Theoretical Computer Science 411 (2010) 3814–3822

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

The approximate swap and mismatch edit distance☆

Yair Dombb, Ohad Lipsky, Benny Porat, Ely Porat*, Asaf Tsur

Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel

ARTICLE INFO

Article history:
Received 22 September 2009
Received in revised form 10 April 2010
Accepted 6 June 2010
Communicated A. Apostolico

Keywords:
Pattern matching
Swap
Edit operations
Mismatch

ABSTRACT

There is no known algorithm that solves the general case of the *approximate edit distance* problem, where the edit operations are insertion, deletion, mismatch, and swap, in time $o(nm)$, where n is the length of the text and m is the length of the pattern.

In the effort to study this problem, the edit operations have been analyzed independently. Karloff [10] showed an algorithm that approximates the edit distance problem with only the mismatch operation in time $O(\frac{1}{\epsilon^2} n \log^3 m)$. Amir et al. [4] showed that if the only edit operations allowed are swap and mismatch, then the exact edit distance problem can be solved in time $O(n\sqrt{m}\log m)$.

In this paper, we discuss the problem of *approximate edit distance with swap and mismatch*. We show a randomized $O(\frac{1}{\epsilon^3} n \log n \log^3 m)$ time algorithm for the problem. The algorithm guarantees an approximation factor of $(1+\epsilon)$ with probability of at least $1-\frac{1}{n}$.

1. Introduction

Approximate string matching is a widely studied area in computer science. In approximate matching, one defines a distance metric between the objects (e.g. strings, matrices) and seeks all text locations where the pattern matches the text within a pre-specified "small" distance.

The earliest and best known distance functions are Levenshtein's *edit distance* [11] and the *Hamming distance*. Let n be the text length and m the pattern length. Lowrance and Wagner [14,15] proposed an $O(nm)$ dynamic programming algorithm for the extended edit distance problem. In [13] the first $O(kn)$ algorithm was given for the edit distance with only k allowed edit operations. Cole and Hariharan [7] presented an $O(nk^4/m + n + m)$ algorithm for this problem. To this moment, however, there is no known algorithm that solves the general case of the extended edit distance problem in time $\tilde{o}(nm)$.

Since the upper bound problem 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 has been done on finding efficient algorithms for string matching with mismatches [1,12,6]. The most efficient deterministic worst-case algorithm for finding the Hamming distance of the pattern at every text location runs in time $O(n\sqrt{m\log m})$. Karloff [10] presented an $O(\frac{1}{\epsilon^2} n \log^3 m)$ time algorithm that approximates the Hamming distance with a factor of $1+\epsilon$. Attalah et al. [2] presented a randomized algorithm for estimating the number of matches in each text location in time $O(kn\log m)$ with variance of $(m-c_i)/k$. Isolating the swap edit operation yielded even better results [3,5], with a worst-case running time of $O(n\log m\log\sigma)$.

Amir et al. [4] faced the challenge of integration of the above two results. Integration has proven tricky since various algorithms often involve different techniques. For example, there are efficient algorithms for string matching with don't

☆ This research was supported by BSF and ISF.
* Corresponding author. Tel.: +972 3 531 8866; fax: +972 3 736 0498.
E-mail addresses: dombby@cs.biu.ac.il (Y. Dombb), ohadlipsky@yahoo.com (O. Lipsky), bennyporat@gmail.com (B. Porat), porately@cs.biu.ac.il (E. Porat), zurasa@cs.biu.ac.il (A. Tsur).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.06.004

cares (e.g. [9]) and efficient algorithms for indexing exact matching (e.g. [16]); both are over 30 years old. Yet there is no known efficient algorithm for indexing with don't cares. In fact, sometimes the integration of two efficiently solvable operations ends up intractable. For example, Wagner [15] proves that edit distance with the *two* operations *insertion* and *swap* is $\mathcal{NP}$-hard, while each case separately can be solved in polynomial time and the general edit distance – consisting of the *four* operations *insertion*, *deletion*, *mismatch* and *swap* – is also polynomially solvable.

In this context, [4] provided an efficient algorithm for the edit distance with *two* operations: *mismatch* and *swap*. Their algorithm runs in time $O(n\sqrt{m}\log m)$.

In this paper we discuss the problem of approximating the edit distance with only swap and mismatch operations allowed. We think that this result is essential for a complete analysis of the "swap and mismatch edit distance" problem. We present an $O(\frac{1}{\epsilon}n\log m)$ time algorithm for a binary alphabet. For a small alphabet, one can use the binary algorithm $|\Sigma|^2$ times (consider each possible pair of symbols separately). We show that in general alphabets the problem can be solved in $O(\frac{1}{\epsilon}n\log n\log^3 m)$ time independently of the size of the alphabet Σ. We also show that the problem of (approximately) counting mismatches linearly reduces to that of the (approximate) swap and mismatch edit distance. The current best time algorithm for approximately counting mismatches runs in time $O(\frac{1}{\epsilon^2}n\log^3 m)$ [10].

The techniques used by the algorithm are novel cases of overlap matching [3] and convolutions as well as a new bounded divide and conquer approach for alphabet size reduction.

2. Problem definition

Definition 1. Let Σ be some finite alphabet. An *edit operation* E is simply a function $E : \Sigma^* \to \Sigma^*$. Let OP be a set of edit operations, and suppose $S, T \in \Sigma^*$. The *edit distance from T to S* (with respect to the set OP) is the minimum number k such that there exist a sequence of k edit operations $\langle E_1, \ldots, E_k\rangle$ for which $E_k(E_{k-1}(\cdots E_1(T)\cdots)) = S$.

Example. The Lowrance and Wagner edit operations are $\{INS_{i,\sigma}, DEL_i, REP_{i,\sigma}$, and $SWAP_i\}$, where
$INS_{i,\sigma}(s_1 \ldots s_n) = s_1 \ldots s_i, \sigma, s_{i+1} \ldots s_n$,
$DEL_i(s_1 \ldots s_n) = s_1 \ldots s_{i-1}, s_{i+1} \ldots s_n$,
$REP_{i,\sigma}(s_1 \ldots s_n) = s_1 \ldots s_{i-1}, \sigma, s_{i+1} \ldots s_n$, and
$SWAP_i(s_1 \ldots s_n) = s_1 \ldots s_{i-1}, s_{i+1}, s_i, s_{i+2} \ldots s_n$.

Definition 2. Let $T = t_1 \ldots t_n$ be a *text* string, and $P = p_1 \ldots p_m$ a *pattern string* over alphabet Σ. The *edit distance problem* of P in T is that of computing, for each $i = 1, \ldots, n$, the minimum edit distance from P to a prefix of $t_i \ldots t_n$.

Lowrance and Wagner [14,15] give an $O(nm)$ algorithm for computing the edit distance problem with the above four edit operations. To date, no better algorithm is known for the general case. We consider the following problem.

Definition 3. The *swap and mismatch edit distance 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 each $i = 1, \ldots, n$, compute the minimum edit distance from P to a prefix of $t_i \ldots t_n$, where the edit operations are $\{REP_{i,\sigma}, SWAP_i\}$.

The following observation plays an important role in our algorithm.

Observation 1. *Every swap operation can be viewed as two replacement operations.*

2.1. Convolutions

Convolutions are used for filtering in signal processing and other applications. A convolution uses two initial functions, T and P, to produce a third function $T \otimes P$. We formally define a discrete convolution.

Definition 4. Let T be a function whose domain is $\{1, \ldots, n\}$ and P a function whose domain is $\{1, \ldots, m\}$. We may view T and P as arrays of numbers, whose lengths are n and m, respectively. The *discrete convolution of T and P* is the polynomial multiplication $T \otimes P$, where

$$(T \otimes P)[j] = \sum_{i=1}^{m} T[j+i-1]P[i], \quad j = 1, \ldots, n-m+1.$$

In the general case, the convolution can be computed by using the Fast Fourier Transform (FFT) [8] on T and P^R, the reverse of P. This can be done in time $O(n\log m)$, in a computational model with word size $\Omega(\log m)$.

The crucial property contributing to the usefulness of convolutions is the following. For every fixed location j^* in T, we are, in essence, overlaying P on T, starting at j^*, i.e. $P[1]$ corresponds to $T[j^*]$, $P[2]$ to $T[j^*+1]$, and in general $P[i]$ corresponds to $T[j^*+i-1]$. We multiply each element of P by its corresponding element of T and add all m resulting products; the result of this process is the convolution's value at location j^*.

Clearly, computing the convolution's value for every text location j, can be done in time $O(nm)$. The fortunate property of convolutions over algebraically closed fields is that they can be computed *for all* n text locations in time $O(n\log m)$ using the

FFT. In the next few sections we will show how this property of convolutions can be used to efficiently compute relations of patterns in texts. This will be done via *linear reductions* to convolutions. In the definition below, $\mathbb{N}$ represents the natural numbers and $\mathbb{R}$ represents the real numbers.

Definition 5. Let P be a pattern of length m and T a text of length n over some alphabet Σ. Let $R(S_1, S_2)$ be a relation on strings of length m over Σ. We say that the relation R *holds between* P *and location* j *of* T if $R(P[0]\cdots P[m-1], T[j]\cdots T[j+m-1])$.

We say that R is *linearly reduced* to convolutions if there exist a natural number c, a constant time computable function $f : \mathbb{N}^c \to \{0, 1\}$, and linear time functions $\ell_1^m, \ldots, \ell_c^m$ and $r_1^n, \ldots, r_c^n$, $\forall n, m \in \mathbb{N}$, where $\ell_i^m : \Sigma^m \to \mathbb{R}^m$, $r_i^n : \Sigma^n \to \mathbb{R}^n$, $i = 1, \ldots, c$, such that R holds between P and location j in T iff $f(\ell_1^m(P) \otimes r_1^n(T)[j], \ell_2^m(P) \otimes r_2^n(T)[j], \ldots, \ell_c^m(P) \otimes r_c^n(T)[j]) = 1$.

Let R be a relation that is linearly reduced to convolutions. It follows immediately from the definition that, using the FFT to compute the c convolutions, it is possible to find all locations j in T where relation R holds in time $O(n \log m)$.

Example. Let $\Sigma = \{a, b\}$ and R be the equality relation. The locations where R holds between P and T are the locations j where $T[j + i - 1] = P[i]$, $i = 1, \ldots, m$. Fischer and Patterson [9] showed that it can be computed in time $O(n \log m)$ by the following trivial reduction to convolutions.

Let $\ell_1 = \chi_a$, $\ell_2 = \chi_b$, $r_1 = \chi_{\bar{a}}$, $r_2 = \chi_{\bar{b}}$ where

$$\chi_\sigma(x) = \begin{cases} 1, & \text{if } x = \sigma \\ 0, & \text{otherwise,} \end{cases} \quad \text{and} \quad \chi_{\bar{\sigma}}(x) = \begin{cases} 1, & \text{if } x \neq \sigma \\ 0, & \text{otherwise} \end{cases}$$

and where we extend the definition of the functions χ_σ to strings in the usual manner, i.e. for $S = s_1 s_2 \ldots s_n$, we have $\chi_\sigma(S) = \chi_\sigma(s_1)\chi_\sigma(s_2)\ldots\chi_\sigma(s_n)$.

Let

$$f(x, y) = \begin{cases} 1, & \text{if } x = y = 0 \\ 0, & \text{otherwise.} \end{cases}$$

Then for every text location j, $f(\ell_1(P) \otimes r_1(T)[j], \ell_2(P) \otimes r_2(T)[j]) = 1$ iff there is an exact matching of P at location j of T.

3. The algorithm for binary alphabets

For the sake of simplicity, we begin with solving the problem for the binary alphabet $\Sigma = \{0, 1\}$; we later show how to handle larger alphabets. When considering a binary alphabet, a swap operation can be effective only in cases where the text has a pair 10 aligned with an 01 in the pattern, or vice versa. Therefore, we are interested in finding alternating sequences of zeros and ones. We define this concept formally, since it is the key to the algorithm's idea.

An alternating segment of a string $S \in \{0, 1\}^*$ is a substring alternating between 0's and 1's. A maximal alternating segment, or *segment* for short, is an alternating segment such that the character to the left of the leftmost character x in the alternating segment, if any, is identical to x, and similarly, the character to the right of the rightmost character y, if any, is identical to y. In other words, a maximal segment is an alternating segment that is not contained in any other (larger) segment. Note that any string over $\Sigma = \{0, 1\}$ can be represented as a concatenation of segments.

Example. Let

$$S = 10100011101010011010011101010 1;$$

then S's segments are: 1010 0 01 1 101010 01 1010 01 1 1010101.

We need to distinguish the cases where aligned text and pattern segments match via swap operations only from the cases where replacements are also necessary. The following lemma, proven in [3], presents the key property necessary to reduce swap matching to overlap matching.

Lemma 1. *The pattern does not (swap-)match in a particular alignment if and only if there exists a segment A in the text and a segment B in the pattern such that* (1) *the characters of A and B misalign in the overlap, and* (2) *the overlap is of odd length.*

The conditions of the above lemma are also useful for our problem.

Lemma 2. *The number of mismatches that are not part of a swap is exactly the number of the overlaps for which conditions* (1) *and* (2) *of Lemma 1 hold.*

Proof. We will examine all possibilities:

1. Condition (1) of the lemma does not hold. Then there is no misalignment of the text. Indeed it matches the pattern.
2. Condition (1) holds but condition (2) does not. According to Lemma 1 there is a swap match.
3. If the two conditions hold then either one of the two segments A and B is entirely contained in the other, or the overlap is a real substring of both segments. For the first case we may assume, without loss of generality, that segment B of the pattern is contained in segment A of the text (the other case is treated in a similar fashion). The situation is that there is a misalignment and the overlap is of odd length. Schematically, we have (with B and A boldfaced):

Text: $\cdots$ **0101** $\cdots$ **1010** $\cdots$
Pattern: $\cdots$ 0**010** $\cdots$ **010**0 $\cdots$

Since swapping B's edges will not help; the only swaps possible are internal to B. This means that there is exactly one element that remains mismatched after the swap.

The other situation is where the overlap is a real substring of both segments. We assume that B starts before A (the other case is handled in a similar fashion). The situation is:

Text: $\cdots$ 0**010** $\cdots$ **0101** $\cdots$
Pattern: $\cdots$ **0101** $\cdots$ **101**1 $\cdots$

Again it is clear that the only possible swaps are internal to the (odd length) overlap, leaving one element mismatched even after all possible swaps. □

The outline of our algorithm is as follows:

1. For every text location i, count the number of mismatches m_i of the pattern starting at location i using two convolutions as described in [9].
2. Partition the text and pattern into segments by parity of starting and ending locations, and by length.
3. For every text location i, count the number r_i of odd length misaligned overlaps, allowing an error of $\pm\epsilon m_i$. An odd length misaligned overlap causes exactly one "real" (non-swap) mismatch.
4. The approximate number of swap errors at location i is $s_i = (m_i - r_i)/2$,
5. The approximate number of swap and mismatch edit errors at location i is $r_i + s_i$.

3.1. Grouping text and pattern segments

We follow some of the implementation ideas of [3]. However, while in their work it was only necessary to check whether odd length mismatched overlaps exist, here we need to count them as well. The main idea that we use is to separate the segments of the text and pattern into a small number of groups. In each of these groups it will be possible to count the required overlaps in time $O(\frac{1}{\epsilon} n \log m)$ using a limited divide and conquer scheme based on polynomial multiplications (convolutions). In the subsections that follow we handle the different cases. Some of these cases necessitate new and creative uses of convolutions.

As Lemma 1 implies, we are interested in two properties of every overlap between a text segment and a pattern segment, namely, whether the characters of the segments misalign, and if so, the length of the overlap. We first divide all the segments in the text and pattern into two types. We say that a segment is a 1-*anchored segment* if it has the 1's in the odd locations, and that it is a 0-*anchored segment* if it has the 0's in the odd locations. The following observation is immediate.

Observation 2. *If the pattern is aligned against the text at an odd location, it is sufficient to check overlaps of* 0*-anchored segments in the text with* 1*-anchored segments in the pattern, and vice versa. If the pattern is aligned against the text at an even location, it is sufficient to check overlaps where the segments of the pattern and text have the same anchor.*

For checking the parity of the overlap, we need to know whether a text segment ends at an odd or even text location. Consequently, we define new texts where each text has *exactly* those segments of a given start and end parity, with all other text elements defined as ϕ (*don't care*) which consequently never contribute an error. (In the polynomial multiplication there will always be a 0 in these text locations.) Henceforth, for short, we will talk of multiplying a text and pattern, by which we mean a polynomial multiplication, where each of the text and pattern is viewed as a vector of coefficients for the corresponding polynomials.

Definition 6. Let $T = t_1 \ldots t_n$ be a string. We define T_0^{oo} as the string of length n in which location i is 1 if t_i is part of a 0-anchor segment that begins and ends in an odd location, and ϕ otherwise. Similarly, T_1^{oo} is the string of length n in which location i is 1 if t_i is part of a 1-anchored segment that begins and ends in an odd location, and ϕ otherwise. In the same manner we define T_0^{ee} and T_1^{ee} as strings that contain 1's only in segments (with anchor 0 or 1, respectively) that begin and end in even locations. Likewise, we define T_0^{eo}, T_1^{eo}, T_0^{oe} and T_1^{oe}.

Example. Let $T = 10100011101010011010011101010 1$. The strings T_0^{oo}, T_1^{oo}, T_0^{ee}, T_1^{ee}, T_0^{oe}, T_1^{oe}, T_0^{eo} and T_1^{eo} are presented below (the spaces show the division into segments):

$$
\begin{array}{lccccccccc}
T_0^{oo} = & \phi\phi\phi\phi & 1 & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi & \phi\phi & \phi\phi\phi\phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi\phi \\
T_1^{oo} = & \phi\phi\phi\phi & \phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi & \phi\phi & \phi\phi\phi\phi & \phi\phi & 1 & \phi\phi\phi\phi\phi\phi\phi \\
T_0^{ee} = & \phi\phi\phi\phi & \phi & \phi\phi & 1 & \phi\phi\phi\phi\phi\phi & \phi\phi & \phi\phi\phi\phi & \phi\phi & \phi & 1111111 \\
T_1^{ee} = & \phi\phi\phi\phi & \phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi & \phi\phi & \phi\phi\phi\phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi\phi \\
T_0^{oe} = & \phi\phi\phi\phi & \phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi & 11 & \phi\phi\phi\phi & 11 & \phi & \phi\phi\phi\phi\phi\phi\phi \\
T_1^{oe} = & 1111 & \phi & \phi\phi & \phi & 111111 & \phi\phi & 1111 & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi\phi \\
T_0^{eo} = & \phi\phi\phi\phi & \phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi & \phi\phi & \phi\phi\phi\phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi\phi \\
T_1^{eo} = & \phi\phi\phi\phi & \phi & 11 & \phi & \phi\phi\phi\phi\phi\phi & \phi\phi & \phi\phi\phi\phi & \phi\phi & \phi & \phi\phi\phi\phi\phi\phi\phi
\end{array}
$$

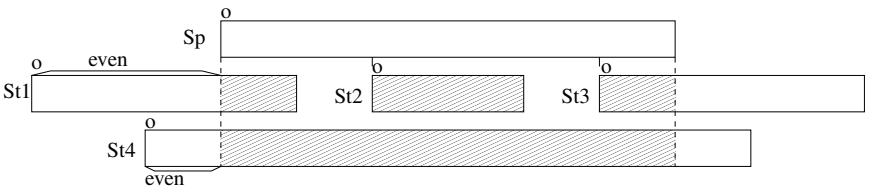

Fig. 1. The cases of both text and pattern segments starting in locations with the same parity.

We now create the strings T_0^{oo}, T_1^{oo}, T_0^{ee}, etc. from the original text string T and likewise the strings P_0^{oo}, P_1^{oo}, P_0^{ee}, etc. from the original pattern string P. As Observation 2 indicates, when considering alignments of the pattern at odd locations we only need to handle segments with different anchors, and for alignments of the pattern at even locations we need only need to handle segments with the same anchor. We can therefore omit the subscript indicating the segment anchors of the text and pattern, since those are chosen according to the alignment location, and have no effect on the algorithm itself.

We are now down to the combinations $T^{a,b}$ and $P^{\alpha,\beta}$ (with $a, b, \alpha, \beta \in \{o, e\}$), which gives us 16 cases. However, many cases are similar, and so we need to consider only cases of three types:

1. $T^{a,b}$ and $P^{\alpha,\beta}$ where $a = \alpha$ or $b = \beta$. (This type covers 12 cases.) These cases are handled in Section 4.
2. T^{oe} and P^{eo}, or T^{eo} and P^{oe}. These cases are handled in Section 5.
3. T^{oo} and P^{ee}, or T^{ee} and P^{oo}. These cases are handled in Section 6.

4. Segments with equal parity start or end

Consider the case of $T^{a,b}$ and $P^{\alpha,\beta}$ where $a = \alpha$.

Observation 3. *For every two segments, S_t in $T^{a,b}$, starting at location x, and S_p in $P^{\alpha,\beta}$, starting at location y, $|x - y|$ is always even (Fig. 1).*

We are interested in the number of odd overlaps. We now show a convolution for which the resulting value at location i is n exactly if there are n odd length overlaps with the pattern starting at location i. (For the convolution with T^{eo} (or T^{oe}, P^{eo}, P^{oe}) we need to do two convolutions, the first with T_1^{eo}, and the second with T_2^{eo}.)

The convolution. Pattern $P' = p'_1 \cdots p'_m$ is constructed as follows:

$$p'_i = \begin{cases} 0, & \text{if } P^{\alpha,\beta}[i] = \phi \\ 1, & \text{otherwise.} \end{cases}$$

The text $T' = t'_1 \cdots t'_n$ is constructed by replacing every ϕ in $T^{a,b}$ by 0, and every segment in $T^{a,b}$ by a segment of alternating 1's and -1's, starting with 1. Then P' and T' are convolved.

Lemma 3. *Let $(T' \otimes P')[q]$ be the qth element in the result of the convolution. $(T' \otimes P')[q]$ is equal to the number of odd overlaps of the relevant text and pattern segments.*

Proof. In the convolution we sum a sequence of alternating 1's and -1's for every overlap. It follows from Observation 3 that every such sequence begins with 1; thus, each overlap of odd length contributes exactly 1 to the total sum, while even length overlaps contribute 0. □

We thus obtain that locations q with $(T' \otimes P')[q] = 0$ are locations with no odd overlaps between the relevant text and pattern segments. This solves all eight cases for $T^{a,b}$ and $P^{\alpha,\beta}$ where $a = \alpha$. For the additional four cases where $b = \beta$ we simply reverse the text and pattern to obtain the case considered above. Note that this gives us the *exact* number of odd length misaligned overlaps of segments with equal parity start or end.

5. The odd–even even–odd segments

Consider the case of T^{oe} and P^{eo} (the case where T^{eo} and P^{oe} are symmetric).

Terminology. Let S_t be a text segment whose starting location is s_t and whose ending location is f_t, and let S_p be a pattern segment being compared to the text at starting position s_p and ending position f_p. If $s_t < s_p < f_p < f_t$ then we say that S_t *contains* S_p. If $s_p < s_t < f_t < f_p$ then we say that S_p *contains* S_t. If $s_t < s_p < f_t < f_p$ then we say that S_t has a *left overlap* with S_p. If $s_p < s_t < f_p < f_t$ then we say that S_t has a *right overlap* with S_p. We will sometimes refer to a left or right overlap as a *side overlap*.

Observation 4. *For every two segments S_t in T_x^{oe} and S_p in P_y^{eo}, if either S_p is contained in S_t or S_t is contained in S_p then the overlap is of even length. If the overlap is a left overlap or right overlap then it is of odd length. All possible cases are shown in Fig. 2 below.*

Note that all segments of these types have even length; thus, if one segment contains the other, the overlap is necessarily of even length. Conversely, in the case of side overlaps, the overlap starts and ends at locations with the same parity, making the length of the overlap odd.

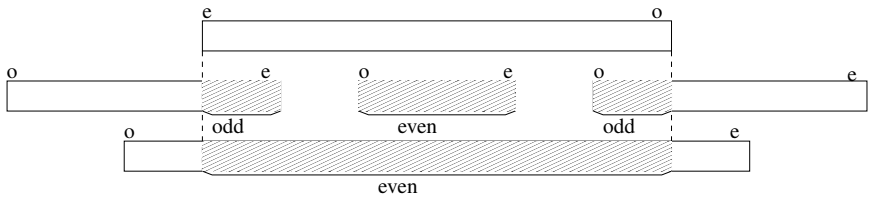

Fig. 2. The cases where the text segment starts at an odd location and ends at an even location, and the pattern segment does the opposite.

Convolution idea. Recall that our goal is to count all locations with odd length overlap between segments. Suppose we create the pattern $P' = p'_1 \dots p'_m$ by setting

$$p'_i = \begin{cases} 0, & \text{if } P^{\alpha,\beta}[i] = \phi \\ 1, & \text{otherwise} \end{cases}$$

and the text $T' = t'_1 \dots t'_n$ by replacing the first and last symbols of every segment in the text with 1's, and all the remaining symbols with 0. Assume that there is an overlap between a text segment S_t and a pattern segment S_p. We have three cases: If it is a side overlap, it contributes exactly 1 to the sum. If S_p is contained in S_t, the overlap contributes 0 to the sum. However, if S_t is contained in S_p, the overlap contributes 2 to the sum, which harms the accuracy of the count.

In order to avoid the last case, we treat segments of different length separately. We use the convolution above only for cases where the pattern segments are not larger than the text segments; for the other case we will use a different (symmetric) convolution.

5.1. Grouping text and pattern segments by length

In order to avoid the problem above, as well as a similar problem in the next section, we group the text and pattern segments by length, in addition to the grouping by the parity of their start and end locations.

Definition 7. For every length ℓ we define a string $T^{a,b,\ell}$ as follows:

$$T^{a,b,\ell}[i] = \begin{cases} 1, & \text{if } T^{a,b}[i] = 1 \text{ and } i \text{ is part of a segment of length } \ell \\ \phi, & \text{otherwise.} \end{cases}$$

Likewise, we define $T^{a,b,\leq\ell}$ and $T_x^{a,b,>\ell}$ with

$$T^{a,b,\leq\ell}[i] = \begin{cases} 1, & \text{if } T^{a,b}[i] = 1 \text{ and } i \text{ is part of a segment of length} \leq \ell \\ \phi, & \text{otherwise} \end{cases}$$

and

$$T^{a,b,>\ell}[i] = \begin{cases} 1, & \text{if } T^{a,b}[i] = 1 \text{ and } i \text{ is part of a segment of length} > \ell \\ \phi, & \text{otherwise.} \end{cases}$$

We similarly define $P^{\alpha,\beta,\ell}$, $P^{\alpha,\beta,\leq\ell}$, and $P^{\alpha,\beta,>\ell}$.

5.2. The convolutions for the odd–even even–odd segments case

We now return to the case of T^{oe} and P^{eo}. Like we said earlier, we will actually need two convolutions for this case. For now we denote the text and pattern simply by T and P, and we later specify the exact lengths that we will use with each convolution.

Convolution 1. The pattern $P' = p'_1 \dots p'_m$ is constructed by setting

$$p'_i = \begin{cases} 0, & \text{if } P[i] = \phi \\ 1, & \text{otherwise.} \end{cases}$$

The text $T' = t'_1 \dots t'_n$ is created by replacing the first and last symbols of every segment in the text with 1's, and all the remaining symbols with 0. (This is exactly the convolution that we described above.)

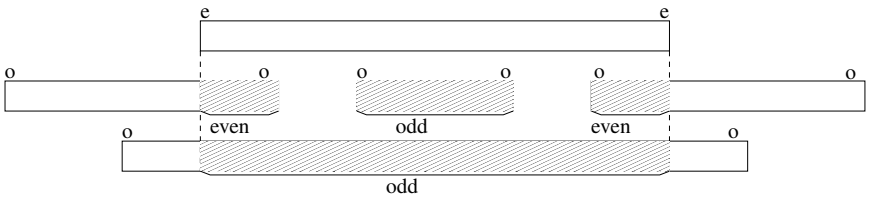

Fig. 3. Every containment has odd length; every side overlap has even length.

Convolution 2. The text $T' = t'_1 \dots t'_n$ is constructed by setting

$$t'_i = \begin{cases} 0, & \text{if } T = \phi \\ 1, & \text{otherwise.} \end{cases}$$

The pattern $P' = p'_1 \dots p'_n$ is created by replacing the first and last symbols of every segment in the pattern with 1's, and all the remaining symbols with 0. Like in convolution 1, every side overlap contributes 1 to the sum. However, here an overlap in which the text segment contains the pattern segment contributes 0, and overlaps in which the pattern segment contains the pattern segment contribute 2.

Using these convolutions for counting the exact number of real mismatches will require treating every length of a pattern segment separately; however, this might take up to $O(\sqrt{m})$ convolutions. In order to decrease the number of convolutions, we will define $L = \frac{4}{\epsilon}$ and treat all segments of length L or larger as one length.

For each length $\ell < L$ we will use convolution 1 to count real mismatches between $T^{oe,\ell}$ and $P^{eo,\leq\ell}$. Since we are comparing text segments only to shorter pattern segments, convolution 1 counts the exact number of mismatches. Also, we will use convolution 2 to count real mismatches between $T^{oe,<\ell}$ and $P^{eo,\ell}$. The total number of convolutions for this part is at most $\frac{8}{\epsilon}$.

In addition, we will use convolution 1 once to count all real mismatches between $T^{oe,>L}$ and $P^{eo,\leq L}$, and use convolution 2 once for counting all real mismatches between $T^{oe,\leq L}$ and $P^{eo,>L}$. Finally, we will use convolution 1 once more to count all real mismatches between $T^{oe,>L}$ and $P^{eo,>L}$. This convolution will count 2 mismatches for every occurrence where S_t is contained in S_p, but note that every such overlap (which is at least $\frac{4}{\epsilon}$ characters long) has at least $\frac{2}{\epsilon}$ swap errors, so the relative error when counting *swap and mismatch* errors will be at most ϵ.

6. The odd–odd even–even segments

Consider the case T^{oo} and P^{ee} (the case T^{ee} and P^{oo} is symmetric).

Observation 5. *For every two segments S_t in T^{oo} and S_p in P^{ee}, if either S_p is contained in S_t or S_t is contained in S_p then the overlap is of odd length. If the overlap is a left overlap or right overlap then it is of even length. All possible cases are shown in Fig. 3 below.*

Segments of these types have odd lengths, and thus if one contains the other then the overlap is necessarily of odd length. Conversely, in the case of a left or right overlap, the overlap starting and ending locations have opposite parity, making the length of the overlap even.

6.1. The convolutions for the odd–odd even–even segments case

The convolution. Text $T' = t'_1 \cdots t'_n$ is constructed as follows:

$$t'_i = \begin{cases} 0, & \text{if } T^{a,b}[i] = \phi \\ 1, & \text{otherwise.} \end{cases}$$

The pattern $P' = p'_1 \dots p'_m$ is constructed by replacing every ϕ in $P^{\alpha,\beta}$ by 0, and every segment in $P^{\alpha,\beta}$ by a segment of alternating 1's and -1's, starting with 1.

Assume that there is an overlap between a text segment S_t and a pattern segment S_p. If the overlap is a side overlap, it will contribute nothing to the sum. If S_p is contained in S_t, it will contribute 1 to the sum. However, if S_t is contained in S_p, it will contribute -1 to the sum.

In order to avoid miscalculations caused by the last case, we will, like in the odd–even even–odd case, treat different sizes of text and pattern segments separately. In this case, we will define $L = \frac{2}{\epsilon}$. For each length $\ell < L$ we will use our convolution to count real mismatches between $T^{oo,\ell}$ and $P^{ee,\leq\ell}$. Since we are comparing text segments only to shorter pattern segments, our convolution counts the exact number of mismatches. Also, we will use our convolution to count real mismatches between $T^{oo,\leq\ell}$ and $P^{ee,\ell}$ (we will get a negative total, and take its absolute value). The total number of convolutions for this part is at most $\frac{4}{\epsilon}$.

Finally, we will use this convolution once to count all real mismatches between $T^{oo,\leq L}$ and $P^{ee,>L}$, and again for counting all real mismatches between $T^{oo,>L}$ and $P^{ee,\leq L}$ (taking the absolute value of the result, like before).

Note that we are not doing the convolution for the case where the text *and* pattern segments are longer than $\frac{2}{\epsilon}$. In this case, real mismatches only occur in cases where text segments are contained in pattern segments (or vice versa), and since all such segments are of length of at least $\frac{2}{\epsilon}$ characters, they are already contributing $\frac{1}{\epsilon}$ swap errors (in addition to the 1 real mismatch that we overlooked); and the relative error when counting *swap and mismatch* errors will be at most ϵ.

7. Approximate solution for a general alphabet

First, we note that in the general alphabet case, every two different symbols that appear in adjacent places somewhere in the pattern define a different *segment type*. To reduce the number of segment types that we handle, we will project the segment types set into a smaller set of size $\frac{1}{\epsilon}$. Once the number of different segment types is bounded, we can use the binary algorithm for each type separately and sum up the results.

Let $S \subseteq \Sigma \times \Sigma$ be the set of different segment types in the pattern, where a segment type is determined only by the alternating two symbols. Note that $|S| \leq m$. We can construct an algorithm that approximates the swap mismatch edit distance in $O(|S|f(n, m, \epsilon) + mis(n, m, \epsilon))$ where $f(n, m, \epsilon)$ is the time needed to obtain a $(1 + \epsilon)$-approximation of the binary alphabet swap and mismatch problem and $mis(n, m, \epsilon)$ is the time needed to approximate counting mismatches. The algorithm will work as follows:

Few Segment Types Algorithm:

1. For every text location $i = 1, \ldots, n - m + 1$, approximately count the number of mismatches M_i of the pattern starting at location i [10].
2. For each segment type (σ_1, σ_2),
   (a) Replace all the symbols in segments of other types with a ϕ symbol.
   (b) Compute s_i, the number of swap errors with segment type (σ_1, σ_2) for every text location $i = 1, \ldots, n - m + 1$, using the algorithm for the binary alphabet case.
   (c) Add s_i to S_i for every $i = 1, \ldots, n - m + 1$.
3. The approximate swap mismatch edit distance at location i is $M_i - S_i$.

Lemma 4. *The algorithm $(1+\epsilon)$-approximates the swap mismatch edit distance and works in time $O(|S|\frac{1}{\epsilon}n \log m + \frac{1}{\epsilon^2}n \log^3 m)$.*

Proof. The inexact parts of this algorithm are the first step of counting mismatches and the use of the binary alphabet case algorithm. Both are guaranteed to have $(1 + \epsilon)$-approximation factor and the approximation is kept under addition operations. We now discuss the time needed for each step. The first step is done using Karloff's algorithm in $O(\frac{1}{\epsilon^2}n \log^3 m)$. For each segment type computing s_i's can be done by counting the number of mismatches using convolutions (since there are only two symbols, this can be done in $O(n \log m)$ time) and running the algorithm for the binary alphabet case which takes $O(\frac{1}{\epsilon}n \log m)$. □

Many Segment Types Algorithm:

1. Randomly choose $\frac{1}{\epsilon} \log n$ projections $\Pi_j : S \to S'$ where $|S'| = \frac{1}{\epsilon}$.
2. For each Π_j replace segments according to Π_j and approximately count swap and mismatch errors using the algorithm for few segment types.
3. Compute the approximate swap and mismatch edit distance by taking the average between the different results.

Observation 6. *Every pair of segments (in the text and in the pattern) that caused* swap *mistakes before the projection will also cause swap mistakes after the projection. Pairs of segments that caused* mismatch *mistakes before the projection might cause mismatch mistakes* or *swap mistakes after the projection. For a given pair, the probability of the projection changing the type of mistakes caused by that pair from mismatch mistakes to swap mistakes is ϵ.*

The correctness of the observation is immediate, since every pair of segment types that caused swap mistakes before the projection must have been a pair of two identical segment types. Therefore, such a pair will be projected to a pair of two identical segment types and will continue to cause swap mistakes after the projection. Pairs of segment types that caused a mismatch mistake must have been a pair of two different segment types; the probability for such a pair to be projected to a pair of two identical segment types is exactly ϵ.

Lemma 5. *The above algorithm correctly $(1 + \epsilon)$-approximates the swap and mismatch edit distance with error probability less than $\frac{1}{n^3}$.*

Proof. Using Observation 6 we have that the swap errors before the projection are necessarily swap errors after the projections. Furthermore, other mismatch errors have the probability of less than ϵ of changing to either matches or swap errors. We use $\frac{1}{\epsilon} \log n$ independent projections. Using Chernoff inequality we get that the probability that in more than $5 \log n$ projections a specific error will change is less than $\frac{1}{n^5}$. $Pr[X > 5 \log n] \leq (\frac{e^4}{(5)^5})^{\log n} < \frac{1}{n^5}$.

The above probability is for a specific location and error. Using the union bound we get a total error probability of at most $\frac{1}{n^3}$. $5 \log n$ is a 5ϵ fraction of $\frac{1}{\epsilon} \log n$ projections; taking $\epsilon' = \epsilon/5$ yields the desired approximation ratio. □

Time complexity. For each projection we need $O(\frac{1}{\epsilon^2} n \log^3 m)$ time. In total our algorithm works in $O(\frac{1}{\epsilon^3} n \log n \log^3 m)$ time complexity.

8. Reduction from *mismatch* to *swap and mismatch*

Theorem 1. *Let A be an algorithm that solves the problem of* swap and mismatch *running in time $O(f(n))$. Then there is an algorithm A' that solves* mismatch *in time $O(n + f(n))$.*

Proof. Let $T = t_1 \dots t_n$ be a text, and $P = p_1 \dots p_n$ be a pattern over some alphabet Σ. We want to find the Hamming distance (i.e. the number of *mismatches*) between the pattern and every location in the text. Let $\psi \notin \Sigma$ and define $T' = t_1 \psi t_2 \psi \dots \psi t_n$; i.e. T' is simply T with the new symbol ψ inserted between every two adjacent characters. Define P' in a similar manner. Now, run the algorithm A on T' and P' and return the result of the odd locations.

It is immediate to observe that all the mistakes that A finds are *mismatches*, since the new ψ symbols between every two adjacent characters of the original text and pattern make swap mistakes impossible. Also note that the additional ψ's don't contribute any additional mismatches, since each ψ in the pattern is aligned with a ψ in the text when we consider only odd locations. We have thus obtained an algorithm A' that solves *mismatch* in time $O(n + f(n))$, as stated. □

References

[1] Karl R. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (6) (1987) 1039–1051.
[2] Mikhail J. Atallah, Frédéric Chyzak, Philippe Dumas, A randomized algorithm for approximate string matching, Algorithmica 29 (3) (2001) 468–486.
[3] Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, Ely Porat, Overlap matching, Inf. Comput. 181 (1) (2003) 57–74.
[4] Amihood Amir, Estrella Eisenberg, Ely Porat, Swap and mismatch edit distance, in: Proc. of the 12th Annual European Symposium on Algorithms, ESA'04, in: Lecture Notes in Computer Science, vol. 3221, Springer, 2004, pp. 16–27.
[5] Amihood Amir, Moshe Lewenstein, Ely Porat, Approximate swapped matching, Inf. Process. Lett. 83 (1) (2002) 33–39.
[6] Amihood Amir, Moshe Lewenstein, Ely Porat, Faster algorithms for string matching with k mismatches, J. Algorithms 50 (2) (2004) 257–275.
[7] R. Cole, R. Hariharan, Approximate string matching: a faster simpler algorithm, in: Proc. 9th ACM-SIAM Symposium on Discrete Algorithms, SODA, 1998, pp. 463–472.
[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, McGraw-Hill, 1992.
[9] M.J. Fischer, M.S. Paterson, String matching and other products, in: R.M. Karp (Ed.), Complexity of Computation, in: SIAM-AMS Proceedings, vol. 7, 1974, pp. 113–125.
[10] H. Karloff, Fast algorithms for approximately counting mismatches, Inf. Process. Lett. 48 (2) (1993) 53–60.
[11] V.I. Levenshtein, Binary codes capable of correcting, deletions, insertions and reversals, Sov. Phys. Dokl. 10 (1966) 707–710.
[12] G.M. Landau, U. Vishkin, Efficient string matching with k mismatches, Theoret. Comput. Sci. 43 (1986) 239–249.
[13] G.M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, J. Algorithms 10 (2) (1989) 157–169.
[14] R. Lowrance, R.A. Wagner, An extension of the string-to-string correction problem, J. ACM (1975) 177–183.
[15] R.A. Wagner, On the complexity of the extended string-to-string correction problem, in: Proc. 7th ACM STOC, 1975, pp. 218–223.
[16] P. Weiner, Linear pattern matching algorithm, in: Proc. 14 IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.