

Fast Packed String Matching for Short Patterns

Simone Faro[†] and M. Oğuzhan Külekci[‡]

[†]Dipartimento di Matematica e Informatica, Università di Catania, Italy

[‡]TÜBİTAK National Research Institute of Electronics and Cryptology, Turkey
faro@dmi.unict.it, oguzhan.kulekci@tubitak.gov.tr

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 very fast string matching algorithms in the case of short patterns. From our experimental results it turns out that, despite their quadratic worst case time complexity, the new presented algorithms become the clear winners on the average for short patterns, when compared against the most 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 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 [20] (KMP). However in many practical cases it is possible to avoid reading all the characters of the text achieving sub-linear performances on average. The optimal average $\mathcal{O}(\frac{n \log_{\sigma} m}{m})$ time complexity [28] was reached for the first time by the Backward-DAWG-Matching algorithm [7] (BDM). However, most of the 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 [6].

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 the *bit-parallelism* technique or on 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 first algorithm based on it, named Shift-Or [1] (SO), simulates efficiently the nondeterministic version of the KMP automaton and runs in $\mathcal{O}(n\lceil\frac{m}{w}\rceil)$, which is still considered among the best practical algorithms in the case of very short patterns and small alphabets [16, 14]. Later a very fast BDM-like algorithm (BNDM), based on the bit-parallel simulation of the nondeterministic suffix automaton, was presented in [24]. Some variants of the BNDM algorithm [11, 13, 8, 25] are among the most practical efficient solutions in literature (see [16, 14]). 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 computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrades considerably as $\lceil\frac{m}{w}\rceil$ grows. Though there are a few techniques to maintain good performance in the case of long patterns [22, 9, 5], 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 $\alpha = \frac{w}{\log\sigma}$.

A first theoretical result in packed string matching was proposed by Fredriks-son [17]. 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 a $\mathcal{O}(n^\varepsilon m)$ -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 [4] 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. Recently 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 $\alpha \leq m \leq \frac{n}{\alpha}$. However, none of these results is of any practical interest.

The first algorithm that achieves good practical and theoretical results was very recently proposed by Ben-Kiki *et al.* [3]. The algorithm is based on two specialized packed string instructions, the `pcmpestrm` and the `pcmpestri` instructions [19], 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 current generation Intel Sandy

Bridge processors, `pcmpestrm` and `pcmpestri` have 2-cycle throughput and 7- and 8-cycle latency, respectively [19].

When the length of the searched pattern increases, another algorithm named Streaming SIMD Extensions Filter (SSEF), presented by Külekci in [21] (and extended to multiple pattern matching in [10]), 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 [16, 14]. Efficient solutions have been also designed for searching on packed DNA sequences [26, 12]. 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 throughput, when compared with those used in [3]. Specifically we present 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 the case of short patterns. The algorithm, named Exact Packed String Matching (EPSM), is based on three different search procedures used for, respectively, very short patterns ($0 < m < \frac{\alpha}{2}$), short patterns ($\frac{\alpha}{2} \leq m < \alpha$) and medium length patterns ($m \geq \alpha$). They use specialized packed string instructions with a low latency and throughput, if compared with those used in [3]. 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 $m \leq \frac{\alpha}{2}$, the first two search procedures achieve, respectively, a $\mathcal{O}(n + occ)$ 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. We then present a new algorithm for the packed string matching problem in Section 3 and report experimental results on short patterns in Section 4. Conclusions are given in Section 5.

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 $0 \leq i < m$, 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 $0 \leq i \leq j < m$. In some cases we will denote by p_i the $(i+1)$ -st character of p , so that $p_i = p[i]$ and $p = p_0p_1 \dots p_{m-1}$.

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 lose 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 $P = P_0P_1P_2\dots P_{k-1}$, where $P_i = p_{i\alpha}p_{i\alpha+1}p_{i\alpha+2}\dots p_{i\alpha+\alpha-1}$, for $0 \leq i < k$. 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 A New Packed String Matching Algorithm

In this section we present a new packed string matching algorithm, named Exact Packed String Matching algorithm (EPSM), which turns out to be efficient in the case of short patterns. EPSM is based on three different auxiliary algorithms, which we name EPSMa, EPSMb and EPSMc, respectively.

The first two auxiliary algorithms 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 third algorithm adopts a filtering based solution.

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 $m \leq \alpha/2$ the EPSMa and EPSMb algorithms reach, respectively, an $\mathcal{O}(m\alpha + \frac{mn}{\alpha} + occ)$ and $\mathcal{O}(\frac{n}{\alpha} + occ)$ time complexity. The first search procedure is designed to be extremely fast in the case of very short patterns, i.e. when $m \leq \frac{\alpha}{2}$, the second algorithm turns out to be a good choice when $\frac{\alpha}{2} \leq m < \alpha$, while the third algorithm turns out to be effective when $m \geq \alpha$. In practical cases we tuned the EPSM algorithm in order to run EPSMa when $0 < m < 4$, EPSMb when $4 \leq m < 16$, and to run EPSMc in all other cases.

In what follows, we first describe in Section 3.1 the computational model we use for the description of our solutions. Then we independently present the three auxiliary algorithms EPSMa, in Section 3.2, EPSMb, in Section 3.3, and EPSMc in Section 3.4.

3.1 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 our model of computation we suppose that w is the number of bits in a word and σ is the size of the alphabet. We indicate with the symbol $\alpha = \frac{w}{\log \sigma}$ the number of characters which fit in a single computer word.

In most practical applications we have $\sigma = 256$ (ASCII code). Moreover SSE specialized instructions allow to work on 128-bit registers, thus reading and processing blocks of sixteen 8-bit characters in a single time unit (thus $\alpha = 16$).

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.

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

Compares two w -bit words, handled as a block of α characters. In particular if $a = a_0a_1 \dots a_{\alpha-1}$ and $b = b_0b_1 \dots b_{\alpha-1}$ are the two w -bit integer parameters, **wscmp**(a, b) returns an α -bit value $r = r_0r_1 \dots r_{\alpha-1}$, where $r_i = 1$ if and only if $a_i = b_i$, and $r_i = 0$ otherwise. Below we give an example of the application of **wscmp**(a, b), assuming $w = 48$, $\gamma = 4$ and $\alpha = 12$.

	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

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)

```

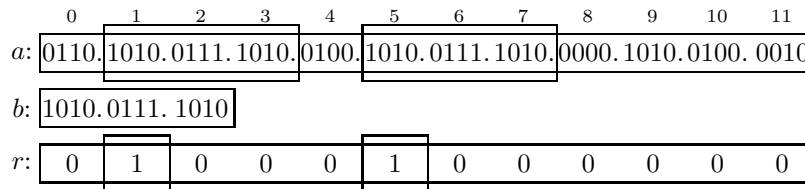
Specifically the `_mm_cmpeq_epi8` instruction compares two 128-bit words, handled as a block of sixteen 8-bit values, and returns a 128-bit value $h = h_0h_1 \dots h_{15}$, where $h_i = 1^8$ if and only if $a_i = b_i$, and $h_i = 0^8$ otherwise. It has a 0.5-cycle throughput and a 1-cycle latency

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.

wsmatch(a, b) (word-size pattern matching 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 $a = a_0a_1 \dots a_{\alpha-1}$, and $b = b_0b_1 \dots b_{k-1}$, then the `wsmatch(a, b)` instruction returns an α -bit integer value, $r = r_0r_1 \dots r_{\alpha-1}$, where $r_i = 1$ if and only if $a_{i+j} = b_j$ for $j = 0 \dots k-1$, 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 $\alpha - k$. Below we give 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 ← _mm_mpsadbw_epu8( $a, b$ )
ℓ ← _mm_cmpeq_epi8( $h, z$ )
r ← _mm_movemask_epi8(ℓ)

```

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 $r = r_0r_1 \dots r_7$, where r_i is computed as $r_i = \sum_{j=0}^4 |a_{i+j} - b_j|$ for $i = 0 \dots 7$. Thus we have that $r_i = 0^{16}$ if and only if $a_{i+j} = b_j$ for $j = 0 \dots 4$, 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.

wsblend(a, b) (word-size blend instruction)

blends two w -bit parameters, handled as two blocks of α characters. Specifically if $a = a_0a_1 \dots a_{\alpha-1}$ and $b = b_0b_1 \dots b_{\alpha-1}$, the instruction returns a w -bit integer $r = r_0r_1 \dots r_{\alpha-1}$, 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. $r = a_{\frac{\alpha}{2}}a_{\frac{\alpha}{2}+1} \dots a_{\alpha-1}b_0b_1 \dots b_{\frac{\alpha}{2}-1}$. Below we give an example of the application of `wsmatch(a, b)`, assuming $w = 48$, $\gamma = 4$ and $\alpha = 12$.

	0	1	2	3	4	5	6	7	8	9	10	11																				
a:	0	1	1	0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	1	1	1	0	1	0	0	0	1	0			
b:	0	1	0	0	0	0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	0	0	1	1	1	0	0	0	1	0	1	0
r:	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0	0	1	0

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

```

h ← _mm_blend_epi16(a, b, c)
r ← _mm_shuffle_epi32(h, _MM_SHUFFLE(1, 0, 3, 2))

```

Such instruction blends two 128-bit integers, $a = a_0a_1 \dots a_7$ and $b = b_0b_1 \dots b_7$, handled as packed 16-bit integers, according to a third parameter c . In particular it returns a 128-bit integer $r = r_0r_1 \dots r_7$ where $r_i = a_i$ if $c_i = 0$, and $r_i = b_i$ otherwise. If we set $c = 0^{64}1^{64}$ we get $r = a_0a_1a_2a_3b_4b_5b_6b_7$. 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, $a = a_0a_1a_2a_3$, handled as four 32-bit values, according to the order of the `_MM_SHUFFLE` macro. In this case we get $r = a_2a_3a_0a_1$. The `_mm_shuffle_epi32` instruction has 1-cycle throughput and a 1-cycle latency.

wsrc(*a*) (word-size cyclic redundancy check 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 `wsrc(a)` instruction can be emulated in constant time by using the `_mm_crc32_u64(a)` SIMD specialized instructions, which 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.

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 $r = r_0r_1r_2 \dots r_{\alpha-1}$, we have $\{r\} = \{i \mid 0 \leq i < \alpha \text{ and } r_i = 1\}$. 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 instructions `_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 a $\mathcal{O}(\alpha 2^\alpha)$ -time preprocessing phase in order to address the 2^α possible registers.

We are now ready to describe the three auxiliary algorithms used in the EPSM algorithm. The pseudocode of the three algorithms is shown in Fig. 1.

3.2 EPSMa: Searching for Very Short Patterns

The EPSMa 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 performances degrades as the length of the patterns increase. In practical cases the EPSM algorithm uses this procedure when $0 < m < 4$. The pseudocode of the algorithm is shown in Fig 1.

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 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 $T = T_0 T_1 \dots T_N$ 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 $s_j = b_0 b_1 \dots b_{\alpha-1}$ 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 $r = r_0 r_1 \dots r_{\alpha-1}$ be the α -bit register defined as $r = s_0 \& (s_1 \ll 1) \& (s_2 \ll 2) \& \dots \& (s_{m'-1} \ll (m' - 1))$.

It is easy to prove that $p[0 \dots m' - 1]$ 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 $0 \leq r < 2^\alpha$, 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 EPSMa algorithm is $\mathcal{O}(nm)$, because in the worst case a naive check is required for each position of the text. However, when $m \leq \frac{\alpha}{2}$ the EPSMa algorithm achieves a $\mathcal{O}(n + occ)$ time complexity, where occ is the number of occurrences of p in t .

3.3 EPSMb: Searching for Short Patterns

The EPSMb 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 practical cases the EPSM algorithm uses this procedure when $m \geq 4$. The pseudocode of the EPSMb algorithm is shown in Fig. 1 (in the middle). Notice that no preprocessing phase is needed by the algorithm.

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 $T = T_0T_1 \dots T_N$ 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 $r = r_0r_1 \dots r_{\alpha-1}$ be the α -bit register returned by the instruction `wsmatch`(T_i, p'), for $0 \leq j < m'$. We have that $r_j = 1$ if and only if an occurrence of p' begins at positions j of the block T_i , for $0 \leq j < \alpha/2$. 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 $0 \leq j < \alpha/2$. 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 require $\mathcal{O}(1)$ additional space. When $m \leq \alpha/2$ the algorithm reaches the optimal $\mathcal{O}(n/\alpha + occ)$ worst case time complexity.

3.4 EPSMc: Searching for Medium Length Patterns

The EPSMc algorithm is designed to be faster for medium length patterns. It is based on a simple filtering method and uses a hash function for computing

```

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

```

Fig. 1. The EPSMa (on the top), the EPSMb (in the middle) and the EPSMc (on the bottom) auxiliary algorithms.

fingerprint values on blocks of α characters. The fingerprint values are computed by using a hash function $h : \Sigma^\alpha \rightarrow \{0, 1, \dots, 2^k - 1\}$, for a constant parameter k (in practice we chose $k = 11$).

The function h 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 $x \in \Sigma^\alpha$.

The pseudocode of the EPSMc algorithm is shown in Fig. 1.

During the preprocessing phase (lines 1-6) a fingerprint values 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 $0 \leq v < 2^k$.

The preprocessing phase of the EPSM algorithm takes $\mathcal{O}(m + 2^k)$ -time.

Let $N = \frac{n}{\alpha} - 1$ and let $T = T_0T_1 \dots T_N$ be the string t represented in chunks of characters. During the searching phase (lines 7-13) the EPSM algorithm inspects the blocks of the text in steps of $(\lfloor m/\alpha \rfloor - 1) \cdot \alpha$ positions. For each inspected block T_i the fingerprint value $h(T_i)$ is computed and all positions in the set $\{i\alpha - j \mid j \in F[h(T_i)]\}$ are naively checked. The EPSM algorithm has a $\mathcal{O}(nm)$ worst case time complexity but turns out to be very effective in practical cases.

4 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 [16, 14]. In particular we compared the following algorithms:

- the Hash algorithm using groups of q characters [23] (HASH q);
- the Extended Backward Oracle Matching algorithm [11, 13] (EBOM);
- the TVSBS algorithm [27] (TVSBS);
- the Shift-Or algorithm [1] (SO)
- the Shift-Or algorithm with q -grams [8] (UFNDM q);
- the Fast-Average-Optimal-Shift-Or algorithm [18] (FAOSO q);
- the Backward DAWG Matching algorithm using q -grams [8] (BNDM q);
- the Simplified BNDM q algorithm [8] (SBNDM q);
- the Forward BNDM q algorithm [11, 13, 25] (FBNDM q);
- the Crochemore-Perrin algorithm using SSE instructions [3] (SSECP);
- the EPSM algorithm presented in this paper.

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 [15] for exact string matching. The experiments were executed locally on a machine running Ubuntu 11.10 (oneiric) with Intel i7-2600 processor with 16GB 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 4MB. The sequences are provided by the Smart research tool. For each input file, we have searched sets of 1000

m	2	4	6	8	12	16	20	24	28	32
HASH q	-	14.7 ⁽³⁾	11.5 ⁽³⁾	10.7 ⁽³⁾	8.78 ⁽³⁾	7.45 ⁽³⁾	6.70 ⁽³⁾	6.15 ⁽⁵⁾	5.75 ⁽⁵⁾	5.47 ⁽⁵⁾
EBOM	11.73	10.60	10.61	10.57	10.83	9.79	8.94	8.33	7.88	7.50
TVSBS	16.17	13.78	12.60	11.93	11.29	10.90	10.74	10.61	10.56	10.45
SO	10.76	10.99	10.62	10.93	10.86	10.67	10.89	10.83	10.77	10.73
FAOSO q	-	13.0 ⁽²⁾	10.7 ⁽²⁾	8.69 ⁽²⁾	7.83 ⁽²⁾	6.56 ⁽⁴⁾	5.90 ⁽⁴⁾	5.76 ⁽⁴⁾	5.66 ⁽⁴⁾	5.58 ⁽⁴⁾
UFNDM q	12.0 ⁽²⁾	9.53 ⁽⁴⁾	7.84 ⁽⁴⁾	6.94 ⁽⁴⁾	5.97 ⁽⁶⁾	5.39 ⁽⁶⁾	5.03 ⁽⁶⁾	4.81 ⁽⁶⁾	4.61 ⁽⁶⁾	4.61 ⁽⁶⁾
BNDM q	12.8 ⁽²⁾	11.3 ⁽²⁾	9.23 ⁽⁴⁾	7.24 ⁽⁴⁾	<u>5.90</u> ⁽⁴⁾	5.36 ⁽⁴⁾	5.09 ⁽⁴⁾	4.78 ⁽⁶⁾	4.61 ⁽⁶⁾	4.46 ⁽⁶⁾
SBNDM q	12.7 ⁽¹⁾	11.2 ⁽²⁾	9.62 ⁽⁴⁾	7.55 ⁽⁴⁾	6.12 ⁽⁴⁾	5.54 ⁽⁴⁾	5.15 ⁽⁶⁾	4.83 ⁽⁶⁾	4.62 ⁽⁶⁾	4.50 ⁽⁶⁾
FBNDM q	16.8 ⁽¹⁾	10.9 ⁽⁴⁾	8.86 ⁽⁴⁾	7.24 ⁽⁴⁾	6.03 ⁽⁴⁾	5.48 ⁽⁴⁾	5.17 ⁽⁶⁾	4.87 ⁽⁶⁾	4.66 ⁽⁶⁾	4.57 ⁽⁶⁾
SSECP	5.31	5.59	<u>5.98</u>	6.50	9.32	9.03	8.73	8.53	8.45	8.37
EPSM	<u>4.45</u>	<u>4.86</u>	6.18	<u>6.12</u>	6.16	<u>4.69</u>	<u>4.77</u>	<u>4.31</u>	<u>4.38</u>	<u>4.22</u>

Table 1. Experimental results for searching 1000 patterns on a genome sequence.

m	2	4	6	8	12	16	20	24	28	32
HASH q	-	14.1 ⁽³⁾	11.3 ⁽³⁾	11.2 ⁽³⁾	8.28 ⁽³⁾	6.98 ⁽³⁾	6.29 ⁽³⁾	5.81 ⁽³⁾	5.51 ⁽³⁾	5.27 ⁽³⁾
EBOM	10.00	6.25	5.50	5.14	4.84	4.69	4.60	4.58	4.53	4.51
TVSBS	11.71	10.52	10.45	9.20	7.68	6.83	6.29	5.93	5.66	5.30
SO	10.68	10.68	10.67	10.62	10.67	10.76	10.70	10.51	10.69	10.21
FAOSO q	-	8.54 ⁽²⁾	7.82 ⁽²⁾	6.42 ⁽⁴⁾	5.70 ⁽⁴⁾	5.67 ⁽⁴⁾	5.15 ⁽⁶⁾	5.12 ⁽⁶⁾	5.10 ⁽⁶⁾	5.09 ⁽⁶⁾
UFNDM q	11.0 ⁽²⁾	7.69 ⁽²⁾	6.44 ⁽²⁾	5.80 ⁽²⁾	5.18 ⁽²⁾	4.85 ⁽²⁾	4.62 ⁽⁴⁾	4.46 ⁽⁴⁾	4.33 ⁽⁴⁾	4.22 ⁽⁴⁾
BNDM q	10.6 ⁽²⁾	7.16 ⁽²⁾	5.95 ⁽²⁾	5.42 ⁽²⁾	4.93 ⁽²⁾	4.68 ⁽²⁾	<u>4.45</u> ⁽⁴⁾	4.30 ⁽⁴⁾	<u>4.18</u> ⁽⁴⁾	<u>4.12</u> ⁽⁴⁾
SBNDM q	10.4 ⁽²⁾	7.01 ⁽²⁾	5.86 ⁽²⁾	5.37 ⁽²⁾	4.89 ⁽²⁾	<u>4.65</u> ⁽²⁾	4.48 ⁽⁴⁾	4.33 ⁽⁴⁾	4.21 ⁽⁴⁾	4.12 ⁽⁴⁾
FBNDM q	10.5 ⁽¹⁾	8.64 ⁽¹⁾	6.85 ⁽²⁾	6.37 ⁽⁴⁾	5.21 ⁽⁴⁾	4.76 ⁽⁴⁾	4.50 ⁽⁴⁾	4.34 ⁽⁴⁾	4.23 ⁽⁴⁾	4.19 ⁽⁴⁾
SSECP	5.31	5.58	5.96	6.49	6.68	6.45	6.33	6.24	6.19	6.15
EPSM	<u>4.47</u>	<u>4.83</u>	<u>4.65</u>	<u>4.65</u>	<u>4.64</u>	<u>4.65</u>	4.73	<u>4.28</u>	4.31	4.18

Table 2. Experimental results for searching 1000 patterns on a protein sequence.

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 genome 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, while the second best results have been boldfaced.

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 BNDM q 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.

m	2	4	6	8	12	16	20	24	28	32
HASH q	-	14.2 ⁽³⁾	11.2 ⁽³⁾	11.1 ⁽³⁾	8.29 ⁽³⁾	6.99 ⁽³⁾	6.26 ⁽³⁾	5.83 ⁽³⁾	5.50 ⁽³⁾	5.27 ⁽³⁾
EBOM	10.26	7.24	6.51	6.14	5.82	5.67	5.55	5.53	5.42	5.37
TVSBS	12.02	10.74	10.25	9.58	8.14	7.24	6.67	6.34	6.01	5.76
SO	10.87	10.80	10.63	10.72	10.72	10.79	10.59	10.71	10.66	10.72
FAOSO q	-	9.22 ⁽²⁾	8.01 ⁽²⁾	6.89 ⁽⁴⁾	5.77 ⁽⁴⁾	5.66 ⁽⁴⁾	5.20 ⁽⁶⁾	5.10 ⁽⁶⁾	5.11 ⁽⁶⁾	5.10 ⁽⁶⁾
UFNDM q	10.6 ⁽²⁾	8.33 ⁽²⁾	7.20 ⁽²⁾	6.35 ⁽⁴⁾	5.48 ⁽⁴⁾	5.02 ⁽⁴⁾	4.77 ⁽⁴⁾	4.62 ⁽⁴⁾	4.47 ⁽⁴⁾	4.39 ⁽⁴⁾
BNDM q	10.6 ⁽²⁾	8.19 ⁽²⁾	7.09 ⁽²⁾	6.49 ⁽²⁾	5.46 ⁽⁴⁾	4.96 ⁽⁴⁾	4.69 ⁽⁴⁾	4.55 ⁽⁴⁾	4.41 ⁽⁴⁾	4.33 ⁽⁴⁾
SBNDM q	10.8 ⁽²⁾	8.02 ⁽²⁾	6.99 ⁽²⁾	6.44 ⁽²⁾	5.60 ⁽⁴⁾	5.07 ⁽⁴⁾	4.78 ⁽⁴⁾	4.64 ⁽⁴⁾	4.48 ⁽⁴⁾	4.39 ⁽⁴⁾
FBNDM q	10.5 ⁽¹⁾	9.13 ⁽¹⁾	8.07 ⁽⁴⁾	6.65 ⁽⁴⁾	5.51 ⁽⁴⁾	5.02 ⁽⁴⁾	4.74 ⁽⁴⁾	4.61 ⁽⁴⁾	4.45 ⁽⁴⁾	4.43 ⁽⁴⁾
SSECP	5.33	5.60	5.98	6.51	7.37	7.03	6.83	6.69	6.62	6.61
EPSM	4.48	4.85	5.15	5.13	5.04	4.68	4.75	4.31	4.35	4.21

Table 3. Experimental results for searching 1000 patterns on a natural language text.

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, underling a linear trend on average. However, the performances of other algorithms based on shift heuristics, slightly increases. 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 to its optimal worst case time complexity, the SSECP algorithm shows an increasing trend on average, while the EPSM algorithm shows a linear behavior.

5 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 and G.H. Gonnet. A new approach to text searching. *Comm. of the ACM*, 35(10):74–82, 1992.
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)*, pages 90–102, 2010.
3. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann. Optimal packed string matching. In *IARCS Annual Conf. on Found. of Software Technology and Theoretical Computer Science*, pages 423–432, 2011.
4. P. Bille. Fast searching in packed strings. *Journal of Discrete Algorithms*, 9(1):49–56, 2011.
5. D. Cantone, S. Faro, and E. Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In *Combinatorial Pattern Matching*, pages 288–298, 2010.
6. C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King’s College, 2004.
7. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994.
8. B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning bndm with q-grams. *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 29–37, 2009.
9. B. Durian, H. Peltola, L. Salmela, and J. Tarhio. Bit-parallel search algorithms for long patterns. In *Intern. Symp. on Experimental Algorithms*, pp. 129–140, 2010.
10. S. Faro and M. O. Külekci. Fast multiple string matching using streaming simd extensions technology. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval*, volume 7608 of *Lecture Notes in Computer Science*, pages 217–228. Springer Berlin, 2012.
11. S. Faro and T. Lecroq. Efficient variants of the backward-oracle-matching algorithm. In *Proc. of the Prague Stringology Conference 2008*, pages 146–160, Czech Technical University in Prague, Czech Republic, 2008.
12. S. Faro and T. Lecroq. An efficient matching algorithm for encoded dna sequences and binary strings. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching, CPM ’09*, pages 106–115, Berlin, Heidelberg, 2009. Springer-Verlag.
13. S. Faro and T. Lecroq. Efficient variants of the backward-oracle-matching algorithm. *Int. J. Found. Comput. Sci.*, 20(6):967–984, 2009.
14. S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *Arxiv preprint arXiv:1012.2547*, 2010.
15. S. Faro and T. Lecroq. Smart: a string matching algorithm research tool. University of Catania and University of Rouen, <http://www.dmi.unict.it/faro/smart/>, 2011.
16. S. Faro and T. Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 45(2):to appear, 2013.
17. K. Fredriksson. Faster string matching with super-alphabets. In *String Processing and Information Retrieval*, pages 207–214. Springer, 2002.
18. K. Fredriksson and S. Grabowski. Practical and optimal string matching. In M. P. Consens and G. Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 376–387. Springer-Verlag, Berlin, 2005.

19. Intel. Intel (R) 64 and IA-32 Architectures Optimization Reference Manual. *Intel Corporation*, 2011
20. D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6:323, 1977.
21. M.O. Külekci. Filter based fast matching of long patterns by using SIMD instructions. In *Proc. of the Prague Stringology Conference*, pages 118–128, 2009.
22. M.O. Külekci. Blim: A new bit-parallel pattern matching algorithm overcoming computer word size limitation. *Mathem. in Computer Science*, 3(4):407–420, 2010.
23. T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.
24. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching*, pages 14–33, 1998.
25. H. Peltola and J. Tarhio. Variations of forward-SBNDM. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 3–14, Czech Technical University in Prague, Czech Republic, 2011.
26. J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, CPM '02, pages 42–52, London, UK, UK, 2002. Springer-Verlag.
27. R. Thathoo, A. Virmani, S. Sai Lakshmi, N. Balakrishnan, and K. Sekar. TVSBS: A fast exact pattern matching algorithm for biological sequences. *J. Indian Acad. Sci., Current Sci.*, 91(1):47–53, 2006.
28. A.C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.