Approximate Swapped Matching

Amihood Amir* Bar-Ilan University and Georgia Tech

Moshe Lewenstein† Bar-Ilan University

Ely Porat‡ Bar-Ilan University and Weizmann Institute

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* P' of P is a length m string derived from P by a series of *local swaps*, (i.e. $p'_\ell \leftarrow p_{\ell+1}$ and $p'_{\ell+1} \leftarrow p_\ell$) where each element can participate in *no more than one swap*. The *Pattern Matching with Swaps* problem is that of finding all locations i of T for which there exists a swapped version P' of P with an exact matching of P' in location i of T.

Recently, some efficient algorithms were developed for this problem. Their time complexity is better than the best known algorithms for pattern matching with mismatches. However, the *Approximate Pattern Matching with Swaps* problem was not known to be solved faster than the pattern matching with mismatches problem.

In the *Approximate Pattern Matching with Swaps* problem the output is, for every text location i where there is a swapped match of P, the *number* of swaps necessary to create the swapped version that matches location i. The fastest known method to-date is that of counting mismatches and dividing by two. The time complexity of this method is $O(n\sqrt{m \log m})$ for a general alphabet Σ.

In this paper we show an algorithm that counts the number of swaps at every location where there is a swapped matching in time $O(n \log m \log \sigma)$, where $\sigma = min(m, |\Sigma|)$. Consequently, the total time for solving the approximate pattern matching with swaps problem is $O(f(n,m) + n \log m \log \sigma)$, where $f(n,m)$ is the time necessary for solving the pattern matching with swaps problem. Since $f(n,m)$ was shown to be $O(n \log m \log \sigma)$ this means our algorithm's running time is $O(n \log m \log \sigma)$.

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

*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-01-04494, ISF grant 282/01, and a BIU internal research grant.

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; `moshe@cs.biu.ac.il`. Partially supported by the Israel Ministry of Science Eshkol Fellowship 061-1-97. Part of this work was done while the author was visiting the Courant Institute at NYU.

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

1 Introduction

The *Pattern Matching with Swaps* problem (the *Swap Matching* problem, for short) requires finding all occurrences of a pattern of length m in a text of length n. The pattern is said to match the text at a given location i if adjacent pattern characters can be swapped, if necessary, so as to make the pattern identical to the substring of the text starting at location i. All the swaps are constrained to be disjoint, i.e., each character is involved in at most one swap.

The importance of the swap matching problem lies in recent efforts to understand the complexity of various *generalized pattern matching* problems. The textbook problem of *exact string matching* that was first shown to be solvable in linear time by Knuth, Morris and Pratt [11] does not answer the growing requirements stemming from advances in Multimedia, Digital libraries and Computational Biology. To this end, pattern matching has to adapt itself to increasingly broader definitions of "matching" [19, 18]. 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.

The above applications motivated research of two new types – *Generalized Pattern Matching*, and *Approximate Pattern Matching*. 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 the *string matching with don't cares* problem defined by Fischer and Paterson [9]. Another example of a generalized matching problem is the *less-than matching* [5] problem defined by Amir and Farach. In this problem both text and pattern are numbers. One seeks all text locations where every pattern number is less than its corresponding text number. Amir and Farach showed that the less-than-matching problem can be solved in time $O(n\sqrt{m}\log m)$.

Muthukrishnan and Ramesh [16] 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. As we have seen, some examples have significantly worse upper bound than this.

The swap matching problem is also a generalized matching problem. It arises from one of the edit operations considered by Lowrance and Wagner [15, 20] to define a distance metric between strings.

Amir et al [2] obtained the first non-trivial results for this problem. They showed how to solve the problem in time $O(nm^{1/3}\log m \log \sigma)$, where $\sigma = \min(|\Sigma|, m)$. Amir et al. [6] also give certain special cases for which $O(m\text{polylog}(m))$ time can be obtained. However, these cases are rather restrictive. Cole and Hariharan [7] give a randomized algorithm that solves the swap matching problem over a binary alphabet in time $O(n \log n)$. Finally, Amir et al [4] solve the swap matching problem over general alphabet in time $O(n \log m \log \sigma)$.

The second important pattern matching paradigm is that of *approximate matching*. Even under the appropriate matching relation there is still a distinction between *exact matching* and *approximate matching*. In the latter case, a distance function is defined on the text. A text location is considered a match if the distance between it and the pattern, under the given distance function, 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 couple of decades.

The earliest and best known distance function is Levenshtein's *edit distance* [14]. The edit distance between two strings is the smallest number of edit operations, in this case insertions, deletions, and mismatches, whereby one string can be converted to the other. Let n be the text length and m the pattern length. A straightforward $O(nm)$ dynamic programming algorithm computes the edit distance between the text and pattern. Lowrance and Wagner [15, 20] proposed an $O(nm)$ dynamic programming algorithm for the extended edit distance problem, where the swap edit operation is added. In [10, 12, 13] $O(kn)$ algorithms are given for the edit distance with only k allowed edit operations. Recently, Cole and Hariharan [8] presented an $O(nk^4/m + n + m)$ algorithm for this problem.

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. By methods similar to those of Fischer and Paterson [9] it can be shown that the string matching with mismatches problem can be solved in time $O(\min(|\Sigma|, m)n \log m)$. For given finite alphabets, this is $O(n \log m)$. Abrahamson [1] developed an algorithm that solves this problem for general alphabets in time $O(n\sqrt{m \log m})$.

The *approximate pattern matching with swaps* problem considers the swaps as the only edit operation and seeks to compute, for each text location i, the number of swaps necessary to convert the pattern to the substring of length m starting at text location i (provided there is a swap match at i). In [3] it was shown that the approximate pattern matching with swaps problem can be reduced to the string matching with mismatches problem. For every location where there is a swap match, the number of swaps is precisely equal to half the number of mismatches (since a swap is two mismatches). Although swap matching as a generalized matching proved to be more efficient than counting mismatches, it remained open whether swap matching as an approximation problem can be done faster than mismatches.

In this paper we answer this question in the affirmative. We show that if all locations where there is a swap match are known, the approximate swap matching problem can be solved in time $O(n \log m \log \sigma)$, where $\sigma = \min(|\Sigma|, m)$. Therefore, since swap matching can be done in time $O(n \log m \log \sigma)$ [4], approximate swap matching can be done in time $O(n \log m \log \sigma)$.

Paper organization. This paper is organized in the following way. In section 2 we give basic definitions. In sections 3, we outline the key idea and intuition behind our algorithm. In section 4 we give a randomized algorithm, which easily highlights the idea of our solution. It turns out that rather than using a generic derandomization strategy, a simple, problem specific, method can be used to obtain the deterministic counterpart. Section 5 presents an easy and efficient code that solves our problem deterministically.

2 Problem Definition

Definition 1: Let $S = s_1 \ldots s_n$ be a string over alphabet Σ. A *swap permutation* for S 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 $s_{\pi(i)} \neq s_i$ (identical characters are not swapped).

For a given string $S = s_1 \ldots s_n$ and swap permutation π for S we denote $\pi(S) = s_{\pi(1)} s_{\pi(2)} \ldots s_{\pi(n)}$. We call $\pi(S)$ a *swapped version* of S.

The *number of swaps* in swapped version $\pi(S)$ of S is the number of pairs $(i, i+1)$ where $\pi(i) = i+1$ and $\pi(i+1) = i$.

For pattern $P = p_1 \ldots p_m$ and text $T = t_1 \ldots t_n$, we say that P *swap matches at location* i if there exists a swapped version P' of P that matches T starting at location i, i.e. $p'_j = t_{i+j-1}$ for $j = 1, \ldots, m$. It is not difficult to see that if P swap matches at location i there is a *unique* swap permutation for that location.

The *Swap Matching Problem* is the following:
INPUT: Pattern $P = p_1 \ldots p_m$ and text $T = t_1 \ldots t_n$ over alphabet Σ.
OUTPUT: All locations i where P swap matches T.

We note that the definition in [2] and the papers that followed is slightly different, allowing the swaps in the text rather than the pattern. However, it follows from Lemma 1 in [2] that both versions are of the same time complexity.

The *Approximate Swap Matching Problem* is the following:
INPUT: Pattern $P = p_1 \ldots p_m$ and text $T = t_1 \ldots t_n$ over alphabet Σ.
OUTPUT: For every location i where P swap matches T, write the number of swaps in the swapped version of P that matches the text substring of length m starting at location i. If there is no swap matching of P at i, write $m+1$ at location i.

Observation 1 *Assume there is a swap match at location i. Then the number of swaps is equal to half the number of mismatches at location i.*

3 Intuition and Key Idea

It would seem from Observation 1 that finding the number of swaps is of the same difficulty as finding the number of mismatches. However, this is not the case. Note that if it is known that there is a swap match at location i, this puts tremendous constraints on the mismatches. It means that if there is a mismatch between pattern location j and text location $i+j-1$ we also know that either $t_{i+j-1} = p_{j-1}$ or $t_{i+j-1} = p_{j+1}$. There is no such constraint in a general mismatch situation!

Since we can "anticipate" for every pattern symbol what would be the mismatch, it gives us some flexibility to change the alphabet to reflect the expected mismatches. Thus we are able to reduce the alphabet to one with a small constant size. For such alphabets, the Fischer and Paterson algorithm [9] allows counting mismatches in time $O(n \log m)$.

In order to be able to anticipate the mismatching symbol, we need to isolate every pattern symbol from its right and left neighbors. This can be done by splitting a pattern P to three patterns, P_1, P_2 and P_3, where each subpattern counts mismatches only in the central element of each triple. P_1, P_2 and P_3 represent the three different offsets of triples in the pattern. For a schema of these three patterns, see Figure 1.

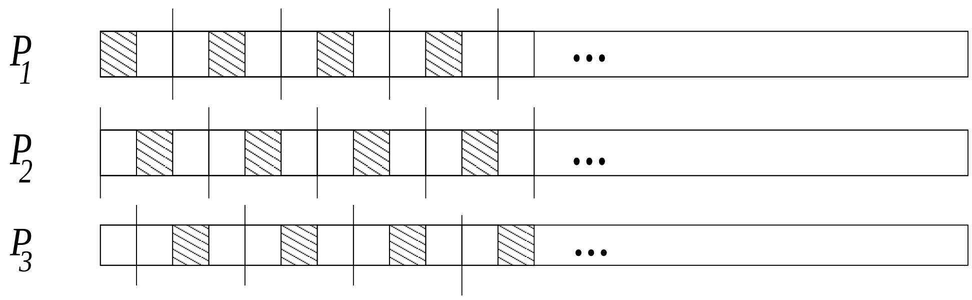

Figure 1: The three patterns resulting from different triple offsets.

For each one of P_1, P_2 and P_3, the central symbol in every triple (the one shaded in Figure 1) has the same value as the respective element of P. All other symbols are "don't care"s (ϕ). The sum of the mismatches of P_i in T, $i = 1, 2, 3$ is precisely the mismatches of P in T. Therefore, half of this sum is the desired number of swaps.

Throughout the remainder of this paper we will concentrate on counting the mismatches of P_2. The cases of P_1 and P_3 are similar. In the next section we will show a randomized algorithm that allows efficient counting of mismatches of P_2 by reducing the alphabet. Section 5 will show a deterministic alphabet reduction.

4 Randomized Alphabet Reduction

Let $h : \Sigma \rightarrow \{1, 2, ..., 4\}$ be chosen randomly. For string $S = s_1, ..., s_m$ define $h(S) = h(s_1), ..., h(s_m)$. Consider $h(P_2)$. Let (x, y, z) be a triple such that $x \neq y$ or $y \neq z$ (i.e. a swap *could* happen). Call such a triple a *potential swap triple*. We say that h *separates* the triple (x, y, z) if $h(x) \neq h(y)$ when $x \neq y$ and $h(y) \neq h(z)$ when $y \neq z$.

If h happens to separate every potential swap triple in the pattern, then the number of mismatches of P_2 in T equals the number of mismatches of $h(P_2)$ in $h(T)$. However, the alphabet of $h(P_2)$ and $h(T)$ is of size 4, hence the mismatches can be counted in time $O(n \log m)$.

We need to be quite lucky to achieve the situation where all potential swap triples get separated by h. However, we really do not need such a drastic event. Every potential swap triple that gets separated, counts all its mismatches. From now on it can be replaced by "don't care"s and never add mismatches. Conversely, every potential swap triple that does not get separated can be masked by "don't care"s and not contribute anything.

Our algorithm, then, is the following.

Algorithm

Let $P_t \leftarrow P_2$
Replace all non potential swap triples of P_t with "don't care"s
while not all triples have been masked do:
  choose a random $h : \Sigma \rightarrow \{1, 2, ..., 4\}$
  Let $P_q \leftarrow h(P_t)$
  Replace all non-separated triples of P_q with "don't care"s
  Count all mismatches of P_q in $h(T)$
  Replace all triples of P_t that were separated by h with "don't care"s

end Algorithm

Since counting all mismatches of P_q in $h(T)$ can be done in time $O(n \log m)$, it is sufficient to know the expected number of times we run through the while loop to calculate the expected running time of the algorithm.

Claim 1 *The expected number of times the* while *loop executed in the above algorithm is* $O(\log \sigma)$, *where* $\sigma = \min(|\Sigma|, m)$.

Proof: The probability that a given potential swap triple gets separated is

$$\left(\frac{3}{4}\right)^2 = \frac{9}{16} > \frac{1}{2}.$$

Therefore, the expectation is that at least half of the triples will be separated in the first execution of the while loop, with every subsequent execution of the while loop separating half of the remaining triples. Since there are no more than $\min(\frac{m}{3}, |\Sigma|^3)$ triples, then the expectation is that in $O(\log(\min(\frac{m}{3}, |\Sigma|^3)) = O(\log \sigma)$ executions of the while loop all triples will be separated. $\square$

Conclude: The expected running time of the algorithm is $O(n \log m \log \sigma)$.

5 Deterministic Alphabet Reduction

Recall that our task is really to separate all triples. There exists in the literature a powerful code that does this separation. Subsequently we show a simple code that solves our problem.

Definition 2: A $(\Sigma, 3)$-*universal set* is a set $S = \{\chi_1, \ldots, \chi_k\}$ of characteristic functions, $\chi_j : \Sigma \rightarrow \{0, 1\}$ such that for every $a, b, c \in \Sigma$, and for each of the eight possible combinations of $0-1$s, there exists χ_j such that $\chi_j(a), \chi_j(b), \chi_j(c)$ equals this combination.

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

Let $S = \{\chi_1, \ldots, \chi_k\}$ be a $(\Sigma, 3)$-universal set such that for every potential swap triple (a, b, c) there exist a j for which $\chi_j(a) = 0, \chi_j(b) = 1, \chi_j(c) = 0$. Note that we will later show (in Claim 2) that the requirement of this particular Boolean triple can be relaxed. However, the fact is that a $(\Sigma, 3)$-universal set does provide this power.

We run the following algorithm, which is very similar to the randomized algorithm in Section 4.

Algorithm

Let $P_t \leftarrow P_2$
Replace all non potential swap triples of P_t with "don't care"s
for $j = 1$ to k do:
    Let $P_q \leftarrow \chi_j(P_t)$
    Replace all non-separated triples of P_q with "don't care"s
    Count all mismatches of P_q in $\chi_j(T)$
    Replace all triples of P_t that were separated by χ_j with "don't care"s

end Algorithm

In [17] it was shown how to construct $(\Sigma, 3)$-universal set of cardinality $k = O(\log \sigma)$ yielding the following.

Corollary 1 *The deterministic algorithm's running time is $O(n \log m \log \sigma)$.*

The Naor and Naor construction of [17] is quite heavy. We conclude with an extremely simple coding of the alphabet that separates triples sufficiently well for our purposes.

First note the following.

Claim 2 *It is sufficient to provide a set $S = \{\chi_1, \ldots, \chi_k\}$ of characteristic functions, $\chi_j : \Sigma \rightarrow \{0,1\}$ such that for every potential swap triple (a,b,c) there either exists a χ_j such that $\chi_j(a) = x, \chi_j(b) = 1-x$ and $\chi_j(c) = x$, where $x \in \{0,1\}$, or there exist χ_{j_1}, χ_{j_2} such that $\chi_{j_1}(a) = x, \chi_{j_1}(b) = 1-x$ and $\chi_{j_1}(c) = 1-x$, and $\chi_{j_2}(a) = y, \chi_{j_2}(b) = y$ and $\chi_{j_2}(c) = 1-y$, where $x, y \in \{0,1\}$.*

Call such a set a swap separating set.

Proof: Let $S = \{\chi_1, \ldots, \chi_k\}$ be a swap separating set. Every potential swap triple for which there exists a χ_j such that $\chi_j(a) = x, \chi_j(b) = 1-x$ and $\chi_j(c) = x$, where $x \in \{0,1\}$, will be separated by χ_j and masked with "don't care"s for all other characteristic functions. In other words, we initially decide, for each function, what are the triples it separates, and mark those triples. If several functions separate the same triple we will, of course, only use one of them.

For every other potential swap triple, there are χ_{j_1}, χ_{j_2} such that $\chi_{j_1}(a) = x, \chi_{j_1}(b) = 1-x$ and $\chi_{j_1}(c) = 1-x$, and $\chi_{j_2}(a) = y, \chi_{j_2}(b) = y$ and $\chi_{j_2}(c) = 1-y$, where $x, y \in \{0,1\}$. Every such triple will participate in the separation of χ_{j_1} **and** χ_{j_2}. For all other characteristic functions it will be masked with "don't care"s.

Note that if there is a match of such a triple with a text location without a swap, then neither χ_{j_1} nor χ_{j_2} will contribute a mismatch. However, if the triple's match requires a swap, then **exactly** one of χ_{j_1} or χ_{j_2} will contribute a mismatch. □

Our remaining task is to provide a simple construction for swap separating set of size $O(\log \sigma)$.

Swap Separating Set Construction

Consider a $\sigma \times \log \sigma$ bit matrix B where the rows are a binary representation of the alphabet elements $(P \cap \Sigma)$. Take $\chi_j(a) = B[a, j]$. In words, the characteristic functions are the columns of B.

For every potential swap triple (a, b, c), if there is a column where the bits of a, b, c are $x, 1-x, x$ then this column provides the function in which the triple participates. If no such column exists, then there clearly are two columns j_1, j_2 such that $B[a, j_1] \neq B[b, j_1]$ and $B[c, j_2] \neq B[b, j_2]$. It is clear that $B[c, j_1] = B[b, j_1]$ and $B[a, j_2] = B[b, j_2]$ (otherwise the first condition holds). The columns j_1 and j_2 provide the functions where triple (a, b, c) participates.

6 Conclusion and Open Problems

We have shown a faster algorithm for the approximate swap matching problem than that of the pattern matching with mismatches problem. This is quite a surprising result considering that it was thought that swap matching may be even harder than pattern matching with mismatches. However, this leads us to conjecture that the current upper bound on the mismatches problem $(O(n\sqrt{m \log m}))$ is not the final word.

The swap operation and the mismatch operation have proven to be relatively "easy" to solve. However, insertion and deletion are still not known to be solvable in time faster than the dynamic programming $O(nm)$ in the worst case. A lower bound or a better upper bound on the complexity of edit distance would be of great interest.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 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, Y. Aumann, G. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37:247–266, 2000. (Preliminary version appeared at FOCS 97.).

[4] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 279–288, 2001.

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

[6] A. Amir, G.M. Landau, M. Lewenstein, and N. Lewenstein. Efficient special cases of pattern matching with swaps. *Information Processing Letters*, 68(3):125–132, 1998.

[7] R. Cole and R. Harihan. Randomized swap matching in $o(m \log m \log |\sigma|)$ time. Technical Report TR1999-789, New York University, Courant Institute, September 1999.

[8] 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.

[9] 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.

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

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

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

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

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

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

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

[17] J. Naor and M. Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comp.*, pages 838–856, 1993.

[18] M. V. Olson. A time to sequence. *Science*, 270:394–396, 1995.

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

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