Linear and Efficient String Matching Algorithms Based on Weak Factor Recognition

DOMENICO CANTONE and SIMONE FARO, University of Catania
ARIANNA PAVONE, University of Messina

We present a simple and very efficient algorithm for string matching based on the combination of weak factor recognition and hashing. Despite its quadratic worst-case running time, our algorithm exhibits a sublinear behaviour. We also propose some practical improvements of our algorithm and a variant with a linear worst-case time complexity. Experimental results show that, in most cases, some of the variants of our algorithm obtain the best running times when compared, under various conditions, against the most effective algorithms present in the literature. For instance, in the case of small alphabets and long patterns, the gain in running time is up to 18%. This makes our proposed algorithm one of the most flexible solutions in practical cases.

CCS Concepts: • **Theory of computation** → *Design and analysis of algorithms*; *Data structures design and analysis*; *Pattern matching*; • **Applied computing** → *Document management and text processing*; *Document searching*;

Additional Key Words and Phrases: String matching, text processing, weak factor, design and analysis of algorithms, experimental evaluation

ACM Reference format:
Domenico Cantone, Simone Faro, and Arianna Pavone. 2019. Linear and Efficient String Matching Algorithms Based on Weak Factor Recognition. *J. Exp. Algorithmics* 24, 1, Article 1.8 (February 2019), 20 pages.
https://doi.org/10.1145/3301295

1 INTRODUCTION

The *exact string matching problem* is one of the most studied problems in computer science. It consists of finding all (possibly overlapping) occurrences of an input pattern x within a text y, both x and y over a common alphabet Σ. A huge number of solutions have been devised since the 1980s [7, 17], and despite such a wide literature, much work has been produced in the past few years, showing that the need for efficient solutions is still high.

Solutions to the exact string matching problem can be divided into two classes: *counting* solutions, which simply return the number of occurrences of the pattern in the text, and *reporting* solutions, which provide also the exact positions at which the pattern occurs in the text. Solutions in the first class are in general faster than the ones in the second class. In this article, we are interested in algorithms belonging to the class of reporting solutions.

The paper is an extended version of a previously paper published in The *Proceedings of the Prague Stringology Conference* [6].
This work was partially supported by G.N.C.S., Istituto Nazionale di Alta Matematica "Francesco Severi."
Authors' addresses: D. Cantone and S. Faro, Department of Mathematics and Computer Science, University of Catania, Viale A.Doria n.6, Catania, 95125, Italy; emails: domenico.cantone@unict.it, faro@dmi.unict.it; A. Pavone, Department of Cognitive Science, University of Messina, Via Concezione n.6, Messina, 98121, Italy; email: apavone@unime.it.

1084-6654/2019/02-ART1.8 $15.00
https://doi.org/10.1145/3301295

Let m and n denote the length of a pattern x and text y, respectively, and let σ denote the size of their common alphabet Σ. The exact string matching problem can be solved in $O(n)$ worst-case time complexity [19]. However, in many practical cases, it is possible to avoid reading each character in the text, thus achieving sublinear performances on the average. The optimal average time complexity $O(\frac{n \log_\sigma m}{m})$ [24] has been reached for the first time by the Backward DAWG Matching (BDM) algorithm [8]. However, in the worst case, all algorithms with a sublinear average behaviour may have to read all the characters in the text. It is interesting to note that several of such algorithms have a worst-case $O(nm)$-time complexity. Interested readers can refer to Charras and Lecroq [7] and Faro and Lecroq [17] for a detailed survey of the most efficient solutions to the exact string matching problem.

The BDM algorithm computes the Directed Acyclic Word Graph (DAWG) of the reverse x^R of the pattern x. Such a graph, which can be computed in $O(m)$-time, is an automaton that recognises all and only the factors of x^R. During the searching phase, the BDM algorithm slides a window of size m over the text. For each new position of the window, the DAWG of x^R is used to search for a factor of x in the window, proceeding from right to left. The basic idea behind the BDM algorithm is that when the backward search fails on a character c, after reading a word u, then the word cu cannot be a factor of x, so it is safe to move the window just past the character c. In addition, the BDM algorithm maintains the length of the last recognised suffix of x^R, which is a prefix of the pattern. When a suffix of length m is recognised, an occurrence of the pattern is reported.

The DAWG of a string x performs an *exact factor recognition*, as the accepted language coincides exactly with the set of all the factors of x. Instead, we say that a given structure performs a *weak factor recognition* when it recognises a possibly proper superset of the set of all the factors. For instance, the Factor Oracle [1] of a given string x is an automaton that performs a weak factor recognition of the factors of x. Owing to its relaxed recognition approach, the Factor Oracle can be constructed and handled using fewer resources than the DAWG, both in terms of space and time.

The Backward Oracle Matching (BOM) algorithm [1] works in the same way as the BDM algorithm but makes use of the Factor Oracle of the reverse pattern in place of the DAWG. In practical cases, the resulting algorithm performs better than the BDM algorithm [17].

Both the BDM and BOM algorithms have been recently improved in various ways. For instance, very fast BDM-like algorithms, based on the bit-parallel simulation of the nondeterministic factor automaton [2], have been presented in several works [1, 5, 13, 21, 22], whereas efficient extensions of the BOM algorithm appeared in others [9, 13].

In this article, we present a new fast string matching algorithm based on a(n) (even) weaker factor recognition approach. Our solution uses a hash function to recognise all the factors of the input pattern, leading to a simple and very fast recognition mechanism, especially effective in practical cases. We also present some practical improvements to our algorithm that lead to very fast variants. These turn out to be among the fastest algorithms for the exact string matching problem, particularly efficient in the case of long patterns. We also present an additional variant of our algorithm, characterised by a linear worst-case time complexity, that maintains much the same performance of the main algorithm.

The article is organised as follows. In Section 2, we introduce and analyse our proposed algorithm, discuss its behaviour in practice, and present some practical improvements. In Section 3, we present a variant of our algorithm, which is based on both weak prefixes and weak factors recognition, and discuss its performances. Then, in Section 4, we present a variant of our algorithm, characterised by a linear worst-case time complexity, and discuss its performance in practice. In Section 5, we illustrate the experimental results of an extensive comparison of our proposed family of algorithms against the most effective solutions present in the literature. Finally, we draw our conclusions in Section 6.

2 AN EFFICIENT WEAK FACTOR RECOGNITION APPROACH

In this section, we present an efficient algorithm for the exact string matching problem based on a weak factor recognition approach with hashing. As will be shown in Section 5, although our proposed algorithm has a quadratic worst-case time complexity, on average it exhibits a sublinear behaviour, turning out to be very competitive in practical cases.

Let x be a pattern of length m and y a text of length n,[1] both drawn from a common alphabet Σ of size σ. Our proposed algorithm, called Weak Factor Recognition (Wfr), counts and reports all the occurrences of x in y.

The preprocessing and searching phases of the Wfr algorithm are presented next. Subsequently, we discuss some of its efficient variants and analyse some experimental evidence.

2.1 Preprocessing Phase of the Wfr Algorithm

In the preprocessing phase, all factors of the pattern x are indexed to speed up the subsequent searching phase. Specifically, we define a hash function $h : \Sigma^* \to \{0 \,..\, 2^\alpha - 1\}$, which associates an integer value $0 \le v < 2^\alpha$ (for a fixed bound α) with any string over the alphabet Σ.

The value of α in the definition of the hash function h may depend on the target machine on which the algorithm is implemented. In our setting, the value of α has been fixed to 16 so that each hash value fits perfectly into a single 16-bit register. Although greater values of α are possible, we observed that the average number of false positives due to the hash function is negligible when the value of α is set to 16 (see Table 1).

We shall assume that the characters in Σ are represented by integers so that arithmetic operations can be performed on them efficiently. For instance, in many practical applications, input strings can be handled as sequences of ASCII characters. In such applications, characters can just be seen as the 8-bit integers corresponding to their ASCII code.

For each string $x \in \Sigma^*$ of length $m \ge 0$, we recursively define a parameterised hash function $h_a(x)$, with $a = 1, 2, \ldots$, by setting

$$h_a(x) := \begin{cases} 0 & \text{if } m = 0 \\ (2^a \cdot h(x[1 \,..\, m-1]) + x[0]) \mod 2^\alpha & \text{otherwise.} \end{cases} \tag{1}$$

Plainly, we have $0 \le h_a(x) < 2^\alpha$ for each string $x \in \Sigma^*$.

Hash functions of the form (1) have already been used in several string matching algorithms: just to mention two relevant examples, they have been used in the well-known Wu-Manber algorithm for the multiple pattern matching problem [23] and in one of the most effective algorithms for the exact pattern matching problem, namely, the Hash$q$ algorithm by Lecroq [20]. As observed in the work of Faro [11], hash functions of the form (1) represent the right tradeoff between computational efficiency and the number of induced collisions. More specifically, our implementations will be based on the hash function $h_2(x)$.[2] However, for simplicity, we shall just denote it by $h(x)$.

The preprocessing phase of our algorithm, reported in Figure 1 (on the left), consists of calculating the hash values of all the substrings of the pattern x.

A *factor table* F will store the hash values of the factors of x. It can be implemented as a table of Boolean values such that, for $0 \le v < 2^\alpha$, $F[v]$ is set (i.e., $F[v] =$ True) if and only if there exists

[1] Here we assume that a string x of length m is a 0-based vector $x[0 \,..\, m-1]$.

[2] Although other values of the parameter a are possible, experimental results show that reductions of the value of a lead to an increment in the number of hash collisions, whereas increments of the value of a lead to a reduction in the size of hashed substrings.

```
SetBit(F, v)
1.  p ← ⌊(v + 1)/w⌋
2.  b ← v mod w
3.  F[p] ← F[p] | (1 ≪ b)

TestBit(F, v)
1.  p ← ⌊(v + 1)/w⌋
2.  b ← v mod w
3.  return (F[p] & (1 ≪ b)) ≠ 0

Preprocessing(x, m, a)
1.  for v ← 0 to 2^α − 1 do
2.      F[v] ← 0
3.  for i ← m − 1 downto 0 do
4.      v ← 0
5.      for j ← i downto max(0, i − w) do
6.          v ← (v ≪ a) + x[j]
7.          SetBit(F, v)
8.  return F

Check(x, m, y, i)
1.  k ← 0
2.  while (k < m and x[k] = y[i + k]) do
3.      k ← k + 1
4.  if k = m then return true
5.  return false

Wfr(x, m, y, n)
1.  F ←Preprocessing(x, m, a)
2.  j ← m − 1
3.  while (j < n) do
4.      v ← y[j]
5.      i ← j − m + 1
6.      while (j > i and TestBit(F, v)) do
7.          j ← j − 1
8.          v ← (v ≪ a) + y[j]
9.      if (j = i and TestBit(F, v) and
            Check(x, m, y, i)) then Output(i)
10.     j ← j + m
```

Fig. 1. The Wfr algorithm and its auxiliary procedures (w is the number of bits in a computer word).

a factor z of x such that $h(z) = v$. More specifically, for each $0 \le v < 2^\alpha$, we have

$$F[v] := \begin{cases} \text{True} & \text{if } h(x[i \,..\, j]) = v, \text{ for some } 0 \le i \le j < m \\ \text{False} & \text{otherwise.} \end{cases} \tag{2}$$

Plainly, for each factor z of x, we have $F[h(z)] = \text{True}$; however, the converse does not hold—that is, if $F[h(z)] = \text{True}$, for some $z \in \Sigma^*$, we cannot conclude that z is a factor of x. This is the reason the factor recognition under consideration is termed as *weak* factor recognition.

Considering that the set of all nonempty factors of a string x of length m has size m^2, the preprocessing phase of our algorithm requires $O(m^2)$-time and $O(2^\alpha)$-space complexity.

We notice that for factors y longer than α/a, the value $h(y)$ gives the same result as for $h(y[|y| - \alpha/a \,..\, |y| - 1])$. For instance, with $a = 2$, $\alpha = 16$, and $m \ge 8$, $h(x[0 \,..\, m - 1])$ and $h(x[m - 8 \,..\, m - 1])$ yield the same result. Thus, the for-loop 5–7 could be iterated downto $\max(0, i - \alpha/a)$, reducing the time complexity of the preprocessing phase.

In practical cases, the factor table F can be implemented in various ways. The most natural implementation uses a bit vector of size 2^α. Thus, if w is the number of bits in a computer word of the target machine, then the factor table can be implemented as a table of $2^\alpha/w$ words. For instance, if $w = 8$ and $\alpha = 16$, then F can be implemented as a table of 8,192 chars (1,024 machine words), corresponding to a bit vector of 65,536 bits. The procedure SetBit(F, v) and the function TestBit(F, v) (both reported in Figure 1, on the left) are used to quickly set and query, respectively, the bit at position i in the table F. Such procedures are very fast and can be executed in constant time.[3]

Alternatively, the factor table could be implemented using a `char` vector F, where each Boolean value is stored in an 8-bit variable. Although in this case space usage may seem quite inefficient,[4] initialisation and queries can be performed much faster than in the case of a bit vector. Thus, it turns out to be a good tradeoff between time and space consumption. In our example, the factor table F has been implemented as a table of 65,536 chars.

[3]The operators or and and in SetBit and TestBit stand for the bitwise 'or' and 'and' operators, respectively. In addition, the operator $\ll$ stands for 'left-shift'.

[4]In the case in which one uses a `char` vector for representing the factor table F, the space consumption is eight times as large as the bit vector implementation.

2.2 Searching Phase of the WFR Algorithm

As in the BDM and BOM algorithms, during the searching phase, a window of size m slides along the text, starting at position 0. After each attempt, the window is shifted to the right until the end of the text is reached. Specifically, in the attempt at a given position i of the text, the window is opened on the substring $y[i \,..\, i+m-1]$.

Let $h_{i,\ell} \coloneqq h(y[i+m-\ell \,..\, i+m-1])$ be the hash value of the suffix of x of length ℓ, for $\ell = 1, \ldots, m$. Plainly, a necessary condition for $y[i \,..\, i+m-1]$ to match the pattern x is that

$$F[h_{i,1}] = F[h_{i,2}] = \cdots = F[h_{i,m}] = \textsc{True}\,. \tag{3}$$

Thus, the WFR algorithm computes the hash values $h_{i,\ell}$, starting from $\ell = 1$ and for increasing values of ℓ, until either $F[h_{i,\ell}] = \textsc{False}$ or $\ell = m$ (and, therefore, $F[h_{i,\ell}] = \textsc{True}$). In the latter case, condition (3) holds.

As soon the first value $\ell \leq m$ such that $F[h_{i,\ell}] = \textsc{False}$ is found, the search attempt is interrupted and the window is shifted $(m - \ell + 1)$ positions to the right for the subsequent matching attempt. Otherwise, for instance, when condition (3) is satisfied, (i) a naive check is performed to verify whether the substring $y[i \,..\, i+m-1]$ actually matches the pattern (see procedure CHECK in Figure 1), and then (ii) the window is shifted one position to the right.

Observe that by exploiting the relation

$$h(y[j-1 \,..\, i+m-1]) = ((h(y[j \,..\, i+m-1]) \ll 1) + y[j-1]) \mod 2^{\alpha}\,,$$

for $i+1 \leq j \leq i+m-1$, the hash value $h_{i,\ell}$ can be computed in constant time in terms of $h_{i,\ell-1}$, for $\ell = 2, \ldots, m$.

Figure 1 reports the pseudocode of the WFR algorithm and its auxiliary procedures.

2.3 Complexity Analysis of the WFR Algorithm

Procedure CHECK takes as input two input strings x (pattern) and y (text), the length m of the pattern, and an index i such that $0 \leq i \leq n-m$ (current window position). It naively checks whether the pattern x actually occurs in y at position i. Plainly, it has a $O(m)$-time and a $O(1)$-space complexity.

Consider next a generic attempt executed by the WFR algorithm at position j of the text (inside the while-loop 3–10). At line 5, the value of i is set to $j - m + 1$; thus, the inner while-loop 6–8 is repeated at most m times, as the value of j is decreased by at most one position at each iteration (line 7). The value of j is then increased only once, at line 10, by m positions. Thus, at each iteration, the advancement of the current window of the text is a value in the range $\{1, \ldots, m\}$. In the worst case, at each iteration of the while-loop 3–10, the window advances by just one position. Hence, such a cycle is repeated at most $n - m$ times.

This leads to a $O(nm)$ worst-case time complexity, as the procedure CHECK could be called at each iteration of the while-loop 3–10 (see the if-statement at line 9). However, the experimental results reported in Section 5 show that despite its quadratic worst-case time complexity, in practical cases the WFR algorithm exhibits a sublinear behaviour.

2.4 Practical Improvements

A first practical improvement of the WFR algorithm can be obtained by means of a *chained-loop* on the characters of the pattern in the implementation of the searching phase. More precisely, such a technique consists of performing the call to TESTBIT in the while-loop at line 6, while computing the hash value, only at each q cycles, for a fixed value of q, rather than at each cycle. This leads to a faster computation of the hash values; however, window shifts turn out to be shorter on average.

```
q-Hash(y, q, j)
 7.   v ← y[j]
 8.   for k ← 1 to q − 1 do
 9.     v ← (v ≪ 2) + y[j − k]
 7.   return v

Wfrq(x, m, y, n, q)
 1.   m′ ← m
 2.   m ← m − (m mod q)
 3.   F ← Preprocessing(x, m)
 4.   j ← m − 1
 5.   while (j < n) do
 6.     v ← q-Hash(y, q, j)
 7.     i ← j − m + q
 8.     while (j > i + q − 1 and TestBit(F, v)) do
 9.       j ← j − q
10.       v ← (v ≪ 2q)+Hash(y, q, j)
11.     if (j = i and TestBit(F, v)) then
12.       i ← i − q + 1
13.       if (Check(x, m′, y, i)) then Output(i)
14.     j ← j + m − q + 1
```

Fig. 2. The Wfr$q$ algorithm.

The pseudocode of the resulting algorithm, named Wfr$q$, is shown in Figure 2. The same figure also reports the pseudocode of the procedure q-Hash, which computes the hash value associated to q-grams. It takes three parameters as input: the input string y, the value q indicating the size of a q-gram, and an index j, and it returns as output the hash value of the q-gram $y[j-q+1\,..\,j]$. Plainly, the procedure q-Hash has a $O(q)$-time and a $O(1)$-space complexity.

Considering that the pattern is processed in bunches of q consecutive characters, each attempt of the Wfr$q$ algorithm is performed on a prefix of the pattern whose length is a multiple of q. Specifically, at start, the length of the pattern is stored in a new variable m'. Then, to be able to process the pattern, the length of the pattern is reduced to $m - (m \mod q)$ inside the while-loop 8–10, without additional verifications.[5] Plainly, the whole pattern needs to be checked when a candidate position is found at line 13.

It is important to observe that the maximal shift performed by the algorithm at the end of each attempt is of $(m - q + 1)$ positions to the right (see line 14). This leads to shorter advancements in practice but allows a faster processing of the current text window during the searching phase. Observe also that the Wfr$q$ algorithm reduces to the Wfr algorithm for $q = 1$.

An additional improvement of the variant Wfr$q$ of the Wfr algorithm could be obtained along the same line of the Tuned-Boyer-Moore algorithm [18]. Specifically, it can be observed that in most practical cases, a mismatch is found at the first comparison of each attempt. This suggests performing a very fast loop at the beginning of each attempt to locate a window of the text which matches at least a suffix of the pattern of length q. The pseudocode of such a variant of the Wfr$q$ algorithm, called Tuned-Wfr$q$, is shown in Figure 3. The fast while-loop is located at lines 8 through 10, at the beginning of the while-loop at lines 6 through 18. To avoid that the current text window shifts beyond the rightmost position, it is convenient to add a sentinel having the form of a copy of the pattern appended to the text (line 4).

Both algorithms Wfr$q$ and Tuned-Wfr$q$ maintain the same space and time complexity as the Wfr algorithm but show in practice sensible performance improvements, as will be noted in the next section.

[5]Notice that when m is a multiple of q, there is no reduction in the pattern length.

```
TUNED-WFRq(x, m, y, n, q)
 1.  m' ← m
 2.  m ← m − (m mod q)
 3.  F ← PREPROCESSING(x, m)
 4.  y ← y.x
 5.  j ← m − 1
 6.  while (j < n) do
 7.    v ← q-HASH(y, q, j)
 8.    while (NOT TESTBIT(F, v))
 9.      j ← j + m − q + 1
10.      v ←HASH(y, q, j)
11.    i ← j − m + q
12.    while (j > i + q − 1 and TESTBIT(F, v)) do
13.      j ← j − q
14.      v ← (v ≪ 2q)+HASH(y, q, j)
15.    if (j = i and TESTBIT(F, v)) then
16.      i ← i − q + 1
17.      if (CHECK(x, m', y, i)) then OUTPUT(i)
18.    j ← j + m − q + 1
```

Fig. 3. The TUNED-WFRq algorithm.

Table 1. Average Number of Occurrences (α Value) Versus Average Number of Verifications (β Value) Per Megabyte

m	4	8	16	32	64	128	256	512	1,024
Genome-α	4068.40	23.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20
Genome-β	4068.40	24.40	0.20	0.20	0.20	0.20	0.20	0.20	0.20
Protein-α	17.00	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20
Protein-β	21.40	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20
English-α	1275.80	28.60	2.00	0.40	0.20	0.20	0.20	0.20	0.20
English-β	1280.40	28.80	2.20	0.40	0.20	0.20	0.20	0.20	0.20

Values have been computed in the searching phase during the experimental tests to be described in Section 5.

2.5 Evaluation in Practice

The experimental results to be presented next allow us to estimate the effectiveness of the searching strategy adopted by the WFR algorithm and its variants WFRq and TUNED-WFRq, and to compare such variants from a practical point of view.

When working with filtering solutions for string matching, it is important to investigate the effectiveness of the underlying filtering strategy in the searching phase. Specifically, we are interested here in counting the percentage of false positives found during the execution of the algorithm—for instance, the candidate positions found during the searching phase that do not correspond to actual occurrences of the pattern in the text. The presence of false positives, in general, downgrades the performance of the algorithm, as the number of verifications increases proportionally.

Table 1 shows the average number of occurrences (α value) versus the average number of verifications (β value) per megabyte. Values have been computed during the searching phase of the WFR algorithm in the experimental tests that will be described in detail in Section 5. Notice that the number of exceeding verifications (namely, $\beta - \alpha$) is negligible and, in most cases, is equal to 0.

In addition, we have run experimental evaluations to compare the behaviour of the WFR algorithm and its variants WFRq and TUNED-WFRq. Specifically, running times for the algorithms WFRq and TUNED-WFRq have been computed for $q = 1, \ldots, 8$. The experimental results reported

in Table 2 have been obtained by running the algorithms on three text buffers: a genome sequence (table Genome), a protein sequence (table Protein), and a natural language text (table NatLang). Running times have been computed as the mean of 1,000 runs over the same set of patterns (see Section 5 for details of the experimental settings).

Experimental results show that the Tuned-Wfr$q$ variant always obtains better results than the Wfr$q$ variant, especially in the case of short patterns or small alphabets. Additionally, as the size of the pattern increases, better results are obtained by choosing larger values for q.

3 INCLUDING A WEAK PREFIXES RECOGNITION

A natural improvement of the Wfr algorithm consists of performing a recognition of the prefixes while scanning the current window of the text for factors of the pattern. This modification brings our algorithm in the same line of the Reverse-Factor algorithm [8], which uses the suffix automaton of the reverse of the pattern while scanning the current text window, reaching an optimal average running time complexity [24]. We name the resulting algorithm Weak Prefix Recognition (Wpr). Its pseudocode is shown in Figure 4.

Specifically, an additional *prefix table* P stores the hash values of the prefixes of x. Much like the factor table F, the prefix table P can be implemented as a Boolean table, where the value at position $h(z)$ in P is set if and only if z is a prefix of x. As shown in Figure 4 (on the left), the preprocessing phase can be easily modified to compute both prefix and factor tables while maintaining the same space and time complexity.

Notice that if the factor table F is maintained by a `char` vector, information about pattern prefixes can be stored in the same table F. Specifically, we may assume that each entry $F[v]$ in the table, with $0 \leq v < 2^{\alpha}$, stores a ternary value in $\{0, 1, 2\}$ with the following meaning:

(i) if v is not the hash value of any factor of x, then $F[v] = 0$;
(ii) if v is the hash value of a prefix of x, then $F[v] = 1$;
(iii) if neither (i) nor (ii) applies (namely, if v is the hash value of some factor of x but of no prefix of x), then $F[v] = 2$.

More formally, we have

$$F[v] = \begin{cases} 0 & \text{if } v \neq h(x[i \,..\, j]), \text{ for all } 0 \leq i \leq j < m \\ 1 & \text{if } v = h(x[0 \,..\, j]), \text{ for some } 0 \leq j < m \\ 2 & \text{otherwise.} \end{cases}$$

During the searching phase, information gathered in the prefix table is used to increase the length of the shift advancements performed at the end of each attempt. The searching phase of the algorithm Wpr is shown in Figure 4 (on the right).

In particular, during each attempt, a variable p is used to maintain the starting position of the longest prefix of the pattern which has been recognised while scanning the current window of the text, from right to left. At the beginning of each attempt, the value of p is initialised to $j + 1$ (line 6), meaning that only the empty prefix has been recognised. When a weak factor is detected (line 7), the algorithm checks whether it is also a candidate prefix of the pattern (line 8); if this is the case, the value of p is updated to the starting position of the factor. At the end of each attempt, the starting position of the text window is advanced to the starting position of the longest prefix of the pattern which has been recognised during the attempt, namely, position p (line 12).

Extensive experimentation shows that on average, the prefixes recognition approach implemented in the Wpr algorithm leads to longer shifts than the approach based on a simple factor recognition. Specifically, the increase in shift advancement varies from 1 character, in the case of

Table 2. Comparison Between the WFRq and TUNED-WFRq Variants of the WFR Algorithm

GENOME	q/m	2	4	8	16	32	64	128	256	512	1,024
WFRq	1	24.13	13.67	8.63	5.43	3.68	2.78	2.42	2.16	1.88	1.72
	2	15.32	10.01	7.15	4.66	2.88	2.29	2.17	2.02	1.80	1.70
	3	—	13.24	5.93	3.54	3.01	2.42	2.12	1.95	1.73	1.67
	4	—	13.93	4.89	3.26	2.57	2.24	2.14	1.99	1.77	1.67
	5	—	—	14.93	3.42	2.55	2.12	2.03	1.91	1.77	1.70
	6	—	—	16.27	4.28	2.62	2.15	2.03	1.91	1.73	1.68
	7	—	—	17.50	4.12	2.77	2.16	2.06	1.93	1.72	1.67
	8	—	—	19.46	4.00	2.73	2.24	2.11	1.95	1.75	1.68
TWFRq	1	23.78	15.29	9.73	6.22	3.95	2.91	2.43	2.14	1.88	1.73
	2	**9.94**	8.71	6.68	4.60	2.92	2.32	2.20	2.04	1.77	1.68
	3	—	**8.16**	4.83	3.31	2.93	2.42	2.16	1.95	1.77	1.69
	4	—	8.42	**3.77**	**2.75**	2.33	2.13	2.07	2.00	1.74	1.67
	5	—	—	9.61	2.94	**2.31**	**2.04**	**1.96**	**1.87**	1.76	1.70
	6	—	—	12.38	3.80	2.53	2.08	1.98	1.88	1.72	1.68
	7	—	—	14.23	3.68	2.58	2.10	2.01	1.88	**1.71**	1.66
	8	—	—	15.22	3.53	2.50	2.14	2.03	1.90	1.72	**1.65**

PROTEIN	q/m	2	4	8	16	32	64	128	256	512	1,024
WFRq	1	11.02	8.42	6.45	4.67	3.24	2.69	2.50	2.23	2.01	1.86
	2	12.64	6.24	4.52	3.43	2.94	2.61	2.37	2.10	1.93	1.83
	3	—	13.55	5.29	3.19	2.62	2.26	2.21	2.13	1.97	1.84
	4	—	15.72	5.07	3.42	2.62	2.28	2.23	2.06	1.93	1.84
	5	—	—	16.70	3.75	2.77	2.31	2.23	2.07	1.90	**1.80**
	6	—	—	18.35	4.80	2.88	2.38	2.28	2.08	1.91	**1.80**
	7	—	—	20.44	4.61	3.05	2.48	2.36	2.15	1.95	1.82
	8	—	—	21.85	4.55	3.03	2.54	2.38	2.19	1.98	1.84
TWFRq	1	9.39	7.74	6.27	4.87	3.45	2.87	2.58	2.25	2.03	1.87
	2	**6.97**	**4.52**	**3.45**	3.02	2.75	2.54	2.35	2.12	1.94	1.82
	3	—	7.77	3.88	**2.79**	2.39	2.15	2.12	2.10	1.98	1.84
	4	—	9.32	3.89	2.83	**2.38**	**2.11**	**2.09**	**2.00**	1.89	1.82
	5	—	—	10.54	3.21	2.54	2.18	2.13	2.01	**1.86**	**1.80**
	6	—	—	14.16	4.20	2.71	2.24	2.19	2.05	1.88	**1.80**
	7	—	—	16.24	4.02	2.83	2.36	2.27	2.09	1.91	1.82
	8	—	—	17.05	3.98	2.78	2.38	2.28	2.11	1.91	1.82

NATLANG	q/m	2	4	8	16	32	64	128	256	512	1,024
WFRq	1	11.83	9.40	7.20	5.00	3.54	2.87	2.63	2.32	2.06	1.91
	2	12.59	6.45	4.66	3.52	2.99	2.61	2.38	2.16	1.97	1.86
	3	—	13.84	5.51	3.22	2.67	2.32	2.23	2.10	1.97	1.85
	4	—	15.73	5.19	3.49	2.68	2.30	2.22	2.08	1.94	1.84
	5	—	—	16.56	3.80	2.81	2.35	2.23	2.07	1.91	**1.82**
	6	—	—	18.33	4.77	2.89	2.42	2.31	2.12	1.94	**1.82**
	7	—	—	20.18	4.61	3.08	2.46	2.34	2.16	1.96	1.85
	8	—	—	21.62	4.52	3.03	2.56	2.38	2.19	1.98	1.87

(Continued)

Table 2. Continued

Genome	q/m	2	4	8	16	32	64	128	256	512	1,024
TWFRq	1	10.12	8.81	7.11	5.21	3.72	2.99	2.68	2.34	2.08	1.94
	2	**7.04**	**4.72**	**3.58**	3.14	2.82	2.52	2.34	2.15	1.98	1.85
	3	—	7.88	4.13	**2.88**	2.46	2.19	2.12	2.06	1.97	1.84
	4	—	9.36	4.05	2.89	**2.45**	**2.16**	**2.09**	**2.02**	**1.89**	1.84
	5	—	—	10.62	3.22	2.57	2.21	2.15	**2.02**	1.92	1.83
	6	—	—	14.23	4.13	2.76	2.29	2.20	2.07	1.92	1.83
	7	—	—	16.21	4.06	2.85	2.37	2.26	2.09	1.94	**1.82**
	8	—	—	16.90	3.98	2.81	2.39	2.28	2.11	1.95	**1.82**

Running times (expressed in thousandths of seconds) have been taken on a genome sequence (table Genome), a protein sequence (table Protein), and a natural language text (table NatLang). For each group of algorithms, the best results are underlined, and the best overall running times are boldfaced. Observe that for $q = 1$, the WFRq algorithm reduces to the WFR algorithm.

```
PREPROCESSING(x, m)
 1.  for v ← 0 to 2^α − 1 do
 2.      F[v] ← 0
 3.  for i ← m − 1 downto 0 do
 4.      v ← 0
 5.      for j ← i downto 1 do
 6.          v ← (v ≪ 2) + x[j]
 8.          SETBIT(F, v)
 9.      v ← (v ≪ 2) + x[0]
10.      SETBIT(F, v)
10.      SETBIT(P, v)
11.  return (F, P)

WPR(x, m, y, n)
 1.  (F, P) ← PREPROCESSING(x, m)
 2.  j ← m − 1
 3.  while (j < n) do
 4.      v ← y[j]
 5.      i ← j − m + 1
 6.      p ← j + 1
 7.      while (j > i and TESTBIT(F, v)) do
 8.          if (SETBIT(P, v)) then p ← j
 9.          j ← j − 1
10.          v ← (v ≪ 2) + y[j]
11.      if (j = i and TESTBIT(F, v) and
               CHECK(x, m, y, i)) then OUTPUT(i)
12.      j ← p + m − 1
```

Fig. 4. The WPR algorithm and its preprocessing.

short patterns of length 4 (corresponding to 25% longer shifts), to 6 characters, in the case of long patterns of length 1,024 (corresponding to 0.5% longer shifts).

However, despite this, the WPR algorithm turns out to be up to 18% slower than the WFR algorithm because of the additional branching condition at line 8, which is necessary for the recognition of the candidate pattern prefixes. For this reason, in the next sections we shall focus only on the WFR algorithm and its variants, overlooking any further consideration concerning the WPR algorithm.

4 A LINEAR VARIANT OF THE WFR ALGORITHM

We present another variant of the WFR algorithm that, in addition to maintain in practice much the same performances as the algorithm WFR, has a linear worst-case time complexity. The searching strategy of the new variant, called the LINEAR-WFR algorithm, improves the one of the WFR algorithm by taking advantage of an efficient verification approach by Knuth-Morris-Pratt.

Much like the WFR algorithm, when the LINEAR-WFR algorithm identifies each candidate position i in the text, such that the substring $y[i \,..\, i + m - 1]$ is recognised as a weaker factor of x, a subsequent verification phase takes place to check whether the strings x and $y[i \,..\, i + m - 1]$ actually match.

However, the LINEAR-WFR algorithm consists of two separated (yet coordinated) procedures which coexist in the same workflow: a filtering procedure, based on the WFR algorithm, which

```
LINEAR-WFR(x, m, y, n)
 1.  F ← PREPROCESSING(x, m)
 2.  π ← PRECOMPUTE-KMP-FAILURE-FUNCTION(x, m)
 3.  j ← m − 1
 4.  k ← 0
 5.  q ← 0
 6.  while (j < n) do
 7.      v ← y[j]
 8.      i ← j − m + 1
 9.      b ← max(i, k − 1)
10.      while (j > b and TESTBIT(F, v)) do
11.          j ← j − 1
12.          v ← (v ≪ 2) + y[j]
13.      if (j = b and TESTBIT(F, v)) then
14.          if (i > k) then
15.              k ← i
16.              q ← 0
17.          while (k < i + m − 1) do
18.              while (q < m and x[q] = y[k]) do
19.                  k ← k + 1
20.                  q ← q + 1
21.              if (q = m) then OUTPUT(i)
22.              q ← π[q]
23.              if (q < 0) then
24.                  k ← k + 1
25.                  q ← q + 1
26.          j ← k + m − 1 − q
27.      else j ← j + m
```

Fig. 5. The LINEAR-WFR algorithm.

is used for locating all candidate occurrences of x in y, and a verification procedure, based on the Knuth-Morris-Pratt (KMP) algorithm, which is used for determining whether each candidate occurrence corresponds to an exact match. When called at a particular position i of the text, the verification procedure executes the KMP algorithm to search for occurrences of the pattern, starting from position i to position $i + m - 1$ of the text, provided that none of such positions has already been verified in a previous verification phase.

In the following sections, we briefly describe the modifications to the verification and filtering phases of the WFR algorithm that lead to the LINEAR-WFR algorithm, and discuss their time complexity and some experimental results.

4.1 Modifications to the Filtering Phase

The filtering approach that will be adopted by the LINEAR-WFR algorithm is a relaxed version of the weak factor recognition approach in the WFR algorithm. We shall refer to it as the *weaker factor recognition approach*. Specifically, a *weaker* factor of x is any string $s \in \Sigma^*$, of length at most m, that is a concatenation of two weak factors of x. More formally, we have the following definition.

Definition 4.1 (Weaker factor). Let x be a string of length m over an alphabet Σ, and let F be the factor table associated with the string x relative to a given hash function $h : \Sigma^* \to \{0, 1, \ldots, 2^\alpha - 1\}$ (compare to (2)). We say that a string $s \in \Sigma^*$ of length at most m is a *weaker factor* of x if there exist strings $t, u \in \Sigma^*$ such that

(a) $s = t.u$,
(b) $F[h(t)] = F[h(u)] = \text{TRUE}$.

The filtering phase of the LINEAR-WFR algorithm is enclosed in lines 7 through 12 of Figure 5. Let i and j be the leftmost and rightmost positions, respectively, of the current text window. Initially, the value of j is set to $m - 1$ (line 3) so that the variable i is initialised to 0 (line 8) for the first

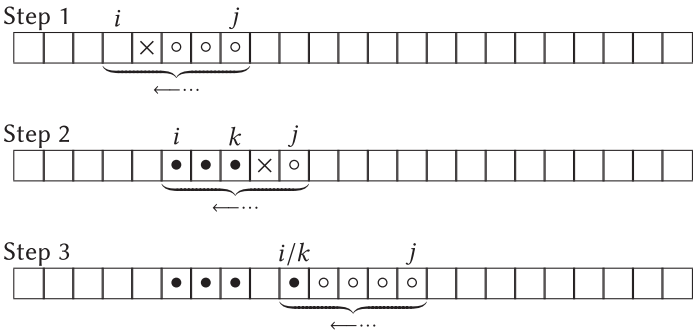

Fig. 6. Three working steps of the Linear-Wfr algorithm's filtering phase, for a pattern of length $m = 5$. Previously inspected positions are identified by the bullet symbol "•" and newly inspected positions by the symbol "○", whereas mismatching positions are identified by "×". Step 1: Only a suffix of length 3 of the current window of the text is recognised as a factor of the pattern. Thus, the window is advanced two positions. Step 2: A suffix of length 1 is recognised as a factor of the pattern, and the window is advanced four positions. Step 3: A suffix of length 4 is recognised as a factor of the pattern. Considering that a prefix of length 1 has been already identified as a weak factor, we have a weaker match at position i of the text so that a verification step can start.

iteration of the while-loop 6–27. At each iteration of the while-loop 10–12, the hash value v of the suffix $y[j \,..\, i + m - 1]$ is computed for decreasing values of j, and the procedure TestBit(F,v) is called to verify whether $y[j \,..\, i + m - 1]$ is a factor of x. At each attempt, the algorithm remembers the rightmost position k of the text window just inspected.

An attempt ends when a mismatch is found, namely, when TestBit(F,v) returns a negative response, or when j reaches the maximum value between i and k. In this latter case, we can distinguish the two following subcases: if $i > k$, then the filtering phase ends with the recognition of the weak factor $y[i \,..\, i + m - 1]$ of length m; otherwise, $k \geq i$ and the filtering phase ends with the recognition of the weaker factor of length m obtained by concatenating the two weak factors $y[i \,..\, k]$ and $y[k + 1 \,..\, i + m - 1]$.

Figure 6 shows an example with three working steps of the filtering phase of the Linear-Wfr algorithm, where it is assumed that the pattern has length $m = 5$.

Plainly, a weaker factor of a pattern x may very well not be a weak factor of x; thus, the adoption of such a more relaxed variant of the factor recognition rule may lead to an increase of the average number of false positives. However, the number of text character inspections in the searching phase is reduced.

4.2 The KMP Algorithm

As stated previously, the verification phase of the Linear-Wfr algorithm is based on the Kmp algorithm. We recall that the Kmp algorithm has been the first string matching algorithm to attain a linear worst-case time complexity. At a high level, it is similar to the brute force algorithm, as all text windows, for $i = 0, 1, \ldots, n - m$, are taken into consideration to determine whether there is a match at position i of the text. However, the Kmp algorithm uses information collected from partial matches between the pattern and text to skip over shifts that are guaranteed to result in no match.

As earlier, let x and y be a pattern and a text of length m and n, respectively, and assume that $x[0 \,..\, q]$ and $y[k - q + 1 \,..\, k]$ match, but $x[q + 1] \neq y[k + 1]$, for some $0 \leq q < m - 1$ and $q - 1 \leq k < n - 2m$. In addition, let $\pi(q)$ be the length of the longest proper prefix of $x[0 \,..\, q]$ that is also a

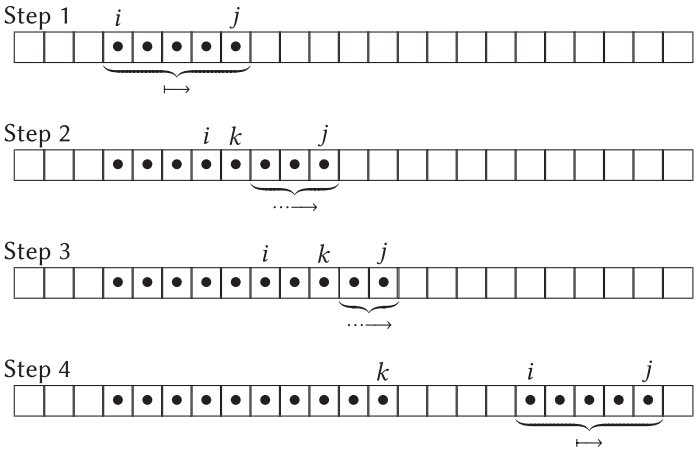

Fig. 7. Four working steps of the Linear-Wfr algorithm for a pattern of length $m = 5$. Verified positions are identified by bullets. Steps 1 and 4: A candidate position i is found while $k < i$; thus, a new Kmp-verification is started from position i. Steps 2 and 3: A new candidate position i is found while $k \geq i$; thus, the Kmp-verification is continued from the previous verified position k.

suffix of $x[0 \,..\, q]$ and such that $x[\pi(q) + 1] \neq x[q + 1]$. Then the Kmp algorithm slides the pattern $(q + 1 - \pi(q))$ positions to the right so that the substrings $x[0 \,..\, \pi(q) - 1]$ and $y[k + 1 - \pi(q) \,..\, k]$ match.[6]

At the beginning of each search attempt, the property $x[0 \,..\, q] = y[k - q + 1 \,..\, k]$ holds as an invariant. Then, if $x[q + 1] = y[k + 1]$, the length of the match is extended, unless $q = m - 1$, in which case a complete match of the pattern in the text is reported. However, if $x[q + 1] \neq y[k + 1]$, then the pattern is slid $(q + 1 - \pi(q))$ positions to the right by setting $q := \pi[q]$, thus maintaining the invariant. In either case, a progress is made at each iteration. The algorithm terminates its computation as soon as the end of the text is reached.

4.3 Modifications to the Verification Phase

The verification phase of the Linear-Wfr algorithm, which will be also referred to as Kmp-*verification*, is reported at lines 14 through 26 in Figure 5.

Let k and q be the two indices involved in the preceding description of the Kmp algorithm such that $x[0 \,..\, q] = y[k - q + 1 \,..\, k]$ holds as an invariant. Initially, both indices k and q are set to 0 (lines 4 and 5). When a verification terminates, the index k contains the last verified position of the text. Thus, when the subsequent verification phase takes place at position i, the algorithm checks whether position k precedes i (line 14). If this is the case, the algorithm starts a new verification phase from position i, by setting $k := i$ and $q := 0$ (lines 15 and 16), thus skipping all the positions from k to $i - 1$ in the text. Otherwise, if $k \geq i$, the previous verification is continued from position k. In both cases, the verification phase stops when k reaches position $i + m - 1$ (namely, the rightmost position of the current text window). If q becomes equal to m, an occurrence of the pattern in the text is reported at position i (line 21).

Figure 7 shows four working steps of the Linear-Wfr algorithm in the case of a pattern of length $m = 5$. Again, text positions that have already been verified are represented with bullets. In

[6]The values $\pi(q)$, for $0 \leq q \leq m$, are precomputed by the Kmp algorithm and stored in a table $\pi[0 \,..\, m]$ of size $m + 1$. In the case of the Linear-Wfr algorithm, this is done by procedure Precompute-Kmp-Failure-Function, called at line 2 of Figure 5.

Step 1, we assume that $k = 0$ and that a candidate position i has been found by the WFR filtering procedure. In this case, we have $k < i$, and thus a KMP-verification is started from position i. At the end of such verification, k is equal to $i + m - 1$. In Steps 2 and 3, new candidate positions i are found while $k \geq i$; thus, in both cases, the KMP-verification is continued from the previous verified position k. Finally, in Step 4, the WFR filtering procedure finds a new candidate position at a position i such that $i > k$. Then positions from $k + 1$ to $i - 1$ are not verified. As in Step 1, a new KMP-verification is started from position i.

4.4 Complexity Analysis

Next, we briefly discuss the time complexity of the LINEAR-WFR algorithm and informally prove that the number of computational steps performed by the algorithm during the searching phase is linear in the length of the text. We separately evaluate the time complexity of the two independent procedures which coexists in the workflow of the algorithm, namely, the filtering and the verification phases.

First, we consider the overall contribution given by the verification phase enclosed in lines 14 through 25 of Figure 5. Observe that the blocks starting at lines 14, 18, and 23 may modify the value of the variable k. However, in all cases, the value of k is always increased. Additionally, observe that the value of k is never decreased during the execution of the algorithm. Thus, because k is initialised to 0 and never exceeds the value n, we can state that the number of times the algorithm can execute such blocks is bounded by the length of the text. Thus, the overall contribution in time complexity given by the verification procedure is $O(n)$.

Next, we consider the overall contribution of the filtering phase in lines 7 through 12 of Figure 5. At a generic attempt of the filtering phase, the current window $y[i \,..\, j]$ of the text is scanned from right to left. In this context, k is the rightmost position of the text window that has been scanned during the previous attempt. Plainly, the inequality $j > k$ is always true at the beginning of each attempt. While scanning from right to left, the algorithm never goes below position $(k + 1)$ (see line 10). Thus, each text position is inspected at most once during the filtering phase. Hence, the overall contribution in time complexity given by the filtering procedure is also $O(n)$.

4.5 Evaluation in Practice

In this section, we briefly discuss the practical behaviour of the LINEAR-WFR algorithm when compared with the best results obtained by the TUNED-WFR algorithm described in the previous sections. The implementation of the LINEAR-WFR algorithm incorporates the improvements described in Section 3, namely, the chained-loop of q consecutive characters and the blind shift cycle with the additional use of a sentinel at the end of the text. We refer to such a variant as the LINEAR-WFRq algorithm.

Table 3 shows the running times obtained on a genome sequence (table GENOME), a protein sequence (table PROTEIN), and a natural language text (table NATLANG), respectively. It also includes the experimental results already reported in Table 2. For each group of algorithms, the best results have been underlined, whereas the best overall running times have been boldfaced.

From the experimental results, it turns out that the TUNED-WFRq variant always obtains better results than the LINEAR-WFRq variant; however, their running times are always very close and show the same sublinear behaviour. With the exception of very short patterns ($m = 2$), in all cases the slowdown produced by the new variant is less than 1.3%, and less than 0.5% in most cases. This behaviour, together with its linear worst-case time complexity, makes the LINEAR-WFRq variant one of the most preferable and competitive solutions.

Table 3. Comparison Among the Linear-Wfr$q$ (LWFRq) and the Tuned-Wfr$q$ (T-WFRq) Variants of the Wfr Algorithm

Genome	q/m	2	4	8	16	32	64	128	256	512	1,024
TWFRq		**9.94**$^{(2)}$	**8.16**$^{(3)}$	**3.77**$^{(4)}$	**2.75**$^{(4)}$	**2.31**$^{(5)}$	**2.04**$^{(5)}$	**1.96**$^{(5)}$	**1.87**$^{(5)}$	**1.71**$^{(7)}$	**1.65**$^{(8)}$
LWFR	1	25.90	15.99	9.91	6.30	4.06	2.93	2.50	2.19	1.91	1.73
	2	10.63	8.93	6.97	4.88	3.05	2.39	2.26	2.05	1.80	1.70
	3	—	8.17	4.89	3.37	3.01	2.50	2.18	1.98	1.76	1.67
	4	—	8.49	3.82	2.78	2.37	2.17	2.09	2.01	1.77	1.69
	5	—	—	9.58	2.93	**2.31**	2.05	**1.96**	1.89	1.79	1.72
	6	—	—	11.37	3.64	2.44	2.09	1.98	1.89	1.72	1.70
	7	—	—	12.51	3.55	2.51	2.09	2.01	1.89	1.73	1.66
	8	—	—	15.60	3.59	2.55	2.17	2.04	1.93	1.75	1.68
Slowdown		7.0%	0.1%	1.3%	1.1%	0.0%	0.5%	0.0%	1.0%	0.6%	0.6%

Protein	q/m	2	4	8	16	32	64	128	256	512	1,024
TWFRq		**6.97**$^{(2)}$	**4.52**$^{(2)}$	**3.45**$^{(2)}$	**2.79**$^{(3)}$	**2.38**$^{(4)}$	**2.11**$^{(4)}$	**2.09**$^{(4)}$	**2.00**$^{(4)}$	**1.86**$^{(5)}$	**1.80**$^{(5)}$
LWFR	1	9.81	8.04	6.58	5.14	3.59	2.93	2.65	2.30	2.03	1.89
	2	7.15	4.56	3.47	3.04	2.76	2.60	2.42	2.14	1.96	1.86
	3	—	7.75	3.84	2.80	2.39	2.17	2.13	2.11	1.97	1.86
	4	—	9.33	3.91	2.84	2.43	2.11	2.10	2.01	1.91	1.86
	5	—	—	10.56	3.22	2.52	2.17	2.15	2.01	1.88	1.81
	6	—	—	12.54	4.03	2.71	2.26	2.20	2.04	1.89	**1.80**
	7	—	—	15.59	4.00	2.79	2.33	2.28	2.11	1.92	1.82
	8	—	—	17.76	4.06	2.79	2.38	2.29	2.13	1.93	1.83
Slowdown		2.6%	0.9%	0.6%	0.3%	0.3%	0.0%	0.5%	0.5%	1.1%	0.0%

NatLang	q/m	2	4	8	16	32	64	128	256	512	1,024
T-WFRq		**7,04**$^{(2)}$	**4,72**$^{(2)}$	**3,58**$^{(2)}$	**2,88**$^{(3)}$	**2,45**$^{(4)}$	**2,16**$^{(4)}$	**2,09**$^{(4)}$	**2,02**$^{(4)}$	**1,89**$^{(4)}$	**1,82**$^{(7)}$
L-WFRq	1	10.80	9.22	7.54	5.44	3.88	3.06	2.72	2.37	2.09	1.94
	2	7.25	4.78	3.64	3.16	2.85	2.60	2.43	2.20	1.98	1.87
	3	—	7.88	4.15	2.89	2.48	2.23	2.14	2.09	1.97	1.87
	4	—	9.40	4.06	2.90	2.47	2.17	2.10	**2.02**	1.92	1.84
	5	—	—	10.60	3.25	2.60	2.21	2.16	2.03	1.92	1.83
	6	—	—	12.83	4.01	2.74	2.30	2.21	2.07	1.91	1.84
	7	—	—	15.28	3.98	2.80	2.35	2.26	2.10	1.94	1.84
	8	—	—	17.45	4.05	2.79	2.40	2.28	2.11	1.96	1.84
Slowdown		3.0%	1.3%	1.7%	0.3%	0.8%	0.5%	0.5%	0.0%	1.1%	0.5%

Running times refer to tests performed on a genome sequence (table Genome), a protein sequence (table Protein), and a natural language text (table NatLang). Patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i | 2 \leq i \leq 10\}$. In all cases, the mean over the running times (expressed in thousandths of seconds) of 1000 runs has been reported. In the case of the Tuned-Wfr$q$ algorithm, only the best results have been reported. For each such result, we reported, in round brackets, the value of q for which the best running time has been obtained. The best running times, for each group, have been underlined while the best overall running times have been also boldfaced.

5 EXPERIMENTAL RESULTS

We report in this section the results of an extensive experimentation of the Wfr algorithm and its variants against the most efficient solutions known in the literature for the online exact string matching problem. Specifically, the following 17 algorithms (implemented in 95 variants, depending on the values of their parameters) have been compared:

Table 4. Experimental Results on a Genome Sequence (Alphabet Size Is 4)

Genome sequence

m	2	4	8	16	32	64	128	256	512	1,024
BNDMq	$10.54^{(2)}$	$8.52^{(2)}$	$4.05^{(4)}$	$3.09^{(4)}$	$2.47^{(4)}$	$2.48^{(4)}$	$2.48^{(4)}$	$2.46^{(6)}$	$2.50^{(4)}$	$2.48^{(4)}$
SBNDMq	$10.10^{(2)}$	$7.71^{(2)}$	$3.98^{(4)}$	$3.05^{(4)}$	$2.47^{(4)}$	$2.45^{(4)}$	$2.49^{(4)}$	$2.47^{(4)}$	$2.48^{(4)}$	$2.47^{(4)}$
FSBNDMqs	$10.15^{(2,0)}$	$7.16^{(3,1)}$	$4.41^{(4,1)}$	$3.25^{(4,1)}$	$2.61^{(4,1)}$	$2.61^{(4,1)}$	$2.60^{(4,1)}$	$2.60^{(4,1)}$	$2.60^{(4,1)}$	$2.58^{(4,1)}$
KBNDM	18.04	10.98	8.49	5.95	4.17	3.19	3.12	3.11	3.10	3.12
BXSq	$13.41^{(2)}$	$8.28^{(3)}$	$4.84^{(4)}$	$3.31^{(4)}$	$2.62^{(4)}$	$2.62^{(4)}$	$2.63^{(4)}$	$2.63^{(4)}$	$2.63^{(4)}$	$2.61^{(4)}$
EBOM	10.98	9.14	8.23	6.41	4.75	3.49	2.92	2.58	2.39	2.46
BSDMq	$10.43^{(2)}$	$\mathbf{6.31}^{(3)}$	$4.06^{(4)}$	$3.08^{(4)}$	$2.46^{(6)}$	$2.18^{(6)}$	$2.07^{(6)}$	$2.01^{(6)}$	$1.98^{(6)}$	$1.97^{(6)}$
FJS	21.67	18.14	16.79	16.71	16.01	15.99	16.50	16.77	16.13	16.95
HASHq	–	$18.32^{(3)}$	$7.89^{(3)}$	$4.87^{(5)}$	$3.29^{(5)}$	$2.93^{(5)}$	$2.79^{(5)}$	$2.82^{(8)}$	$2.60^{(8)}$	$2.47^{(8)}$
FS-w	$17.72^{(4)}$	$12.32^{(2)}$	$9.52^{(2)}$	$7.97^{(2)}$	$6.96^{(2)}$	$6.27^{(2)}$	$5.72^{(2)}$	$5.31^{(4)}$	$4.90^{(4)}$	$4.59^{(4)}$
IOM	19.63	14.75	11.91	11.38	11.20	11.23	11.31	11.33	11.29	11.47
WOM	21.70	16.84	12.40	10.16	8.86	7.95	7.28	6.84	6.38	6.10
JOM	22.73	17.48	12.90	9.66	5.84	5.26	4.84	4.62	4.47	4.32
WFRq	$15.32^{(2)}$	$10.01^{(2)}$	$4.89^{(4)}$	$3.26^{(4)}$	$2.55^{(5)}$	$2.12^{(5)}$	$2.03^{(5)}$	$1.91^{(5)}$	$1.72^{(7)}$	$1.67^{(7)}$
TWFRq	$\mathbf{9.94}^{(2)}$	$8.16^{(3)}$	$\mathbf{3.77}^{(4)}$	$\mathbf{2.75}^{(4)}$	$\mathbf{2.31}^{(5)}$	$\mathbf{2.04}^{(5)}$	$\mathbf{1.96}^{(5)}$	$\mathbf{1.87}^{(5)}$	$\mathbf{1.71}^{(7)}$	$\mathbf{1.65}^{(8)}$
LWFRq	$10.63^{(2)}$	$8.17^{(3)}$	$3.82^{(4)}$	$2.78^{(4)}$	$\mathbf{2.31}^{(5)}$	$2.05^{(5)}$	$\mathbf{1.96}^{(5)}$	$1.89^{(5)}$	$1.72^{(6)}$	$1.66^{(7)}$
Speedup	**−1.5%**	+30%	**−5.3%**	**−9.8%**	**−6.1%**	**−6.4%**	**−5.3%**	**−6.9%**	**−14%**	**−16%**
EPSM	2.31	2.40	3.09	2.54	2.00	1.82	1.80	1.70	1.68	1.70
TSOq	$10.43^{(5)}$	$5.45^{(5)}$	$3.87^{(5)}$	$3.11^{(5)}$	$2.45^{(5)}$	$2.11^{(5)}$	–	–	–	–

Patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. In all cases, the mean over the running times (expressed in thousandths of seconds) of 1000 runs has been reported.

- BNDMq: the Backward Nondeterministic DAWG Matching algorithm [21] implemented using q-grams, with $1 \leq q \leq 8$;
- SBNDMq: the Simplified version of the Backward Nondeterministic DAWG Matching algorithm [1] implemented using q-grams, with $1 \leq q \leq 8$;
- FSBNDMqs: the Forward Simplified version [13, 22] of the BNDM algorithm [21] implemented using q-grams s-forward characters, with $1 \leq q \leq 8$ and $1 \leq s \leq 6$;
- KBNDM: the Factorized variant [5] BNDM algorithm [21];
- BXSq: the Backward Nondeterministic DAWG Matching algorithm [21] with Extended Shift [9] implemented using q-grams, with $1 \leq q \leq 8$;
- EBOM: the extended version [13] of the BOM algorithm [1];
- BSDMq: the Backward SNR DAWG Matching algorithm [15] using condensed alphabets with groups of q characters, with $1 \leq q \leq 8$;
- HASHq: the Hashing algorithm [20] using q-grams, with $3 \leq q \leq 5$;
- FS-w: the Multiple Windows version [16] of the Fast Search algorithm [3] implemented using w sliding windows, with $2 \leq w \leq 6$;
- IOM: the Improved Occurrence Matcher [4]
- WOM: the Worst Occurrence Matcher [4];
- JOM: the Jumping Occurrence Matcher [4];
- WFRq: the new Weak Factors Recognition variants implemented with a q-chained-loop, (with $1 \leq q \leq 8$);
- TWFRq: the new Tuned Weak Factors Recognition variants implemented with a q-chained-loop (with $1 \leq q \leq 8$) and a fast blind test-loop;

Table 5. Experimental Results on a Protein Sequence (Alphabet Size Is 21)

PROTEIN SEQUENCE

m	2	4	8	16	32	64	128	256	512	1,024
BNDMq	$8.75^{(2)}$	$4.61^{(2)}$	$3.39^{(2)}$	$2.73^{(2)}$	$2.32^{(2)}$	$2.38^{(2)}$	$2.40^{(2)}$	$2.39^{(2)}$	$2.39^{(2)}$	$2.36^{(2)}$
SBNDMq	$7.57^{(2)}$	$4.34^{(2)}$	$3.29^{(2)}$	**$2.72^{(2)}$**	**$2.31^{(2)}$**	$2.32^{(2)}$	$2.30^{(2)}$	$2.31^{(2)}$	$2.30^{(2)}$	$2.30^{(2)}$
FSBNDMqs	$7.24^{(2,1)}$	**$4.32^{(2,0)}$**	$3.32^{(2,0)}$	$2.74^{(2,0)}$	**$2.31^{(2,0)}$**	$2.32^{(2,0)}$	$2.30^{(2,0)}$	$2.31^{(2,0)}$	$2.32^{(2,0)}$	$2.30^{(2,0)}$
KBNDM	14.65	8.50	5.42	4.09	3.39	3.12	3.23	3.18	3.19	3.19
BXSq	$10.02^{(2)}$	$5.16^{(2)}$	$3.68^{(2)}$	$2.89^{(2)}$	$2.44^{(2)}$	$2.44^{(2)}$	$2.44^{(2)}$	$2.43^{(2)}$	$2.44^{(2)}$	$2.34^{(1)}$
EBOM	7.19	4.37	**3.19**	2.73	2.48	2.33	2.35	2.36	2.32	2.42
BSDMq	$7.31^{(2)}$	$4.72^{(2)}$	$3.69^{(3)}$	$3.00^{(3)}$	$2.63^{(4)}$	$2.36^{(6)}$	$2.25^{(6)}$	$2.20^{(6)}$	$2.18^{(6)}$	$2.18^{(6)}$
FJS	11.89	8.17	7.29	5.40	6.76	4.79	7.58	5.57	5.18	7.03
HASHq	—	$20.08^{(3)}$	$8.68^{(3)}$	$5.22^{(3)}$	$3.84^{(5)}$	$3.30^{(5)}$	$3.15^{(5)}$	$3.15^{(5)}$	$2.91^{(5)}$	$2.79^{(5)}$
FS-w	$7.99^{(4)}$	$5.16^{(4)}$	$3.71^{(4)}$	$3.11^{(4)}$	$2.81^{(4)}$	$2.73^{(4)}$	$2.67^{(4)}$	$2.62^{(4)}$	$2.67^{(4)}$	$3.00^{(4)}$
IOM	12.82	9.26	6.49	5.16	4.47	4.14	4.04	3.91	3.92	3.87
WOM	13.23	9.49	6.58	5.19	4.38	3.99	3.71	3.53	3.39	3.35
JOM	11.41	9.14	6.82	5.31	4.27	3.64	3.30	3.14	3.03	2.87
WFRq	$11.02^{(1)}$	$6.24^{(2)}$	$4.52^{(2)}$	$3.19^{(3)}$	$2.62^{(3)}$	$2.26^{(3)}$	$2.21^{(3)}$	$2.06^{(4)}$	$1.90^{(5)}$	$1.80^{(5)}$
TWFRq	**$6.97^{(2)}$**	$4.52^{(2)}$	$3.45^{(2)}$	$2.79^{(3)}$	$2.38^{(4)}$	**$2.11^{(4)}$**	**$2.09^{(4)}$**	**$2.00^{(4)}$**	**$1.86^{(5)}$**	**$1.80^{(5)}$**
LWFRq	$7.15^{(2)}$	$4.56^{(2)}$	$3.47^{(2)}$	$2.80^{(3)}$	$2.39^{(3)}$	**$2.11^{(4)}$**	$2.10^{(4)}$	$2.01^{(4)}$	$1.88^{(5)}$	**$1.80^{(6)}$**
SPEEDUP	**−3.1%**	+4.6%	+8.1%	+2.6%	+3.0%	**−9.0%**	**−7.1%**	**−9.1%**	**−14%**	**−17%**
EPSM	2.62	2.73	2.79	2.85	2.25	2.01	2.02	1.95	1.88	1.92
TSOq	$10.42^{(5)}$	$5.36^{(5)}$	$3.92^{(5)}$	$3.24^{(5)}$	$2.66^{(5)}$	$2.28^{(5)}$	—	—	—	—

Patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. Running times are expressed in thousandths of seconds.

- LWFRq: the new Linear Weak Factors Recognition variants implemented with a q-chained-loop (with $1 \leq q \leq 8$) and a fast blind test-loop.

For the sake of completeness, we also evaluated the following two string matching algorithms for *counting* occurrences (however, we did not take them into account in our comparison because they simply count the number of matching occurrences without reporting their positions):

- EPSM: the Exact Packed String Matching algorithm [12] based on SIMD instructions;
- TSOq: the Two-Way variant of [10] the Shift-Or algorithm [2] implemented with a loop unrolling of q characters, with $q \in \{3, 5, 9\}$.

All algorithms have been implemented in C and have been tested using the SMART tool [14] and executed locally on a MacBook Pro with 4 cores, a 2GHz Intel Core i7 processor, 16GB RAM 1,600MHz DDR3, 256KB of L2 cache, and 6MB of cache L3.[7] Comparisons have been performed in terms of running times, including any preprocessing time. Experimental evaluations relative to three real datasets are reported in Tables 4 through 6.

Our tests have been run on a genome sequence, a protein sequence, and an English text (each of size 10MB). Such sequences are provided by the research tool SMART, available online for download (additional details on the sequences can be found in Faro et al. [14]).

In the experimental evaluation, patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. In all cases, the mean over the running times (expressed in thousandths of seconds) of 1,000 runs has been reported.

[7] The SMART tool is available online at http://www.dmi.unict.it/~faro/smart/.

Table 6. Experimental Results on a Natural Language Sequence (Alphabet Size Is 120)

Natural language sequence

m	2	4	8	16	32	64	128	256	512	1,024
BNDMq	9.00$^{(2)}$	5,07$^{(2)}$	3,87$^{(2)}$	3,12$^{(2)}$	2,65$^{(4)}$	2,67$^{(4)}$	2,66$^{(4)}$	2,67$^{(4)}$	2,66$^{(4)}$	2,65$^{(4)}$
SBNDMq	7,89$^{(2)}$	4,74$^{(2)}$	3,72$^{(2)}$	3,06$^{(2)}$	2,63$^{(4)}$	2,62$^{(4)}$	2,63$^{(4)}$	2,64$^{(4)}$	2,63$^{(4)}$	2,64$^{(4)}$
FSBNDMqs	7,69$^{(2,1)}$	4,78$^{(2,0)}$	3,79$^{(2,0)}$	3,01$^{(3,1)}$	2,57$^{(3,1)}$	2,57$^{(3,1)}$	2,59$^{(3,1)}$	2,59$^{(3,1)}$	2,57$^{(3,1)}$	2,57$^{(3,1)}$
KBNDM	15.03	8.94	5.90	4.63	3.76	3.39	3.27	3.19	3.21	3.16
BXSq	10,32$^{(2)}$	5,55$^{(2)}$	4,16$^{(2)}$	3,14$^{(3)}$	2,63$^{(3)}$	2,73$^{(4)}$	2,73$^{(1)}$	2,52$^{(1)}$	2,38$^{(1)}$	2,13$^{(1)}$
EBOM	7.71	4.91	3.77	3.27	3.01	2.80	2.70	2.57	2.47	2.59
BSDMq	7,19$^{(2)}$	4,85$^{(2)}$	3,86$^{(2)}$	3,12$^{(3)}$	2,67$^{(6)}$	2,30$^{(6)}$	2,21$^{(6)}$	2,18$^{(6)}$	2,18$^{(6)}$	2,21$^{(6)}$
FJS	**6.52**	6.33	6.93	6.69	6.55	6.36	6.40	6.41	7.00	10.41
HASHq	—	20,19$^{(3)}$	8,68$^{(3)}$	5,20$^{(3)}$	3,85$^{(3)}$	3,28$^{(5)}$	3,17$^{(8)}$	3,17$^{(8)}$	2,88$^{(5)}$	2,68$^{(3)}$
FS-w	8,81$^{(4)}$	5,75$^{(4)}$	4,18$^{(4)}$	3,29$^{(4)}$	2,95$^{(4)}$	2,77$^{(4)}$	2,62$^{(4)}$	2,57$^{(4)}$	2,48$^{(4)}$	2,36$^{(4)}$
IOM	13.40	9.59	6.84	5.35	4.59	4.07	3.75	3.57	3.40	3.24
WOM	14.00	10.03	7.12	5.33	4.46	3.87	3.45	3.30	3.16	3.10
JOM	12.49	10.05	7.82	5.71	4.54	3.77	3.30	3.17	2.95	2.68
WFRq	11,83$^{(1)}$	6,45$^{(2)}$	4,66$^{(2)}$	3,22$^{(3)}$	2,67$^{(3)}$	2,30$^{(4)}$	2,22$^{(4)}$	2,07$^{(5)}$	1,91$^{(5)}$	1,82$^{(5)}$
TWFRq	7,04$^{(2)}$	**4,72**$^{(2)}$	**3,58**$^{(2)}$	**2,88**$^{(3)}$	**2,45**$^{(4)}$	**2,16**$^{(4)}$	**2,09**$^{(4)}$	**2,02**$^{(4)}$	**1,89**$^{(4)}$	**1,82**$^{(7)}$
LWFRq	7,25$^{(2)}$	4,78$^{(2)}$	3,64$^{(2)}$	2,89$^{(3)}$	2,47$^{(4)}$	2,17$^{(4)}$	2,10$^{(4)}$	**2,02**$^{(4)}$	1,91$^{(6)}$	1,83$^{(5)}$
Speedup	+7.9%	**−0.4%**	**−3.7%**	**−4.3%**	**−4.7%**	**−6.0%**	**−5.4%**	**−7.3%**	**−13%**	**−18%**

EPSM	2.62	2.74	3.00	2.83	2.24	2.01	2.02	1.95	1.89	1.93
TSOq	—	5.45$^{(5)}$	4.02$^{(5)}$	3.25$^{(5)}$	2.69$^{(5)}$	2.29$^{(5)}$	—	—	—	—

Patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. Running times are expressed in thousandths of seconds.

Tables 4, 5 and 6 summarise the running times of our evaluations. Each table is divided into three blocks. The first block present results relative to the most effective algorithms based on automata and character comparisons present in the literature. Best results among the first set of algorithms have been boldfaced to ease their localization. The second block contains the running times of our newly proposed algorithm Wfr and its variant, including the speedup (in percentage) obtained against the best running time in the first block: positive percentages denote running times worsening, whereas negative values denote performance improvements. Running times representing performance improvements have been boldfaced.

The last block, reported in a separated subtable, concerns the running times obtained by the best two algorithms for *counting* occurrences (however, as already remarked, these have not been included in our comparison).

Experimental results show that in the case of long patterns, the BSDMq algorithm obtains the best running times among the algorithms already present in the literature. However, in the case of short patterns, the BSDMq algorithm is second to the EBOM algorithm and some variants of the BNDM algorithm.

Our proposed Wfr algorithm performs well in several cases and turns out to be competitive against previous solutions. It even turns out to be faster than the BSDMq algorithm in the case of very long patterns ($m \geq 256$), as the shifts performed by the Wfr algorithm are quite long on average.

When the Wfr algorithm is implemented using an unchained-loop, its performance increases further. Specifically, the Wfr$q$ algorithm turns out to be the fastest solution for patterns with a moderate length and for long patterns ($m \geq 32$). Better performances are obtained in the case of

small alphabets, where the gain is up to 25%, whereas in the case of large alphabets, the gain is up to 14%.

6 CONCLUSION

We investigated a weak factor recognition approach based on hashing to the exact string matching problem and devised an algorithm which, despite its quadratic worst-case time complexity, exhibits a sublinear behaviour in practical cases. Experimental results show that under suitable conditions, our algorithm obtains better running times than the most efficient algorithms known in the literature. We also proposed a worst-case linear-time variant of our algorithm that maintains much the same practical behaviour.

We plan to investigate whether multiple hashing functions could be used to reduce the number of false positives in the searching phase. A deeper analysis of the hash function and data structures implemented will be performed in future work.

REFERENCES

[1] C. Allauzen, M. Crochemore, and M. Raffinot. 1999. Factor oracle: A new structure for pattern matching. In *SOFSEM'99: Theory and Practice of Informatics*. Lecture Notes in Computer Science, Vol. 1725. Springer, 291–306.

[2] R. Baeza-Yates and G. H. Gonnet. 1992. A new approach to text searching. *Communications of the ACM* 35, 10 (1992), 74–82.

[3] D. Cantone and S. Faro. 2005. Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm. *Journal of Automata, Languages and Combinatorics* 10, 5–6 (2005), 589–608.

[4] D. Cantone and S. Faro. 2014. Improved and self-tuned occurrence heuristics. *Journal of Discrete Algorithms* 28, C (2014), 73–84.

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

[6] D. Cantone, S. Faro, and A. Pavone. 2017. Speeding up string matching by weak factor recognition. In *Proceedings of the Prague Stringology Conference*. 42–50.

[7] C. Charras and T. Lecroq. 2014. *Handbook of Exact String Matching Algorithms*. King's College.

[8] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, et al. 1994. Speeding up two string-matching algorithms. *Algorithmica* 12, 4 (1994), 247–267.

[9] B. Durian, H. Peltola, L. Salmela, and J. Tarhio. 2010. Bit-parallel search algorithms for long patterns. In *SEA 2010: Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 6049. Springer, 129–140.

[10] B. Durian, T. Chhabra, S. S. Ghuman, T. Hirvola, H. Peltola, and J. Tarhio. 2014. Improved two-way bit-parallel search. In *Proceedings of the Stringology Conference*. 71–83.

[11] S. Faro. 2016. Evaluation and improvement of fast algorithms for exact matching on genome sequences. In *AlCoB 2016: Algorithms for Computational Biology*. Lecture Notes in Computer Science, Vol. 9702. Springer, 145–157.

[12] S. Faro and O. Külekci. 2014. Fast and flexible packed string matching. *Journal of Discrete Algorithms* 28, (2014), 61–72.

[13] S. Faro and T. Lecroq. 2009. Efficient variants of the backward-oracle-matching algorithm. *International Journal on Foundations in Computer Science* 20, 6 (2009), 967–984.

[14] S. Faro, T. Lecroq, S. Borzì, S. Di Mauro, and A. Maggio. 2016. The string matching algorithms research tool. In *Proceedings of the Stringology Conference*. 99–111.

[15] S. Faro and T. Lecroq. 2012. A fast suffix automata based algorithm for exact online string matching. In *CIAA 2012: Implementation and Application of Automata*. Lecture Notes in Computer Science, Vol. 7381. Springer, 149–158.

[16] S. Faro and T. Lecroq. 2012. A multiple sliding windows approach to speed up string matching algorithms. In *SEA 2012: Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 7276. Springer, 172–183.

[17] S. Faro and T. Lecroq. 2013. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys* 45, 2 (2013), Article 13.

[18] A. Hume and D. Sunday. 1991. Fast string searching. *In Proceedings of the Summer 1991 USENIX Conference*. 221–234.

[19] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing* 6, 1 (1977), 323–350.

[20] T. Lecroq. 2007. Fast exact string matching algorithms. *Information Processing Letters* 102, 6 (2007), 229–235.

[21] G. Navarro and M. Raffinot. 1998. A bit-parallel approach to suffix automata: Fast extended string matching. In *CPM 1998: Combinatorial Pattern Matching*. Lecture Notes in Computer Science, Vol. 1448. Springer, 14–33.

[22] H. Peltola and J. Tarhio. 2011. Variations of Forward-SBNDM. In *Proceedings of the Stringology Conference*. 3–14.

[23] S. Wu and U. Manber. 1994. *A Fast Algorithm for Multi-Pattern Searching*. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson.
[24] A. C. Yao. 1979. The complexity of pattern matching for a random string. *SIAM Journal on Computing* 8, 3 (1979), 368–387.

Received December 2017; revised November 2018; accepted December 2018