance metric. Until now, the definition we gave for swapped matching was a generalized matching definition. However, we can view a swap permutation operationally, as resulting from a sequence of swaps of adjacent pairs, with no symbol participating in more than one swap. We are now interested in counting the number of swap operations that a text location needs to undergo in order to equal the pattern. Let us, for the moment, restrict ourselves to the swapped pattern matching with internal endpoints problem. It is clear that the number of such operations is exactly half the numbers of mismatches between the text and its swapped version (recall that we never swap a pair of equal symbols). For the general case, where the allowed swaps are not only internal, we need to count half of the mismatches resulting from internal swaps and then take care of the start and end point, in case there occurred external swaps there. Formally,

Definition: Let $T = t_1, \ldots, t_n$ be a *text* string over alphabet Σ. Let π be a swap permutation for T. The *number of swaps* in $\pi(T)[i, j]$ (the substring $t_{\pi(i)}, \ldots, t_{\pi(j)}$) is $\lfloor \frac{1}{2} d(T[i, j], \pi(T)[i, j]) \rfloor + L + R - B$, where $d(T[i,j],\pi(T)[i,j])$ is the number of mismatches between $T[i,j]$ and $\pi(T)[i,j]$,

$$L = \begin{cases} 1, & \text{if } \pi(i) = i-1; \\ 0, & \text{otherwise.} \end{cases}, \quad R = \begin{cases} 1, & \text{if } \pi(j) = j+1; \\ 0, & \text{otherwise.} \end{cases}, \quad \textit{and} \quad B = \begin{cases} 1, & \text{if } R = L = 1; \\ 0, & \text{otherwise.} \end{cases}.$$

The *Pattern Matching with Swaps Metric Problem* is the following:
INPUT: Text string $T = t_1, \ldots, t_n$ and pattern string $P = p_1, \ldots, p_m$ over alphabet Σ.
OUTPUT: For every location i where P has a swapped match in T, where T' is the swapped version of T, write the number of swaps in $T'[i, i+m-1]$.

Theorem 3 *The Swaps Metric problem can be solved in $O(n\sqrt{m}\log^2 m)$ steps.*

Proof: Pattern matching with mismatches as a metric is well-known to run in $O(n\sqrt{m}\log^2 m)$ time [1, 14]. Now, consider a text location where the pattern matches with k mismatches and the pattern also has a swapped matching. If the swapped match has internal endpoints then the number of swaps is exactly $\frac{k}{2}$. If exactly one of the endpoints swaps externally then the number of swaps is $\frac{k+1}{2}$. If both swap externally there are $\frac{k+2}{2}$ swaps. It is straightforward to adapt the solution to the swapped match problem to announce for each swapped match whether it matches with an external swap on either endpoint. Therefore we can implement pattern matching with swaps as a metric in $O(n\sqrt{m}\log^2 m)$ time. $\Box$

Acknowledgments: The authors kindly thank Alex Schäffer for sharing with us the biological inspiration and reference [19]. We are also indebted to a number of friends, known and anonymous – the first of whom being Pyotr Indyk – for pointing out to us reference [25].

References

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

[2] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.

[3] A. Amir, G. Landau, M. Lewenstein, and N. Lewenstein. Efficient special cases of swaped matching. *Information Processing Letters*, 68(3):125–132, 1997.

[4] A. Amir, M. Lewenstein, and E. Porat. Faster string matching with k mismatches. *Proceedings of the Eleventh Symposium on Discrete Algorithms, San Fransisco, CA, (to appear)*, 2000.

[5] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[6] V. Chvatal, D.A. Klarner, and D.E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Stanford University, 1972.

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

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

[9] Z. Galil. Open problems in stringology. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 1–8. NATO ASI Series F, 1985.

[10] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[11] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. Computing*, 19(6):989–999, 1990.

[12] P. Indyk. Deterministic superimposed coding with applications to pattern matching. *Proc. 38th IEEE FOCS*, pages 127–136, 1997.

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

[14] S. Rao Kosaraju. Efficient string matching. Manuscript, 1987.

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

[16] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[17] G.M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM J. Computing*, 27(2):557–582, 1998.

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

[19] B. Lewin. Genes for SMA: Multum in parvo. *Cell*, 80:1–5, 1995.

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

[21] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *Proc. 6th Combinatorial Pattern Matching Conference*, pages 298–317. Lecture Notes in Computer Science 937, Springer-Verlag, 1995.

[22] S. Muthukrishnan. Boolean convolution reduction to swapped matching. Personal communication, 1997.

[23] S. Muthukrishnan and K. Palem. Non-standard stringology: Algorithms and complexity. In *Proc. 26th Annual Symposium on the Theory of Computing*, pages 770–779, 1994.

[24] S. Muthukrishnan and H. Ramesh. String matching under a general matching relation. *Information and Computation*, 122(1):140–148, 1995.

[25] Joseph (Seffi) Naor and Moni Naor. Small-bias probability spaces: efficient constructions and applications. *SIAM Journal on Computing*, pages 838–856, 1993.

[26] A. Pentland. Invited talk. NSF Institutional Infrastructure Workshop, 1992.

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

[28] E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5:313–323, 1990.

[29] R. A. Wagner. On the complexity of the extended string-to-string correction problem. In *Proc. 7th ACM STOC*, pages 218–223, 1975.