Efficient Special Cases of Pattern Matching with Swaps

Amihood Amir* — Georgia Tech and Bar-Ilan University

Gad M. Landau† — Polytechnic University and 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 where there is an exact matching of P at location i of T'.

It was recently shown that the *Pattern Matching with Swaps* problem has a solution in time $O(nm^{1/3} \log m \log^2 \sigma)$, where $\sigma = min(|\Sigma|, m)$.

We consider some interesting special cases of patterns, namely, patterns where there is no *length-one run*, i.e. there are no $a, b, c \in \Sigma$ where $b \neq a$ and $b \neq c$ and where the substring abc appears in the pattern. We show that for such patterns the pattern matching with swaps problem can be solved in time $O(n \log^2 m)$.

Key Words: Design and analysis of algorithms, combinatorial algorithms on words, pattern matching, pattern matching with swaps, approximate pattern matching, generalized pattern matching.

1 Introduction

The *exact string matching* problem of seeking all locations in a text string where a pattern string matches, is a classical problem in computer science. Much effort was devoted to its solution in the early 70's (e.g. [4, 5, 8, 7, 15]) which provided algorithms, data structures, and techniques that became textbook materials.

The recent explosion in multimedia, digital libraries and the world wide web, has spurred renewed efforts in the pattern matching area [13]. Two important research directions are *generalized matching* and *approximate matching*.

In *generalized matching*, one still seeks all exact occurrences of the pattern in the text, but a *match* is not *equality*. Rather, a more general matching relation is defined. Examples are string matching with "don't cares" [5, 14], less-than matching [3], a matching relation defined by a graph on the alphabet [12], or a matching relation defined over commutative monoids [6].

*Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8770; amir@cs.biu.ac.il; Partially supported by NSF grant CCR-96-10170 and the Israel Ministry of Science and the Arts grants 6297 and 8560.

†Department of Computer Science, Haifa University, Haifa 31905, Israel (972-4)824-0103; landau@poly.edu; partially supported by NSF grants CCR-9305873 and CCR-9610238.

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

§Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; noa@cs.biu.ac.il; Partially supported by the Israel Ministry of Science and the Arts grant 8560.

In *approximate matching*, a distance metric between strings is defined. One then seeks all text locations where the pattern matches the text within the allowed distance under the given metric. An approximation can clearly be defined on a generalized matching relation, however the earliest distance metrics were defined on exact matching. Levenshtein used the *edit distance* of *insertions*, *deletions* and *mismatches* [10]. Lowrance and Wagner [11] added to the distance metric the *swap* as a unit cost error. The *swap*, where the order of two consecutive symbols is reversed, is one of the most typical typing errors. A much more complex version of swaps occurs in gene mutations.

The *approximate matching problem* is that of finding the smallest distance between every text location and the pattern. In the case of the Lowrance and Wagner edit distance, as in the case of the Levenshtein distance, we want the smallest number of edit operations by which every text location will be brought to match the pattern. The fastest known deterministic worst-case solution to this problem, for both the Lowrance-Wagner and Levenshtein versions, takes time $O(nm)$, where n is the text length and m is the pattern length. This solution uses dynamic programming. Despite numerous atempts to "break" the $O(nm)$ barrier, there is still no known algorithm that solves the approximate pattern matching problem with the above four edit operations in time $o(nm)$.

Attempts were made in the past to limit the edit operations, and see if more efficient algorithms result. It was shown [1, 9] that if only mismatches are used, then one can indeed solve the approximate pattern matching problem in time $O(n\sqrt{m \log m})$.

In a recent paper [2], the swap edit operation was isolated. Both the approximate matching problem with the number of swaps as the error metric, and the exact swapped matching problem, were considered. In the latter, a generalized matching relation - *the swap match* is defined.

A pattern P *swap matches* a text T at location i if an unrestricted number of local swaps can bring the text to exactly match the pattern, provided only adjacent symbols are swapped, and no symbol is swapped more than once. In [2] it was shown that the problem of pattern matching with swaps can be solved in time $O(nm^{1/3} \log m \log^2 \sigma)$, where $\sigma = min(|\Sigma|, m)$. It was conjectured there that the problem can be solved in time $O(n \text{ polylog } m)$

In this paper we consider several special cases of patterns. If every time a new symbol appears in the pattern, there is a *run* of at least *four* consecutive appearances of the symbol, then we show that the problem can be solved in linear time for general alphabets.

If the length of the minimum run is limited to *two* rather than four, we provide an $O(n \log^2 m)$ algorithm for general alphabets, thus giving evidence in support of the conjecture of [2].

2 Problem Definition – Pattern Matching with Swaps

Definition: Let $T = t_1, \ldots, t_n$ be a *text* string over alphabet Σ and $P = p_1, \ldots, p_m$ be a *pattern* string over Σ. The string $\overline{T}$ is *derived from* T *via a swap* if $\exists \ell,\ 1 \leq \ell \leq n-1$, such that $\overline{t}_\ell = t_{\ell+1},\ \overline{t}_{\ell+1} = t_\ell$, and for every $j,\ j \neq \ell, \ell+1,\ \overline{t}_j = t_j$. T' is a *swapped version* of T if there is a sequence $T = T^1, \cdots, T^s = T'$ where T^{x+1} is derived from T^x via a swap and where each element t_ℓ is involved in no more than one swap derivation in the sequence.

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 for which there exists a swapped version T' of T where there is an exact occurrence of P in T' starting from i, i.e. where $t'_i = p_1, t'_{i+1} = p_2, \ldots, t'_{i+m-1} = p_m$.

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

Restriction: When matching a pattern P with swapped version T' at location i we do not allow the symbol t'_i to have participated in any swap. We call this the *Swapped Pattern Matching with Fixed Starting Symbol* problem.

The following lemma, proven in [2], assures us that this restriction does not limit generality.

Lemma 1 *The Swapped Pattern Matching with Fixed Starting Symbol can be solved in time $O(f(n,m))$ iff the Pattern Matching with Swaps problem can be solved in time $O(f(n,m))$.*

Further Restriction: As we did in the Swapped Pattern Matching with Fixed Starting Point, we may in addition restrict the Swapped Matching so that the ending point is not allowed to participate in a swap. We get the *Swapped Matching with Fixed Endpoints* problem.

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 Fixed 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 fixed endpoints version of the problem as convenience dictates.

3 Special Patterns - Linear Time Algorithm

In the next sections we will analyze special properties of the pattern and show that under many circumstances swapped pattern matching can be done significantly faster than our general algorithm. There are some special cases where the swapped matching problem can be solved by an extremely simple method in linear time. One of them is the case where the pattern is composed of runs whose length is not smaller than 4. We start with this case because of its simplicity. We later relax the restrictions on the minimum run-length and still provide an efficient algorithm.

Definitions:

1. Let $S = s_1 \ldots s_m$ be a string. We say that a *run of length l* starts at position i if $s_{i-1} \neq s_i, s_i = s_{i+1} = \cdots = s_{i+l-1}$, and $s_{i+l-1} \neq s_{i+l}$. We denote such a run by s_i^l.
   The run lengths can be no less than 1 and no longer than m.
2. The *run-length notation* of string S is $a_1^{\ell_1} a_2^{\ell_2} \cdots a_r^{\ell_r}$, where $a_i \neq a_{i+1},\ i = 1, \ldots, r-1$.

Note that if every run length in the pattern is at least 4, a swapped version of the text that matches the pattern has a very distinct look. Clearly swapping elements within a run does not change the text's appearance. The only time that swaps can have an effect is on a seam between runs. However, because the pattern has no runs of length smaller than 4, a swap would force a situation of the type *aababb*, i.e. two length 1 runs surrounded by runs of length 2 or more. In fact, this is the only possibility of a swapped text matching the pattern. This suggests the following algorithm.

Algorithm for Swapped Matching with Long Runs (Fixed Endpoints)

1. Scan the text T and construct text T' where every situation of the type $xxyxyy$ is converted to $xxxyyy$
2. Run any linear time exact matching algorithm on text T' and pattern P.
3. For every match found at location i declare it a swapped match if $p_1 = t_i$ and if $p_m = t_{i+m-1}$ in T.

End Algorithm

Time: $O(n)$.

Correctness: It follows from the discussion that every swapped match of P in the text is found in step 2. We still need to make sure that we didn't declare a match in step 2 for a location that doesn't have a swapped match. Once again, from the discussion it is clear that the only way that this can happen is on the endpoints of the pattern. In step 3 we verify which of those found have fixed endpoints.

4 Special Patterns - Runs of Length 3

Assumption: Let us further reduce the restriction on the maximum run-length in the pattern, and assume it is 3.

The following observation will be the key to a more efficient algorithm:

Observation 1 *Let $a_1^{\ell_1} a_2^{\ell_2} \cdots a_r^{\ell_r}$ be the run length notation of P, and let i be a text location where there is a swapped occurrence of P. If $\ell_j \geq 3,\ j = 1, \ldots, r$ then all the following locations do not swap (i.e. the pattern symbol exactly matches the text symbol in that location):*

$t_{i+1}, \ldots, t_{i+\ell_1-2}$
$t_{i+\ell_1+1}, \ldots, t_{i+\ell_1+\ell_2-2}$
$\vdots$
$t_{i+\ell_1+\ell_2+\cdots\ell_{r-1}+1}, \ldots, t_{i+\ell_1+\ell_2+\cdots\ell_r-2}$.

The above observation is true since there is no point in swapping a pair of equal symbols. It means that in every run, only the two symbols at the edge may be swapped. All the other symbols appear in the text and pattern in exactly the same location, call them the *fixed part*. Call the leftmost (rightmost) symbol in the fixed part in a run the *left (right) fixed edge*.

Example: Let $P = aaaabbbbcccaaabbbb = a^4b^4c^3a^3b^4$. Consider a swapped matching starting at text location i (we use T_i to denote the text substring $t_i \cdots t_{i+|P|-1}$). The connected locations have no swaps.

$$\begin{array}{ccccccccccccccccccc}
P = & a & a & a & a & b & b & b & b & c & c & c & a & a & a & b & b & b & b \\
 & & | & | & & & | & | & & & | & & & | & & & | & | & \\
T_i = & * & a & a & * & * & b & b & * & * & c & * & * & a & * & * & b & b & *
\end{array}$$

There are five fixed parts and accordingly five left (and right) fixed edges. a in the second pattern location is an example of a left fixed edge and b in the seventh location is an example of a right fixed edge.

This leads to the following idea. Let us first find all locations where there is an *exact matching* between the part of the pattern that does not swap and the respective text locations. We mark all starts of such matchings as *potential starts*. The second step is verifying that the edges are legal. Fortunately, the legality of the edge is a local property, as can be seen from the following lemma.

Lemma 2 (Fixed Edge Local Property) *Let P be a pattern whose shortest run has length at least 3. Then there is a swapped occurrence of P at location i in T iff there is an exact matching of the pattern fixed part and the corresponding text location, and for every left fixed edge t_j either* (i) $(t_j = t_{j-1})$ *or* (ii) $(t_j = t_{j-2})$ *and for every right fixed edge t_j either* (i) $(t_j = t_{j+1})$ *or* (ii) $(t_j = t_{j+2})$.

Proof: Note that condition (i) means there is no swap on the edge and condition (ii) means there is a swap. Also note that for every two consecutive runs, condition (i) of the rightmost fixed edge holds for a run iff condition (i) of the leftmost fixed edge holds for the following run. With this in mind the lemma is obvious, since it states that there is a swapped occurrence iff the symbols of the fixed part are not swapped, and for every edge, either there is a swap or there isn't one. $\square$

The importance of the above lemma is that it allows us to check whether a text pair is legal or not, *independently of the pattern.*

Before proceeding with our algorithm, we define one more tool that will be required.

Definition: Let $T = t_1 \cdots t_n$ and $P = p_1 \cdots p_m$ be a text and pattern over alphabet $\Sigma \cup \{\phi\}$, where $\phi \notin \Sigma$ represents a "don't care" symbol. The *String Matching with Don't Cares Problem* is that of finding all locations i in T where $\forall j,\ j=0,\ldots,m-1$ either $t_{i+j}=p_{1+j}$ or $t_{i+j}=\phi$ or $p_{1+j}=\phi$. In other words, this is the exact matching problem but there is a wildcard symbol that is permitted to match every other alphabet symbol.

Fischer and Paterson [5] showed a convolution-based algorithm for the string matching with don't cares problem whose time complexity is $O(n\log m)$ for alphabets of finite fixed size, and $O(n\log^2 m)$ for general alphabets (in a computation model where a word has at least $\log m$ bits).

Our algorithm is now complete. Let $P=a_1^{\ell_1}a_2^{\ell_2}\cdots a_r^{\ell_r}$, and let $\ell_j\geq 3,\ j=1,\ldots,r$. Let $\phi\notin\Sigma$ represent the "don't care" symbol.

Algorithm for Swapped Matching with Runs of length at least 3

1. **The fixed part:** Construct $P_{fixed}=\phi a_1^{\ell_1-2}\phi\phi a_2^{\ell_2-2}\phi\phi\cdots\phi\phi a_r^{\ell_r-2}\phi$. Do a string matching with don't cares of P_{fixed} in T. Declare every match a potential start of a swapped occurrence of P in T.
2. **The leftmost fixed edges:**
   (a) Construct new text T' where
   $$t'_j=\begin{cases}1, & \text{if } j{=}2 \text{ and } t_1=t_2\\ & \text{or if } j\geq 3 \text{ and } [(t_j=t_{j-1})\ \vee\ (t_j=t_{j-2})];\\ 0, & \text{otherwise.}\end{cases}$$
   (b) Construct $P_{left-edge}=\phi 1\phi^{\ell_1-1}1\phi^{\ell_2-1}1\cdots\phi^{\ell_{r-1}-1}1$.
   (c) Do a string matching with don't cares of $P_{left-edge}$ in T'. For every match j, declare j a potential start of a swapped occurrence of P in T.
3. **The rightmost fixed edges:**
   (a) Construct new text T' where
   $$t'_j=\begin{cases}1, & \text{if } j{=}n-1 \text{ and } t_{n-1}=t_n\\ & \text{or if } j\leq n-2 \text{ and } [(t_j=t_{j+1})\ \vee\ (t_j=t_{j+2})];\\ 0, & \text{otherwise.}\end{cases}$$
   (b) Construct $P_{right-edge}=\phi^{\ell_1-2}1\phi^{\ell_2-1}1\cdots\phi^{\ell_{r-1}-1}1\phi^{\ell_r-1}1$.
   (c) Do a string matching with don't cares of $P_{right-edge}$ in T'. For every match j, declare j a potential start of a swapped occurrence of P in T.
4. The intersection of the potential starts in the fixed part step, the leftmost edges step and the rightmost edges step is exactly the set of starts of all swapped matchings of P in T.

End Algorithm

Correctness: Immediate from the fixed edge local property lemma.

Time: $O(n\log m)$ for the edge pairs step and $O(n)$ for the intersection. For constant size Σ, the fixed part step is also done in time $O(n\log m)$. For general alphabets the fixed part step takes time $O(n\log^2 m)$. We conclude:

Total Algorithm Time: $O(n\log m)$ for finite fixed size alphabets. $O(n\log^2 m)$ for general alphabets.

Note: The algorithm can be easily modified to find all swapped matches of a pattern with don't care symbols as long as all runs of alphabet symbols ($\neq\phi$) are of length at least three and the runs of don't care symbols are at least of length two.

5 Special Patterns - Runs of Length 2

We further reduce our restriction to patterns where the minimum run has length 2. At this point we no longer have any fixed part to the pattern. For example, if all runs have length 2, every element may swap. However, in such a dire situation we take advantage of another pattern property.

The following lemma establishes conditions under which it is possible to concatenate two strings and assume that if there are consecutive swapped appearances of each, there is a swapped appearance of the concatenated string. This is the local property we will need for patterns with runs of length two.

Lemma 3 (Concatenating Patterns Property) *Let $S = s_1 \cdots s_m$, $S' = s'_1 \cdots s'_{m'}$, be two strings where $s_{m-1} = s_m = \sigma$ and $s'_1 = s'_2 = \rho$. Assume that there is a swapped matching of S starting at location i of text T, and that there is a swapped matching of S' starting at location $i+m$ of the text.*
Let $P = SS'$, i.e. P is the concatenation of S and S'. Then there exists a swapped matching of P starting at location i of T.

Proof: We assume that there is a sequence of text swaps that allows matching S starting at location i, and *another* sequence of text swaps that allows matching S' starting at location $i+m$. The lemma says that these two sequences do not conflict. In fact, we will show that both S and S' have to use exactly the same text swaps on the seam (locations $i+m-1$ and $i+m$). If this is the case, we are done since the other swaps have no affect upon each other.

Since $s_{m-1} = s_m$ there can not be a swap between t_{i+m-2} and t_{i+m-1}. For the same reason t_{i+m} and t_{i+m+1} can not swap. In fact t_{i+m-1} and t_{i+m} can only swap with each other.
Therefore, if $t_{i+m-1} = \sigma$ then t_{i+m} must equal ρ and there will be no external swaps in the set of swaps of the matchings of S and S'. Otherwise if $t_{i+m-1} \neq \sigma$ it forces t_{i+m} to be σ and t_{i+m-1} to be ρ. In this case there must be external swaps in both S and S'. This swap serves P. $\square$

This leads us to an algorithm for patterns with run-length 2 only. The idea is as follows: for each $\sigma \in \Sigma$ scan the text and mark each location that has a swapped matching of $\sigma\sigma$ with 1 and every other location 0. Mark each pattern location in which $\sigma\sigma$ begins with 1 and every other pattern location with ϕ. Now simply do pattern matching with don't cares. Merging this with the algorithm of the previous section yields an algorithm with time complexity $O(|\Sigma| n \log m)$ and correctness following from the Concatenating Patterns Property.

We now show a method that eliminates the $|\Sigma|$ factor from the complexity. If at each text location there is at most one type of pair that has a swapped matching then we can adapt the idea we just displayed by building a new text containing at each location σ if $\sigma\sigma$ matches with swaps at this location. Then apply don't care pattern matching with pattern $p_1 \phi p_3 \cdots p_{n-1} \phi$.
In general there may be more than one type of pair with a swapped matching at any text location but clearly not more than two. If both $\sigma\sigma$ and $\rho\rho$ have a swapped matching at text location i then the text beginning at text location $i-1$ must be of the form $\sigma\rho\rho\sigma$, $\sigma\rho\sigma\rho$, $\rho\sigma\rho\sigma$ or $\rho\sigma\sigma\rho$. This leads us to the following lemma:

Lemma 4 (Concatenating Sandwiched Pairs Property) *Let $S = s_1 \cdots s_m$, $S' = s'_1 \cdots s'_{m'}$ and $U = u_0^2 u_1^2 \ldots u_k^2$, such that $s_{m-1} = s_m \neq u_0$ and $u_k \neq s'_1 = s'_2$ (for notation, we assume $s_m = u_{-1}$ and $s'_1 = u_{k+1}$). Assume that there is a swapped matching of S starting at location i of text T, and that there is a swapped matching of S' starting at location $i+m+2k+2$ of the text.*
Let $P = SUS'$ then P has a swapped matching at text location i iff for every $0 \leq j \leq k$

a. if $u_{j-1} = u_{j+1}$ then both, and only, the pairs u_j^2 and u_{j-1}^2 have a swapped matching at text location $i+m+2j$.

b. if $u_{j-1} \neq u_{j+1}$ then u_j^2 is the only pair that has a swapped matching at text location $i+m+2j$.

Proof: The 'only if' side follows from induction on the concatenating patterns property.
Therefore, let us assume that $P = SUS'$ has a swapped matching at text location i. Let $0 \leq j \leq k$. Clearly u_j^2 must match with swaps at text location $i+m+2j$. But since $p_{m+2j+1} = p_{m+2j+2}$ $(= u_j)$, t_{i+m+2j} can not swap with $t_{i+m+2j+1}$. Following the same reasoning $t_{i+m+2j+2}$ and $t_{i+m+2j+3}$ cannot swap and likewise for $t_{i+m+2j-2}$ and $t_{i+m+2j-1}$. The only swaps possible between these elements is a swap between $t_{i+m+2j-1}$ and t_{i+m+2j} and a swap between $t_{i+m+2j+1}$ and $t_{i+m+2j+2}$. This means that the only other pair that can match with swaps at location $i+m+2j$ is u_{j-1}^2 and this happens only if $u_{j-1} = u_{j+1}$. $\square$

We are now ready for our algorithm. The *concatenating patterns property* assures us that text locations that have a match with swaps for all runs of length at least 3 and have a match with swaps for runs of length 2 separately are exactly the text locations where the whole pattern has a swapped matching.
Checking for locations where the runs of length 3 match with swaps is done using the algorithm from the previous section. For the runs of length 2 we use the *concatenating sandwiched pairs property*. The idea is to create a new text with symbols corresponding to the letters of the pairs that have a swapped matching at that text location, more precisely if at text location i no pair matches with swaps then t'_i will be 0, if only one pair matches with swaps, e.g. 'aa', t'_i will be 'a' and if two pairs have a swapped matching, e.g. 'aa' and 'cc', t'_i will be the new symbol $< a, c >$ which is formed from the corresponding letters in lexicographic order.
We also create a new pattern with a symbol at the beginning of every pair depending on the neighboring pairs. If the neighboring pairs are of different types then we leave the letter already there, e.g. for 'bb aa cc' the first 'a' remains. If, on the other hand, the neighboring pairs are of the same type we replace it with a new symbol consisting of a letter from each type, e.g. for 'cc aa cc' the first 'a' will be replaced by $< a, c >$. The second symbol is always replaced by a don't care symbol. The matches of the new pattern in the new text capture the locations at which the length 2 runs have a swapped matching. Note that even though we have created a special pattern we still allow swaps only in the text.

The pattern we are handling is of the form $P = a_1^{\ell_1} B_1 a_2^{\ell_2} B_2 \cdots a_{r-1}^{\ell_{r-1}} B_{r-1} a_r^{\ell_r}$ where
$a_i \in \Sigma, \quad i = 1, \ldots, r.$
$\ell_i \geq 3, \quad i = 2, \ldots, r-1.$
$\ell_i = 0$ or $\ell_i \geq 3, \quad i = 1, r.$
$B_i =$ substrings over Σ where all runs are of length *exactly* 2. The first symbol of B_i is not equal to a_i, and the last symbol of B_i is not equal to a_{i+1}. The length of B_i is $h_i, \quad i = 1, \ldots, r-1$. Note that B_i may be empty, i.e. $h_i = 0$. In this case $a_i \neq a_{i+1}$.

For the sake of simplicity we also assume $\ell_1 \neq 0$ and $\ell_r \neq 0$, since this is the case in the concatenated sandwiched pair property. The only addition to the algorithm below for cases where the pattern starts (ends) with a run of length two, is the following. Create pattern P_d by deleting the first (last) two letters of the pattern P. Run the algorithm below for finding all locations of P_d and then, for each location found, check the leading (trailing) pair of characters. Retain only the locations where the leading (trailing) pair swap-match as well.

Example for the case of $\ell_1 \neq 0$ and $\ell_r \neq 0$:

$$
\begin{array}{lccccccccccccccc}
T = & c & c & a & c & b & a & b & a & c & a & a & c & c & d & e & \cdots \\
T' = & c & c & c & a & b & <a,b> & a & a & a & <a,c> & 0 & c & 0 & \cdot & \cdot & \cdots \\
P = & c & c & c & a & a & b & b & a & a & c & c & a & a & a & & \\
P' = & & & & a & \phi & <a,b> & \phi & a & \phi & <a,c> & \phi & & & & &
\end{array}
$$

As can be seen, there is an exact matching with "don't cares" of P' starting at location 4 of T'. Note that P' represents only the run-length-two part of P.

Algorithm for Swapped Matching with Runs of length at least 2

1. **Runs of length at least 3:** Construct $P' = a_1^{\ell_1} \phi^{h_1} a_2^{\ell_2} \phi^{h_2} \cdots \phi^{h_{r-1}} a_r^{\ell_r}$. Run the algorithm for swapped matching with runs of length at least 3 and find all text locations where there is a swapped match of P'.

2. **Runs of length 2 exactly:**

(a) Construct new text T' where

$$t'_j = \begin{cases} 0, & \text{if there is no swapped match of } \sigma\sigma \text{ for any } \sigma \in \Sigma \text{ starting at location } j; \\ \sigma, & \text{if } \sigma \text{ is the only letter for which there exists a swapped} \\ & \text{match of } \sigma\sigma \text{ starting at location } j; \\ <\sigma, \rho>, & \text{if } \sigma < \rho \text{ and } \sigma\sigma \text{ and } \rho\rho \text{ both have a swapped matching at text location } j. \end{cases}$$

(b) Construct pattern P' where

$p'_j = \phi$ for all p_j that are not in the B_i parts of the pattern, $i = 1, \ldots, r-1$. For the elements of B_i, $i = 1, \ldots, r-1$, with first element p_{k_i+2j}, we set $p'_{k_i+2j+1} = \phi$, and

$$p'_{k_i+2j} = \begin{cases} <p_{k_i+2j-1}, p_{k_i+2j}>, & \text{if } p_{k_i+2j-1} = p_{k_i+2j+2} \text{ and } p_{k_i+2j-1} < p_{k_i+2j}; \\ <p_{k_i+2j}, p_{k_i+2j-1}>, & \text{if } p_{k_i+2j-1} = p_{k_i+2j+2} \text{ and } p_{k_i+2j-1} > p_{k_i+2j}; \\ p_{k_i+2j}, & \text{otherwise.} \end{cases}$$

(c) Do a string matching with don't cares of P' in T'.

3. The intersection of the potential starts in the length 3 runs step and the potential starts in the length 2 runs step is exactly the desired set of starts of all swapped matchings of P in T.

End Algorithm

Correctness: Follows from the concatenating sandwiched pairs property and the concatenating patterns property.

Time: $O(n \log m)$ for finite fixed size alphabets, $O(n \log^2 m)$ for general alphabets.

6 Conclusions and Open Problem

We have shown that for all cases where the minimum run in the pattern is at least 2 there is a $O(n \log^2 m)$ algorithm. Yet, for the general case, the best known algorithm is $O(nm^{1/3} \text{ polylog } m)$. We believe that the general swapped pattern matching problem can be solved in time $O(n \text{ polylog } m)$.

References

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

[2] A. Amir, Y. Aumann, G. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Proc. 38th IEEE FOCS*, pages 144–153, 1997.

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

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

[5] 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.

[6] K. Hashiguchi and K. Yamada. Two recognizable string-matching problems over free partially commutative monoids. *Theoretical Computer Science*, 92(1):77–86, 1992.

[7] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *Symposium on the Theory of Computing*, 4:125–136, 1972.

[8] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

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

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

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

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

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

[14] R. Y. Pinter. Efficient string matching with don't care patterns. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 11–29. NATO ASI Series F, 1985.

[15] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.