Journal of Discrete Algorithms 28 (2014) 61–72

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Fast and flexible packed string matching[☆]

Simone Faro[a,*], M. Oğuzhan Külekci[b]

[a] Dipartimento di Matematica e Informatica, Università di Catania, Italy
[b] İstanbul Medipol University, Faculty of Engineering and Natural Sciences, Turkey

ARTICLE INFO

Article history:
Available online 24 July 2014

Keywords:
Exact string matching
Text algorithms
Experimental algorithms
Online searching
Information retrieval

ABSTRACT

Searching for all occurrences of a pattern in a text is a fundamental problem in computer science with applications in many other fields, like natural language processing, information retrieval and computational biology. In the last two decades a general trend has appeared trying to exploit the power of the word RAM model to speed-up the performances of classical string matching algorithms. In this model an algorithm operates on words of length w, grouping blocks of characters, and arithmetic and logic operations on the words take one unit of time.
In this paper we use specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology, to design a very fast string matching algorithm. We evaluate our solution in terms of efficiency, stability and flexibility, where we propose to use the deviation in running time of an algorithm on distinct equal length patterns as a measure of stability.
From our experimental results it turns out that, despite their quadratic worst case time complexity, the new presented algorithm becomes the clear winner on the average in many cases, when compared against the most recent and effective algorithms known in literature.

1. Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the *exact string matching problem* consists in finding *all* occurrences of the pattern p in t. This problem has been extensively studied in computer science because of its direct application to many areas. Moreover, string matching algorithms are the basic components in many software applications and play an important role in theoretical computer science by providing challenging problems.

In a computational model where the matching algorithm is restricted to read all the characters of the text one by one the optimal complexity is $\mathcal{O}(n)$, and was achieved the first time by the well known Knuth–Morris–Pratt algorithm [26] (KMP). However, in many practical cases it is possible to avoid reading all the characters of the text achieving sub-linear performances on the average. The optimal average $\mathcal{O}(\frac{n \log_\sigma m}{m})$ time complexity [35] was reached for the first time by the Backward-DAWG-Matching algorithm [11] (BDM). However, all algorithms with a sub-linear average behavior may have to read all the text characters in the worst case. It is interesting to note that many of those algorithms have an even worse $\mathcal{O}(nm)$-time complexity in the worst-case [10,19,22].

[☆] A preliminary version of the results presented in this paper has been previously published in [15].
[*] Corresponding author.
E-mail addresses: faro@dmi.unict.it (S. Faro), okulekci@medipol.edu.tr (M.O. Külekci).

http://dx.doi.org/10.1016/j.jda.2014.07.003
1570-8667/© 2014 Elsevier B.V. All rights reserved.

In the last two decades a lot of work has been made in order to exploit the power of the word RAM model of computation to speed-up classical string matching algorithms. In this model, the computer operates on words of length w, thus blocks of characters are read and processed at once. This means that usual arithmetic and logic operations on the words all take one unit of time.

Most of the solutions which exploit the word RAM model are based on either the *bit-parallelism* technique or the *packed string matching* technique.

The *bit-parallelism* technique [1] takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w. Bit-parallelism is particularly suitable for the efficient simulation of nondeterministic automaton. The Shift-Or [1] (SO) algorithm is the first of this genre, which simulates efficiently the nondeterministic version of the KMP automaton and runs in $\mathcal{O}(n\lceil \frac{m}{w} \rceil)$. It is still considered among the best practical algorithms in the case of very short patterns and small alphabets [22,19]. Later a very fast BDM-like algorithm (BNDM), based on the bit-parallel simulation of the nondeterministic suffix automaton, was presented in [31]. Some variants of the BNDM algorithm [16,18,12,32] are among the most practical efficient solutions in literature (see [22,19]). However, the bit-parallel encoding requires one bit per pattern symbol, for a total of $\lceil \frac{m}{w} \rceil$ computer words. Thus, as long as a pattern fits in a single computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrade considerably as $\lceil \frac{m}{w} \rceil$ grows. Though there are a few techniques to maintain good performances in the case of long patterns [28,13,8,9], such limitation is intrinsic.

In the *packed string matching* technique multiple characters are packed into one larger word, so that the characters can be compared in bulk rather than individually. In this context, if the characters of a string are drawn from an alphabet of size σ, then $\lfloor \frac{w}{\log \sigma} \rfloor$ different characters fit in a single word, using $\lceil \log \sigma \rceil$ bits per characters. The packing factor is $\alpha = \lfloor \frac{w}{\log \sigma} \rfloor$.[1]

A first theoretical result in packed string matching was proposed by Fredriksson [23]. He presented a general scheme that can be applied to speed-up many pattern matching algorithms. His approach relies on the use of the *four Russian* technique (i.e. tabulation), achieving in favorable cases an $\mathcal{O}(n^{\varepsilon} m)$-space and $\mathcal{O}(\frac{n}{m \log \sigma} + n^{\varepsilon} m + occ)$-time complexity, where $\varepsilon > 0$ denotes an arbitrary small constant, and *occ* denotes the number of occurrences of p in t. Bille [5] presented an alternative solution with $\mathcal{O}(\frac{n}{\log_\sigma n} + m + occ)$-time and $\mathcal{O}(n^{\varepsilon} + m)$-space complexities by an efficient segmentation and coding of the KMP automaton. Belazzougui [2] proposed a packed string matching algorithm which works in $\mathcal{O}(\frac{n}{m} + \frac{n}{\alpha} + m + occ)$ time and $\mathcal{O}(m)$ space, reaching the optimal $\mathcal{O}(\frac{n}{\alpha} + occ)$-time bound for $\alpha \le m \le \frac{n}{\alpha}$. More recently, Belazzougui and Raffinot [3] introduced an average-optimal time string matching algorithm for packed strings, which achieves $\mathcal{O}(n/m)$ query time. However, none of these results leads to practical algorithms.

The first algorithm that achieves good practical and theoretical results was very recently proposed by Ben-Kiki et al. [4]. The algorithm is based on two specialized packed string instructions, the pcmpestrm and the pcmpestri instructions [29], and reaches the optimal $\mathcal{O}(\frac{n}{\alpha} + occ)$-time complexity requiring only $\mathcal{O}(1)$ extra space. Moreover the authors showed that their algorithm turns out to be among the fastest string matching solutions in the case of very short patterns. However, it has to be noticed that on the family of Intel Sandy Bridge processors, which we consider as the benchmark platform for the implementations throughout the study, pcmpestrm and pcmpestri have 2-cycle throughput and 7- and 8-cycle latency, respectively [29].

When the length of the searched pattern increases, another algorithm named Streaming SIMD Extensions Filter (SSEF), presented by Külekci in [27] (and extended to multiple pattern matching in [14]), exploits the advantages of the word-RAM model. Specifically it uses a filter method that inspects blocks of characters instead of reading them one by one. Despite its $\mathcal{O}(nm)$ worst case time complexity, the SSEF algorithm turns out to be among the fastest solutions when searching for long patterns [22,19].

Efficient solutions have been also designed for searching on packed DNA sequences [33,17]. However in this paper we do not take into account this type of solutions since they require a different type of data representation.

Streaming SIMD technology offers single-instructions to perform a variety of tests on packed strings. Unfortunately those instructions are heavier than other instructions provided in the same family as a consequence of their relatively high latencies. Hence, in this paper we focus on design of algorithms using instructions with low latency and high throughput, when compared with those used in [4].

Specifically we introduce a new practical and efficient algorithm for the exact packed string matching problem that turns out to be faster than the best algorithms known in literature in most practical cases [15].

The newly presented algorithm, named Exact Packed String Matching (EPSM), is based on four different search procedures used for, respectively, very short patterns ($0 < m < \frac{\alpha}{2}$), short patterns ($\frac{\alpha}{2} \le m < \alpha$), medium length patterns ($m \ge \alpha$) and long patterns ($m \ge w$). All search procedures have an $\mathcal{O}(nm)$ worst case time complexity. However, they have very good performances on average. In the case of very short patterns, i.e. when $m \le \frac{\alpha}{2}$, the first two search procedures achieve, respectively, an $\mathcal{O}(n + occ)$ and an optimal $\mathcal{O}(\frac{n}{\alpha} + occ)$-time complexity.

The paper is organized as follows. In Section 2, we introduce some notions and terminologies, while in Section 3 we describe the model of computations we assume for describing our solutions. We then present a new algorithm for the packed

[1] However, it is noteworthy that in practice supporting varying packing factors seems not very possible in todays SIMD technologies such as the Intel's SSE instruction set. The practical implementations assume the ASCII alphabet with size 8-bits per symbol and the packing factor used is 16 (32) symbols per block in 128-bits (256-bits) SIMD technologies.

	0	1	2	3	4	5	6	7	8	9	10	11
a:	0110.	0010.	0111.	1010.	0010.	1110.	0010.	0100.	0110.	0111.	0100.	0010
b:	0100.	0010.	0000.	0111.	1111.	0010.	0010.	1100.	0110.	0100.	1110.	0010
r:	0	1	0	0	0	0	1	0	1	0	0	1

Fig. 1. An example of the application of wscmp(a, b), assuming $w = 48$, $\gamma = 4$ and $\alpha = 12$.

string matching problem in Section 4 and report experimental results under various conditions in Section 5. Conclusions are given in Section 6.

2. Notions and terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $m > 0$ is represented as a finite array $p[0..m-1]$ of characters from a finite alphabet Σ of size σ. Thus, $p[i]$ will denote the $(i+1)$-st character of p, for $0 \le i < m$, and $p[i..j]$ will denote the *factor* (or *substring*) of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \le i \le j < m$. In some cases we will denote by p_i the $(i+1)$-st character of p, so that $p_i = p[i]$ and $p = p_0 p_1 \dots p_{m-1}$.

We indicate with symbol w the number of bits in a computer word and with symbol $\gamma = \lceil \log \sigma \rceil$ the number of bits used for encoding a single character of the alphabet Σ. The number of characters of the alphabet that fit in a single word is shown by $\alpha = \lfloor w/\gamma \rfloor$. Without loss in generality we will assume along the paper that γ divides w and that α is an even value.

In chunks of α characters, the string p is represented by an array $P[0..k-1]$ of length $k = (m-1)/\alpha + 1$. In particular we denote $P = P_0 P_1 P_2 \dots P_{k-1}$, where $P_i = p_{i\alpha} p_{i\alpha+1} p_{i\alpha+2} \dots p_{i\alpha+\alpha-1}$, for $0 \le i < k$. The last block P_{k-1} is not complete if $m \bmod \alpha \neq 0$. In that case, the rightmost remaining characters of the block are set to zero.

Although different values of α and γ are possible, in most cases we assume that $\alpha = 16$ and $\gamma = 8$, which is the most common case when working with characters in ASCII code and in a word RAM model with 128-bit registers, which are almost all available in recent commodity processors supporting single instruction multiple data (SIMD) operations.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|" and the `left shift` "$\ll$" operator (which shifts to the left its first argument by a number of bits equal to its second argument).

3. The model

In the design of our algorithms we use specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data simultaneously via a set of special instructions working on limited number of special registers.

Although the usage of SIMD is explored deeply in multimedia processing, implementation of encryption/decryption algorithms, and on some scientific calculations, it has not been much addressed in pattern matching.

In the design of our algorithms we make use of the following specialized word-size packed instructions. For each instruction we describe how it could be emulated by using SSE specialized intrinsics.

3.1. **wscmp(a, b) *(*word-size compare instruction*)***

The wscmp instruction compares two w-bit words, handled as a block of α characters. In particular if $a = a_0 a_1 \dots a_{\alpha-1}$ and $b = b_0 b_1 \dots b_{\alpha-1}$ are the two w-bit integer parameters, wscmp(a, b) returns an α-bit value $r = r_0 r_1 \dots r_{\alpha-1}$, where $r_i = 1$ if and only if $a_i = b_i$, and $r_i = 0$ otherwise. Fig. 1 shows an example of the application of wscmp(a, b), assuming $w = 48$, $\gamma = 4$ and $\alpha = 12$.

The wscmp specialized instruction can be emulated in constant time by using the following sequence of specialized SIMD instructions

```
h ← _mm_cmpeq _epi8(a, b)
r ← _mm _movemask_epi8(h)
```

Specifically the _mm_cmpeq_epi8 instruction compares two 128-bit words, handled as a block of sixteen 8-bit values, and returns a 128-bit value $h = h_0 h_1 \dots h_{15}$, where $h_i = 1^8$ if and only if $a_i = b_i$, and $h_i = 0^8$ otherwise. It has a 0.5-cycle throughput and a 1-cycle latency.

The _mm_movemask_epi8 instruction gets a 128-bit parameter h, handled as sixteen 8-bit integers, and creates a 16-bit mask from the most significant bits of the 16 integers in h, and zero extends the upper bits.

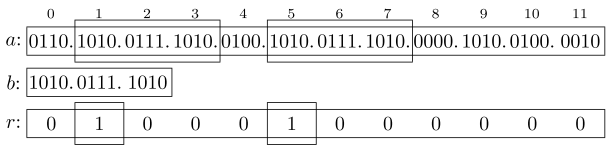
0 1 2 3 4 5 6 7 8 9 10 11
a: 0110. 1010. 0111. 1010. 0100. 1010. 0111. 1010. 0000. 1010. 0100. 0010
b: 1010. 0111. 1010
r: 0 1 0 0 0 1 0 0 0 0 0 0

Fig. 2. An example of the application of wsmatch(a, b), assuming $w = 48$, $\gamma = 4$, $\alpha = 12$ and $k = 3$.

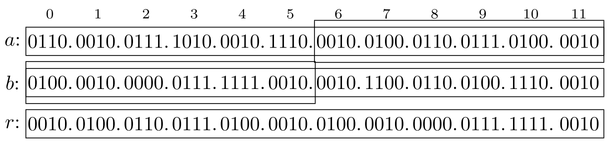
0 1 2 3 4 5 6 7 8 9 10 11
a: 0110. 0010. 0111. 1010. 0010. 1110. 0010. 0100. 0110. 0111. 0100. 0010
b: 0100. 0010. 0000. 0111. 1111. 0010. 0010. 1100. 0110. 0100. 1110. 0010
r: 0010. 0100. 0110. 0111. 0100. 0010. 0100. 0010. 0000. 0111. 1111. 0010

Fig. 3. An example of the application of wsblend(a, b), assuming $w = 48$, $\gamma = 4$ and $\alpha = 12$.

3.2. wsmatch$(\boldsymbol{a}, \boldsymbol{b})$ (word-size matching instruction)

The wsmatch instruction reports all occurrences of a short string b in a w-bit parameter a, handled as a string of α characters. The parameter b is a string of length $k \leq \alpha$.

Specifically, if $a = a_0 a_1 \ldots a_{\alpha-1}$, and $b = b_0 b_1 \ldots b_{k-1}$, then the wsmatch$(a, b)$ instruction returns an α-bit integer value, $r = r_0 r_1 \ldots r_{\alpha-1}$, where $r_i = 1$ if and only if $a_{i+j} = b_j$ for $j = 0 \ldots k-1$, i.e. an occurrence of b in a begins at position i. Notice that $r_i = 0$ for $\alpha - k < i < \alpha$, since no occurrence of b in a could begin at a position greater than $\alpha - k$. Fig. 2 shows an example of the application of wsmatch(a, b), assuming $w = 48$, $\gamma = 4$, $\alpha = 12$ and $k = 3$.

The wsmatch(a, b) instruction can be emulated in constant time by using the following sequence of SIMD specialized instructions

$$h \leftarrow \text{\_mm\_mpsadbw\_epu8}(a, b)$$

$$\ell \leftarrow \text{\_mm\_cmpeq\_epi8}(h, z)$$

$$r \leftarrow \text{\_mm\_movemask\_epi8}(\ell)$$

where z is a 128-bit register with all bits set to 0, i.e. $z = 0^{128}$.

Specifically the _mm_mpsadbw_epu8(a, b) instruction gets two 128-bit words, handled as a block of sixteen 8-bit values, and returns a 128-bit value $r = r_0 r_1 \ldots r_7$ (handled as a block of eight 16-bit values), where r_i is computed as $r_i = \sum_{j=0}^{3} |a_{i+j} - b_j|$ for $i = 0 \ldots 7$.

Thus we have that $r_i = 0^{16}$ if and only if $a_{i+j} = b_j$ for $j = 0 \ldots 3$, i.e. an occurrence of the prefix of b with length 4 begins in a at position i. The _mm_mpsadbw_epu8 instruction has 1-cycle throughput and a 4-cycle latency. The _mm_cmpeq_epi8 and _mm_movemask_epi8 instructions have been described above.

3.3. wsblend$(\boldsymbol{a}, \boldsymbol{b})$ (word-size blend instruction)

The wsblend instruction blends two w-bit parameters, handled as two blocks of α characters. Specifically if $a = a_0 a_1 \ldots a_{\alpha-1}$ and $b = b_0 b_1 \ldots b_{\alpha-1}$, the instruction returns a w-bit integer $r = r_0 r_1 \ldots r_{\alpha-1}$, where $r_i = a_{i+\alpha/2}$, if $0 \leq i < \alpha/2$, and $r_i = b_{i-\alpha/2}$ if $\alpha/2 \leq i < \alpha$, i.e. $r = a_{\frac{\alpha}{2}} a_{\frac{\alpha}{2}+1} \ldots a_{\alpha-1} b_0 b_1 \ldots b_{\frac{\alpha}{2}-1}$. Fig. 3 shows an example of the application of wsblend(a, b), assuming $w = 48$, $\gamma = 4$ and $\alpha = 12$.

The wsblend(a, b) instruction can be emulated in constant time by using the following sequence of SIMD specialized instructions

$$h \leftarrow \text{\_mm\_blend\_epi16}(a, b, c)$$

$$\text{SHUFFLE} \leftarrow \text{\_MM\_SHUFFLE}(1, 0, 3, 2)$$

$$r \leftarrow \text{\_mm\_shuffle\_epi32}(h, \text{SHUFFLE})$$

Such instruction blends two 128-bit integers, $a = a_0 a_1 \ldots a_7$ and $b = b_0 b_1 \ldots b_7$, handled as packed 16-bit integers, according to a third parameter c. In particular it returns a 128-bit integer $r = r_0 r_1 \ldots r_7$ where $r_i = a_i$ if $c_i = 0$, and $r_i = b_i$ otherwise. If we set $c = 1^{64} 0^{64}$ we get $r = b_0 b_1 b_2 b_3 a_4 a_5 a_6 a_7$. The _mm_blend_epi16 instruction has 0.5-cycle throughput and a 1-cycle latency.

The _mm_shuffle_epi32 instruction shuffles a w-bit parameter, $a = a_0 a_1 a_2 a_3$, handled as four 32-bit values, according to the order of the _MM_SHUFFLE macro. In this case we get $r = a_2 a_3 a_0 a_1$. The _mm_shuffle_epi32 instruction has 1-cycle throughput and a 1-cycle latency.

3.4. **wscrc(*a*) *(*word-size cyclic redundancy check*)***

The wscrc instruction computes the 32-bit cyclic redundancy checksum (CRC) signature for a w-bit parameter. It is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data and can also be used as a hash function.

The wscrc(a) instruction can be emulated in constant time by using the following SIMD specialized instruction

$$r \leftarrow \text{\_mm\_crc32\_u64}(a)$$

Specifically the _mm_crc32_u64(a) instruction computes the 32 bit cyclic redundancy check of a 64-bit block according to a polynomial. Such instruction has a 1-cycle throughput and a 3-cycle latency, thus provides a robust and fast way of computing hash values.

3.5. Additional specialized instructions

In addition to the above listed instructions, given an α-bit register r, in our description we make use of the symbol $\{r\}$ to indicate the set of bits in r whose value is set. More formally, given an α-bit register $r = r_0r_1r_2\ldots r_{\alpha-1}$, we have $\{r\} = \{i \mid 0 \le i < \alpha \text{ and } r_i = 1\}$. Moreover, given a value $s \in \mathbb{N}$, we use for simplicity the expression $s + \{r\}$ to indicate the set of values $\{s + i \mid i \in \{r\}\}$.

The cardinality of the set $\{r\}$ can be computed in constant time by using the SIMD specialized instruction

$$n \leftarrow \text{\_mm\_popcnt\_u32}(r)$$

which calculates the number of bits of the parameter r that are set to 1. Such instruction has 1-cycle throughput and a 3-cycle latency.

Differently the list of values in $\{r\}$ can be efficiently listed in $\mathcal{O}(\alpha)$-time and $\mathcal{O}(1)$-space, or using a tabulation approach, in $\mathcal{O}(|\{r\}|)$-time and $\mathcal{O}(2^\alpha)$-space. In the latter case we need an $\mathcal{O}(\alpha 2^\alpha)$-time preprocessing phase in order to address the 2^α possible registers.

4. A new packed string matching algorithm

In this section we present the new packed string matching algorithm, named Exact Packed String Matching (EPSM) algorithm. EPSM is based on three different auxiliary algorithms, which we name EPSM*a*, EPSM*b* and EPSM*c*, respectively. The EPSM*a*, EPSM*b* and EPSM*c* procedures have been previously described in a preliminary result presented in [14].

The first two auxiliary algorithms, EPSM*a* and EPSM*b*, are designed to search for patterns of length, at most, $\alpha/2$. When the length of the pattern is longer than $\alpha/2$ the algorithms adopt a filter mechanism: they first search for a substring of the pattern of length $\alpha/2$ and, when a candidate occurrence has been found, a naive check follows. The EPSM*c* algorithm adopts a filtering based solution and has been designed for searching medium length and long patterns.

All three algorithms run in $\mathcal{O}(nm)$ worst case time complexity and use, respectively, $\mathcal{O}(\min\{m, \alpha\})$, $\mathcal{O}(1)$ and $\mathcal{O}(2^k)$ additional space, where k is a constant parameter. However, when $m \le \alpha/2$ the EPSM*a* and EPSM*b* algorithms reach, respectively, an $\mathcal{O}(m\alpha + \frac{mn}{\alpha} + occ)$ and $\mathcal{O}(\frac{n}{\alpha} + occ)$ time complexity.

The EPSM*a* procedure is designed to be extremely fast in the case of very short patterns, i.e. when $m \le \frac{\alpha}{2}$, the EPSM*b* procedure turns out to be a good choice when $\frac{\alpha}{2} \le m < \alpha$, while EPSM*c* turns out to be effective when $\alpha \le m < w$.

In practical cases we tuned the EPSM algorithm in order to run EPSM*a* when $0 < m < 4$, EPSM*b* when $4 \le m < 16$, EPSM*c* when $m \ge 16$. The pseudocode of the three algorithms is shown in Fig. 4.

4.1. EPSMa: searching for very short patterns

The EPSM*a* algorithm is designed to be extremely fast in the case of very short patterns and although it could be adapted to work for longer patterns its performance degrades as the length of the patterns increases.

The main idea in EPSM*a* algorithm is to mark the positions of the very short pattern's symbols on the investigated text chunk. Assume we have m α-bits long bitmaps, where the bits of the ith bitmap are set to 1 at the positions of the appearances of the corresponding symbol p_i, and to 0 elsewhere. For instance, if $P = ab$, the first bitmap will indicate the positions that letter a is observed, and the second one will do the same for letter b. If ab appears on the current block such that $t_it_{i+1} = ab$, for $0 \le i < (\alpha - 1)$, then the position i on the first bitmap and position $i + 1$ on the second bitmap should be set to 1. Thus, the bitwise *and* between the one bit left-shifted second bitmap and the first bitmap should report a 1 at position i. Careful readers will quickly realize that, the occurrence of the reverse pattern ba will also produce a 1 at ith position. To avoid this error, we follow a sequential procedure such that at each step we perform the *and* operation between the previous bitmask and the newly computed bitmap that marks the positions of the current pattern symbol. Notice that initially the bitmask is set to all 1s. The details of the EPSM*a* is as follows.

The preprocessing of the algorithm (lines 1–4) is computed on the prefix of the pattern of length $m' = \min\{m, \frac{\alpha}{2}\}$. If $m' = m$ the whole pattern is preprocessed and searched, otherwise the algorithm works as a filter, searching for all

```
EPSMa($p, m, t, n$)
1.  $m' \leftarrow \min\{m, \alpha/2\}$
2.  for $i \leftarrow 0$ to $(m'-1)$ do
3.      for $j \leftarrow 0$ to $\alpha - 1$ do
4.          $B_i[j] \leftarrow p[i]$
5.  for $i \leftarrow 0$ to $(n/\alpha) - 1$ do
6.      $r \leftarrow 1^\alpha$
7.      for $j \leftarrow 0$ to $m' - 1$ do
8.          $s_j \leftarrow$ wscmp($T_i, B_j$)
9.          $r \leftarrow r$ & $(s_j \ll j)$
10.     if $m = m'$
11.     then report occurrences at $i\alpha + \{r\}$
12.     else check positions $i\alpha + \{r\}$
13.     for $j \leftarrow 0$ to $m - 2$ do
14.         check position $(i+1)\alpha - j$

EPSMb($p, m, t, n$)
1.  $m' \leftarrow \min\{m, \alpha/2\}$
2.  $p' \leftarrow p[0..m'-1]$
3.  for $i \leftarrow 0$ to $(n/\alpha) - 1$ do
4.      $r \leftarrow$ wsmatch($T_i, p'$)
5.      if $r \neq 0^\alpha$ then
6.          if $m = m'$
7.          then report occurrences at $i\alpha + \{r\}$
8.          else check positions $i\alpha + \{r\}$
9.      $S \leftarrow$ wsblend($T_i, T_{i+1}$)
10.     $r \leftarrow$ wsmatch($S, p'$)
11.     if $r \neq 0^\alpha$ then
12.         if $m = m'$
13.         then report occurrences at $i\alpha + \frac{\alpha}{2} + \{r\}$
14.         else check positions $i\alpha + \frac{\alpha}{2} + \{r\}$

EPSMc($p, m, t, n$)
1.  $mask \leftarrow 0^{\alpha-k}1^k$
2.  for $i \leftarrow 0$ to $m - \alpha$ do
3.      $v \leftarrow$ wscrc($p[i..i+\alpha-1]$)
4.      $v \leftarrow v$ & $mask$
5.      $L[v] \leftarrow L[v] \cup \{i\}$
6.  $sh \leftarrow (\lfloor m/(\alpha/2) \rfloor - 1)$
7.  for $i \leftarrow 0$ to $(n/(\alpha/2)) - 1$ do
8.      $v \leftarrow$ wscrc($T_i$)
9.      $v \leftarrow v$ & $mask$
10.     for all $j \in L[v]$ do
11.         if $0 \leq i - j < n - m$
12.         then check position $i - j$
13.     $i \leftarrow i + sh$

EPSM($p, m, t, n$)
1.      if $m \leq \alpha/2$ then return EPSMa($p, m, t, n$)
2.      if $m \leq \alpha$ then return EPSMb($p, m, t, n$)
3.      return EPSMc($p, m, t, n$)
```

Fig. 4. The EPSM algorithm and its EPSM*a*, the EPSM*b* and the EPSM*c* procedures.

occurrences of the prefix with length m' and, after an occurrence has been found, naively checking the whole occurrence of the pattern.

Specifically the preprocessing phase consists in constructing an array B of m' different strings of length α. Each string of the array exactly fits in a word of w bits. The i-th string in the array B consists of α copies of the character p_i. More formally the string $B[i]$, for $0 \leq i < m'$, is defined as $B[i] = (p_i)^\alpha$.

For instance, if $p = ab$ is a pattern of length $m = 2$, $\gamma = 8$ and $w = 128$, then B consists of two strings of length $\alpha = 16$, defined as $B[0] = a^{16}$ and $B[1] = b^{16}$. The preprocessing phase of the algorithm requires $\mathcal{O}(\min\{m, \frac{\alpha}{2}\}\alpha)$-time and $\mathcal{O}(\min\{m, \frac{\alpha}{2}\})$-space.

The searching phase of the algorithm (lines 5–14) processes the text t in chunks of α characters. Let $N = \frac{n}{\alpha} - 1$ and let $T = T_0T_1 \ldots T_N$ be the string t represented in chunks of characters. Each block of the text, T_i, is compared with the strings in the array B using the instruction wscmp.

Let $s_j = b_0b_1 \ldots b_{\alpha-1}$ be the α-bit register returned by the instruction wscmp($T_i, B[j]$), for $0 \leq j < m'$. It can be easily proved that $b_k = 1$ if and only if the k-th character of the block T_i is equal to p_j, i.e. if and only if $T_i[k] = p_j$ (remember that $B[j] = (p_j)^\alpha$). Finally let $r = r_0r_1 \ldots r_{\alpha-1}$ be the α-bit register defined as $r = s_0$ & $(s_1 \ll 1)$ & $(s_2 \ll 2)$ & $\cdots$ & $(s_{m'-1} \ll (m'-1))$.

It is easy to prove that $p[0..m'-1]$ has an occurrence beginning at position j of T_i if and only if $r_j = 1$. In fact $r_j = 1$ only if $s_k[j+k] = 1$, for $k = 0 \dots m'-1$, which implies that $T_i[j+k] = p_k$, for $k = 0 \dots m'-1$.

Then, if $m = m'$ the algorithm reports the occurrences of the pattern at positions $i\alpha + \{r\}$, if any. Otherwise we know that occurrences of the prefix of the pattern with length $\alpha/2$ begin at positions $i\alpha + \{r\}$. Thus the algorithm checks the occurrences beginning at those positions.

If we maintain, for each value r, with $0 \le r < 2^\alpha$, a list of the values in the set $\{r\}$, the naive check of the occurrences can be done in $\mathcal{O}(|\{r\}|m)$-time. When $m = m'$ the occurrences can be reported in $\mathcal{O}(|\{r\}|)$-time.

Finally, observe that the $m'-1$ possible occurrences crossing the blocks T_i and T_{i+1} are naively checked by the algorithm (lines 13–14).

The overall time complexity of the EPSM*a* algorithm is $\mathcal{O}(nm)$, because in the worst case a naive check is required for each position of the text. However, when $m \le \frac{\alpha}{2}$ the EPSM*a* algorithm achieves an $\mathcal{O}(n + occ)$ time complexity, where *occ* is the number of occurrences of the pattern p in the text t.

4.2. EPSMb: searching for short patterns

The EPSM*b* algorithm searches for the whole pattern when its length is less or equal to $\alpha/2$ and works as a filter algorithm for longer patterns. However, it is based on a more efficient filtering technique and turns out to be faster in the second case.

In a chunk of α characters, the occurrences of the pattern are investigated via the simple *wsmatch* function described above. Since the length of P is less than or equal to $\alpha/2$, the appearances beginning in the first half of the investigated block end in the second half ordinarily, and need no further processing. However, it is possible that an occurrence beginning in the second half of the chunk may extend to the next chunk. Thus, instead of scanning in chunks of α symbols, we traverse the text in chunks of $\alpha/2$ characters. We perform the *wsblend* operation to create an α-symbols long chunk by concatenating the second half of the current chunk with the first half of the next chunk, and check whether an occurrence exists on the boundary of the text blocks. The formal definition of the EPSM*b* is as follows.

Let m' be the minimum between $\alpha/2$ and m. Moreover let p' be the prefix of p of length m'. The searching phase of the algorithm (lines 3–14) processes the text t in chunks of α characters.

Let $N = \frac{n}{\alpha} - 1$ and let $T = T_0 T_1 \dots T_N$ be the string t represented in chunks of characters. Each block of the text, T_i, is searched one by one for occurrences of the string p' using the instruction wsmatch.

Specifically, let $r = r_0 r_1 \dots r_{\alpha-1}$ be the α-bit register returned by the instruction wsmatch(T_i, p'), for $0 \le j < m'$. We have that $r_j = 1$ if and only if an occurrence of p' begins at positions j of the block T_i, for $0 \le j < \alpha/2$. Then, if $m' = m$ (and hence $p = p'$) the algorithm simply returns positions $i\alpha + j$, such that $r_j = 1$. Otherwise, if $m' < m$, the algorithm naively checks for the whole occurrences of the pattern starting at positions $i\alpha + j$, such that $r_j = 1$.

Notice that generally packed string matching instructions allow to read only blocks T_i of α characters (128 bits in the case of SSE instructions), where $T_i = t[i\alpha..(i+1)\alpha - 1]$. Occurrences of the pattern beginning in the second half of the block T_i are checked separately. In particular a new block, S, obtained by applying the instruction wsblend(T_i, T_{i+1}), is processed in a similar way as block T_i. In this case we report all occurrences of the pattern beginning at positions $i\alpha + \alpha/2 + j$, with $0 \le j < \alpha/2$. One may argue that why blending is used instead of simply shifting the window. The reason is the SSE instructions used in this context require the operands to be 16-byte aligned in memory, where the performance degrades significantly otherwise. Thus, blending is more advantageous.

The resulting algorithm has an $\mathcal{O}(nm)$ worst case time complexity and requires $\mathcal{O}(1)$ additional space. When $m \le \alpha/2$ the algorithm reaches the optimal $\mathcal{O}(n/\alpha + occ)$ worst case time complexity.

4.3. EPSMc: searching for long patterns

The EPSM*c* algorithm is designed to be faster for medium and long patterns. It is based on a simple filtering method and uses a hash function for computing fingerprint values on blocks of α characters in a similar way as in Rabin–Karp algorithm [25]. The fingerprint values are computed by using a hash function $h : \Sigma^\alpha \to \{0, 1, \dots, 2^k - 1\}$, for a constant parameter $k \le \alpha$, that may vary according to the text or the pattern structures. In practical cases we chose a value of $k = 11$, which gave us best results during the benchmarks.

The hash function h used for computing the fingerprint value is computed in a very fast way by using the wscrc specialized instruction, and in particular

$$h(a) = \text{wscrc}(a)\ \&\ 0^{\alpha-k}1^k$$

for each $A \in \Sigma^\alpha$, and where we remember that & is the bitwise and operation.

During the preprocessing phase (lines 1–6) a fingerprint value of k bits is computed for all substrings of the pattern of length α. Then a table L of size 2^k is computed in order to store starting positions of all substrings of the pattern, indexed by their fingerprint values. In particular we have

$$L[v] = \{i \mid h(p[i..i+\alpha-1]) = v\}$$

for all $0 \le v < 2^k$. Thus the preprocessing phase of the EPSM*c* algorithm takes $\mathcal{O}(m + 2^k)$-time.

Let $N = \frac{n}{\alpha} - 1$ and let $T = T_0 T_1 \ldots T_N$ be the string t represented in chunks of characters.

During the searching phase (lines 7–13) the EPSM*c* algorithm inspects the blocks of the text in steps of $(\lfloor m/(\alpha/2) \rfloor - 1)$ positions.[2] For each inspected block T_i the fingerprint value $h(T_i)$ is computed and all positions in the set $\{i\alpha - j \mid j \in L[h(T_i)]\}$ are naively checked.

It is easy to observe that the EPSM*c* algorithm has an $\mathcal{O}(nm)$ worst case time complexity. However, despite its worst case time behavior it turns out to be very effective in practical cases.

5. Experimental results

In this section we present experimental results in order to compare the performances of our newly presented algorithms against the best solutions known in literature in the case of short patterns. We consider all the fastest algorithms in the case of short patterns as listed in a recent experimental evaluation by Faro and Lecroq [22,19]. In particular we compared EPSM with the following algorithms:

- the Hash algorithm using groups of q characters [30] (HASHq);
- the Extended Backward Oracle Matching algorithm [16,18] (EBOM);
- the Fast-Search algorithm using h sliding windows [6,7,21] (TVSBS);
- the TVSBS algorithm using h sliding windows [34,21] (TVSBS);
- the Shift-Or algorithm [1] (SO);
- the Shift-Or algorithm with q-grams [12] (UFNDMq);
- the Fast-Average-Optimal-Shift-Or algorithm [24] (FAOSOq);
- the q-gram filtering algorithm [13] (QFqf);
- the Forward Simplified BNDM algorithm using q-grams and f forward characters [16,18,32] (FSBNDMqf);
- the Forward Simplified BNDM algorithm using h sliding windows [16,18,21] (FSBNDM-Wh);
- the Packed SSE-Filter algorithm using SIMD instructions [27] (SSEF);
- the Packed Crochemore-Perrin algorithm using SIMD instructions [4] (SSECP).

We remember that the EPSM algorithm consists of the EPSM*a* algorithm, when $m < 4$, of the EPSM*b* algorithm when $4 \leq m \leq 16$, and of the EPSM*c* algorithm when $m > 16$.

In the case of algorithms making use of q grams, the value of q ranges in the set $\{2, 4, 6\}$. All algorithms have been implemented in the C programming language and have been tested using the Smart tool [20] for exact string matching. The experiments were executed locally on a machine running Ubuntu 11.10 (oneiric) with Intel i7-2600 processor with 16 GB memory. Algorithms have been compared in terms of running times, including any preprocessing time. For the evaluation we used a genome sequence, a protein sequence and a natural language text (English language), all sequences of 4 MB. The sequences are provided by the Smart research tool. For each input file, we have searched sets of 1000 patterns of fixed length m randomly extracted from the text, for m ranging from 2 to 32 (short patterns). Then, the mean of the running times has been reported.

Table 1, Table 2 and Table 3 show the experimental results obtained for a gnome sequence, a protein sequence and a natural language text, respectively.

In the case of algorithms using q-grams we have reported only the best result obtained by its variants. The values of q which obtained the best running times are reported as apices. Running times are expressed in hundredths of seconds, best results have been boldfaced and underlined.

5.1. Efficiency

From experimental results it turns out that the EPSM algorithm has mostly the best performances for short patterns. When searching on a genome sequence it is second only to the BNDMq algorithm for $12 \leq m \leq 14$ and to the SSECP algorithm when $m = 6$. Observe however that the EPSM algorithm is (up to 2 times) faster than the SSECP algorithm in most cases.

When searching on a natural language text the EPSM algorithm obtains in most cases the best results, and is second to BNDM based algorithms only for $20 \leq m \leq 22$.

For increasing lengths of the pattern the performances of the EPSM algorithm remain stable, underlining a linear trend on average. However, the performances of other algorithms based on shift heuristics, slightly increase. This is more evident when searching on a protein sequence, where the algorithms based on bit-parallelism and q grams turn out to be the faster solutions for longer patterns. However, in this latter cases the EPSM algorithm is always very close the best solutions.

It is interesting to observe that the EPSM algorithm is faster than the SSECP algorithm in almost all cases, and the gap is more evident in the case of longer patterns. In fact, despite its optimal worst case time complexity, the SSECP algorithm shows an increasing trend on average, while the EPSM algorithm shows a linear behavior.

[2] Actually, using $(\alpha/2)$ term instead of α directly stems from the limitation in practice that, the *crc* value can be computed on 64 bits rather than 128 bits in the current SSE instruction sets. Thus, any *crc* of a block defines the *crc* of the largest possible initial portion of the block.

Table 1
Experimental results for searching short (on top) and long (on bottom) patterns on a genome sequence. Running times are expressed in hundredths of seconds, best results have been boldfaced and underlined.

m	2	4	6	8	12	16	20	24
HASHq	–	10.87$^{(3)}$	7.98$^{(3)}$	7.40$^{(3)}$	5.78$^{(3)}$	4.57$^{(3)}$	4.05$^{(3)}$	3.70$^{(5)}$
EBOM	8.01	7.76	7.96	7.61	7.62	6.68	6.08	5.52
FS-Wh	12.73$^{(2)}$	9.70$^{(2)}$	8.67$^{(2)}$	8.33$^{(4)}$	7.71$^{(4)}$	7.84$^{(4)}$	7.44v	7.61$^{(4)}$
TVSBS-Wh	11.93$^{(2)}$	9.72$^{(2)}$	8.75$^{(2)}$	8.34$^{(2)}$	7.62$^{(2)}$	7.81$^{(2)}$	7.38$^{(2)}$	7.54$^{(2)}$
SO	7.86	7.80	7.91	7.89	7.77	7.93	7.80	7.88
FAOSOq	–	10.65$^{(2)}$	8.19$^{(2)}$	6.35$^{(2)}$	5.55$^{(2)}$	4.40$^{(4)}$	3.65$^{(4)}$	3.46$^{(4)}$
QFqs	–	7.54$^{(3,3)}$	6.12$^{(4,3)}$	5.04$^{(4,3)}$	3.11$^{(4,3)}$	2.65$^{(4,3)}$	2.42$^{(4,3)}$	2.22$^{(6,2)}$
FSBNDMqf	10.38$^{(2,0)}$	7.61$^{(4,2)}$	5.98$^{(4,1)}$	4.71$^{(4,1)}$	3.58$^{(4,1)}$	3.06$^{(6,2)}$	2.67$^{(6,2)}$	2.44$^{(6,2)}$
UFNDMq	8.54$^{(2)}$	6.12$^{(4)}$	4.71$^{(4)}$	4.07$^{(4)}$	3.30$^{(6)}$	2.84$^{(6)}$	2.55$^{(6)}$	2.36$^{(6)}$
SSECP	2.65	2.87	3.17	3.60	6.53	5.96	5.80	5.72
EPSM	**2.09**$^{(a)}$	**2.27**$^{(b)}$	**3.23**$^{(b)}$	**3.25**$^{(b)}$	**3.28**$^{(b)}$	**2.39**$^{(b)}$	**2.47**$^{(c)}$	**1.91**$^{(c)}$

m	32	64	128	256	512	1024	2048	4096
HASHq	3.10$^{(5)}$	2.61$^{(5)}$	2.15$^{(5)}$	1.84$^{(5)}$	1.66$^{(5)}$	1.56$^{(5)}$	1.54$^{(5)}$	1.53$^{(5)}$
EBOM	4.85	3.48	2.56	2.13	1.99	2.17	2.81	4.26
FS-Wh	7.86$^{(2)}$	7.37$^{(2)}$	7.04$^{(2)}$	6.11$^{(2)}$	5.99$^{(2)}$	5.32$^{(2)}$	4.97$^{(2)}$	4.61$^{(2)}$
TVSBS-Wh	7.15$^{(2)}$	6.62$^{(2)}$	6.69$^{(2)}$	6.52$^{(2)}$	6.52$^{(2)}$	6.66$^{(2)}$	6.59$^{(2)}$	6.58$^{(2)}$
SO	7.80	6.68	6.77	6.70	6.71	6.54	6.49	6.62
FAOSOq	4.30$^{(4)}$	4.33$^{(4)}$	4.34$^{(4)}$	4.31$^{(4)}$	4.35$^{(4)}$	4.32$^{(4)}$	4.34$^{(4)}$	4.33$^{(4)}$
QFqs	1.99$^{(6,2)}$	1.66$^{(6,2)}$	1.40$^{(6,2)}$	1.26$^{(6,2)}$	1.20$^{(6,2)}$	**1.17**$^{(6,2)}$	**1.13**$^{(6,2)}$	**1.13**$^{(6,2)}$
FSBNDM-Wh	3.56$^{(2)}$	3.55$^{(2)}$	3.57$^{(2)}$	3.55$^{(2)}$	3.55$^{(2)}$	3.56$^{(2)}$	3.57$^{(2)}$	3.55$^{(2)}$
FSBNDMqf	2.15$^{(6,1)}$	2.16$^{(6,1)}$	2.16$^{(6,1)}$	2.15$^{(6,1)}$	2.15$^{(6,1)}$	2.16$^{(6,1)}$	2.16$^{(6,1)}$	2.01$^{(6,2)}$
UFNDMq	2.24$^{(6)}$	2.24$^{(6)}$	2.23$^{(6)}$	2.23$^{(6)}$	2.23$^{(6)}$	2.23$^{(6)}$	2.24$^{(6)}$	2.24$^{(6)}$
SSEF	2.91	2.03	1.53	1.33	1.26	1.31	1.37	1.49
SSECP	5.52	5.32	5.20	5.18	5.17	5.10	5.20	5.26
EPSM	**1.75**$^{(c)}$	**1.46**$^{(c)}$	**1.26**$^{(c)}$	**1.21**$^{(c)}$	**1.19**$^{(c)}$	1.21$^{(c)}$	1.26$^{(c)}$	1.43$^{(c)}$

Table 2
Experimental results for searching short (on top) and long (on bottom) patterns on a protein sequence. Running times are expressed in hundredths of seconds, best results have been boldfaced and underlined.

m	2	4	6	8	12	16	20	24
HASHq	–	10.70$^{(3)}$	7.86$^{(3)}$	7.63$^{(3)}$	5.30$^{(3)}$	4.23$^{(3)}$	3.62$^{(3)}$	3.31$^{(3)}$
EBOM	6.54	3.58	2.83	2.62	2.33	2.20	2.11	2.03
FS-Wh	7.40$^{(6)}$	5.07$^{(6)}$	4.00$^{(6)}$	3.42$^{(6)}$	2.85$^{(6)}$	2.60$^{(6)}$	2.46$^{(6)}$	2.40$^{(6)}$
TVSBS-Wh	7.45$^{(4)}$	6.16$^{(6)}$	4.89$^{(6)}$	4.24$^{(6)}$	3.45$^{(6)}$	2.56$^{(6)}$	2.68$^{(6)}$	2.50$^{(6)}$
SO	7.88	7.91	7.83	7.78	7.79	8.03	7.79	7.99
FAOSOq	–	6.14$^{(2)}$	5.50$^{(2)}$	4.22$^{(4)}$	3.41$^{(4)}$	3.37$^{(4)}$	2.77$^{(6)}$	2.72$^{(6)}$
QFqs	–	4.72$^{(2,8)}$	3.25$^{(2,6)}$	2.96$^{(3,4)}$	2.49$^{(3,4)}$	2.18$^{(3,4)}$	2.00$^{(3,4)}$	1.89$^{(3,4)}$
FSBNDM-Wh	8.66$^{(8)}$	5.52$^{(4)}$	4.24$^{(4)}$	3.61$^{(4)}$	3.03$^{(4)}$	2.74$^{(4)}$	2.54$^{(4)}$	2.40$^{(4)}$
FSBNDMqf	7.80$^{(2,1)}$	4.53$^{(2,0)}$	3.11$^{(2,0)}$	3.00$^{(3,1)}$	2.42$^{(3,1)}$	**2.11**$^{(3,1)}$	**1.96**$^{(3,1)}$	**1.88**$^{(3,1)}$
UFNDMq	6.95$^{(2)}$	4.53$^{(2)}$	3.55$^{(2)}$	3.13$^{(2)}$	2.63$^{(2)}$	2.37$^{(2)}$	2.18$^{(4)}$	2.04$^{(4)}$
SSECP	2.67	2.87	3.17	3.62	3.97	3.70	3.55	3.47
EPSM	**2.11**$^{(a)}$	**1.95**$^{(b)}$	**2.26**$^{(b)}$	**2.25**$^{(b)}$	**2.24**$^{(b)}$	2.37$^{(b)}$	2.44$^{(c)}$	1.91$^{(c)}$

m	32	64	128	256	512	1024	2048	4096
HASHq	2.91$^{(3)}$	2.55$^{(5)}$	2.06$^{(5)}$	1.78$^{(5)}$	1.59$^{(5)}$	1.52$^{(5)}$	1.52$^{(5)}$	1.53$^{(5)}$
EBOM	1.93	1.77	1.65	1.61	1.65	1.94	2.67	4.19
FS-Wh	2.31$^{(6)}$	2.17$^{(6)}$	2.04$^{(6)}$	1.98$^{(6)}$	1.95$^{(6)}$	1.94$^{(6)}$	1.92$^{(6)}$	1.92$^{(6)}$
TVSBS-Wh	2.27$^{(6)}$	2.02$^{(6)}$	1.71$^{(6)}$	1.53$^{(6)}$	1.44$^{(6)}$	1.41$^{(6)}$	1.39$^{(6)}$	1.38$^{(6)}$
SO	7.90	7.62	6.72	6.74	6.36	6.75	6.68	6.77
FAOSOq	4.30$^{(4)}$	4.27$^{(4)}$	4.31$^{(4)}$	4.33$^{(4)}$	4.35$^{(4)}$	4.29$^{(4)}$	4.27$^{(4)}$	4.33$^{(4)}$
QFqs	1.75$^{(3,4)}$	1.50$^{(4,3)}$	1.28$^{(4,3)}$	**1.16**$^{(4,3)}$	**1.09**$^{(4,3)}$	**1.07**$^{(4,3)}$	**1.07**$^{(4,3)}$	**1.06**$^{(4,3)}$
FSBNDM-Wh	2.21$^{(4)}$	2.22$^{(4)}$	2.20v	2.22$^{(4)}$	2.21$^{(4)}$	2.21$^{(4)}$	2.22$^{(4)}$	2.21$^{(4)}$
FSBNDMqf	1.74$^{(3,1)}$	1.74$^{(3,1)}$	1.74$^{(3,1)}$	1.74$^{(3,1)}$	1.74$^{(3,1)}$	1.74$^{(3,1)}$	1.75$^{(3,1)}$	1.74$^{(3,1)}$
UFNDMq	1.94$^{(4)}$	1.94$^{(4)}$	1.95$^{(4)}$	1.95$^{(4)}$	1.95$^{(4)}$	1.95$^{(4)}$	1.95$^{(4)}$	1.95$^{(4)}$
SSEF	2.90	2.02	1.57	1.35	1.29	1.30	1.36	1.50
SSECP	3.35	3.28	3.24	3.21	3.20	3.22	3.24	3.27
EPSM	**1.73**$^{(c)}$	**1.45**$^{(c)}$	**1.24**$^{(c)}$	1.18$^{(c)}$	1.17$^{(c)}$	1.19$^{(c)}$	1.25$^{(c)}$	1.41$^{(c)}$

Table 3
Experimental results for searching short (on top) and long (on bottom) patterns on a natural language text (English). Running times are expressed in hundredths of seconds, best results have been boldfaced and underlined.

m	2	4	6	8	12	16	20	24
HASHq	–	10.43$^{(3)}$	7.79$^{(3)}$	7.59$^{(3)}$	5.31$^{(3)}$	4.23$^{(3)}$	3.67$^{(3)}$	3.29$^{(3)}$
EBOM	7.14	4.42	3.76	3.45	3.25	3.08	2.98	2.87
FS-Wh	7.44$^{(6)}$	6.11$^{(6)}$	5.02$^{(6)}$	4.36$^{(6)}$	3.48$^{(6)}$	3.24$^{(6)}$	3.02$^{(6)}$	2.87$^{(6)}$
TVSBS-Wh	7.49$^{(6)}$	6.42$^{(6)}$	5.34$^{(6)}$	4.74$^{(6)}$	3.68$^{(6)}$	3.25$^{(6)}$	2.94$^{(6)}$	2.74$^{(6)}$
SO	7.88	7.66	7.87	7.73	7.81	7.69	7.84	7.95
FAOSOq	–	7.05$^{(2)}$	5.75$^{(2)}$	4.75$^{(4)}$	3.49$^{(4)}$	3.40$^{(4)}$	2.82$^{(6)}$	2.73$^{(6)}$
QFqs	–	5.93$^{(2,6)}$	4.38$^{(2,6)}$	3.67$^{(3,4)}$	2.85$^{(4,3)}$	**2.41**$^{(4,3)}$	**2.20**$^{(4,3)}$	2.08$^{(4,3)}$
FSBNDM-Wh	8.53$^{(1)}$	6.65$^{(2)}$	5.31$^{(2)}$	4.56$^{(4)}$	3.80$^{(4)}$	3.39$^{(4)}$	3.16$^{(4)}$	2.93$^{(4)}$
FSBNDMqf	7.75$^{(2,0)}$	5.77$^{(2,0)}$	4.04$^{(2,0)}$	3.60$^{(3,1)}$	3.01$^{(3,1)}$	2.65$^{(3,1)}$	2.40$^{(4,1)}$	2.22$^{(4,1)}$
UFNDMQ4	7.23$^{(2)}$	5.12$^{(2)}$	4.23$^{(4)}$	3.54$^{(4)}$	2.91$^{(4)}$	2.55$^{(4)}$	2.33$^{(4)}$	2.18$^{(4)}$
SSECP	2.66	2.87	3.17	3.62	4.64	4.17	4.05	3.90
EPSM	**2.11**$^{(a)}$	**2.29**$^{(b)}$	**2.58**$^{(b)}$	**2.58**$^{(b)}$	**2.57**$^{(b)}$	**2.41**$^{(b)}$	2.48$^{(c)}$	**1.93**$^{(c)}$

m	32	64	128	256	512	1024	2048	4096
HASHq	2.92$^{(3)}$	2.65$^{(5)}$	2.11$^{(5)}$	1.80$^{(3)}$	1.59$^{(3)}$	1.47$^{(3)}$	1.42$^{(3)}$	1.40$^{(3)}$
EBOM	2.75	2.46	2.14	1.91	1.88	2.12	2.80	4.24
FS-Wh	2.68$^{(6)}$	2.39$^{(6)}$	2.05$^{(6)}$	1.85$^{(6)}$	1.71$^{(6)}$	1.60$^{(6)}$	1.55$^{(6)}$	1.52$^{(6)}$
TVSBS-Wh	2.49$^{(6)}$	2.23$^{(6)}$	1.87$^{(6)}$	1.64$^{(6)}$	1.52$^{(6)}$	1.43$^{(6)}$	1.40$^{(6)}$	1.38$^{(6)}$
SO	7.86	6.62	6.91	6.79	6.69	6.80	6.67	6.80
FAOSOq	4.27$^{(4)}$	4.31$^{(4)}$	4.35$^{(4)}$	4.29$^{(4)}$	4.31$^{(4)}$	4.34$^{(4)}$	4.32$^{(4)}$	4.38$^{(4)}$
QF$q,s$	1.91$^{(4,3)}$	1.62$^{(4,3)}$	1.38$^{(4,3)}$	**1.08**$^{(6,2)}$	**1.15**$^{(6,2)}$	**1.11**$^{(6,2)}$	**1.09**$^{(6,2)}$	**1.08**$^{(6,2)}$
FSBNDM-Wh	2.72$^{(4)}$	2.72$^{(4)}$	2.72$^{(4)}$	2.73$^{(4)}$	2.73$^{(4)}$	2.74$^{(4)}$	2.72$^{(4)}$	2.74$^{(4)}$
FSBNDMqf	2.07$^{(4,1)}$	2.07$^{(4,1)}$	2.07$^{(4,1)}$	2.08$^{(4,1)}$	2.08$^{(4,1)}$	2.08$^{(4,1)}$	2.07$^{(4,1)}$	2.08$^{(4,1)}$
UFNDMQ4	2.08	2.08	2.07	2.09	2.09	2.08	2.08	2.08
SSEF	2.89	2.00	1.48	1.30	1.26	1.32	1.36	1.50
SSECP	3.79	3.31	3.03	2.85	2.86	2.85	2.85	2.86
EPSM	**1.76**$^{(c)}$	**1.47**$^{(c)}$	**1.26**$^{(c)}$	1.19$^{(c)}$	1.18$^{(c)}$	1.20$^{(c)}$	1.26$^{(c)}$	1.44$^{(c)}$

5.2. Flexibility

Flexibility is used as an attribute of various types of systems. In the field of string matching, it refers to algorithms that can adapt when changes in the input data occur. Thus a string matching algorithm can be considered flexible when, for instance, it maintains good performances for both short and long patterns, or in the case of both small and large alphabets.

Most string matching algorithms obtain good performances only in the case of long patterns sacrificing their performance for short ones. This is a common behavior, for instance, for all algorithm which make use of a sliding window approach (HASHq, EBOM, FS-Wh and TVSBS-Wh). Such approach allows the pattern to slide along the text by performing subsequent shifts. Each shift can be at most as long as the length of the pattern. It turns out that statistically the shift increases when the length of the pattern increases, or when the size of the alphabet increases.

A decreasing trend in running times can be observer also in the case of suffix automata based algorithms (FSBNDM-Wh, FSBNDMqf and QFqs). Although bit-parallel algorithms are designed to be extremely efficient in the case of short patterns, also this class of algorithms suffers of a lack in flexibility.

Only packed string matching algorithms turn out to have good performances for short patterns. This is the case of the SSECP algorithm whose performances, unfortunately, degrade when the length of the pattern increases.

On the contrary, the performances of the EPSM algorithm do not depend on pattern lengths and thus it is the only algorithm which maintains very good performances for both short and long patterns. The performances of the EPSM algorithm are maintained also when the size of the alphabet decreases.

Thus we can state that the EPSM algorithm is the most flexible algorithm among the best solutions known in literature to date.

5.3. Stability

We evaluate the stability of an algorithm as the standard deviation of running times observed during the evaluation. Algorithm stability is an important feature in string matching when real time processing is needed. Such value shows how much variation exists from the average, i.e. the mean of the running times. A low standard deviation indicates that the running times tend to be very close to the mean, underlying a high stability of the algorithm. On the other hand an high standard deviation indicates that the running times are spread out over a large range of values, thus indicating a low stability. See Tables 4–6.

Table 4
Values of standard deviation observed while searching short patterns on a genome sequence. Values are expressed in hundredths of seconds, best results have been boldfaced and underlined.

m	2	4	6	8	10	12	14	16	18	20	24	28
HASH*q*	–	2.35	2.43	1.93	0.94	0.67	0.43	0.54	0.26	0.33	0.30	0.21
EBOM	2.58	2.65	2.55	2.43	2.14	1.67	1.59	1.27	0.93	0.87	0.79	0.68
FS-W*h*	2.47	2.57	2.61	2.71	2.50	2.56	2.52	2.79	2.60	2.64	2.55	2.50
TVSBS-W*h*	2.42	2.31	2.23	2.56	2.48	2.45	2.53	2.33	2.14	2.07	1.84	1.65
SO	2.47	2.52	2.42	2.50	2.46	2.48	2.51	2.46	2.49	2.49	2.52	2.47
FAOSO*q*	–	2.48	2.38	1.10	0.58	0.71	0.74	0.60	0.72	0.67	0.63	0.18
QF*qs*	–	2.45	1.01	0.66	0.34	0.14	0.14	0.14	0.14	0.09	0.12	0.08
FSBNDM*qf*	2.36	2.41	1.00	0.40	0.28	0.29	0.21	0.14	0.17	0.10	0.13	0.09
UFNDM*q*	2.55	0.86	0.57	0.27	**0.22**	**0.11**	**0.11**	0.14	0.11	0.11	0.14	0.11
SSECP	**0.05**	**0.07**	**0.09**	**0.25**	2.44	1.97	1.68	1.50	1.35	1.19	0.95	0.88
EPSM	0.09	0.11	0.34	0.36	0.34	0.33	0.33	**0.11**	**0.10**	**0.07**	**0.10**	**0.07**

Table 5
Values of standard deviation observed while searching short patterns on a protein sequence. Values are expressed in hundredths of seconds, best results have been boldfaced and underlined.

m	2	4	6	8	10	12	14	16	18	20	24	28
HASH*q*	–	2.57	2.46	1.41	0.86	0.62	0.41	0.29	0.25	0.33	0.08	0.11
EBOM	1.34	0.18	0.39	0.13	0.11	0.15	0.11	**0.09**	0.12	0.09	0.11	0.07
FS-W*h*	2.23	0.91	0.52	0.37	0.32	0.30	0.25	0.24	0.23	0.23	0.23	0.21
TVSBS-W*h*	2.60	1.21	0.81	0.56	0.46	0.33	0.31	0.77	0.22	0.21	0.19	0.15
SO	2.52	2.52	2.49	2.44	2.50	2.44	2.52	2.45	2.43	2.41	2.44	2.49
FAOSO*q*	–	1.13	2.58	0.44	0.40	0.24	0.17	0.18	0.29	0.13	0.14	0.09
QF*qs*	–	1.47	0.21	**0.08**	**0.09**	**0.10**	**0.09**	**0.09**	0.09	0.09	0.08	0.10
FSBNDM-W*h*	–	1.30	0.65	0.38	0.26	0.22	0.17	0.16	0.14	0.14	0.11	0.13
FSBNDM*qf*	2.75	0.68	0.50	0.12	0.09	0.10	0.10	**0.09**	0.09	0.09	0.09	0.08
UFNDM*q*	1.33	0.40	0.33	0.15	0.20	0.16	0.11	0.11	0.11	0.10	0.09	0.08
SSECP	**0.06**	**0.15**	**0.09**	0.22	2.15	1.72	1.29	1.24	1.06	0.96	0.78	0.60
EPSM	0.11	0.53	0.10	0.10	0.10	0.13	0.10	**0.09**	**0.08**	**0.08**	**0.07**	**0.06**

Table 6
Values of standard deviation observed while searching short patterns on a natural language text (English). Values are expressed in hundredths of seconds, best results have been boldfaced and underlined.

m	2	4	6	8	10	12	14	16	18	20	24	28
HASH*q*	–	2.16	2.54	1.48	0.89	0.63	0.49	0.29	0.24	0.16	0.18	0.13
EBOM	1.68	1.32	1.02	0.89	0.78	0.74	0.61	0.60	0.56	0.51	0.45	0.41
FS-W*h*	3.16	2.36	1.58	1.27	1.05	0.92	0.82	0.77	0.71	0.65	0.59	0.52
TVSBS-W*h*	3.08	2.63	1.94	1.50	1.16	0.96	0.82	0.78	0.68	0.60	0.50	0.43
SO	2.47	2.53	2.47	2.49	2.51	2.56	2.47	2.44	2.40	2.51	2.40	2.42
FAOSO*q*	–	1.90	0.85	0.55	0.72	0.77	0.52	0.82	0.67	0.64	0.57	0.18
QF*qs*	–	2.31	1.03	0.85	0.67	0.23	0.18	0.16	0.17	0.14	0.12	0.11
FSBNDM*qf*	–	2.23	0.76	0.55	0.34	0.31	0.25	0.24	0.23	0.21	0.17	0.16
UFNDM*q*	2.13	0.94	0.36	0.30	**0.12**	**0.08**	**0.10**	0.15	0.10	**0.12**	0.11	**0.10**
SSECP	**0.10**	**0.11**	**0.10**	**0.14**	2.03	1.74	1.44	1.28	1.22	1.17	1.09	1.02
EPSM	0.11	0.13	0.82	0.84	0.79	0.78	0.77	**0.12**	**0.08**	**0.12**	**0.07**	**0.10**

It turns out from our observations that almost all algorithms have a low stability for short patterns while their stability increases when the length of the pattern increases. Such behavior becomes more evident for larger alphabets.

Sometimes an opposite behavior can be observed when searching on texts over a small alphabet like DNA sequences. This is the case, for instance, of FS-W*h* and TVSBS-W*h* algorithms, whose stability decreases for small alphabets when the length of the pattern gets longer. Observe also that the SSECP algorithm shows such behavior for both small and large alphabets.

6. Conclusions

We presented a new packed exact string matching algorithm based on the Intel streaming SIMD extensions technology. The presented algorithm, named EPSM, is based on three auxiliary algorithms which are used when $0 < m < 4$, $m \geq 4$, and $m \geq 16$, respectively. Despite the $\mathcal{O}(nm)$-worst case time complexity the resulting algorithm turns out to be very fast in the case of very short patterns. From our experimental results it turns out that the EPSM algorithm is in general the best solutions when $m \leq 32$. It could be interesting to investigate the possibility to improve the performances of packed string matching algorithms by introducing shift heuristics.

References

[1] R. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Commun. ACM 35 (10) (1992) 74–82.
[2] D. Belazzougui, Worst case efficient single and multiple string matching in the RAM model, in: Proceedings of the 21st International Workshop on Combinatorial Algorithms, IWOCA, 2010, pp. 90–102.
[3] D. Belazzougui, M. Raffinot, Average optimal string matching in packed strings, in: P.G. Spirakis, M. Serna (Eds.), Proceedings of the 8th International Conference on Algorithms and Complexity, CIAC, in: Lecture Notes in Computer Science, vol. 7878, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 37–48.
[4] O. Ben-Kiki, P. Bille, D. Breslauer, L. Gąsieniec, R. Grossi, O. Weimann, Optimal packed string matching, in: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 13, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011, pp. 423–432.
[5] P. Bille, Fast searching in packed strings, J. Discrete Algorithms 9 (1) (2011) 49–56.
[6] D. Cantone, S. Faro, Fast-Search: a new efficient variant of the Boyer–Moore string matching algorithm, in: Proceedings of the Second International Workshop Experimental and Efficient Algorithms, WEA, Ascona, Switzerland, in: Lecture Notes in Computer Science, vol. 2647, Springer-Verlag, Berlin, 2003, pp. 247–258.
[7] D. Cantone, S. Faro, Fast-search algorithms: new efficient variants of the Boyer–Moore pattern-matching algorithm, J. Autom. Lang. Comb. 10 (5/6) (2005) 589–608.
[8] D. Cantone, S. Faro, E. Giaquinta, A compact representation of nondeterministic (suffix) automata for the bit-parallel approach, in: Combinatorial Pattern Matching, 2010, pp. 288–298.
[9] D. Cantone, S. Faro, E. Giaquinta, A compact representation of nondeterministic (suffix) automata for the bit-parallel approach, Inf. Comput. 213 (2012) 3–12.
[10] C. Charras, T. Lecroq, Handbook of Exact String Matching Algorithms, King's College, 2004.
[11] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (4) (1994) 247–267.
[12] B. Durian, J. Holub, H. Peltola, J. Tarhio, Tuning BNDM with q-grams, in: Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX, 2009, pp. 29–37.
[13] B. Durian, H. Peltola, L. Salmela, J. Tarhio, Bit-parallel search algorithms for long patterns, in: Paola Festa (Ed.), Proceedings of the 9th International Symposium on Experimental Algorithms, SEA, Ischia Island, Naples, Italy, in: Lecture Notes in Computer Science, vol. 6049, Springer-Verlag, Berlin, 2010, pp. 129–140.
[14] S. Faro, M.O. Külekci, Fast multiple string matching using streaming SIMD extensions technology, in: Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, Nivio Ziviani (Eds.), Proceedings of the 19th International Symposium on String Processing and Information Retrieval, SPIRE, Colombia, in: Lecture Notes in Computer Science, vol. 7608, Springer-Verlag, Berlin, 2012, pp. 217–228.
[15] S. Faro, M.O. Külekci, Fast packed string matching for short patterns, in: Peter Sanders, Norbert Zeh (Eds.), Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX, SIAM, New Orleans, LA, USA, 2013, pp. 113–121.
[16] S. Faro, T. Lecroq, Efficient variants of the backward-oracle-matching algorithm, in: Jan Holub, Jan Žďárek (Eds.), Proceedings of the Prague Stringology Conference 2008, Czech Technical University in Prague, Czech Republic, 2008, pp. 146–160.
[17] S. Faro, T. Lecroq, An efficient matching algorithm for encoded DNA sequences and binary strings, in: Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 5577, 2009, pp. 106–115.
[18] S. Faro, T. Lecroq, Efficient variants of the backward-oracle-matching algorithm, Int. J. Found. Comput. Sci. 20 (6) (2009) 967–984.
[19] S. Faro, T. Lecroq, The exact string matching problem: a comprehensive experimental evaluation, preprint, arXiv:1012.2547, 2010.
[20] S. Faro, T. Lecroq, Smart: a string matching algorithm research tool, University of Catania and University of Rouen, http://www.dmi.unict.it/~faro/smart/, 2011.
[21] S. Faro, T. Lecroq, A multiple sliding windows approach to speed up string matching algorithms, in: R. Klasing (Ed.), 11th International Symposium on Experimental Algorithms, SEA 2012, in: Lecture Notes in Computer Science, vol. 7276, Springer-Verlag, Bordeaux, France, 2012, pp. 172–183.
[22] S. Faro, T. Lecroq, The exact online string matching problem: a review of the most recent results, ACM Comput. Surv. 45 (2) (2013) 1–42.
[23] K. Fredriksson, Faster string matching with super-alphabets, in: String Processing and Information Retrieval, Springer, 2002, pp. 207–214.
[24] K. Fredriksson, S. Grabowski, Practical and optimal string matching, in: M.P. Consens, G. Navarro (Eds.), Proceedings of the International Symposium on String Processing and Information Retrieval, SPIRE, in: Lecture Notes in Computer Science, vol. 3772, Springer-Verlag, Berlin, 2005, pp. 376–387.
[25] R.M. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Dev. 31 (2) (1987) 249–260.
[26] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323.
[27] M.O. Külekci, Filter based fast matching of long patterns by using SIMD instructions, in: Proceedings of the Prague Stringology Conference, 2009, pp. 118–128.
[28] M.O. Külekci, Blim: a new bit-parallel pattern matching algorithm overcoming computer word size limitation, Math. Comput. Sci. 3 (4) (2010) 407–420.
[29] Intel (R) 64 and IA-32 Architectures Optimization Reference Manual, Intel Corporation, 2011.
[30] T. Lecroq, Fast exact string matching algorithms, Inf. Process. Lett. 102 (6) (2007) 229–235.
[31] G. Navarro, M. Raffinot, A bit-parallel approach to suffix automata: fast extended string matching, in: Combinatorial Pattern Matching, Springer, 1998, pp. 14–33.
[32] H. Peltola, J. Tarhio, Variations of forward-SBNDM, in: Jan Holub, Jan Žďárek (Eds.), Proceedings of the Prague Stringology Conference 2011, Czech Technical University in Prague, Czech Republic, 2011, pp. 3–14.
[33] J. Rautio, J. Tanninen, J. Tarhio, String matching with stopper encoding and code splitting, in: Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching, CPM '02, Springer-Verlag, London, UK, 2002, pp. 42–52.
[34] R. Thathoo, A. Virmani, S. Sai Lakshmi, N. Balakrishnan, K. Sekar, TVSBS: a fast exact pattern matching algorithm for biological sequences, J. Indian Acad. Sci., Current Sci. 91 (1) (2006) 47–53.
[35] A.C. Yao, The complexity of pattern matching for a random string, SIAM J. Comput. 8 (3) (1979) 368–387.