Information and Computation 208 (2010) 1020–1030

Contents lists available at ScienceDirect

Information and Computation

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

String matching with up to k swaps and mismatches

Ohad Lipsky[a], Benny Porat[a], Ely Porat[a,*], B. Riva Shalom[b], Asaf Tzur[a]

[a] *Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel*
[b] *Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel*

ARTICLE INFO

Article history:
Received 18 2 2008
Revised 4 March 2010
Available online 5 May 2010

Keywords:
Pattern matching
Swap

ABSTRACT

Finding the similarity between two sequences is a major problem in computer science. It is motivated by many issues from computational biology as well as from information retrieval and image processing. These fields take into account possible corruptions of the data caused by genome rearrangements, typing mistakes, and more. Therefore, many applications do not require merely complete resemblance of the sequences, but rather an approximate matching. We consider mismatches and swaps as natural mistakes which are allowed in a meagre number. The edit distance problem with swap and mismatch operations was solved in $O(n\sqrt{m}\log m)$ time. Yet, the problem of string matching with at most k swaps and mismatches errors was open.

In this paper, we present an algorithm that finds all locations where the pattern has at most k mismatch and swap errors in time $O(n\sqrt{k\log m})$.

1. Introduction

In the last few decades various scientific and business applications requested solutions for the approximate string matching problem. In approximate matching, a distance between objects (e.g. strings, matrices) is defined and the goal is to find all text locations where the pattern matches the text with a pre-specified "small" distance. The necessity mainly derives from information retrieval, image processing, and computational biology. In the former fields the benefit from approximate string matching is obvious. In the latter, the presence of a region of conserved gene order allows, for example, the prediction of groups of functionally associated genes [18]. Nevertheless, as rearrangements do occur, the approximate matching is sought after, as well as the exact matching. Similarly, detecting that a certain protein is in proximity to a known one, may yield functional similarity between them, giving biologists a lead for their research.

Approximate string matching is defined by finding text positions similar to the pattern, where similarity should be defined by a function such as Edit Distance, hereafter defined.

Definition 1. Let S and Q be two strings over alphabet Σ and let E be a set of edit operations. Then the *Edit Distance* (S, Q) with respect to E is the minimum number d, such that exists a sequence of d edit operations $\in E$ for which $e_d(e_{d-1}(\ldots e_1(Q)\ldots)) = S$.

In this paper, we consider the set of edit operations $\{swap, mismatch\}$.

Definition 2. Given a string $S = s_0 \ldots s_{n-1}$ over alphabet Σ and $\sigma \in \Sigma$,
$\mathbf{swap}((i)(S)) = s_0 \ldots s_{i-1}s_{i+1}s_is_{i+2}\ldots s_{n-1}$.
$\mathbf{mis}((i,\sigma)(S)) = s_0 \ldots s_{i-1}\sigma s_{i+1}\ldots s_{n-1}$.

* Corresponding author.
E-mail addresses: lipsky@gmail.com (O. Lipsky), bennyporat@gmail.com (B. Porat), porately@cs.biu.ac.il (E. Porat), riva.shalom@gmail.com (B.R. Shalom), zurasa@cs.biu.ac.il (A. Tzur).

0890-5401/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2010.04.001

The earliest and well-known string distance measures are Levenshtein's *edit distance* [15] and the *Hamming distance*, considering merely mismatches. Let n be the text length and m the pattern length. Lowrance and Wagner [16,17] proposed an $O(nm)$ dynamic programming algorithm for an extended edit distance problem, where the set of edit operations is $\{mismatch, delete, insert\}$. In [14] the first $O(kn)$ algorithm was given for the edit distance with only k allowed edit operations. Cole and Hariharan [9] presented an $O(nk^4/m + n + m)$ algorithm for this problem. To date, however, there is no known algorithm that solves the general case of the extended edit distance problem, where the edit operations are: insertion, deletion, mismatch, and swap, in time $o(nm)$.

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 substantial amount of work was done on finding efficient algorithms for string matching with mismatches, e.g. [7,14]. The most efficient deterministic worst-case algorithm for finding the Hamming distance of the pattern at every text location runs in time $O(n\sqrt{m \log m})$. Isolating the swap edit operation yielded even better results [3,6], with a worst-case running time of $O(n \log m \log \sigma)$.

Amir et al. [4] coped with the challenge of integration of the above two results, providing an efficient algorithm for edit distance with mismatch and swap. Integration can be tricky and the problem of indexing with do not cares is an example for that. There are efficient algorithms for string matching with do not cares (e.g. [10,13]) and efficient algorithms for indexing exact matching (e.g. [19]), both are over 30 years old. Yet, no efficient algorithm for indexing with do not cares is known, though, Cole et al. [8] give an efficient algorithm for indexing in which a bounded number of "don't care" characters is allowed.

In fact, sometimes the integration of two efficiently solvable operations ends up intractable. For example, Wagner [17] proves that edit distance with the *two* operations: *insertion* and *swap* is $\mathcal{NP}$-hard, while each one 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, Amir et al. [5] provided an efficient algorithm for edit distance with *two* operations: *mismatch* and *swap*. Their algorithm runs in time $O(n\sqrt{m}\log m)$.

String matching with up to k swaps and mismatches errors has been an open problem for over four years, worked on by some researchers groups to no avail. Lately, a randomized solution was suggested [11].

In this paper, we discuss the problem of approximate string matching, with only swap and mismatch operations allowed, as formally defined in Definition 3.

Definition 3. The string matching with up to k swaps and mismatches problem (SMkSM):

Input: A text $T = t_0 \ldots t_{n-1}$ and a pattern $P = p_0 \ldots p_{m-1}$, over alphabet Σ, a constant k and a set of editing operations $E = \{swap(i), mis(i, \sigma)\}$.

Output: All text locations i, for which the edit distance, with respect to E, between P and the substring $t_i \ldots t_{i+m-1}$ is $\leq k$, with the restriction that each character can participate in no more than one swap.

The restriction for characters not to participate in more than one swap is motivated by text editing common typing mistakes [2]. Moreover, removing this restriction gives rise to a different distance measure, the interchange distance, which was first defined and discussed in pattern matching settings by Amir et al. [1].

We present an $O(n\sqrt{k \log m})$ time algorithm for a constant alphabet, using counting techniques, convolutions and other combinatorial methods.

The paper is organized as follows: in the next section, we give some related previous work. In Section 3, we describe our algorithm, split into three cases, one of them uses the results of Section 4. An overview of the algorithm is given at the end of this section. Section 4 describes an algorithm performing simultaneously many computations. This algorithm is used by a certain case of the main algorithm, but as it can stand for its own, it was singled out to occupy a distinct section. Section 5 concludes the paper.

2. Previous work

The SMkSM (string matching with up to k swaps and mismatches) problem sets challenges of both approximate matching and that of integrating the swap and mismatch operations. We therefore briefly describe the results regarding both edit operations.

Landau and Vishkin [14] solved the approximate matching problem, allowing k mismatches, in $O(nk)$ time, for a text of length n. They introduced a method of using suffix trees and Lowest Common Ancestor queries in order to allow constant-time jumps over equal substrings in the text and the pattern. Their solution can be easily adjusted to solve our problem, allowing swaps as well. Start at each text location, pass over the longest equal substring, until a mismatch is reached, consuming constant time (using a Lowest Common Ancestor query). After the next 'jump' check whether the last mismatches are adjacent, and if so, whether a single swap can replace the two mismatches. In case a swap was found, the number of corrections activated so far is updated. If a location has more than k errors, we stop. Thus, verification of every location takes time of $O(k)$. Hence, the total time required is $O(nk)$, a bound which we improve in this paper.

Amir et al. [7] solved the problem of approximate pattern matching, where k mismatches are allowed per location. They introduced a counting technique reducing the number of text locations needed to be checked using filtering. Their algorithm requires $O(n\sqrt{k \log k})$ time. Their solution does not return the locations of mismatches in the text during its processing, hence, it cannot efficiently solve the SMkSM problem. Nevertheless, we follow some ideas from their algorithm, but instead of referring to symbols we refer to segments of the text and the pattern.

Amir et al. [2] showed that if the swap operation is isolated as the only allowed edit distance operation, the approximate string matching problem can be solved in time $O(n\sqrt{m \log m})$, where n, m stands for the length of the text, pattern, correspondingly.

The first integration of the swap and mismatch as the set of legitimate operations of the edit distance was suggested by Amir et al. [4,5]. They considered the case of binary alphabet and developed an algorithm using new cases of overlap matching and convolutions, consuming $O(n \log m)$ time, where n, m stands for the length of the text and pattern, correspondingly. For general alphabet they reduced part of the problem to binary alphabet, yielding a total solution in $O(n\sqrt{m} \log m)$ time. Obviously, their algorithm solves our problem: compute the edit distance between all text locations and the pattern, then save only locations where the edit distance is less than $k + 1$. Using their application, the time required for solving our problem is $O(n\sqrt{m} \log m)$, which we wish to ameliorate.

Lately, Porat et al. [11] solved the approximate swap and mismatch edit distance using a randomized algorithm consuming $O(\frac{1}{\epsilon^3} n \log n \log^3 m)$ time. Their algorithm guarantees an approximation factor of $(1 + \epsilon)$ with probability of at least $1 - \frac{1}{n}$.

3. The algorithm

The algorithm we suggest for the string matching with up to k swaps and mismatches problem uses several filters yielding possible candidates and then verifies these text locations.

As a first step, we partition the text to pieces of length $2m$ starting from the first text symbol. In order to avoid neglecting pattern appearances that are split between two adjacent text pieces, we perform yet another partitioning starting from index m of the text. As a consequence we have the following texts: $[T_0, T_{2m-1}]$, $[T_{2m}, T_{4m-1}]$, ... and $[T_m, T_{3m-1}]$, $[T_{3m}, T_{5m-1}]$, ... The creation of the text pieces is done in linear time and space. Throughout the algorithm we consider these text pieces as texts of size $2m$. The results returned by the algorithm are later normalized by the start index of the text piece, in the input text of size n, to form the output.

Consider the combination of our edit operations, swap and mismatch. It is clear that every swap error can be viewed as two mismatch errors, yielding the following observation.

Observation 1. Every text location that has a swap plus mismatch distance of at most k must have a mismatch only distance of at most $2k$.

As a primary filter we use the algorithm of Amir et al. [7] for string matching with a constant number of mismatches. Due to Observation 1 we apply their algorithm with $2k$ allowed mismatches. This procedure requires $O(m\sqrt{k \log k})$ time. To every text location we attach a flag, which is set only if its corresponding location begins an occurrence of the pattern with at most $2k$ mismatches. From now on, we process the whole text but consider merely results associated with possible candidates.

The next step is to perform a relabelling over the text and pattern, by converting the building blocks of the sequences from single symbols to segments.

Definition 4. *An alternating segment* of a string S over alphabet Σ, is a substring alternating between $\sigma, \rho \in \Sigma$. A special case is a substring consisting of a single $\sigma \in \Sigma$. *A maximal alternating segment*, or a *segment* for short, is an alternating segment that cannot be extended to either side.

In other words, if S' is a segment $\sigma\rho\sigma$ . . ., then the character to the left of S', cannot be ρ and the one to the right of S' is distinguished from the symbol before the last symbol of S'. A segment is therefore uniquely defined by its first and second characters and by its length. To each segment we attach its appearance index in the sequence. When partitioning a string S into segments, we start a segment at the last symbol of the preceding one, implying a single character overlap between two consecutive segments, as was previously done [5]. An example for segmentation follows. The numbers in round brackets refer the startin location of the segment in the concatenated string. Let $S = ababcbbacacbabab$, then the segments are: $abab[a, b, 4](1)$, $bcb[b, c, 3](4)$, $b[b, 2](6)$, $ba[b, a, 2](7)$, $ac[a, c, 4](8)$, $cb[c, b, 2](11)$, $babab[b, a, 5](12)$.

Trying to avoid the segments overlap might cause an erroneous counting of the swaps mistakes between pattern segments and text segments. For example, suppose $P = ababacac$, its non-overlapping segment are $Sp_1 = ababa$ and $Sp_2 = cac$. If the text contains $babacaca$, we will count for Sp_1 two swaps and a mismatch, and for Sp_2 a single swap and a single mismatch, summing up to 5 mistakes, while the correct number of mistakes is 4 swaps. The partitioning including border symbol overlaps prevents such pitfalls. However, we must ensure that overlapping symbol will not participate in two swap operations.

The segmentation is therefore, done by going over the string with two distinct character variables and a counter. As long as the string consists of these characters merely, in alternating order, the counter is increased. When the alternation is disturbed, we seal the segment, attaching it its characters and length, which equals the counter. We then clear the counter, and decrease by one the current index, to induce overlap between segments. In addition, we update the characters of the new segment. Having started a new segment, we proceed in the same fashion, till the end of the string. Obviously the segmentation is done in linear time in the size of the string.

Note that a swap operation performed on single character segments as "*bbbb*" are meaningless. Consequentially, in the rest of the paper we consider only segments consisting of two distinct characters.

Lemma 1. *For every two consecutive pattern segments Sp_1, Sp_2,either only the last symbol of Sp_1 or only the first symbol of Sp_2 participates in a swap correction or neither of them does.*

Proof. Let Sp_1, Sp_2 be adjacent pattern segments compared to a text segment St_i. Suppose $Sp_1 = \ldots ab$. Since Sp_2 is adjacent to Sp_1, it must begin in b due to the border symbol overlap. In addition, we know that the consecutive symbol is not a, otherwise contradicting the maximality of Sp_1, therefore $Sp_2 = bc\ldots$ for some $c \in \Sigma$. Suppose the last ab of Sp_1 are swapped, hence the text segment St_i compared to Sp_1 is $St_i = \ldots baba\ldots$ If Sp_2 is also compared to St_i right after Sp_1, it will confront ba aligned to $Sp_2 = bc\ldots$, yielding no need for swapping. Nevertheless, if St_i's end is aligned to the end of Sp_1, even though Sp_2 will be aligned to a different segment, it is the consecutive segment of St_i, therefore starting in a and continuing in a symbol distinct from b, here again implying no profit from a swap operation. The proof for the case Sp_2 participate in a swap is symmetric. □

After executing these primary procedures of constructing a new text and pattern *ST*, and *SP* by partitioning them to segments and replacing their symbols by the appropriate segments, we are ready for efficiently solving the SMkSM problem. For this purpose we consider three cases of pattern instances, for each we suggest appropriate methods. An outline of the overall algorithm is presented at the end of this section in Fig. 6.

3.1. Pattern with many different segments

In this subsection, we deal with patterns having more than $3k$ distinct segments. A distinct segment refers to a segment different from all other segments, by at least one of its properties, starting letter, second letter or its length, avoiding duplicity. For this case we suggest finding all pattern occurrences in the text with no more than k swaps and mismatches in linear time. This will be done using a counting filter based on the following lemma and on a verification.

Lemma 2. *A pattern segment that does not match exactly an identical text segment produces at least one mismatch error.*

Proof. Suppose a text segment *St* is aligned against segment *Sp* of the pattern. By definition, if the segments do not match in the overlap, there are mismatches. Suppose they do match in the overlap area, but one of them, say *St*, ends after the other, i.e. *Sp* ends at location i and *St* ends at location $i + j$, for $j > 0$. As a consequence, there is a mismatch at the location $i + 1$, otherwise, the symbol at location $i + 1$ of the pattern was the same as that of *St* and the segment *Sp* could have been longer, contradicting the maximality of the segmentation. The case where one segment starts before the other is symmetric. □

We would like to count, for every text location l, the number of pattern segments which are identical to the text segments included in $T[l\ldots l + m - 1]$. Let $\{Sp_1, \ldots, Sp_{3k}\}$ be the first $3k$ different pattern segments. We select for each Sp_j its first occurrence in the pattern starting at index i_j. Now, for every text location l, if l is a start of a text segment *St* and there exists a pattern segment Sp_j that is **identical** to *St*, then mark, increment by one a counter related to text location $l - i_j$. This counting technique is formally described in Fig. 1.

Recall that according to Observation 1, every text location with no more than k swap and mismatch errors cannot have more than $2k$ mismatch errors. Since we mark $3k$ pattern segment occurrences in text segments, a possible matching must be marked by at least k pattern segments out of the $3k$, allowing at most $2k$ mismatches with $2k$ pattern segments that did not match. Thus, locations marked by less than k pattern segments are discarded.

Lemma 3. *At the conclusion of the marking stage there are at most $O(m/k)$ candidate locations.*

Proof. The algorithm performs at most one marking per text location, the total number of marks cannot exceed $2m$. Every location that was not discarded has at least k marks, there could be no more than $O(2m/k)$ such locations. □

In order to verify the $2m/k$ new candidates we use the Landau and Vishkin [14] method of moving from one mismatch to the next one, till the $(k+1)$th mismatch is reached. As we go over the mismatches we consider consecutive ones and check,

```
COUNT_SEGMENT_MATCHING (Segmented text $ST$, $\{Sp_i\}$ )

$\{Sp_i\} = \{Sp_1, \ldots, Sp_{3k}\}$ contains the first $3k$ different pattern segments.
1  Let $counters$ be an array of size $2m$.
2  For $j = 1$ to $3k$ do
3      $i_j \leftarrow$ first occurrence of pattern segment $Sp_j$ in P .
5  For every $St \in ST$
3      $l \leftarrow$ the starting location of $St$ in $T$
6      If $\exists Sp_j \in \{Sp_i\}$ such that $Sp_j$ is identical to $St$
7          $counters[l - i_j] + +$
8  Return $counters$
```

Fig. 1. Counting algorithm.

```
FIRST_CASE_ALGORITHM (Segmented text $ST$, Segmented pattern $SP$ )

1  $\{Sp_i\} \leftarrow 3k$ distinct pattern segments.
2  $counters$ = $Count\_Segment\_Matching(ST, \{Sp_i\})$ (Fig. 1)
3  For $l = 1$ to $2m$
4      If $counters[l] > k$
5  $Candidates \leftarrow l$ //text locations marked by more than $k$ pattern segments.
6  Verify $Candidates$. //using modified string matching with $k$ mismatches alg. [15].
```

Fig. 2. An outline of the SMkSM algorithm for the case of many pattern segments.

with no additional complexity, whether they can be corrected by a single swap and if so we increase the number of yet allowed mistakes by one. This verification requires $O(2k)$ time per candidate location. As a consequence the time required for verifying all candidates is $O(km/k) = O(m)$. Lemma 4 follows.

Lemma 4. *The SMkSM problem for patterns with more than* $3k$ *distinct segments can be solved in linear time.*

An outline of the procedure suggested to the case of more than $3k$ pattern segments appears in Fig. 2.

3.2. Pattern with frequent segments

In this subsection we deal with patterns having less than $3k$ distinct segments, yet, these segments appear in high frequency, compensating for their small number. Formally, in this case there are at least $\sqrt{k/\log m}$ different frequent segments each occurring at least $3\sqrt{k \log m}$ times in the pattern.

Note that in order to apply the marking and counting technique, we need to select for marking purposes 3k pattern segments, a condition which is fulfilled by selecting the first $3\sqrt{k \log m}$ appearances of each of the different frequent segments.

For the counting step we go over the text and whenever we encounter a text segment identical to a pattern segment, we would like to mark a possible starting of an occurrence of the pattern in the text. Yet, in contrast to the previous case, here we have to mark several text locations, one for each of the first $3\sqrt{k \log n}$ occurrence of this segment in the pattern. Therefore, for each frequent pattern segment Sp_j, we construct a list of indices i_j, starting locations of the $3\sqrt{k \log m}$ first occurrences of Sp_j in the pattern. We go over the text and for each text location l that is a start of a segment St in the text, identical to a segment Sp_j, we mark locations $l - i_j$ for each i_j in the list. Since we choose $3k$ segments' occurrences in total, we need $O(k)$ additional space for the lists.

After the marking and counting step, we discard every text location that is marked with less than k marks. The counting procedure for frequent pattern segments is described in Fig. 3.

Lemma 5. *At the conclusion of the marking stage there are at most* $3m\cdot \sqrt{\log m/k}$ *candidate locations.*

Proof. Similar to the proof of Lemma 3, the algorithm can perform $3\sqrt{k \log m}$ marking per text location, the total number of marks cannot exceed $3m\sqrt{k \log m}$. Every location that was not discarded has at least k marks, hence, there could be no more than $3m\sqrt{k \log m}/k = 3m\sqrt{\log m/k}$ such locations. □

For verifying the candidates we use, here again, the Landau and Vishkin [14] method, as was previously described. Since there are at most $3m\sqrt{\log m/k}$ candidate locations, and the verification time for each candidate is $O(k)$, the total verification time is $O(m\sqrt{k \log m})$. Lemma 6 follows.

```
COUNT_SEGMENT_MATCHING2 (Segmented text $ST$, $\{Sp_i\}$, $\{List_i\}$ )

// $\{Sp_i\} = \{Sp_1, \ldots, Sp_{\sqrt{k/\log m}}\}$ frequent pattern segments.
// $\{List_j\}$ contains $3\sqrt{k \log m}$ occurrences of $Sp_j$ in $SP$.
1  Let counters be an array of size $2m$.
2  For every $St \in ST$
3      $l \leftarrow$ the starting location of $St$ in $T$
4      If $\exists Sp_j \in \{Sp_i\}$ such that $Sp_j$ is identical to $St$
5          For $f = 1$ to $3\sqrt{k \log m}$
6              $i_{j_f} \leftarrow$ next occurrence of $Sp_j$ in $\{List_j\}$
7              $counters[l - i_{j_f}]++$
8  Return counters
```

Fig. 3. Counting algorithm for frequent pattern segments.

```
SECOND_CASE_ALGORITHM (Segmented text $ST$, Segmented pattern $SP$ )

// $Sp$ is a frequent segments if occurs at least $3\sqrt{k \log m}$ times in the pattern.
1 $\{Sp_i\} \leftarrow$ all distinct frequent segments.
2 For every $Sp_j \in \{Sp_i\}$
3     $List_j \leftarrow$ occurrences of $Sp_j$ in $SP$.
4  counters = Count_Segment_Matching2($ST$, $\{Sp_i\}$, $\{List_i\}$) (Fig. 3)
5  For $l = 1$ to $2m$
6      If $counters[l] > k$
7  Candidates $\leftarrow l$ //text locations marked by more than $k$ pattern segments.
8  Verify Candidates. //using modified string matching with $k$ mismatches alg. [15].
```

Fig. 4. An outline of the SMkSM algorithm for the case of frequent pattern segments.

Lemma 6. *The case of patterns with $\sqrt{k/\log m}$ frequent segments is solvable in time $O(m\sqrt{k \log m})$.*

An outline of the procedure suggested to the case of $\sqrt{k/\log m}$ frequent pattern segments appears in Fig. 4.

If there are less than $\sqrt{k/\log m}$ frequent segments, we choose $3\sqrt{k}$ occurrences of every frequent segment, and then we pick unfrequent segments and their occurrences. If we manage to gather $3k$ segments' occurrences altogether, then the necessary condition is satisfied and we obtain $O(m\sqrt{k \log m})$ time algorithm once again using counting arguments.

3.3. Few segments and fewer frequent segments

The last case is where there are less than $\sqrt{k/\log m}$ frequent pattern segments, and there are no $3k$ segments appearances in the pattern required for the counting filtering.

For the current case we suggest to compute the swaps caused by all possible alignments of the pattern on the text. Recall we have calculated, at the beginning of the algorithm, all text locations matching the pattern with at most $2k$ mismatches. If we can detect the number of swaps, subtracting the latter from the former gives us the number of required swaps and mismatches in matching the pattern to each of the candidate text location. Locations that do not exceed k errors are reported as output.

For text T of length $2m$, we consider all its first $m+1$ substrings of length m, which are possible full overlaps with the pattern, and call these substrings *text sections*.

So far, we have considered frequency of segments originating in the pattern only. In this subsection the text segments are also measured by their frequency. A *text section segment* is called frequent if it appears at least $5\sqrt{klogm}$ times in the text section. We therefore determine that a *text segment* is called frequent if it appears at least $10\sqrt{klogm}$ times in the text (of length 2m). Recall that a *pattern segment* is called frequent if it appears at least $3\sqrt{klogm}$ times in the pattern. Note that these rather high constants were selected for the ease of the reader, in practice they can be substantially reduced.

For each text section we count the number of its frequent segments. Observe, that as a substring differs from its following substring by omitting its first segment and by the adding of a single segment at its end, using a sliding window, this process is done in linear time, for all text sections.

Lemma 7. *In case the pattern contains less than $\sqrt{k/\log m}$ frequent segments and a text section consists of at least $2\sqrt{k/\log m}$ frequent segments, this section cannot be matched to the pattern under the problem restrictions.*

```
THIRD_CASE_ALGORITHM (Segmented text ST, Segmented pattern SP )

   // SP contains less than $\sqrt{k/\log m}$ frequent segments.
1  Partition T into $m+1$ text sections.
2  Consider frequency of text section segments.
3  Discard text sections consisting of $2\sqrt{k/\log m}$ frequent text segments.
4  Count differences between frequent pattern segments and all text segments
5  Count differences between unfrequent pattern segments and frequent text segments
6  For unfrequent segments from both SP and ST use alg. described in Section 4.
7  Return the number of swaps induced by aligning SP to ST.
```

Fig. 5. An outline of the SMkSM algorithm for the case of pattern segments.

Proof. Suppose, in the best case, all frequent segments of P are contained in the text section frequent segments set, hence, there are at least $\sqrt{k/\log m}$ frequent segments of the text section which do not frequently occur in P. Even if we claim that all these segments do appear in P almost frequently, up to $3\sqrt{klogm}-1$ times, these $\sqrt{k/\log m}$ segments still have additional $2\sqrt{klogm}+1$ appearances in the text section segment that have no correspondence in P, due to the definition of $5\sqrt{klogm}$ appearances of a frequent text section segment. Hence, we get at least $\sqrt{k/\log m}\cdot 2\sqrt{klogm}=2k$ mismatches, preventing an approximate matching between this text section and the pattern. □

As this subsection considers merely patterns with less than $\sqrt{k/\log m}$ frequent segments, due to Lemma 7, we can rule out text locations by their number of frequent segments information. We go over our text, first from right to left and then from left to right and seek the first location possessing less than $2\sqrt{k/\log m}$ frequent text segments. These text locations, bounds the area in the text of possible candidates.

Lemma 8. *The bounded area contains $c\sqrt{k/\log m}$ frequent text segments, c a small constant.*

Proof. The bounds of the bounded area are locations where the text sections starting there have less than $2\sqrt{k/\log m}$ frequent text section segments. Even if these locations are far enough, where their frequent text section segments are not overlapping, and the frequent segments are identical, implying their frequency is multiplied by two, we get only $4\sqrt{k/\log m}$ frequent text segments.

Moreover, though there may be more frequent text section segments, because a non-frequent text section segment now having more occurrences and can become frequent, nevertheless, it cannot attain the frequency demand in the total text. □

For the frequent segments we use the Amir et al. [4] algorithm for calculating {swap, mismatch} edit distance of sequences with binary alphabets, requiring $O(n\log m)$ time, where the text is of size n and the pattern of size m. Their algorithm computes the number of real mismatches due to certain pattern and text segments alignments, yet their output can easily be translated to swaps, (by executing an additional convolution of the appropriate sequences, and subtracting the former output from it and dividing by two). Due to Lemma 8 and the current case of few frequent pattern segments, there are merely $c\sqrt{k/\log m}$ frequent segments from the pattern and text, to consider. In order to avoid duplicate counting, we operate the algorithm for every frequent pattern segment with all segments of the text, and for frequent text segments with merely unfrequent pattern segments. Therefore, the time consumed by calculating their swaps contribution is $O(m\sqrt{k/\log m}\log m) = O(m\sqrt{k\log m})$. Lemma 9 follows.

Lemma 9. *Counting the number of swaps induced by frequent segments from both pattern and text can be done in $O(m\sqrt{k\log m})$ time.*

Having calculated the number of the swaps caused by the frequent segments from both pattern and text, we are left to count swaps caused by non-frequent pattern segments (appearing at most $3\sqrt{k\log m}$ times in the pattern), that come across a non-frequent text segment (appearing at most $10\sqrt{k\log m}$).

In Section 4 we suggest a procedure, that given a pattern and text segments accordingly, Sp and St, marks the number of swaps between the compared segments in constant time, enabling to retrieve by two passes over the text the total number of swaps due to all Sps. First we consider matching of Sp of length i with all text segments of length greater than or equal to i and then the other way around.

Lemma 10. *Marking the swaps caused by alignments of pattern segments and texts segments of greater or equal size can be done in $O(m\sqrt{klogm})$ time.*

Proof. For a text segment St of length $|St|$, the number of different pattern segments of length $|St|$ or less is bounded by $2|St||\Sigma|^2$ due to $|St|$ possible lengths, $|\Sigma|^2$ options for alphabet and two options for the first character. Each of the pattern segments can appear at most $3\sqrt{k\log m}$ times and every appearance implies a $O(1)$ time for marking the swaps by

```
SMkSM Algorithm (text $T$ of length $2m$, pattern $P$ of length $m$ )

1  Apply the string matching with $2k$ mismatches alg. [7].
2  Consider possible candidates.
3  Apply segmentation to $P$ and $T$.
4  Construct $SP$ and $ST$ consisting of the segments of $P$ and $T$ accordingly.
5  If $SP$ has $3k$ distinct segments
6       Apply First_Case_Algorithm Fig. 2.      \\ Subsection 3.1
7  Else if $SP$ has $\sqrt{k/\log m}$ distinct frequent segments each occurring at least
8        $3\sqrt{k \log m}$ times in the pattern.
9        Apply Second_Case_Algorithm Fig. 4.   \\ Subsection 3.2
10       Else \\ Subsection 3.3
11           $swaps \leftarrow$ Third_Case_Algorithm Fig. 5. \\ Subsection 3.3
12           Subtract $swaps$ from the number of mismatches found on line 1.
             Discard unappropriate text locations.
13 Normalize the results to locations in the original text of length $n$.
```

Fig. 6. A general outline of the SMkSM algorithm.

Lemma 13. We get that a single text segment St requires $6|St|\sqrt{k \log m}|\Sigma|^2$ time for the marking. Since we deal with small sizes alphabets we say the marking per a text segment is done in $O(|St|\sqrt{k \log m})$. The time required for all text segments matched to shorter pattern segments is $\sum_{St \in T} O(|St|\sqrt{k \log m})$. All text segments compose the text itself, so we get a time of $O(m\sqrt{k \log m})$.

Now we need to compare p of length i with all text segments of length less than i. We apply the marking procedure here again, but count the number of operations from the pattern segment point of view, implying it can be matched to shorter text segments, whose number is bounded by $2(|Sp| - 1)|\Sigma|^2$. Recall that Sp's appearances in the current case, cannot exceed $3\sqrt{k \log m}$ in the pattern. Therefore, the time required for swap detecting between Sp and a shorter text segments is $6|\Sigma|^2(|Sp| - 1)\sqrt{k \log m} = O(|Sp|\sqrt{k \log m})$. Considering all pattern segments, the time is $\sum_{Sp \in P} O(|Sp|\sqrt{k \log m})$ and since all Sps construct P, the overall time is $O(m\sqrt{k \log m})$.

An additional pass over the text is required, to sum up all swaps caused by alignments of all unfrequent Sps to all unfrequent Sts, as described in the following section, requires linear time, hence subsumed by $O(m\sqrt{k \log m})$. □

An outline of the algorithm suggested to the third case is shown in Fig. 5.

Lemma 11. *The time required for solving the SMkSM problem for the third case of patterns is* $O(m\sqrt{k \log m})$.

Proof. Lemma 9 implies that counting the number of swaps induced by frequent text or pattern segments requires $O(m\sqrt{k \log m})$ time. The rest of the swap operation are caused by aligning non-frequent pattern and text segments. According to Lemma 10 these swaps can be counted in time $O(m\sqrt{k \log m})$. After counting the total number of swaps in $O(m\sqrt{k \log m})$ time we are left to subtract for every text location the swaps number from the number of mismatches found by the Amir et al. [7] algorithm, which can be done in linear time, yielding no additional contribution to the time complexity. □

Theorem 1. *The string matching with up to k swaps and mismatches problem is solvable in* $O(n\sqrt{k \log m})$ *time.*

Proof. Due to Lemmas 4, 6, and 11 we have the problem solved for $2m$ sized texts in $O(m\sqrt{k \log m})$. We perform the algorithm for each of the $2n/m$ text pieces, yielding the required time. □

An outline of the overall SMkSM algorithms is given in Fig. 6.

Suppose $k = O(m)$ then, $\sqrt{k} = O(\sqrt{m})$. For small value k there is an additional algorithm [12], solving the SMkSM problem in $O(m\sqrt{k \log k})$ time, to be published.

4. Detecting swaps

The problem we solve in this section is efficiently marking the number of swaps between two segments St and Sp. The proposed algorithm is used in Section 3.3.

Lemma 12. *Aiming at detecting swaps between two segments, only segments sharing the same alphabet need to be considered.*

Proof. For two segments, there are three possible relations concerning their alphabets: they can be distinct, share a single character or they can be identical. For the first case, no swap can obtain a match between the segments. If the segments share a single symbol, for example, $Sp = abab$, $St = bcbcbc$ then activating a swap operations over Sp, will not reduce the

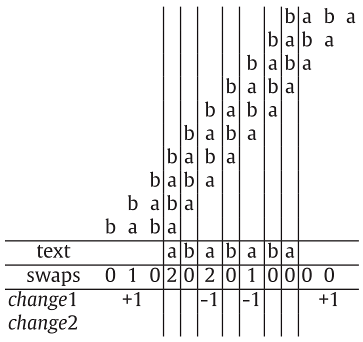

Fig. 7. Alignments of $t = ababab a$ and $p = baba$.

edit distance between the segments, as each pair of symbols will require a swap and a replacement corrections instead of two replacements, so all mismatches are real. When the alphabets of the segments *St*, *Sp* are identical, swaps do occur. The first symbol of *St* and the last of *Sp* determine the number of swaps required for matching the segments. □

For simplicity, we suppose hereafter that the pattern segment *Sp* is the first segment of the pattern, therefore, when aligning *P* to index *j* of *T*, all mismatches found are due to index *j* of *T*. If this is not the situation, and *Sp* begins at index *i* of the pattern, the number of mismatches calculated should be written associated to location $j - i$ of *T*.

Marking each text location with the number of swaps caused by aligning the relevant pattern segments can be easily done in linear time. However, looking closely on the problem, we see there is a constancy in the slide over the text segment. There is a certain point where the end of the pattern segment starts overlapping the text segment. The overlap will increase as well as the number of swaps, as long as we continue the pass over the text segment. When the overlap reaches its peak the number of swaps is stabilized. Decrease of the overlap implies the same effect over the number of swaps.

Consider, for example, $St = abababa$ and $Sp = baba$ all possible alignments of these segments are depicted in Fig. 7. Beneath a text location *j* we write the number of swaps between *St* and *Sp* due to placing *p* on location *j* of *T*.

We wish to capture the overlap orientation in a way requiring a constant number of operations per a pattern and text segments. For this purpose, we use, for every text segment, an array named *change* , in which we save change points reflecting the trend of swaps as detailed hereafter. When *p* start overlapping *t*, the overlap contains a single symbol. Hereafter, the overlap increases until it reaches its maximum. When there is a first swap due to aligning the pattern segment to the *j*th index of the text, we denote $change[j] = change[j] + 1$ meaning that from now on the number of swaps increases by one for every alternate offset.

Having reached the largest possible overlap, starting at index j', the number of swaps is maximal, so we want to stop the incrementing of the swaps in the following offsets, by applying $change[j' + 2] = change[j' + 2] - 1$.

The next change occurs when the number of swaps decreases due to the shortening of the overlap, say at index j''. We need a further decrement and update $change[j''] = change[j''] - 1$. Nevertheless, if the length of the overlap reaches its pick, at index j' and start decreasing in the following location, we will then need to decrease 2 from $change[j' + 2]$, to counteract the first +1 as well as start decreasing by one.

When the overlap ends at index j''' and the number of swaps is zero, we want to avoid a negative number of mismatches formed by the continuation of the -1 usage. Performing $change[j''' + 2] = change[j''' + 2] + 1$ we stabilize the zero number of swaps due to the current *Sp*, *St*.

We save two *change* arrays, one for odd alignments and the other for the evens. In the example of Fig. 7, suppose the text segment begins at location 3 of the text and the pattern is placed over the first text location which is 0, the odd offsets are the swapping locations, they first occur when the pattern is aligned to location 1 of the text, so we use *change*1. The arrays are updated for every relevant *Sp*, *St*. Similarly, in the example of Fig. 8 *change*2 is used.

After all segments of the same alphabet as that of a certain *St* were compared, and *change* arrays updated, we calculate, via a single pass over the *change* arrays, the number of swaps caused by all relevant pattern segments and the current text segment *St*. The number of swaps per text location is saved in array *swaps*. We also use a counter *dif* accumulating the difference between the current *swaps* entry and the preceding one. The computation is done consecutively by the following assignments:

$$\textit{if } i \textit{ is odd} \quad dif1 = dif1 + change1[i], \quad swap[i] = swap[i-2] + dif1. \tag{1}$$

$$\textit{if } i \textit{ is even} \quad dif2 = dif2 + change2[i], \quad swap[i] = swap[i-2] + dif2. \tag{2}$$

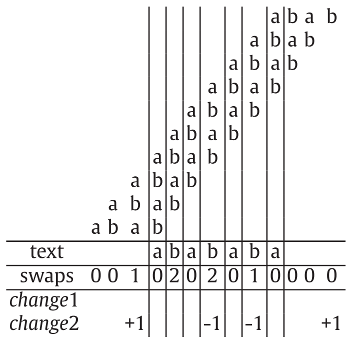

Fig. 8. Alignments of $t = abababa$ and $p = abab$.

```
MARK_SWAPS ($St$, $\{Sp_i\}$)

1   Let $St$ be a text segment starting at location $l$ of the text.
2   $List \leftarrow$ all shorter pattern segments, sharing $St$'s $\Sigma$.
3   For $i = 1$ to $|List|$ do
4       $Sp \leftarrow i$th pattern segment of $List$.
5       $j \leftarrow l - |Sp| - 1$
6       If $Sp$'s last symbol = $St$'s first symbol
7           $change1[j + 1] + +$
8           $change1[j + 2even\ overlap + 1] - -$
9           $change1[j + |Sp| + |St| - 4] - -$
10          If $|Sp| mod 2 = |St| mod 2 = 0$ then $j''' \leftarrow j + |Sp| + |St| + 1$
11          Else If $|Sp| mod 2 = |St| mod 2 = 1$ then $j''' \leftarrow j + |Sp| + |St| - 1$
12              Else $j''' \leftarrow j + |Sp| + |St|$
13          $change1[j'''] + +$
```

Fig. 9. Algorithm for marking swaps.

Note, that there may be several swapped overlaps in a certain offset l of P on T due to different pattern segments, yet they will be noted when compared, and the appropriate adding will be marked on *change*$[l]$. As a consequence, in the final pass the appropriate values will be added to *swap*$[l]$, capturing all swaps altogether (this is the reason for adding values to *change* instead of assigning it the value).

Determining the text locations in which *change* should be updated, and the values added there, a procedure performed for every relevant Sp, can be easily done in constant time. Suppose the pattern segment Sp of length $|Sp|$ is first aligned to location j of the text, and then slides along the text segment St of length $|St|$. We discuss hereafter the case where the first symbol of St is the same as the last symbol of Sp. Such a scenario is depicted in Fig. 7. The other case of distinct symbols can be computed similarly.

First we increment by 1 entry $j + 1$ of the appropriate change array, indicating that when there is a two length overlap, one swap is counted. Note that we are interested merely in an even length overlap since a swap requires two characters. The size of such possible overlap between St and Sp is $2\lfloor \frac{min\{|Sp|,|St|\}}{2} \rfloor$ and we refer to it as *even overlap*.

The maximal overlap and accordingly the maximal number of swaps is reached at j' which equals $j + 2even\ overlap - 1$, therefore we decrement *change*$[j' + 2]$ by one to stop the increase.

The overlap starts diminishing at index j'', when the pattern segment is not fully contained in the text segment and we decrement *change*$[j'']$ by 1. The last symbol of St is located at index $j + |Sp| - 1 + |St| - 1$, consequently $j'' = j + |Sp| - 1 + |St| - 1 - 2$.

The last step is the disappearance of the even overlap, where the swaps number reaches zero, at location j'''. We would like to increment *change*$[j''' + 2]$ by one in order to set the swaps number attributed to the current segments to zero. Here we distinguish between cases of parity of $|Sp|$ and $|St|$. When one of the segments is of odd length and the other is even lengthen, j''' is the index of the last symbol of St. In case both segments are even j''' is the index after the last symbol of St. For the last case of odd segments j''' is the index before the last symbol of St.

A summarized description of the algorithm for odd alignments, is given in Fig. 9. Other cases are solved similarly, with slight modifications. Lemma 13 follows.

Lemma 13. *Marking the number of swaps between a pattern segment and a text segment requires constant time.*

Proof. As was described in this subsection, marking swaps between pattern and text segments requires merely denoting turning points in the even lengthen overlap's life cycle. This implies four marks are needed: the starting point of an even overlap from which it increases, the peak of the overlap, from which it does not change, the location in which it starts decreasing and the final point of disappearance. The algorithm proposed in Fig. 9 perform these four computations in constant time, as all the computations can be combinatorially done without actually comparing the two segments. □

5. Conclusions

The main contribution of this paper is presenting a simple yet efficient algorithm for the important problem of approximate matching with swap and mismatch errors. We have used counting and convolution techniques, adjusting them to this problem unique requirements, as well as other combinatorial methods. Other questions of finding the distance or approximate matching between two sequences with regard to other sets of edit operations are still open.

Acknowledgments

The authors thank anonymous reviewers for their helpful comments

References

[1] A. Amir, Y. Aumann, G. Benson, A. Levy, O. Lipsky, E. Porat, S. Skiena, U. Vishne, Pattern matching with address errors: rearrangement distances, in: Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006, pp. 1221–1229.
[2] A. Amir, Y. Aumann, G.M. Landau, M. Lewenstein, N. Lewenstein, Pattern matching with swaps, in: Proceedings of the 38th IEEE FOCS, 1997, pp. 144–153.
[3] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, E. Porat, Overlap matching, Information and Computation 181 (1) (2003) 57–74.
[4] A. Amir, E. Eisenberg, E. Porat, Swap and mismatch edit distance, in: Proceedings of the 12th Annual European Symposium (ESA), LNCS, vol. 3221, 2004, pp. 16–27.
[5] A. Amir, E. Eisenberg, E. Porat, Swap and mismatch edit distance, Algorithmica 45 (1) (2006) 109–120.
[6] Amihood Amir, Moshe Lewenstein, Ely Porat, Approximate swapped matching, Information Processing Letters 83 (1) (2002) 33–39.
[7] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, Journal of Algorithms 50 (2) (2004) 257–275.
[8] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: Proceedings of the 36th Annual ACM Symposium on the Theory of Computing, 2004, pp. 91–100.
[9] R. Cole, R. Hariharan, Approximate string matching: a faster simpler algorithm, in: Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA), 1998, pp. 463–472.
[10] R. Cole, R. Hariharan, Verifying candidate matches in sparse and wildcard matching, in: Proceedings of the 34th Annual ACM Symposium on the Theory of Computing, 2002, pp. 592–601.
[11] Y. Dombb, O. Lipsky, B. Porat, E. Porat, A. Zur, Approximate swap and mismatch edit distance, in: Proceedings of the 14th String Processing and Information Retrieval Symposium (SPIRE), 2007, pp. 149–163.
[12] E. Porat, Swap and mismatch for small distance, Manuscript, 2007.
[13] M.J. Fischer, M.S. Paterson, String matching and other products, in: R.M. Karp (Ed.), Complexity of Computation, SIAM-AMS Proceedings, vol. 7, 1974, pp. 113–125.
[14] G.M. Landau, U. Vishkin, Efficient String Matching with k Mismatches, Theoretical Computer Science 43 (1986) 239–249.
[15] V.I. Levenshtein, Binary codes capable of correcting, deletions, insertions and reversals, Soviet Physics Doklady 10 (1966) 707–710.
[16] R. Lowrance, R.A. Wagner, An extension of the string-to-string correction problem, Journal of the ACM (1975) 177–183.
[17] R.A. Wagner, On the complexity of the extended string-to-string correction problem, in: Proceedings of the 7th ACM STOC, 1975, pp. 218–223.
[18] I. Yanai, C. DeLisi, The society of genes: networks of functional links between genes from comparative genomics, Genome Biology 3 (11) (2002) 1–12.
[19] P. Weiner, Linear pattern matching algorithm, in: Proceedings of the 14 IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.