

An efficient matching algorithm for encoded DNA sequences and binary strings

Simone Faro¹ and Thierry Lecroq²

¹ Dipartimento di Matematica e Informatica, Università di Catania, Italy

² University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Abstract. We present a new efficient algorithm for exact matching in encoded DNA sequences and on binary strings. Our algorithm combines a multi-pattern version of the BNDM algorithm and a simplified version of the COMMENTZ-WALTER algorithm. We performed also experimental comparisons with the most efficient algorithms presented in the literature. Experimental results show that the newly presented algorithm outperforms existing solutions in most cases.

Keywords: string matching, binary strings, DNA sequences, experimental algorithms, compressed text processing.

1 Introduction

In this article we consider the problem of searching for all exact occurrences of a pattern p , of length m , in a text t , of length n , where p and t are both bitstreams. In particular we consider the cases where each character of p and t consists in a single bit (binary sequences) or a couple of bits (encoded DNA sequences).

Matching binary data is an interesting problem in computer science, since binary data are omnipresent in telecom and computer network applications. Many formats for data exchange between nodes in distributed computer systems as well as most network protocols use binary representations. Binary images often arise in digital image processing as masks or as the results of certain operations such as segmentation, thresholding and dithering. Moreover some input/output devices, such as laser printers and fax machines, can only handle binary images.

Also DNA sequences can be handled as bitstreams. Since a DNA sequence is constructed with four bases (A, C, T, and G), an efficient fixed-length encoding method [7] can be used, where only two bits for each character are needed.

In molecular biology, DNA sequences are the fundamental information for each species and a comparison between DNA sequences is an interesting and basic problem. There are various kinds of comparison tools which provide approximate matching. However most of them are based on exact matching in order to speed up the process. Moreover because the total number of sequences is rapidly increasing, efficient methods are needed, not only for fast matching but also for efficient sequences storage. Thus the need for fast matching algorithms on encoded sequences.

The first non trivial algorithm for searching on bitstreams was presented in [8] by Klein and Ben-Nissan. They proposed an efficient variant of the BOYER-MOORE [2] algorithm for the binary case without referring to bits. The algorithm is projected to process only entire blocks such as bytes or words and achieves a significantly reduction in the number of text character inspections.

Recently in [5] two efficient algorithms have been presented for the problem adapted to completely avoid any reference to bits allowing to process pattern and text byte by byte. The first solution, called BINARY-HASH-MATCHING algorithm, is an adaptation of the q -HASH family [10] for exact pattern matching to the case of binary strings. The second solution, called BINARY-SKIP-SEARCH algorithm, extends the SKIP-SEARCH algorithm [3] for the single exact pattern matching problem. Experimental results conducted in [5] on various conditions showed that the proposed algorithms perform better than existing solutions and even than the most effective algorithms for standard pattern matching.

All previous solutions have been proposed for searching on binary data but can be easily adapted to the case of encoded DNA sequences with minor changes.

The FED algorithm [7] (Fast matching with Encoded DNA sequences) is a string matching algorithm specifically tuned for matching DNA sequences compressed using a fixed-length encoding. Specifically, the FED algorithm combines a multi-pattern version of the QUICK-SEARCH algorithm [12] and a simplified version of the COMMENTZ-WALTER algorithm [4]. However, its strategy is general enough to be adapted also for matching binary sequences.

In this article we present a new efficient algorithm for matching on binary strings and encoded DNA sequences which, despite its $\mathcal{O}(nm)$ worst case time complexity, obtains very good results in practical cases.

The rest of the paper is organized as follows. In Section 2, we describe in details the new solution tuned for binary data. Next, in Section 3, we explain how to handle also encoded DNA sequences. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 4. Finally, we draw our conclusions in Section 5.

2 A new efficient algorithm

Before entering into details, we need a bit of notations and terminology. A string p of length $m \geq 0$ is represented as a finite array $p[0..m-1]$ of characters from a finite alphabet Σ . In particular, for $m = 0$ we obtain the empty string, also denoted by ε . By $p[i]$ we denote the $(i+1)$ -th character of p , for $0 \leq i < m$. Likewise, by $p[i..j]$ we denote the substring of p contained between the $(i+1)$ -th and the $(j+1)$ -th characters of p , for $0 \leq i \leq j < m$. A substring of p of length ℓ is called a ℓ -substring of p .

A string p over the binary alphabet $\Sigma = \{0, 1\}$ is said to be a *binary string* and is represented as a binary vector $p[0..m-1]$, whose elements are bits. Binary vectors are usually structured in blocks of k bits, typically bytes ($k = 8$), halfwords ($k = 16$) or words ($k = 32$), which can be processed at the cost of a single operation. If p is a binary string of length m we use the symbol $P[i]$ to indicate the $(i+1)$ -th block of p and use $p[i]$ to indicate the $(i+1)$ -th bit of

p . If B is a block of k bits we indicate with symbol B_j the j -th bit of B , with $0 \leq j < k$. Thus, for $i = 0, \dots, m-1$ we have $p[i] = P[\lfloor i/k \rfloor]_{i \bmod k}$.

In this section we present a new efficient algorithm, called BFL algorithm (BINARY-FARO-LECROQ), for searching on binary strings and encoded DNA sequences. The proposed algorithm exploits the block structure of text and pattern to speed up the searching phase avoiding to work with bitwise operations. The core of the algorithm is based on a multiple-pattern version of the BNDM algorithm [11] (Backward Nondeterministic Dawg Match) for the single exact pattern matching problem, which makes use of bit-parallelism [1]. Moreover the algorithm implements also an efficient shift strategy based on a simplified version of the COMMENTZ-WALTER algorithm [4].

During the preprocessing phase the algorithm constructs a set of tables that can be accessed later, during the searching phase, in order to speed up the overall performances. The procedures used in the preprocessing phase are presented in Figure 2. Specifically the algorithm computes: **(1)** a table of several copies of p , in order to process text and pattern block by block (as in [8]); **(2)** a bit mask-vector used to implement a multi-pattern version of the BNDM algorithm; **(3)** an index-list table, λ , in order to identify candidate alignments during the searching phase; **(4)** a shift table ls , based on the bad-character heuristic, with the aim of increasing the shift advancements.

In what follows we suppose that the block size k is fixed, so that all references to both text and pattern will only be to entire blocks of k bits. We refer to a k -bit block as a *byte*, though larger values than $k = 8$ could be supported as well. Moreover we suppose that $T[i]$ and $P[i]$ denote, respectively, the $(i+1)$ -th byte of the text and of the pattern, starting for $i = 0$ with both text and pattern aligned at the leftmost bit of the first byte. Since the lengths in bits of both text and pattern are not necessarily multiples of k , the last byte may be only partially defined. In particular if the pattern has length m then its last byte is that of position $\lceil m/k \rceil$ and only the leftmost $(m \bmod k)$ bits of the last byte are defined. We suppose that the undefined bits of the last byte are set to 0.

Finally we notice that the following description of the algorithm is tuned for processing binary data where each character consists in a single bit. In the next section we explain how to handle also encoded DNA sequences.

Preprocessing of the pattern

In the preprocessing phase we define several copies of the pattern, identified by a set of indexes \mathbf{P} , and memorized in the form of a matrix of bytes, $Patt$, of size $k \times (\lceil m/k \rceil + 1)$. Each index $i \in \mathbf{P}$ refers to a row of the matrix $Patt$ containing a copy of the pattern shifted by i position to the right. In each pattern $Patt[i]$, for $i \in \mathbf{P}$, the i leftmost bits of the first byte remain undefined and are set to 0. Similarly the rightmost $((k - ((m+i) \bmod k)) \bmod k)$ bits of the last byte are set to 0. Formally the j -th bit of the byte $Patt[i, h]$ is defined by

$$Patt[i, h]_j = \begin{cases} p[kh - i + j] & \text{if } 0 \leq kh - i + j < m \\ 0 & \text{otherwise} \end{cases} .$$

<i>Patt</i>	0	1	2	3	m_i	s_i	m'_i	<i>F1</i>	<i>F2</i>
<i>i</i> 0	<u>11001011</u>	<u>00101100</u>	<u>10110000</u>		2	0	2	<u>11111111</u>	<u>11111000</u>
1	01100101	<u>10010110</u>	01011000		2	1	1	<u>01111111</u>	<u>11111100</u>
2	<u>00110010</u>	<u>11001011</u>	<u>00101100</u>		2	1	1	<u>00111111</u>	<u>11111110</u>
3	00011001	<u>01100101</u>	<u>10010110</u>		2	1	2	<u>00011111</u>	<u>11111111</u>
4	00001100	<u>10110010</u>	<u>11001011</u>	00000000	3	1	2	<u>00001111</u>	<u>10000000</u>
5	00000110	<u>01011001</u>	<u>01100101</u>	<u>10000000</u>	3	1	2	<u>00000111</u>	<u>11000000</u>
6	00000011	<u>00101100</u>	<u>10110010</u>	<u>11000000</u>	3	1	2	<u>00000011</u>	<u>11100000</u>
7	00000001	<u>10010110</u>	<u>01011001</u>	<u>01100000</u>	3	1	2	<u>00000001</u>	<u>11110000</u>

Fig. 1. Precomputed tables for a pattern $p = 110010110010110010110$ of length $m = 21$ and block size $k = 8$. Blocks containing a k -substring of p are presented with light gray background color.

for $0 \leq j < k$, $0 \leq h < \lceil (m+i)/k \rceil$ and $i \in \mathbf{P}$.

For binary strings we have $\mathbf{P} = \{i \mid 0 \leq i < k\}$. Observe that each k -substring of the pattern appears once in the table *Patt*. In particular, the k -substring starting at position j of p is memorized in $Patt[k - (j \bmod k), \lceil j/k \rceil]$.

We indicate with symbol m_i , for each $i \in \mathbf{P}$, the index of the last byte of the pattern $Patt[i]$, i.e., $m_i = \lceil (m+i)/k \rceil - 1$. Moreover we define the values s_i and m'_i which represent, respectively, the position of the first byte in $Patt[i]$ containing a k -substring of p and the number of bytes in $Patt[i]$ containing k -substrings of p . Observe that $s_i = 0$ if $i = 0$, while $s_i = 1$ when $i > 0$.

The BFL algorithm uses bytes in the matrix *Patt* to compare the pattern block by block against the text, for any possible alignment of the pattern. However when comparing the first or the last byte of P against its counterpart in the text, the bit positions not belonging to the pattern have to be neutralized. For this purpose we define two vectors, *F1* and *F2*, containing binary masks of length k . Formally, for each $i \in \mathbf{P}$,

$$F1[i]_j = \begin{cases} 1 & \text{if } i \leq j < k \\ 0 & \text{otherwise} \end{cases}, F2[i]_j = \begin{cases} 1 & \text{if } 0 \leq j \leq (m+i-1) \bmod k \\ 0 & \text{otherwise} \end{cases}.$$

Figure 1 shows the precomputed tables defined above for a pattern of length $m = 21$ and $k = 8$.

Definition of the NFA and construction of the bit mask vector

The BFL algorithm uses bit-parallelism to simulate the behavior of a nondeterministic finite state automaton constructed over the set of patterns identified by \mathbf{P} . However, in order to let the automaton fit in a single machine word of size ω , only the substrings $Patt[i][s_i \dots s_i + m' - 1]$ are handled by the automaton, for each $i \in \mathbf{P}$, where $m' = \min(\{m'_i \mid i \in \mathbf{P}\} \cup \{\omega\})$.

Specifically the NFA constructed by the BFL algorithm has $m' + 1$ different states, say $Q = \{0, 1, 2, 3, \dots, m'\}$, and m' different transitions. In particular each state q , with $0 < q \leq m'$, has a transition towards state $q - 1$ labeled with the class of characters $\{Patt[i][s_i + q] \mid i \in \mathbf{P}\}$. State m' is the initial state.

In order to simulate the NFA the algorithm initializes a bit mask $M[B]$ of dimension ω , for each block $B \in \{0 \dots 2^k - 1\}$. In particular the j -th bit of $M[B]$

<pre> PREPROCESS (<i>Patt</i>, <i>k</i>) 1. for <i>i</i> ← 0 to <i>k</i> - 1 do 2. if <i>i</i> = 0 then <i>s_i</i> ← 0 else <i>s_i</i> ← 1 3. <i>m_i</i> ← ⌈(<i>m</i> + <i>i</i>)/<i>k</i>⌉ - 1 4. for <i>h</i> ← 0 to <i>m_i</i> do 5. <i>Patt</i>[<i>i</i>, <i>h</i>] ← (<i>P</i>[<i>h</i>] ≫ <i>i</i>) 6. if <i>h</i> > 0 then 7. <i>Patt</i>[<i>i</i>, <i>h</i>] ← <i>Patt</i>[<i>i</i>, <i>h</i>] (<i>P</i>[<i>h</i> - 1] ≪ (k - <i>i</i>)) 8. <i>F1</i>[<i>i</i>] ← 1^{<i>k</i>} ≫ <i>i</i> 9. <i>F2</i>[<i>i</i>] ← 1^{<i>k</i>} ≪ k - ((<i>m</i> + <i>i</i>) mod <i>k</i>) 10. if <i>F2</i>[<i>i</i>] = 1^{<i>k</i>} then <i>m'_i</i> ← <i>m_i</i> - <i>s_i</i> + 1 11. else <i>m'_i</i> ← <i>m_i</i> - <i>s_i</i> 12. return (<i>Patt</i>, <i>Mask</i>) </pre>	<pre> COMPUTE-LONG-SHIFT(<i>Patt</i>, <i>k</i>) 1. for <i>B</i> ← 0 to 2^{<i>k</i>} - 1 do 2. <i>ls</i>[<i>B</i>] = 2^{<i>m'</i>} - 1 3. <i>i</i> ← 0 4. <i>h</i> ← 0 5. for <i>j</i> ← 0 to <i>m</i> - <i>k</i> do 6. <i>ls</i>[<i>Patt</i>[<i>i</i>, <i>h</i>]] ← <i>s_i</i> + 2^{<i>m'</i>} - <i>h</i> - 2 7. <i>i</i> ← <i>i</i> - 1 8. if <i>i</i> < 0 then 9. <i>i</i> ← 7 10. <i>h</i> ← <i>h</i> + 1 11. return <i>ls</i> </pre>
<pre> INITIALIZE-BIT-MASK(<i>Patt</i>, <i>k</i>) 1. for <i>B</i> ← 0 to 2^{<i>k</i>} - 1 do <i>M</i>[<i>B</i>] = 0 2. for <i>i</i> ← 0 to <i>k</i> do 3. <i>A</i> ← 10^{ω-1} 4. for <i>j</i> = 0 to <i>m'</i> - 1 do 5. <i>B</i> ← <i>Patt</i>[<i>i</i>, <i>s_i</i> + <i>m'</i> - <i>j</i> - 1] 6. <i>M</i>[<i>B</i>] ← <i>M</i>[<i>B</i>] <i>A</i> 7. <i>A</i> ← <i>A</i> ≫ 1 8. return <i>M</i> </pre>	<pre> COMPUTE-INDEX-LIST(<i>Patt</i>, <i>k</i>) 1. for <i>B</i> ← 0 to 2^{<i>k</i>} - 1 do 2. λ[<i>B</i>] = ∅ 3. <i>i</i> ← 0 4. <i>h</i> ← 0 5. for <i>i</i> ← 0 to <i>k</i> do 6. <i>B</i> ← <i>Patt</i>[<i>i</i>, <i>s_i</i> + <i>m'</i> - 1] 7. λ[<i>B</i>] ← λ[<i>B</i>] ∪ {<i>i</i>} 8. return λ </pre>

Fig. 2. Procedures in the preprocessing phase of the BFL algorithm for binary strings.

is set to 1 if the block B appears at position $s_i + m' - j - 1$ in, at least, one of the patterns $Patt[i]$, with $i \in \mathbf{P}$. Otherwise the j -th bit of $M[B]$ is set to 0.

For each block $B \in \{0 \dots 2^k - 1\}$, the definition of the bit mask $M[B]$ can be done in two steps. First we define, for each $i \in \mathbf{P}$, a bit mask $M_i[B]$ where, for $0 \leq j < \omega$,

$$M_i[B]_j = \begin{cases} 1 & \text{if } j < m' \text{ and } Patt[i, s_i + m' - j - 1] = B \\ 0 & \text{otherwise.} \end{cases}$$

Then the j -th bit of the mask $M[B]$, with $0 \leq j < \omega$, is defined as

$$M[B]_j = (M_0[B]_j \mid M_1[B]_j \mid \dots \mid M_{k-1}[B]_j).$$

Construction of the index list

The automaton defined above recognizes also words that are not substrings of the pattern. Formally the automaton recognizes any block sequence x , of length $\ell \leq m'$, of the form $x = x_0.x_1 \dots x_{\ell-1}$ where, for $0 \leq j < \ell$,

$$x_j \in \{Patt[i][s_i + m' - \ell + j] \mid i \in \mathbf{P}\}.$$

Despite this fact, the automaton can be used to search for a pattern in a text. In particular when a candidate substring is found, the algorithm could naively check for the occurrence of any pattern $Patt[i]$, with $i \in \mathbf{P}$.

However, in order to make a filter the algorithm maintains, for each block $B \in \{0 \dots 2^k - 1\}$, a linked list λ which is used to find candidate patterns. In particular, for each block $B \in \{0 \dots 2^k - 1\}$, the entry $\lambda[B]$ is a set of indexes, defined by $\lambda[B] = \{i \mid i \in \mathbf{P} \text{ and } P[i][s_i + m' - 1] = B\}$.

Thus, when a block sequence is recognized by the automaton, ending at block position j of the text, the algorithm naively checks for the occurrence of any pattern $Patt[i]$, with $i \in \lambda[T[j]]$.

In practical cases each set in the table can be implemented as a linked list.

Construction of the shift table

The BFL algorithm makes also use of a *long shift* rule which is a multi-pattern version of the original bad-character shift heuristic, improved with an efficient look-ahead. The shift rule defined here is used by the algorithm when no substring is recognized while scanning the window of the text from right to left. In such a case the current window of the text can be safely advanced by $m' - 1$ positions to the right.

Observe that, if j is the ending position of the current window of the text, the block at position $j + m' - 1$ is always involved in the next alignment. Thus we can use it to compute the next window alignment.

Specifically the algorithm computes a *long shift* table $ls() : \{0, \dots, 2^k - 1\} \rightarrow \{m' - 1, \dots, 2 \times m' - 1\}$, whose definition can be done in two steps.

First, for each pattern $Patt[i]$, with $i \in \mathbf{P}$, we compute a shift table $sh[i, B]$, where, for each $B \in \{0 \dots 2^k - 1\}$,

$$sh[i, B] = \min(\{m'\} \cup \{s_i + m' - h - 1 \mid Patt[i, h] = B \text{ and } s_i \leq h < s_i + m'\}).$$

Then, for each $B \in \{0 \dots 2^k - 1\}$, the long shift table ls is defined by

$$ls[B] = \min\{sh[i, B] \mid i \in \mathbf{P}\} + m' - 1.$$

Thus the next alignment to be processed in the text is that ending at position $j + ls[T[s + m' - 1]]$.

The searching phase

The searching phase of the BFL algorithm can be divided into two parts: a match phase and a shift phase. The pseudocode of the algorithm is shown in Figure 3.

The NFA is represented by a state vector D of size m' (the first state of the automaton is not represented). Like in the SBNDM algorithm [6], the BFL algorithm starts each iteration with a test of two consecutive text characters and implements a fast-loop to obtain better results on average (LINE 7). Such a fast loop makes use of the long shift table to compute the next window alignment.

In the match phase the same kind of right to left scan in a window of size m' , ending at position j in the text, is performed as in the BNDM algorithm (LINE 9). The state vector is updated in a similar fashion as in the SHIFT-AND algorithm [1]. If the state vector D is equal to 0 after $\ell + 1$ updates of D ,

```

BFL ( $P, m, T, n$ )
1.  $Patt \leftarrow \text{Preprocess}(P, m, k)$ 
2.  $M \leftarrow \text{Initialize-Bit-Mask}(Patt, k)$ 
3.  $\lambda \leftarrow \text{Compute-Index-List}(Patt, k)$ 
4.  $ls \leftarrow \text{Compute-Long-Shift}(Patt, k)$ 
5.  $j \leftarrow \lfloor n/k \rfloor$ 
6. while  $j < \lceil n/k \rceil$  do
7.   while  $((M[T[j]] \ll 1) \& M[T[j-1]]) = 0$  do  $j \leftarrow j + ls[T[j + m' - 1]]$ 
8.    $pos \leftarrow j$ 
9.    $D \leftarrow (M[T[j]] \ll 1) \& M[T[j-1]]$ 
10.  while  $D \leftarrow (D \ll 1) \& M[T[j-2]]$  do  $j \leftarrow j - 1$ 
11.   $j \leftarrow j + m'$ 
12.  if  $j = pos$  then
13.    for each  $i \in \lambda[T[j]]$  do
14.       $h \leftarrow 1, s \leftarrow j - m' - s_i + 1$ 
15.      while  $h < m_i$  and  $Patt[i, h] = T[s + h]$  do  $h \leftarrow h + 1$ 
16.      if  $h = m_i$  and
17.         $Patt[i, 0] = (T[s] \& F1[i])$  and
18.         $Patt[i, h] = (T[s + h] \& F2[i])$ 
19.      then Output( $s$ )

```

Fig. 3. The searching phase of the BFL algorithm.

then a word of length ℓ has been recognized by the automaton. If $\ell = m'$ a candidate alignment has been found (LINE 12) and the algorithm naively checks the occurrence of any pattern contained in the index list $\lambda[T[j]]$ (LINE 13).

In all cases, after the match phase, the index j is advanced of $m' - \ell + 1$ to the right and the algorithm restarts its computation with the shift phase.

If a candidate alignment is found for a text position j , for each index $i \in \lambda[T[j]]$, the algorithm uses the precomputed table $Patt[i]$ to check whether $s = j - m' - s_i + 1$ is a valid shift. Specifically the algorithm reports a match if the following three conditions hold (LINES 13-18):

1. $Patt[i, h] = T[s + h]$, for $h = s_i, \dots, s_i + m_i - 1$ and
2. $Patt[i, 0] = T[s] \& F1[i]$ and
3. $Patt[i, m_i] = T[s + m_i] \& F2[i]$.

After the matching phase the algorithm restarts with the shift phase.

Complexity issues

In this section we analyze the time complexity of the newly presented algorithm: **(1)** Procedure PREPROCESS requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space; **(2)** procedure INITIALIZE-BIT-MASK, used to initialize the vector of bit masks M , requires $\mathcal{O}(2^k + km)$ time and $\mathcal{O}(2^k)$ extra-space; **(3)** procedure COMPUTE-INDEX-LIST, used to construct the index list λ , requires $\mathcal{O}(2^k + k)$ time and $\mathcal{O}(2^k)$ extra-space; **(4)** procedure COMPUTE-LONG-SHIFT, used to construct the long shift table ls , requires $\mathcal{O}(2^k + m)$ time and $\mathcal{O}(2^k)$ extra-space; finally **(5)** the searching phase, shown in Figure 3, takes $\mathcal{O}(\lceil n/k \rceil \lceil m/k \rceil k) = \mathcal{O}(nm)$ time. Thus the BFL algorithm has a $\mathcal{O}(2^k + nm)$ overall time complexity and requires $\mathcal{O}(2^k)$ extra-space in the worst case.

3 Handling encoded DNA sequences

In a fix-length encoded DNA sequence each base is represented by a couple of bits. Specifically we define a map ζ which associates to any character in the set $\{\text{A, C, G, T}\}$ an element in the set $\{00, 01, 10, 11\}$. Thus a DNA sequence γ can be represented with a bitstream $t = \zeta(\gamma)$ of $(2 \times |\gamma|)$ bits.

Due to the structure of the encoded sequence t , any occurrence of a given encoded pattern p , starts at an even position of the text. This suggests that only even alignments of the pattern have to be processed.

The only change to be applied, when handling encoded DNA sequences, is in the preprocessing of the set of patterns. Specifically the set \mathbf{P} is defined by

$$\mathbf{P} = \{i \mid 0 \leq i < k \text{ and } (i \bmod 2) = 0\}$$

For instance, if each block consists of $k = 8$ bits, we have $\mathbf{P} = \{0, 2, 4, 6\}$.

4 Experimental Results

Here we present experimental data which allow to compare, in terms of running time, the following string matching algorithms on binary strings and encoded DNA sequences: the BINARY-BOYER-MOORE algorithm (BBM) [8] by Klein and Ben-Nissan, the BINARY-HASH-MATCHING algorithm (BHM) [5], the BINARY-SKIP-SEARCH algorithm (BSKS) [5], FED algorithm (FED) [7] and the new BFL (BFL) algorithm. All algorithms have been implemented in the C programming language and were used to search for the same binary strings in large fixed text buffers on a PC with Intel Core2 processor of 1.66GHz with 1GB memory. Moreover we use a word size $\omega = 32$ and a block size $k = 8$.

To simulate the different conditions which can arise when processing binary data the algorithms have been tested on two $\text{Rand}(1/0)_\gamma$ problems, with a different distribution of zeros and ones. For the case of compressed strings it is quite reasonable to assume a uniform distribution of characters. For compression scheme using Huffman coding, such randomness has been shown to hold in [9]. In contrast when processing binary images we expect a non-uniform distribution of characters. For instance in a fax-image usually more than 90% of the total number of bits is set to zero.

In particular each $\text{Rand}(1/0)_\gamma$ problem consists of searching a set of 1000 random patterns of a given length in a random binary text of 4×10^6 bits. The distribution of characters depends on the value of the parameter γ : bit 0 appears with a percentage equal to $\gamma\%$.

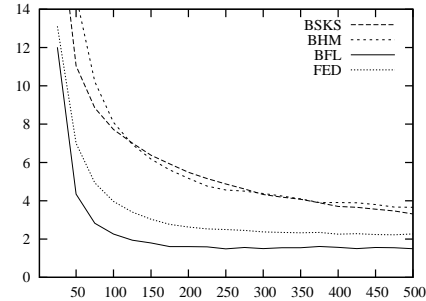
The genome we used for tests on encoded DNA sequences is a sequence of 4,638,690 base pairs of *Escherichia coli*. We used the encoded version of the file E.coli of the Large Canterbury Corpus¹.

Searching have been performed for patterns, of length m from 25 to 500, which have been taken as substring of the text at random starting positions.

¹ <http://www.data-compression.info/Corpora/CanterburyCorpus/>

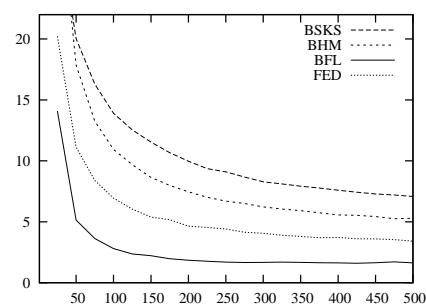
In the following tables, running times are expressed in hundredths of seconds. Best results are bold faced.

m	BBM	BSKS	BHM	BFL	FED
25	161.905	19.484	28.625	12.014	13.107
50	90.109	11.047	14.671	4.344	7.000
75	70.718	8.846	10.220	2.828	4.936
100	65.797	7.720	8.094	2.264	3.968
125	58.780	7.016	6.939	1.938	3.406
150	52.593	6.375	6.171	1.798	3.032
200	42.032	5.484	5.171	1.609	2.625
250	50.751	4.875	4.563	1.485	2.500
300	47.564	4.327	4.375	1.498	2.375
350	45.498	4.079	4.094	1.546	2.328
400	42.502	3.702	3.904	1.564	2.253
450	45.344	3.562	3.800	1.562	2.234
500	44.345	3.311	3.658	1.497	2.267



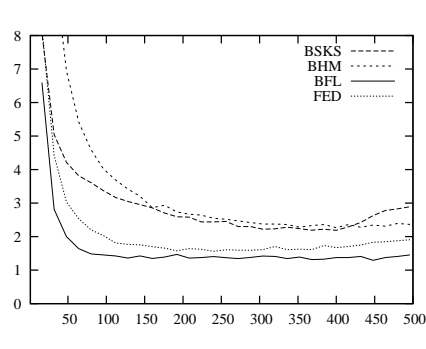
Experimental results for a $\text{Rand}(0/1)_{50}$ problem

m	BBM	BSKS	BHM	BFL	FED
25	188.469	29.842	33.110	14.095	20.219
50	112.720	20.031	17.860	5.142	11.125
75	88.953	16.299	13.251	3.624	8.390
100	82.360	13.909	10.938	2.797	6.938
125	74.671	12.531	9.686	2.358	6.032
150	69.875	11.531	8.641	2.218	5.389
200	58.952	9.967	7.452	1.843	4.641
250	64.921	9.093	6.690	1.689	4.406
300	61.219	8.283	6.218	1.671	4.063
350	58.141	7.921	5.908	1.670	3.796
400	54.420	7.595	5.563	1.624	3.718
450	57.402	7.284	5.423	1.642	3.594
500	55.296	7.077	5.281	1.625	3.405



Experimental results for a $\text{Rand}(0/1)_{70}$ problem

m	BBM	BSKS	BHM	BFL	FED
16	41.266	8.062	19.407	6.594	8.249
32	28.955	5.046	10.046	2.814	4.422
64	29.485	3.813	5.420	1.641	2.533
96	26.764	3.375	4.031	1.453	2.032
128	26.436	3.047	3.422	1.361	1.766
160	24.577	2.859	2.862	1.347	1.701
192	25.624	2.592	2.733	1.469	1.578
224	33.170	2.438	2.641	1.373	1.623
256	28.595	2.453	2.517	1.372	1.608
288	26.421	2.299	2.421	1.377	1.593
320	27.596	2.234	2.374	1.407	1.703
352	24.251	2.235	2.281	1.391	1.625
384	23.593	2.221	2.359	1.327	1.734
448	24.063	2.626	2.343	1.294	1.830
496	24.659	2.891	2.362	1.452	1.906



Experimental results for an encoded DNA sequence

Experimental results show that the BFL algorithm obtains the best run-time performance in all cases. In particular it turns out that the BFL algorithm is two times faster than the FED algorithm when searching on binary data.

In the case of encoded DNA sequences such a difference is less evident and the BFL algorithm turns out to be 15% faster than the FED algorithm which is specifically tuned for matching encoded DNA sequences.

Finally we report that, in most cases, the FED algorithm turns out to be more efficient than the BINARY-SKIP-SEARCH and BINARY-HASH-MATCHING algorithms, for both binary data and encoded DNA sequences.

5 Conclusion

An efficient algorithm for exact matching on binary strings and encoded DNA sequences has been presented. The algorithm combines a multi-pattern version of the BNDM algorithm with a simplified shift strategy of COMMENTZ-WALTER algorithm. Moreover it exploits the block structure of the binary strings and process text and pattern with no use of any bit manipulations. From our experimental results it turns out that the presented algorithm is the most effective in practical cases and outperforms existing solutions especially in the case of binary data.

References

1. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
2. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
3. C. Charras, T. Lecroq, and J. D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In *CPM*, volume 1448 of *LNCS*, pages 55–64. Springer-Verlag, 1998.
4. B. Commentz-Walter. A string matching algorithm fast on the average. In *ICALP*, volume 71 of *LNCS*, pages 118–132, 1979.
5. S. Faro and T. Lecroq. Efficient pattern matching on binary strings. In *Current Trends in Theory and Practice of Computer Science*, 2009. Poster.
6. J. Holub and B. Durian. Fast variants of bit parallel approach to suffix automata. *Talk given in: The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation*, <http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf>, 2005.
7. J. W. Kim, E. Kim, and K. Park. Fast matching method for DNA sequences. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *LNCS*, pages 271–281, 2007.
8. S. T. Klein and M. K. Ben-Nissan. Accelerating Boyer Moore searches on binary texts. In *CIAA*, volume 4783 of *LNCS*, pages 130–143. Springer-Verlag, 2007.
9. S. T. Klein, A. Bookstein, and S. Deerwester. Storing text retrieval systems on cdrom: Compression and encryption considerations. *ACM Trans. on Information Systems*, 7:230–245, 1989.
10. T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.
11. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *CPM*, volume 1448 of *LNCS*, pages 14–33. Springer-Verlag, 1998.
12. D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.