An Efficient Skip-Search Approach to Swap Matching

Simone Faro[†] and Arianna Pavone[‡]

[†] *University of Catania, Department of Mathematics and Computer Science, Italy*
[‡] *University of Messina, Department of Cognitive Science, Italy*
Email: faro@dmi.unict.it

The *swap matching* problem consists in finding all occurrences of a pattern x of length m in a text y of length n, allowing for disjoint local swaps of characters in the pattern. In 2003 Amir *et al.* solved the problem in $\mathcal{O}(n \log m \log \sigma)$ worst case time complexity, where σ is the size of the alphabet. In recent years much research has focused on practical solutions and efficient algorithms have been devised by means of the bit-parallel simulation of non-deterministic automata. In this paper we present a new efficient algorithm for the swap matching problem based on character comparison and structured as a generalisation of the Skip-Search algorithm for the exact string matching problem. Although our solution has a quadratic worst case time complexity, it shows a sub-linear behaviour on average. According to experimental results, our algorithm obtains in most practical cases the best running times, when compared against the most effective solutions. The gain in speed-up, in terms of running times, is up to 48%. This makes the new algorithm one of the most efficient solutions in practical cases.

Keywords: Automated Auctions; Analytical Models; Autonomic Systems; Internet Technologies; E-Commerce; Queueing Systems

1. INTRODUCTION

The *string matching problem with swaps* (swap matching problem, for short) is a well-studied variant of the classic string matching problem, and was introduced for the first time in 1995 as one of the open problems in nonstandard string matching [1].

It consists in finding all occurrences, up to character swaps, of a pattern x of length m in a text y of length n, with x and y sequences of characters drawn from a same finite alphabet Σ of size σ. More precisely, the pattern is said to *swap-match the text at a given location j* if adjacent pattern characters can be swapped, if necessary, so as to make it identical to the substring of the text starting (or, equivalently, ending) at location j. All swaps are constrained to be disjoint, i.e., each character can be involved in at most one swap. Moreover, we make the agreement that identical adjacent characters are not allowed to be swapped.

For instance the pattern $x =$ "*abaab*" swap-matches the text $y =$ "*baababa*" at three different locations. Specifically, at position 0 the substring "*baaba*" needs two swaps to match the pattern, while at positions 1 and 2 the substrings "*aabab*" and "*ababa*" need a single swap to match the pattern, respectively.

This problem arises from one of the edit operations considered by Lowrance and Wagner [2] to define a distance metric between strings and turns out to be of relevance in practical applications such as text and music retrieval, data mining, network security, and many others.

For instance, in the field of natural language processing the transposition of two adjacent characters in a text is a most common typing error. Thus several algorithms for the spell-checking problem are designed in order to identify swaps of characters in their matching engines. In musical information retrieval the swap of two adjacent notes in a melody (or two beats in a rhythmic sequence) is used as a basic operation to compute the similarity between two musical sequences [3]. Swap matching is also strongly connected with the Kendall's Tau distance [4], which computes the number of pairs that are in different order in two given rankings. In this context it serves to find the rankings whose ordering can be restored by means of direct swaps of adjacent elements.

According to [5], the swap matching problem finds also application in the process of translation in molecular biology, with the genetic triplets (otherwise called *codons*). In such application one wants to detect the possible positions of the start and stop codons of a mRNA in a biological sequence and find hints as to where the flanking regions are, relative to the translated mRNA region.

1.1. Previous works

The first nontrivial result to the Swap Matching problem was reported by Amir *et al.* [6], who provided a $\mathcal{O}(nm^{\frac{1}{3}}\log m)$-time algorithm in the case of alphabet sets of size 2, showing also that the case of alphabets of size exceeding 2 characters can be reduced to that of size 2 with a $\mathcal{O}(\log^2\sigma)$-time overhead, subsequently reduced to $\mathcal{O}(\log\sigma)$ in the journal version [7].

Amir *et al.* [8] studied some rather restrictive cases in which a $\mathcal{O}(m\log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* [9] solved the swap matching problem in $\mathcal{O}(n\log m\log\sigma)$-time. The above solutions are all based on the fast Fourier transform and consist in reducing the problem to convolutions. For this reason they are only of theoretical interest.

The first attempt to provide a practical solution to the problem goes back to 2000 and is due to Fredriksson [10], who presented a generalisation of the nondeterministic finite automaton for the language $\Sigma^* x$ adapted to the Swap Matching problem, together with a fast method to simulate it using bit-parallelism [11]. The resulting algorithm runs in $\mathcal{O}(n\lceil m/w\rceil)$-time and uses $\mathcal{O}(\sigma\lceil m/w\rceil)$-space, where w is the size in bits of a word in the target machine. In the same paper Fredriksson also presented a variant of the Bndm algorithm [11], which generalises the nondeterministic suffix automaton of x and achieves sub-linear time on average, while requires $\mathcal{O}(nm\lceil m/w\rceil)$-time in the worst-case.

In 2008 Iliopoulos and Rahman introduced in [5] a new graph-theoretic approach to model the problem (then extended in a journal version [12] by Ahmed *et al.*) and devised an efficient algorithm based on the bit-parallel encoding, which runs in $\mathcal{O}((n+\lceil m/w\rceil)\log m)$-time. However Blâzej *et al.* recently [13] pointed out a fatal flaw in the algorithm presented in [12] and devised a corrected algorithm which, although slower than the previous, is based on the same graph theoretical model.

In 2009, Cantone and Faro [14] presented a new efficient algorithm, named Cross-Sampling (Cs), which simulates a non-deterministic automaton with $2m$ states and $3m-2$ transitions. The Cs algorithm though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named Bit-Parallel-Cross-Sampling (Bpcs), which achieves $\mathcal{O}(n\lceil m/w\rceil)$ worst-case time and $\mathcal{O}(\sigma\lceil m/w\rceil)$ space complexity. In a subsequent paper [15] a more efficient algorithm, named Backward-Cross-Sampling (Bcs) was proposed, with a $\mathcal{O}(nm^2)$-time complexity, whereas its bit-parallel implementation, named Bit-Parallel-Backward-Cross-Sampling (Bpbcs), works in $\mathcal{O}(n\lceil m/w\rceil)$-time and $\mathcal{O}(\sigma\lceil m/w\rceil)$-space, showing better practical performances.

More recently Faro [16] presented a new theoretical model to solve the problem based on reactive automata [17, 18]. The model is based on an automaton with only m states, at most $3m-2$ transitions and $8m-12$ reactive links. The author proposed also two non-standard bit-parallel simulations of the automaton. The first simulation, named Bit-Parallel Swap Reactive Automata (Bpsra), works by encoding the transitions of the reactive automaton requiring $\mathcal{O}(n\lceil m/w\rceil)$ worst case time complexity and $\mathcal{O}(\sigma\lceil m/w\rceil)$-space complexity. The second simulation, name Bit-Parallel Swap Reactive Oracle (Bpsro), uses a simpler encoding and, under suitable conditions, it turns out to be very efficient in practice, achieving $\mathcal{O}(n\lceil m/w\rceil)$ worst case time complexity and requiring $\mathcal{O}(\sigma^2\lceil m/w\rceil)$-extra space. It is associated with the definition of a string with disjoint triplets (see [16]) and, since it works as an oracle, in the general case it needs an extra verification phase when a candidate occurrence is found. In this case its worst case time complexity is $\mathcal{O}(nm)$.

For the sake of completeness we mention also a recent study [19] by Fredriksson and Giaquinta, about on the bit-parallel simulation of the automaton due to Fredriksson [10]. By exploiting the method presented by Cantone *et al.* [20], the authors obtained a compact bit-parallel encoding of the swap automaton which takes only $\mathcal{O}(\sigma^2\lceil k/w\rceil)$ space and allows one to simulate the automaton in time $\mathcal{O}(n\lceil k/w\rceil)$, where $\lceil m/\sigma\rceil \leq k \leq m$.

1.2. Our results

In this paper we present a new algorithm for the swap matching problem in strings, based on comparison of characters. Specifically the algorithm can be seen as a generalisation to swap matching of the well known Skip-Search algorithm [21] for the exact string matching problem. In order to speed-up the searching procedure, our solutions extend the original Skip-Search approach in order to take into account substrings of characters. To the best of our knowledge this is the first solution to the swap matching problem which is based on character's comparison.

Our solution achieves a quadratic worst case time complexity in the worst case, but shows a sub-linear behaviour in practice. In particular from our experimental results it turns out that the proposed algorithm obtains the best results in most cases. The gain, in terms of running times, is up to 48% in the case of large alphabets, and is in most cases it is over 20%.

1.3. Organization of the paper

The paper is organised as follows. In Section 2 we introduce some notions and definitions. Then in Section 3 we introduce our new solution based on the Skip Search approach, describe its preprocessing and its searching phase and discuss its practical behaviour. In Section 4 we compare our solution, in terms of running times, against the most effective algorithms known in literature for the Swap Matching problem. Finally we draw our conclusions in Section 5.

2. NOTIONS AND BASIC DEFINITIONS

Given a string x of length $m \geq 0$, we represent it as a finite array $x[0\,..\,m-1]$ and denote by $|x|$ the length of x. In particular, for $m = 0$ we obtain the empty string ε. We denote by $x[i]$ the $(i+1)$-st character of x, for $0 \leq i < |x|$, and by $x[i\,..\,j]$ the substring of x contained between the $(i+1)$-st and the $(j+1)$-st characters of x, for $0 \leq i \leq j < |x|$. A k-substring of a string z is a substring of z of length k. We denote by x_i the nonempty prefix $x[0\,..\,i]$ of x of length $i+1$, for $0 \leq i < m$, whereas, if $i < 0$, we agree that x_i is the empty string ε. Moreover, we say that x' is a proper prefix (suffix) of x if x' is a prefix (suffix) of x and $|x'| < |x|$. We write $x \cdot z$ to denote the concatenation of x and z.

Definition 2.1. *A* swap permutation *for a string x of length m is a permutation $\pi : \{0, ..., m-1\} \to \{0, ..., m-1\}$ such that:*

(a) if $\pi(i) = j$ then $\pi(j) = i$
(characters at positions i and j are swapped);

(b) for all i, $\pi(i) \in \{i-1, i, i+1\}$
(only adjacent characters can be swapped);

(c) if $\pi(i) \neq i$ then $x[\pi(i)] \neq x[i]$
(identical characters can not be swapped).

For a given string x and a swap permutation π for x, we write $\pi(x)$ to denote the *swapped version* of x, namely $\pi(x) = x[\pi(0)] \cdot x[\pi(1)] \cdots x[\pi(m-1)]$.

Definition 2.2. *Given a text y of length n and a pattern x of length m, x is said to* swap-match *(or to have a* swapped occurrence*)* at location $j \geq m-1$ of y *if there exists a swap permutation π of x such that $\pi(x)$ matches y at location j, i.e., $\pi(x) = y[j-m+1\,..\,j]$. In such a case we write $x \propto y_j$.*

If a pattern x of length m has a swap-match ending at location j of a text y, then the number k of swaps needed to transform x into its swapped version $\pi(x) = y[j-m+1\,..\,j]$ is equal to half the number of mismatches of x at location j. Thus the value of k lies between 0 and $\lfloor m/2 \rfloor$.

Definition 2.3 (Pattern Matching Problem with Swaps). *Given a text y of length n and a pattern x of length m, find all locations $j \in \{m-1, ..., n-1\}$ such that x swap-matches with y at location j, i.e., $x \propto y_j$.*

The following elementary result will be used later (its proof is given in [14]).

Lemma 2.1 ([14]). *Let x and R be strings of length m over an alphabet Σ and suppose that there exists a swap permutation π such that $\pi(x) = R$. Then π is unique.*

Corollary 2.1. *Given a text y of length n and a pattern x of length m, if $x \propto y_j$, for a given position $j \in \{m-1, \ldots, n-1\}$, then there exists a unique swapped occurrence of x in y ending at position j.* □

3. A NEW EFFICIENT ALGORITHM

In this section we present an efficient algorithm for the Swap Matching Problem, which takes inspiration from the Skip-Search algorithm. While most of the previous solutions were based on automata and their simulation by using bit-parallelism, to the best of our knowledge this is the first time that an approach based on character's comparison is adopted for solving the pattern matching problem with swaps. The resulting algorithm has a quadratic worst case time complexity while in practical cases it shows a sub-linear behaviour.

The Skip Search algorithm is an elegant and efficient solution to the exact pattern matching problem, firstly presented in [21] and subsequently adapted to many other problems and variants of exact pattern matching.

Let x and y be a pattern and a text of length m and n, respectively, over a common alphabet Σ of size σ. For each character c of the alphabet, the Skip Search algorithm collects in a bucket $B[c]$ all the positions of that character in the pattern x, so that for each $c \in \Sigma$ we have:

$$B[c] = \{i \ : \ 0 \leq i \leq m-1 \text{ and } x[i] = c\}.$$

Plainly, the space and time complexity needed for the construction of the array B of buckets is $\mathcal{O}(m + \sigma)$.

Thus if a character occurs k times in the pattern, there are k corresponding positions in the bucket of the character.

The search phase of the Skip Search algorithm examines all the characters $y[j]$ in the text at positions $j = km - 1$, for $k = 1, 2, \ldots, \lfloor n/m \rfloor$. For each such character $y[j]$, the bucket $B[y[j]]$ allows one to compute the possible positions h of the text in the neighborhood of j at which the pattern could occur. By performing a character-by-character comparison between x and the substring $y[h\,..\,h+m-1]$ until either a mismatch is found, or all the characters in the pattern x have been considered, it can be tested whether x actually occurs at position h of the text.

The Skip Search algorithm has a quadratic worst-case time complexity, however, as shown in [21], the expected number of text character inspections is $\mathcal{O}(n)$.

Among the variants of the Skip Search algorithm, the most relevant one for our purposes is the Alpha Skip Search algorithm [21], which collects buckets for substrings of the pattern rather than for its single characters, and the Skip Search using q-grams [22].

We are now ready to present our new algorithm for the Swap Matching Problem. In the same line of the algorithm presented in [22] our solution makes use of a q-gram approach and a fingerprint function. In the following sections we will introduce the fingerprint function used for finding candidate occurrences of the pattern, the preprocessing phase and the searching phase. Finally we will briefly discuss some practical behaviours of the algorithm.

3.1. The fingerprint value

The main idea behind the new algorithm for the Swap Matching problem consists in using same kind of resemblance between the substrings of the pattern and the substrings of the text in order to easily locate candidate occurrences of the pattern in the text. Specifically, we define a hash function

$$h : \Sigma^* \to \{0 \,..\, 2^\alpha - 1\} \tag{1}$$

which associates an integer value $0 \leq v < 2^\alpha$ (for a given bound α) with any string over the alphabet Σ.

Here we shall make the assumption that each character $c \in \Sigma$ can be handled as an integer value, so that arithmetic operations can be performed on characters. For instance, in many practical applications, input strings can be handled as sequences of ASCII characters. Thus each character can be seen as an 8-bit value corresponding to its ASCII code.

For each string $x \in \Sigma^*$ of length $m \geq 0$, the value of $h(x)$ is defined as follows

$$h(x) = \left[\sum_{j=0}^{m-1} \Big(x[i+j] \times 4(m-1-j) \Big) \right] \mod 2^\alpha \tag{2}$$

Observe that, due to the modulo operation, for each string $x \in \Sigma^*$, we have $0 \leq h(x) < 2^\alpha$.

The use of the bound α in the definition of the hash function given in (1) is for avoiding space consumption and may depend on the target machine on which the algorithm is implemented. In our setting, the value α has been fixed to 16, so that each hash value fits into a single 16-bit register. Although greater values of α are possible, it turns out by observations conducted in [23] that by setting the value of α to 16 the average number of collisions due to the hash function computed as in (2) is negligible.

The hash function defined in (2) is not original; it has been used in many other string matching algorithms in order to associate an integer value with any string over the input alphabet. Just to mention two relevant examples, it has been used in the well known Wu-Manber algorithm [24] for the multiple pattern matching problem and in one of the most effective algorithms for the exact pattern matching problem, the Hash$q$ algorithm [25] by Lecroq. Although a different hash function can be used in the implementation of our algorithm, it has been noticed [23] that the function given in (2) is the right tradeoff between the number of induced collisions and the computation conciseness.

Procedure fng (shown in Fig. 2) is used for computing the fingerprint value of a string. Given a sequence x of length m and a swap permutation π, the procedure fng computes the fingerprint of the substring $\pi(x)$. For efficiency, multiplications in the definition of the hash function are translated in bitwise shift operations in line 3, i.e. $x[i] \times 4$ is translated into programming code by $x[i] \ll 2$. Plainly, its time complexity is $\mathcal{O}(m)$.

Length	#Strings	Collisions	Percentage
2	16	0	0%
3	64	1	1.5%
4	256	13	5.0%
5	1024	97	9.4%
6	4,096	589	14.3%
7	16,384	3,188	19.4%

TABLE 1. The number of collisions of the fingerprint values in the case of Dna sequences. The table reports also the percentage of strings which share the same fingerprint among all possible strings of a given length.

Example 1. A Dna sequence is a string over the alphabet $\Sigma = \{$a,c,g,t$\}$. Let x be the Dna sequence of length $|x| = 5$, and specifically $x =$ "agcgt". If we indicate by Ascii(c) the Ascii code corresponding to a given character $c \in \Sigma$, we have Ascii(a) = 97, Ascii(c) = 99, Ascii(g) = 103 and Ascii(t) = 116. According to the definition of the hash function given in equation (2) we have

$h(x[0..1]) = (97 \times 4) + 103 = 491$
$h(x[1..3]) = (103 \times 16) + (99 \times 4) + 103 = 2147$
$h(x[2..4]) = (99 \times 16) + (103 \times 4) + 116 = 2112$

Observe that the the fingerprint value is not unique for each substring of length q, i.e. two different strings can be associated with the same fingerprint value. For instance in the case of Dna sequences it turns out that the two sequences on length 3 "gcg" and "ctc" share the same fingerprint value. In particular we have

$h($"gcg"$) = (103 \times 16) + (99 \times 4) + 103 = 2147$
$h($"ctc"$) = (99 \times 16) + (116 \times 4) + 99 = 2147$

However it has to be noticed that, for short strings, such collisions are rare in practical cases. Just to mention some examples concerning strings over the Dna alphabet, we have only one collision among the 64 possible different sequences of length 3 (just 1.5%) and only 13 collisions among the 256 possible different sequences of length 4 (just 5.0%). Such results are clearly reported in Table 1. On the other hand, when the length of the strings increases the number of collisions drastically rises: we have 97 collisions among the 1024 possible different sequences of length 5 (9.4%), 589 collisions for sequences of length 6 (14.3%), and 3,188 collisions for sequences of length 7 (20%).

For the sake of completeness we mention a specific approach [26] which can be used to drastically reduce (or completely remove, under particular conditions) the number of collisions in the case of short patterns. However this approach introduces an additional overhead in terms of computational time. In what follows we will not take into account these specific enhancements.

3.2. The preprocessing phase

The preprocessing phase of the algorithm, which is reported in Fig. 2, consists in compiling the fingerprints values associated with of all possible substrings of length q contained in the pattern x, for a given parameter $1 \leq q \leq m$.

Specifically the preprocessing phase works as follows. Let "*" be a jolly symbol, not appearing in the alphabet Σ. For each substring of length q, $z = x[i \,..\, i+q-1]$, for $0 \leq i < m-q$, an extended version of z, $\mathsf{ext}(z)$, is computed by adding the symbol "*" at the beginning (and at the end) of it, provided that z is not already a prefix (or a suffix, respectively) of x (lines 3-5). More formally, for a string x of length m, we have

$$\mathsf{ext}(x[i..j]) = \begin{cases} x[i..j] & \text{if } i = 0 \text{ and } j = m-1 \\ x[i..j] \cdot * & \text{if } i = 0 \text{ and } j < m-1 \\ * \cdot x[i..j] & \text{if } 0 < i \text{ and } j = m-1 \\ * \cdot x[i..j] \cdot * & \text{if } 0 < i \leq j < m-1 \end{cases}$$

Example 2. Let $x =$ "*cbabbcc*" be a string of length 7 and let $q = 4$. Then the set of all substrings of length 4, together with their corresponding extended versions, contains the following strings:

(a) $cbab \rightarrow cbab*$
(b) $babb \rightarrow *babb*$
(c) $abbc \rightarrow *abbc*$
(d) $bbcc \rightarrow *abbc$

Observe that (a) is extended only on its right side, since it is a prefix of x, while (d) is extended only in its left side, since it is a suffix of x.

Subsequently, for each such extended z the algorithm computes all the possible swap permutations $\pi(z)$ and the corresponding fingerprint values (lines 6-7). The number of swap permutations of a string depends on its length and, according to Definition 2.1, on its structure. Fig. 1 presents the set of all possible swap permutations of $\mathsf{ext}(z)$ for a generic string z, with length varying in the range $\{0 \ldots 5\}$. In Fig. 1 we assume that the string doesn't contain any couple of equal adjacent characters.

Regarding the maximal number of swap permutations of a string of length n, we can observe that it is equal to the $(n+1)$-th number in the Fibonacci Sequence. Specifically we obtain the following technical result.

Lemma 3.1. *Let x be a string of length n. Then the number of swap permutations of x is bounded by the $(n+1)$-th number in the Fibonacci Sequence.*

Proof. Let F_n be the n-th number in the Fibonacci Sequence and let $\tau(n)$ be the number of swap permutations of a string of length n.

We first assume the base case where x is a string of length 1. Since a single character cannot be swapped the string admits a single swap permutation, thus $\tau(1) = 1 = F_2$. In addition, if x has length 2, we have exactly 2 swap permutations, namely $x[0] \cdot x[1]$ and $x[1] \cdot x[0]$. Thus $\tau(2) = 2 = F_3$.

All Possible Swap Permutations

[2:2]	[4:5]	[6:13]	[7:21]
* * * *	* a b * * b a * a * b * a * * b * a * b	* a b c d * * b a c d * * a c b d * * a b d c * * b a d c * a * b c d * a * c b d * a * b d c * a * b c * d a * c b * d * a b c * d * b a c * d * a c b * d	* a b c d e * * b a c d e * * a c b d e * * a b d c e * * a b c e d * * b a d c e * * a c b e d * * b a c e d * a * b c d e * a * c b d e * a * b d c e * a * b c e d * a * c b e d * a * b c d * e a * c b d * e a * b d c * e * a b c d * e * b a c d * e * a c b d * e * a b d c * e * b a d c * e
[3:3]	[5:8]		
* a b * * b a * a * b * a * * b * a * b	* a b c * * b a c * * a c b * a * b c * a * c b * a * b * c * a b * c * b a * c		

FIGURE 1. All possible swap permutations of $* \cdot z \cdot *$ for a generic string z, with length varying in the range $\{0 \ldots 5\}$. In the first row of the table the original string is depicted. Swapped couples of characters have been underlined. Each permutation list is labeled by a couple of values in square brachets $[\alpha, \beta]$, where α is the length of the string z, while β is the number of permutations.

Assume now that the relation holds for $n > 1$, i.e. $\tau(n) \leq F_{n+1}$, and let x be a string of length $n+1$. If we avoid the leftmost character $x[0]$ to swap, the number of swap permutations is computed on the substring $x[1..n]$ and coincides with $\tau(n)$. On the other hand if we assume that the leftmost character $x[0]$ swaps with $x[1]$, the number of swap permutations is computed on the substring $x[2..n]$ and coincides with $\tau(n-1)$. Thus the total number of swaps permutation is given by $\tau(n-1) + \tau(n) \leq F_n + F_{n+1} = F_{n+2}$. □

Corollary 3.1. *Let $\tau(n)$ be the number of swap permutations of a string of length n. Then we have that $\tau(n) \in \mathcal{O}(2^n)$.* □

Let z be $\mathsf{ext}(x[i \,..\, i+q-1])$, let π be a swap permutation of z and let v be the corresponding fingerprint value. In this case we say that the index i is associated with the fingerprint value v. During the preprocessing phase a table T, of size 2^α, is computed in order to maintain, for any possible fingerprint value v, the set of all positions i associated with v.

More precisely, for $0 \leq v < 2^\alpha$, we have

$$T[v] = \Big\{ i \mid \begin{array}{l} z = \mathsf{ext}(x[i..i+q-1]) \text{ and} \\ 0 \leq i < m-q \text{ and} \\ \exists\, \pi(z) \mid \textsc{fng}(z, |z|, \pi) = v \end{array} \Big\}$$

Thus the preprocessing phase of the algorithm requires some additional space to store the $2^q(m-q)$ possible alignments in the 2^α locations of the table T. Thus, the space requirement of the algorithm is $\mathcal{O}(2^q(m-q)+2^\alpha)$.

However if we assume that q is a constant smaller than 5 and α is a parameter smaller than 16 the above requirement could be approximated to $\mathcal{O}(m)$.

From our experimental tests conducted on a 32 bit machine, assuming that each set in T is represented by an ordered linked list and each node of the list consists of an integer and a next pointer, it turns out that the total memory requirement for maintaining the table T is always less 140 KB, for patterns with a length $m < 256$.

Concerning the time complexity of the preprocessing phase, observe that the first loop (line 1) just initialises the table T, while the second loop (line 2) requires $\mathcal{O}(2^q(m-q))$ time, which makes the overall time complexity of such phase $\mathcal{O}(2^q m + 2^\alpha)$ that, again, could be approximated to $\mathcal{O}(m)$.

```
VERIFY(x, m, y, i)
 1.  j ← 0
 2.  while (j < m) do
 3.    if (x[j] = y[i + j]) then j ← j + 1
 4.    else
 5.      if (j < m − 1 and
 6.            x[j] = y[i + j + 1] and
 7.            x[j + 1] = y[i + j])
 8.        then j ← j + 2
 9.        else return FALSE
10.  return TRUE

FNG(x, m, π)
 1.  v ← 0
 2.  for i ← m − 1 downto 0 do
 3.    if (x[π(i)] ≠ " ∗ ") then
 3.      v ← (v ≪ 2) + x[π(i)]
 4.  return v

PREPROCESSING(x, q, m)
 1.  for v ← 0 to 2^α − 1 do T[v] ← ∅
 2.  for i ← 1 to m − q − 1 do
 3.    z = x[i..i + q − 1]
 4.    if (i > 0) then z = ∗ · z
 5.    if (i < m − q) then z = z · ∗
 6.    foreach swap permutation π(z) do
 7.      v ← FNG(z, |z|, π)
 8.      if (i ∉ T[v]) then T[v] ← T[v] ∪ {i}
 9.  return T

SKIPq(x, m, y, n, q)
 1.  T ←PREPROCESSING(x, q, m, q)
 2.  π ← ⟨0, 1, . . . , q − 1⟩
 3.  for j ← m − q to n − m step m − q + 1 do
 4.    v ← FNG(y[j..j + q − 1], q, π)
 5.    for each i ∈ T[v] do
 6.      if VERIFY(x, m, y, j − i)
 7.        then output (j − i)
```

FIGURE 2. The pseudo-code of the Skip$q$ algorithm for the swap matching problem and its auxiliary procedures.

3.3. The searching phase

Along the same line of the Skip Search algorithm, the basic idea of the searching phase is to compute a fingerprint value every $(m-q+1)$ positions of the text y and to check whether the pattern appears in y, involving the block $y[j\,..\,j+q-1]$. If the fingerprint value associated with such block indicates that some of the alignments are possible, then the candidate positions are checked naively for matching.

The pseudo-code provided in Fig. 2 reports the skeleton of the SKIP-SEARCH algorithm for the Swap Matching problem. The main loop investigates the blocks of the text y in steps of $(m-q+1)$ blocks (line 3). If the fingerprint v computed on $y[j\,..\,j+q-1]$ points to a nonempty bucket of the table T, then the positions listed in $T[v]$ are verified accordingly.

In particular $T[v]$ contains a linked list of the values i marking the pattern x and the beginning position of the candidate occurrence in the text.

While looking for occurrences on $y[j\,..\,j+q-1]$, if $T[v]$ contains the value i, this indicates that the pattern x may potentially begin at position $(j-i)$ of the text. In that case, a matching test has to be performed between x and $y[j-i\,..\,j-i+m-1]$ via a character-by-character inspection. This is done by means of procedure VERIFY.

Procedure VERIFY naively checks if the pattern x swap-matches at a given position i of the text, i.e. if $x \propto y_i$. It simply compares the characters of the text, proceedings from left to right, i.e. from position i to position $i+m-1$. At each position j the procedure checks if a swap is involved in the match $(x[j] = y[i+j+1]$ and $x[j+1] = y[i+j])$ or if the the character $y[j]$ matches directly the corresponding character in the pattern. If none of the above conditions are verified the procedure stops. It is easy to prove that if x swap-matches at position i, according to Corollary 2.1 such a match is unique, and procedure VERIFY correctly detects it.

Thus procedure VERIFY works in $\mathcal{O}(m)$-worst case time complexity and $\mathcal{O}(1)$ space.

Regarding the time complexity of the searching phase, we can observe that the total number of attempts performed by the algorithm is exactly $n/(m-q)$. At each attempt, the maximum number of verification requests is $(m-q)$, since the filter provides information about that number of appropriate alignments of the pattern. On the other hand, if the computed fingerprint points to an empty location in T, then there is obviously no need for verification. Hence, in the worst case the time complexity of the verification is $\mathcal{O}(m(m-q))$, which happens when all alignments in x must be verified at any possible beginning position. Hence, the best case complexity is $\mathcal{O}(n/(m-q))$, while the worst case complexity is $\mathcal{O}(nm)$.

However, despite its quadratic worst case time complexity, the algorithm has a sub-linear behaviour in practice, as shown in the next sections.

3.4. Practical behaviour

In this section we shortly present some experimental evaluations in order to understand how effective is the searching strategy adopted by our Skip Search algorithm for the Swap Matching problem described above, and how its variants, implemented with different values of q, compare from a practical point of view.

When working with filtering solutions for string matching it is important to investigate how effective is the underlying filtering strategy during the searching phase. Specifically we are interested in counting the number of verifications performed during the execution of the algorithm, i.e candidate positions found during the searching phase which have to be naively verified. Such verifications, in general, downgrades the performance of the algorithm indeed.

Fig. 3 shows the average number of verifications performed, for each attempt, by the Skip Search algorithm, implemented with values of q ranging in the set $\{1, \ldots, 5\}$. A symbol "$\sim$" is reported to indicate a value smaller than 0.01. Experimental results have been performed on a genome sequence, a protein sequence and a natural language text (see Section 4 for all the details of experimental settings). It turns out that by implementing the algorithm with increasing values of q the number of verifications sensibly decreases. However for $q > 4$ the number of verifications drastically increases (of course it also depends on the value of α). This could be attributed with the increase in the number of collisions due to the hash function, as reported in Section 3.1. In addition we perform experimental evaluations, in terms of running times, by testing the algorithms on three text buffers: a genome sequence, a protein sequence and a natural language text. Running times have been computed as the mean of 1000 runs over the same set of patterns and are shown in Fig 4. For each table the best overall running times have been boldfaced. Observe that, according to observations shown in Fig. 3, the implementation of the Skip Search with $q = 4$, obtains the best results in almost all cases. On the top of Fig. 4 we report also preprocessing times of the Skip Search algorithms, for different values of m. The preprocessing times sensibly increases as the value of m increases. This further suggests to avoid values of q greater than 4.

Genome Sequence

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	2.0	$\sim$	$\sim$	$\sim$	-
8	4.3	0.9	$\sim$	$\sim$	$\sim$
16	9.1	2.9	0.9	$\sim$	$\sim$
32	18.1	6.5	2.1	**0.6**	0.8
64	36.6	14.0	5.1	**1.8**	2.0
128	73.6	28.9	10.9	**4.1**	4.9
256	147.0	58.4	22.6	**8.8**	10.2
512	294.3	118.0	45.7	**18.4**	21.4
1024	591.6	237.9	92.3	**38.0**	44.4

Protein Sequence

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	0.6	$\sim$	$\sim$	$\sim$	-
8	1.2	$\sim$	$\sim$	$\sim$	$\sim$
16	2.6	$\sim$	$\sim$	$\sim$	$\sim$
32	5.2	1.0	$\sim$	$\sim$	0.7
64	10.8	2.7	0.9	$\sim$	1.8
128	21.3	5.8	1.9	$\sim$	4.0
256	43.5	12.1	4.1	**1.1**	8.9
512	86.1	24.8	8.9	**3.0**	18.3
1024	171.7	49.4	18.4	**6.9**	37.0

Natural Language Text

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	0.8	$\sim$	$\sim$	$\sim$	-
8	1.6	$\sim$	$\sim$	$\sim$	$\sim$
16	3.5	$\sim$	$\sim$	$\sim$	$\sim$
32	7.2	1.0	$\sim$	$\sim$	$\sim$
64	14.5	2.1	0.7	$\sim$	0.3
128	29.3	5.1	1.9	**0.2**	1.4
256	58.1	10.5	4.0	**1.2**	3.5
512	115.9	21.3	8.2	**3.1**	7.6
1024	230.2	43.8	17.0	**6.9**	15.7

FIGURE 3. Average number of verifications performed, for each attempt, by the Skip Search algorithm, implemented with values of q ranging in the set $\{1, \ldots, 5\}$. Experimental results have been performed on a genome sequence, a protein sequence and a natural language text.

Preprocessing Time

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	0.02	0.02	0.04	0.08	-
8	0.02	0.02	0.05	0.07	0.27
16	0.02	0.02	0.04	0.08	0.27
32	0.02	0.02	0.05	0.09	0.28
64	0.02	0.03	0.06	0.10	0.31
128	0.03	0.04	0.08	0.13	0.36
256	0.05	0.07	0.12	0.19	0.45
512	0.08	0.13	0.21	0.32	0.63
1024	0.14	0.23	0.36	0.56	0.99

Genome Sequence

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	31.37	20.72	12.78	**11.59**	-
8	32.44	19.71	12.11	**7.60**	8.71
16	30.86	16.45	10.08	**6.04**	6.94
32	29.64	14.55	8.59	**5.46**	6.03
64	28.36	13.19	7.49	**4.80**	5.21
128	27.97	12.75	6.98	**4.47**	4.86
256	28.15	12.70	6.71	**4.20**	4.63
512	28.34	13.07	7.03	**4.24**	4.55
1024	30.33	14.09	7.30	**4.56**	5.29

Protein Sequence

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	10.70	7.50	**7.29**	11.04	-
8	8.95	6.03	4.78	**4.39**	7.09
16	8.05	5.79	4.07	**3.28**	5.93
32	7.02	4.88	3.56	**2.82**	4.99
64	6.35	4.40	3.37	**2.64**	4.14
128	5.96	3.84	3.09	**2.58**	3.89
256	6.01	3.76	2.94	**2.63**	3.78
512	6.07	3.70	2.89	**2.46**	3.95
1024	7.61	4.03	3.00	**2.62**	4.44

Natural Language Text

m	SKIP	SKIP2	SKIP3	SKIP4	SKIP5
4	12.68	**7.33**	7.35	12.00	-
8	11.03	5.99	5.00	**4.76**	6.10
16	9.77	5.41	4.04	**3.43**	4.48
32	8.65	4.81	3.43	**3.02**	3.68
64	7.63	4.45	3.16	**2.68**	3.35
128	7.26	3.82	2.95	**2.56**	3.12
256	7.41	3.68	3.13	**2.85**	3.50
512	7.60	3.62	2.97	**2.71**	3.24
1024	8.98	4.02	3.12	**2.64**	3.48

FIGURE 4. Average running times of the Skip Search algorithm, implemented with values of q ranging in the set $\{1, \ldots, 5\}$. Results have been performed on a genome sequence, a protein sequence and a natural language text. On the top, preprocessing times are reported.

4. EXPERIMENTAL COMPARISONS

We report in this section the experimental results of the performance evaluation of the Skip Search algorithm and its variants against the most efficient solutions known in literature for the Swap Matching problem. Specifically, the following 5 algorithms have been compared. According to [16] the first four algorithms in the list are the most effective solutions for this problem:

- Bpcs: Bit Parallel Cross Sampling [14]
- Bpbcs: Bit Parallel Backward Cross Sampling [15]
- Bpsra: Bit Parallel Swap Reactive Automata [16]
- Bpsro: Bit Parallel Swap Reactive Oracle [16]
- Skip$q$: Skip Search, with $1 \leq q \leq 5$ (this paper)

All algorithms have been implemented in the C programming language[1] and have been tested using the Smart tool [27].[2] All experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 16 GB RAM 1600 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. All algorithms have been compared in terms of their running times, excluding any preprocessing time.

We report experimental evaluations on three real data sets (see Fig. 5). Specifically, we used a genome sequence, a protein sequence, and an English text. All sequences have a length of 5MB; they are provided by the Smart research tool and are available online for download (for additional details about the sequences visit the Smart tool web-page).

In the experimental evaluation, patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. For each case, the mean over the running times (expressed in hundredths of seconds) of 1000 runs has been reported.

Tables reported in Fig. 5 summarise the running times of our evaluations, including the speedup (in percentage) obtained by the Skip Search algorithm against the best running time among the previous solutions. Any positive values indicate a performance improvement. Running times representing best results have been bold-faced.

From experimental results it turns out that the Skip Search algorithm obtains the best results in almost all cases. Specifically it is second to the Bpsro and Bpbcs algorithms in the case of small alphabets and short patterns. The speed-up obtained by our solution increases as the size of the alphabet increases. In particular in the case of the genome sequence the improvement in running time is up to 19%, while it is even up to 45% in the case of natural language texts. This makes our Skip Search approach one of the most effective solutions for the Swap Matching problem.

[1]The C code of all algorithms used in our experimental comparison are available online for download at the following link `http://www.dmi.unict.it/~faro/swapalgos.zip`

[2]The Smart tool is available online at the following link: `http://www.dmi.unict.it/~faro/smart/`.

Genome Sequence

m	BPCS	BPBCS	BPSRA	BPSRO	SKIPq	SPEED-UP
4	12.31	21.75	11.87	**9.37**	11.6$^{(4)}$	-
8	10.38	12.39	10.76	8.05	**7.60**$^{(4)}$	5%
16	10.34	7.83	10.68	8.02	**6.04**$^{(4)}$	23%
32	10.37	**5.17**	10.80	7.92	5.46$^{(4)}$	-
64	10.27	5.18	10.69	7.81	**4.80**$^{(4)}$	7%
128	10.23	5.13	10.75	7.89	**4.47**$^{(4)}$	13%
256	10.25	5.19	10.74	8.00	**4.20**$^{(4)}$	19%
512	10.31	5.14	10.87	7.93	**4.24**$^{(4)}$	17%
1024	10.53	5.26	10.87	7.96	**4.56**$^{(4)}$	13%

Protein Sequence

m	BPCS	BPBCS	BPSRA	BPSRO	SKIPq	SPEED-UP
4	11.67	12.11	12.01	8.55	**7.29**$^{(3)}$	15%
8	11.67	8.53	11.53	8.47	**4.39**$^{(4)}$	48%
16	11.04	5.49	11.83	8.47	**3.28**$^{(4)}$	40%
32	11.29	3.92	12.26	8.63	**2.82**$^{(4)}$	28%
64	11.59	3.95	12.61	8.94	**2.64**$^{(4)}$	33%
128	11.92	4.08	12.70	8.61	**2.58**$^{(4)}$	37%
256	11.86	4.05	12.22	8.89	**2.63**$^{(4)}$	35%
512	11.96	4.01	11.93	8.74	**2.46**$^{(4)}$	39%
1024	12.20	3.94	12.53	8.63	**2.62**$^{(4)}$	33%

Natural Language Text

m	BPCS	BPBCS	BPSRA	BPSRO	SKIPq	SPEED-UP
4	12.66	14.05	13.07	9.14	**7.33**$^{(2)}$	20%
8	12.22	9.60	12.66	9.02	**4.76**$^{(4)}$	47%
16	12.24	6.28	12.70	9.23	**3.43**$^{(4)}$	45%
32	12.32	4.72	13.01	9.17	**3.02**$^{(4)}$	36%
64	11.88	4.52	13.08	9.15	**2.68**$^{(4)}$	41%
128	12.02	4.73	12.59	9.13	**2.56**$^{(4)}$	46%
256	12.11	4.80	12.95	9.46	**2.85**$^{(4)}$	41%
512	12.30	4.66	12.47	9.19	**2.71**$^{(4)}$	42%
1024	12.23	4.84	12.70	9.28	**2.64**$^{(4)}$	45%

FIGURE 5. Experimental results obtained by running 5 swap matching algorithms on three text buffers. Experimental results have been conducted on three text buffers: (on the top) a genome sequence, (in the middle) a protein sequence and (on the bottom) a natural language text.

5. CONCLUSIONS

We presented a Skip-Search based approach to Swap Matching, which is, to the best of our knowledge, the first solution to the problem based on characters comparison. The algorithms has a quadratic worst case time complexity but turns out to be very effective in practical cases, especially when implemented by using a q-gram approach. We conducted an extensive set of experimental results in order to compare our new solution against the most effective algorithms known in literature. From such experimental results it turned out that the Skip Search approach obtains the best results in most practical cases. The speed-up achieved by the algorithm is up to 20% in the case of small alphabets, while it is up to 45% in the case of large alphabets. Best results are always obtained by choosing a value of q equal to 4, which turns out to be a good trade-off between the number of collisions and the computational overhead.

REFERENCES

[1] Muthukrishnan, S. (1995) New results and open problems related to non-standard stringology. *CPM*, pp. 298–317.

[2] Lowrance, R. and Wagner, R. A. (1975) An extension of the string-to-string correction problem. *J. ACM*, **22**, 177–183.

[3] Toussaint, G. T. (2004) The geometry of musical rhythm. *Discrete and Computational Geometry, Japanese Conference, JCDCG 2004, Tokyo, Japan, October 8-11, 2004, Revised Selected Papers*, Lecture Notes in Computer Science, **3742**, pp. 198–212. Springer.

[4] Kendall, M. and Gibbons, J. D. (1990) *Rank Correlation Methods*, 5 edition. A Charles Griffin Title.

[5] Iliopoulos, C. S. and Rahman, M. S. (2008) A new model to solve the swap matching problem and efficient algorithms for short patterns. *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 19-25, 2008, Proceedings*, Lecture Notes in Computer Science, **4910**, pp. 316–327. Springer.

[6] Amir, A., Aumann, Y., Landau, G. M., Lewenstein, M., and Lewenstein, N. (1997) Pattern matching with swaps. *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pp. 144–153. IEEE Computer Society.

[7] Amir, A., Aumann, Y., Landau, G. M., Lewenstein, M., and Lewenstein, N. (2000) Pattern matching with swaps. *J. Algorithms*, **37**, 247–266.

[8] Amir, A., Landau, G. M., Lewenstein, M., and Lewenstein, N. (1998) Efficient special cases of pattern matching with swaps. *Inf. Process. Lett.*, **68**, 125–132.

[9] Amir, A., Cole, R., Hariharan, R., Lewenstein, M., and Porat, E. (2003) Overlap matching. *Inf. Comput.*, **181**, 57–74.

[10] Fredriksson, K. (2000) Fast algorithms for string matching with and without swaps. Technical report. Unpublished manuscript, `http://www.cs.uef.fi/~fredriks/pub/papers/sm-w-swaps.pdf`.

[11] Baeza-Yates, R. A. and Gonnet, G. H. (1992) A new approach to text searching. *Commun. ACM*, **35**, 74–82.

[12] Ahmed, P., Iliopoulos, C. S., Islam, A. S. M. S., and Rahman, M. S. (2014) The swap matching problem revisited. *Theor. Comput. Sci.*, **557**, 34–49.

[13] Blazej, V., Suchý, O., and Valla, T. (2016) A simpler bit-parallel algorithm for swap matching. *CoRR*, **abs/1606.04763**.

[14] Cantone, D. and Faro, S. (2009) Pattern matching with swaps for short patterns in linear time. *SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 24-30, 2009. Proceedings*, Lecture Notes in Computer Science, **5404**, pp. 255–266. Springer.

[15] Campanelli, M., Cantone, D., and Faro, S. (2009) A new algorithm for efficient pattern matching with swaps. *Combinatorial Algorithms, 20th International Workshop, IWOCA 2009, Hradec nad Moravicí, Czech Republic, June 28-July 2, 2009, Revised Selected Papers*, Lecture Notes in Computer Science, **5874**, pp. 230–241. Springer.

[16] Faro, S. (2013) Swap matching in strings by simulating reactive automata. *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pp. 7–20. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague.

[17] Gabbay, D. M. (2008) Introducing reactive kripke semantics and arc accessibility. *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, Lecture Notes in Computer Science, **4800**, pp. 292–341. Springer.

[18] Crochemore, M. and Gabbay, D. M. (2011) Reactive automata. *Inf. Comput.*, **209**, 692–704.

[19] Fredriksson, K. and Giaquinta, E. (2014) On a compact encoding of the swap automaton. *Inf. Process. Lett.*, **114**, 392–396.

[20] Cantone, D., Faro, S., and Giaquinta, E. (2010) A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, Lecture Notes in Computer Science, **6129**, pp. 288–298. Springer.

[21] Charras, C., Lecroq, T., and Pehoushek, J. D. (1998) A very fast string matching algorithm for small alphabeths and long patterns (extended abstract). *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, Lecture Notes in Computer Science, **1448**, pp. 55–64. Springer.

[22] Faro, S. (2016) A very fast string matching algorithm based on condensed alphabets. *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016, Proceedings*, Lecture Notes in Computer Science, **9778**, pp. 65–76. Springer.

[23] Faro, S. (2016) Evaluation and improvement of fast algorithms for exact matching on genome sequences. *Algorithms for Computational Biology - Third International Conference, AlCoB 2016, Trujillo, Spain, June 21-22, 2016, Proceedings*, Lecture Notes in Computer Science, **9702**, pp. 145–157. Springer.

[24] Wu, S. and Manber, U. (1994) A fast algorithm for multi-pattern searching. Technical report. University of Arizona, Tucson, AZ.

[25] Lecroq, T. (2007) Fast exact string matching algorithms. *Inf. Process. Lett.*, **102**, 229–235.

[26] Cantone, D., Faro, S., and Giaquinta, E. (2014) Text searching allowing for inversions and translocations of factors. *Discrete Applied Mathematics*, **163**, 247–257.

[27] Faro, S., Lecroq, T., Borzi, S., Mauro, S. D., and Maggio, A. (2016) The string matching algorithms research tool. *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2015*, pp. 99–111. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague.

[28] Campanelli, M., Cantone, D., Faro, S., and Giaquinta, E. (2012) Pattern matching with swaps in practice. *Int. J. Found. Comput. Sci.*, **23**, 323–342.

[29] Cantone, D., Faro, S., and Giaquinta, E. (2012) A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Inf. Comput.*, **213**, 3–12.

[30] Cantone, D., Faro, S., and Giaquinta, E. (2012) On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns. *J. Discrete Algorithms*, **11**, 25–36.

[31] Faro, S. and Külekci, M. O. (2012) Fast multiple string matching using streaming SIMD extensions technology. *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, Lecture Notes in Computer Science, **7608**, pp. 217–228. Springer.

[32] Grabowski, S., Faro, S., and Giaquinta, E. (2011) String matching with inversions and translocations in linear average time (most of the time). *Inf. Process. Lett.*, **111**, 516–520.