Efficient Pattern Matching on Binary Strings

Simone Faro[1] and Thierry Lecroq[2]

[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. The *binary string matching* problem consists in finding all the occurrences of a pattern in a text where both strings are built on a binary alphabet. This is an interesting problem in computer science, since binary data are omnipresent in telecom and computer network applications. Moreover the problem finds applications also in the field of image processing and in pattern matching on compressed texts. Recently it has been shown that adaptations of classical exact string matching algorithms are not very efficient on binary data. In this paper we present two efficient algorithms for the problem adapted to completely avoid any reference to bits allowing to process pattern and text byte by byte. Experimental results show that the new algorithms outperform existing solutions in most cases.

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

1 Introduction

Given a text t and a pattern p over some alphabet Σ of size σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t. It is a very extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

In this article we consider the problem of searching for a pattern p of length m in a text t of length n, with both strings are built over a binary alphabet, where each character of p and t is represented by a single bit. Thus memory space needed to represent t and p is, respectively, $\lceil n/8 \rceil$ and $\lceil m/8 \rceil$ bytes.

This 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.

The main reason for using binaries is size. A binary is a much more compact format than the symbolic or textual representation of the same information.

Consequently, less resources are required to transmit binaries over the network. For this reason the binary string matching problem finds applications also in pattern matching on compressed texts, when using the Huffman compression strategy [KS05,SD06,FG06].

Observe that the text t, and the pattern p to search for, cannot be directly processed as strings with a super-alphabet [Fre02,FG06], i.e., where each group of 8 bits is considered as a character of the text. This is because an occurrence of the pattern can be found starting at the middle of a group. Suppose, for instance, that $t = \texttt{011001001000100110100101000101001001}$ and $p = \texttt{0100110100}$. If we write text and pattern as groups of 8 bits then we obtain

$$\begin{aligned} t &= \texttt{01100100 10001001 10100101 00010100 1001} \\ p &= \texttt{01001 10100} \end{aligned}$$

The occurrence of the pattern at position 11 of the text cannot be located by a classical pattern matching algorithm based on super-alphabet.

It is possible to simply adapt classical efficient algorithms for exact pattern-matching to binary-matching with minor modifications. We can substitute in the algorithms reference to the character at position i with reference to the bit at position i. Roughly speaking we can substitute occurrences of $t[i]$ with $\textsc{getBit}(t, i)$ which returns the i-th bit of the text t. This transformation does not affect the time complexity of the algorithm but is time consuming and in general could be not very efficient.

In [KBN07] Klein and Ben-Nissan proposed an efficient variant of the Boyer-Moore 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. In [KBN07] the authors showed also by empirical results that the new variant performs better than the regular binary Boyer-Moore algorithm and even than binary versions of the most effective algorithms for classical pattern matching.

In this note we present two new efficient algorithms for matching on binary strings which, despite their $\mathcal{O}(nm)$ worst case time complexity, obtain very good results in practical cases. The first algorithm is an adaptation of the q-Hash algorithm [Lec07] which is among the most efficient algorithms for the standard pattern matching problem. We show how the technique adopted by the algorithm can be naturally translated to allow for blocks of bits.

The second solution can be seen as an adaptation to binary string matching of the Skip-Search algorithm [CLP98]. This algorithm can be efficiently adapted to completely avoid any reference to bits allowing to process pattern and text proceeding byte by byte.

The paper is organized as follows. In Section 2 we introduce basic definitions and the terminology used along the article. In Section 3 we introduce a high level model used to process binary strings avoiding any reference to bits. Next, in Section 4, we introduce the new solutions. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 5. Finally, we draw our conclusions in Section 6.

2 Preliminaries and basic definitions

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 $(j+1)$-st characters of p, for $0 \leq i \leq j < m$. Moreover, for any $i, j \in \mathbb{Z}$, we put $p[i\,..\,j] = \varepsilon$ if $i > j$ and $p[i\,..\,j] = p[\max(i,0), \min(j, m-1)]$ if $i \leq j$. A substring of p is also called a *factor* of p. A substring of the form $p[0\,..\,i]$ is called a *prefix* of p and a substring of the form $p[i\,..\,m-1]$ is called a *suffix* of p for $0 \leq i \leq m-1$. For any two strings u and w, we write $w \sqsupseteq u$ to indicate that w is a suffix of u. Similarly, we write $w \sqsubseteq u$ to indicate that w is a prefix of u.

Let t be a text of length n and let p be a pattern of length m. When the character $p[0]$ is aligned with the character $t[s]$ of the text, so that the character $p[i]$ is aligned with the character $t[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern p has *shift* s in t. In this case the substring $t[s\,..\,s+m-1]$ is called the *current window* of the text. If $t[s\,..\,s+m-1] = p$, we say that the shift s is *valid*.

Most string matching algorithms have the following general structure. First, during a *preprocessing phase*, they calculate useful mappings, generally in the form of tables, which later are accessed to determine nontrivial shift advancements. Next, starting with shift $s = 0$, they look for all valid shifts, by executing a *matching phase*, which determines whether the shift s is valid and computes a *positive* shift increment, Δs. Such increment Δs is used to produce the new shift $s + \Delta s$ to be fed to the subsequent matching phase.

For instance, in the case of the NAIVE string matching algorithm, there is no preprocessing phase and the matching phase always returns a unitary shift increment, i.e. all possible shifts are actually processed.

3 A High Level Model for Matching on Binary Strings

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 $i = 0, \ldots, m-1$ we have $p[i] = P[\lfloor i/k \rfloor]_{i \bmod k}$.

In this section we present a high level model to process binary strings which exploits the block structure of text and pattern to speed up the searching phase avoiding to work with bitwise operations. 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. The idea to eliminate any reference to bits and to proceed block by block has been first suggested in [CKP85] for fast decoding

(A) *Patt*	0	1	2	3
0	11001011	00101100	10110000	
1	01100101	10010110	01011000	
2	00110010	11001011	00101100	
3	00011001	01100101	10010110	
4	00001100	10110010	11001011	00000000
5	00000110	01011001	01100101	10000000
6	00000011	00101100	10110010	11000000
7	00000001	10010110	01011001	01100000

(C) *Last*
2
2
2
2
3
3
3
3

(B) *Mask*	0	1	2	3
0	11111111	11111111	11111000	
1	01111111	11111111	11111100	
2	00111111	11111111	11111110	
3	00011111	11111111	11111111	
4	00001111	11111111	11111111	10000000
5	00000111	11111111	11111111	11000000
6	00000011	11111111	11111111	11100000
7	00000001	11111111	11111111	11110000

Fig. 1. Let P =11001011001011001011O. **(A)** The matrix *Patt*. **(B)** The matrix *Mask*. **(C)** The array *Last*. In *Patt* and *Mask* bits belonging to P are underlined. Blocks containing a factor of P are presented with light gray background color.

of binary Huffman encoded texts. A similar approach has been adopted also in [KBN07,Fre02]. For the sake of uniformity we use in the following, when it is possible, the same terminology adopted in [KBN07].

Let $T[i]$ 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 $(m \mod k)$ bits of the last byte are defined. We suppose that the undefined bits of the last byte are set to 0.

In our high level model we define a sequence of several copies of the pattern memorized in the form of a matrix of bytes, *Patt*, of size $k \times (\lceil m/k \rceil + 1)$. Each row i of the matrix *Patt* contains a copy of the pattern shifted by i position to the right. The i leftmost bits of the first byte remain undefined and are set to 0. Similarly the rightmost $k - ((m+i) \mod k)$ bits of the last byte are set to 0. Formally the j-th bit of byte $Patt[i,h]$ is defined by

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

for $0 \le i < k$ and $0 \le h < \lceil (m+i)/k \rceil$.

Observe that each factor of length k of the pattern appears once in the table *Patt*. In particular, the factor of length k starting at position j of p is memorized in $Patt[k - (j \mod k), \lceil j/k \rceil]$.

The high level model uses bytes in the matrix *Patt* to compare the pattern block by block against the text for any possible shift of the pattern. However when comparing the first or last byte of P against its counterpart in the text,

```
PREPROCESS (P, m)
1   M ← 1^m 0^{k−plast}
2   for i = 0 to k-1 do
3     Last[i] = ⌈(m + i)/k⌉ − 1
4     for h = 0 to Last[i] do
5       Patt[i, h] ← (P[h] >> i)
6       Mask[i, h] ← (M[h] >> i)
7       if h > 0 then
8         X ← Patt[i, h] | (P[h − 1] << (k − i))
9         Patt[i, h] ← X
10        Y ← Mask[i, h] | (M[h − 1] << (k − i))
11        Mask[i, h] ← Y
12  return (Patt, Last, Mask)
```

(A)

```
BINARY-NAIVE (P, m, T, n)
1   (Patt, L, M) ← PREPROCESS (P, m)
2   s ← i ← w ← 0
3   while s < n do
4     j ← 0
5     while j < L[i] and
6       Patt[i, j] = (T[w + j]&M[i, j])
7       do j ← j + 1
8     if j = L[i] then Output(s)
9     i ← i + 1
10    if i = k then
11      w ← w + 1
12      i ← 0
13    s ← s + 1
```

(B)

Fig. 2. (A) The PREPROCESS procedure for the computation of the tables *Patt*, *Mask* and *Last*. **(B)** The BINARY-NAIVE algorithm for the binary string matching problem.

the bit positions not belonging to the pattern have to be neutralized. For this purpose we define a matrix of bytes, *Mask*, of size $k \times (\lceil m/k \rceil + 1)$, containing binary masks of length k. In particular a bit in the mask $Mask[i, h]$ is set to 1 if and only if the corresponding bit of $Patt[i, h]$ belongs to P. More formally

$$Mask[i, h]_j = \begin{cases} 1 & \text{if } 0 \leq kh - i + j < m \\ 0 & \text{otherwise} \end{cases} .$$

for $0 \leq i < k$ and $0 \leq h < \lceil (m + i)/k \rceil$.

Finally we need to compute an array, *Last*, of size k where $Last[i]$ is defined to be the index of the last byte in the row $Patt[i]$. Formally, for $0 \leq i < k$ we define $Last[i] = \lceil (m + i)/k \rceil$.

The procedure PREPROCESS used to precompute the tables defined above is presented in Figure 2(A). It requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space. Figure 1 shows the precomputed tables defined above for a pattern $P =$`110010110010110010110` of length $m = 21$ and $k = 8$.

The model uses the precomputed tables to check whether s is a valid shift without making use of bitwise operations but processing pattern and text byte by byte. In particular, for a given shift position s (the pattern is aligned with the s-th bit of the text), we report a match if

$$Patt[i, h] = T[j + h] \ \& \ Mask[i, h], \text{ for } h = 0, 1, ..., Last[i]. \tag{1}$$

where $j = \lfloor s/k \rfloor$ is the starting byte position in the text and $i = (s \mod k)$.

A simple BINARY-NAIVE algorithm, obtained with this high level model, is shown in Figure 2(B). The algorithm starts by aligning the left ends of the pattern and text. Then, for each value of the shift $s = 0, 1, \ldots, n - m$, it checks whether p occurs in t by simply comparing each byte of the pattern with its corresponding byte in the text, proceeding from left to right. At the end of the matching phase, the shift is advanced by one position to the right. In the worstcase, the BINARY-NAIVE algorithm requires $\mathcal{O}(\lceil m/k \rceil n)$ comparisons.

4 New Efficient Binary String Matching Algorithms

In this section we present two new efficient algorithms for matching on binary strings based on the high level model presented above. The first algorithm is an adaptation of the q-HASH algorithm [Lec07] which is among the most efficient algorithms for the standard pattern matching problem. We show how the technique adopted by the algorithm can be naturally translated to allow for blocks of bits.

The second solution can be seen as an adaptation to binary string matching of the SKIP-SEARCH algorithm [CLP98]. This algorithm can be efficiently adapted to completely avoid any reference to bits allowing to process pattern and text proceeding byte by byte.

4.1 The Binary-Hash-Matching Algorithm

Algorithms in the q-HASH family for exact pattern matching have been introduced in [Lec07] where the author presented an adaptation of the Wu and Manber multiple string matching algorithm [WM94] to single string matching problem.

The idea of the q-HASH algorithm is to consider factors of the pattern of length q. Each substring w of such a length q is hashed using a function $hash$ into integer values within 0 and 255. Then the algorithm computes in a preprocessing phase a function $Hs : \{0, 1, \ldots, 255\} \rightarrow \{0, 1, \ldots, m - q\}$, such that for each $0 \leq c \leq 255$ the value $Hs(c)$ is defined by

$$Hs(c) = \min\Big(\{0 \leq k < m - q \mid hash(p[m - k - q \,..\, m - k - 1]) = c\} \cup \{m - q\}\Big).$$

The searching phase of the algorithm consists of reading, for each shift s of the pattern in the text, the substring $w = t[s + m - q \,..\, s + m - 1]$ of length q. If $Hs(hash(w)) > 0$ then a shift of length $Hs(hash(w))$ is applied. Otherwise, when $Hs(hash(w)) = 0$ the pattern p is naively checked in the text. In this case a shift of length $(m - 1 - i)$ is applied, where i is the starting position of the rightmost occurrence in p of a factor $p[j \,..\, j + q - 1]$ such that $hash(p[j \,..\, j + q - 1]) = hash(p[m - q + 1 \,..\, m - 1])$.

If the pattern p is a binary string we can directly associate each substring of length q with its numeric value in the range $[0, 2^q - 1]$ without making use of the $hash$ function. In order to exploit the block structure of the text we take into account substrings of length $q = k$. This means that, if $k = 8$, each block B of k bits can be considered as a value $0 \leq B \leq 255$. Thus we define a function $Hs : \{0, 1, \ldots, 2^k - 1\} \rightarrow \{0, 1, \ldots, m\}$, such that for each byte $0 \leq B < 2^k$

$$Hs(B) = \min\Big(\{0 \leq u < m \mid p[m - u - k \,..\, m - u - 1] \sqsupseteq B\} \cup \{m\}\Big) \ .$$

Observe that if $B = p[m - k \,..\, m - 1]$ then $Hs[B]$ is defined to be 0.

```
COMPUTE-HASH(Patt, Last, Mask, m)
 1.  for B ← 0 to 2^k − 1 do
 1.    Hs[B] ← m
 2.  for i ← k − 1 downto 1 do
 3.    for B ← 0 to 2^k − 1 do
 4.      if Patt[i, 0] = B & Mask[i, 0]
 5.        then Hs[B] ← m − k + i
 6.  i ← h ← 0
 7.  for j ← 0 to m − k − 1 do
 8.    Hs[Patt[i, h]] ← m − k − j
 9.    i ← i − 1
10.    if i < 0 then
11.      i ← k − 1
12.      h ← h + 1
13.  return Hs
```

```
BINARY-HASH-MATCHING (P, m, T, n)
 1.  (Patt, Last, Mask) ← PREPROCESS (P, m)
 2.  Hs ← COMPUTE-HASH(Patt, Last, Mask, m)
 3.  gap ← k − (m mod k)
 4.  B ← Patt[i][Last[i]]
 5.  shift ← Hs[B], Hs[B] ← 0
 6.  j ← 0, sℓ ← m − 1
 7.  while j < ⌈n/k⌉ do
 8.    while sℓ ≥ k do
 9.      sℓ ← sℓ − k
10.      j ← j + 1
11.    B ← T[j] ≫ k − sℓ
12.    B ← B | (T[j − 1] ≪ (sℓ + 1))
13.    if Hs[B] = 0 then
14.      i ← (sℓ + gap) mod k
15.      h ← Last[i], q ← 0
16.      while h > 0 and
           Patt[i, h] = (T[j − q] & Mask[i, h])
17.        do h ← h − 1, q ← q + 1
18.      if h < 0 then Output(j × k + sℓ)
19.      sℓ ← sℓ + shift
20.    else sℓ ← sℓ + Hs[B]
```

Fig. 3. The BINARY-HASH-MATCHING algorithm for the binary string matching problem.

For example, in the case of the pattern $P = \texttt{11001011001011001011}$0, presented in Figure 1, we have $Hs[\texttt{01100101}] = 2$, $Hs[\texttt{11001011}] = 1$, and moreover $Hs[\texttt{10010110}] = 0$.

The code of the BINARY-HASH-MATCHING algorithm and its preprocessing phase are presented in Figure 3.

The preprocessing phase of the algorithm consists in computing the function Hs defined above and requires $\mathcal{O}(m + k2^{k+1})$ time complexity and $\mathcal{O}(m + 2^k)$ extra space. During the search phase the algorithm reads, for each shift position s of the pattern in the text, the block $B = t[s + m - q \,..\, s + m - 1]$ of k bits (lines `11-12`). If $Hs(B) > 0$ then a shift of length $Hs(B)$ is applied (line `20`). Otherwise, when $Hs(B) = 0$ the pattern p is naively checked in the text block by block (lines `15-18`).

After the test an advancement of length *shift* is applied (line `19`) where

$$shift = \min\Big(\{0 < u < m \mid p[m - u - k \,..\, m - u - 1] \sqsupset p[m - k \,..\, m - 1]\} \cup \{m\}\Big)$$

Observe that if the block B has its $s\ell$ rightmost bits in in the j-th block of T and the $(k - s\ell)$ leftmost bits in the block $T[j-1]$, then it is computed by performing the following bitwise operation

$$B = \Big(T[j] \gg (k - s\ell)\Big) \mid \Big(T[j-1] \ll (s\ell + 1)\Big)$$

The BINARY-HASH-MATCHING algorithm has a $\mathcal{O}(\lceil m/k \rceil n)$ time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

For blocks of length k the size of the *Hs* table is 2^k, which seems reasonable for $k = 8$ or even 16. For greater values of k it is possible to adapt the algorithm to choose the desired time/space tradeoff by introducing a new parameter $K \leq k$, representing the number of bits taken into account for computing the shift advancement. Roughly speaking, only the K rightmost bits of the current window of the text are taken into account, reducing the total sizes of the tables to 2^K at the cost of sometimes shifting the the pattern less than could be done if the full length of a block had been considered.

4.2 The Binary-Skip-Search Algorithm

The SKIP-SEARCH algorithm has been presented in [CLP98] by Charras, Lecroq and Pehoushek. The idea of the algorithm is straightforward. Let p be a pattern of length m and t a text of length n, both over a finite alphabet Σ. For each character c of the alphabet, a bucket collects all the positions of that character in the pattern. When a character occurs ℓ times in the pattern, there are ℓ corresponding positions in the bucket of that character. Formally, for $c \in \Sigma$ the SKIP-SEARCH algorithm computes the table $S[c]$ where

$$S[c] = \{i \mid 0 \leq i < m \ \wedge \ P[i] = c\}.$$

It is possible to notice that when the pattern is much shorter than the alphabet, many buckets are empty. The main loop of the search phase consists in examining every m-th text character, $t[j]$ (so there will be n/m main iterations). For each character $t[j]$, it uses each position in the bucket $S[t[j]]$ to obtain all possible starting positions of p in t. For each position the algorithm performs a comparison of p with t, character by character, until there is a mismatch, or until an occurrence is found.

For each possible block B of k bits, a bucket collects all pairs (i, h) in the table *Patt* such that $Patt[i, h] = B$. When a block of bits occurs more times in the pattern, there are different corresponding pairs in the bucket of that block. Observe that for a pattern of length m there are $m - k + 1$ different blocks of length k corresponding to the blocks $Patt[i, h]$ such that $kh - i \geq 0$ and $k(h + 1) - i - 1 < m$.

However, to take advantage of the block structure of the text, we can compute buckets only for blocks contained in the suffix of the pattern of length $m' = k\lfloor m/k \rfloor$. In such a way m' is a multiple of k and we could reduce to examine a block for each m'/k blocks of the text.

Formally, for $0 \leq B < 2^k$

$$S_k[B] = \{(i, h) : (m \mod k) \leq kh - i \leq m - k \ \wedge \ Patt[i, h] = B\}.$$

For example in the case of the pattern $P = \texttt{11001011001011001011}0$ we have $S_k[\texttt{01011001}] = \{(7, 2)\}$, $S_k[\texttt{01100101}] = \{(3, 1), (5, 2)\}$, $S_k[\texttt{11001011}] = \{(2, 1)\}$, $S_k[\texttt{10010110}] = \{(1, 1), (3, 2)\}$ and $S_k[\texttt{10110010}] = \{(4, 2), (6, 2)\}$.

```
PRECOMPUTE-SKIP-TABLE(Patt, m)
 1.  for b = 0 to 2^k − 1 do S_k[b] ← ∅
 2.  i ← h ← 0
 3.  for j = 0 to m − k do
 4.    if j ≥ (m mod k) then
 5.      b ← Patt[i, h]
 6.      S_k[b] = S_k[b] ∪ {(i, h)}
 7.    i ← i − 1
 8.    if i < 0 then
 9.      i ← k − 1
10.      h ← h + 1
11.  return S_k
```

```
BINARY-SKIP-SEARCH (P, m, T, n)
 1.  (Patt, Last, Mask) ← PREPROCESS(P, m)
 2.  S_k ← PRECOMPUTE-SKIP-TABLE(Patt, m)
 3.  shift ← ⌊m/k⌋ − 1
 4.  j ← shift − 1
 5.  while j < ⌈n/k⌉ do
 6.    for each (i, pos) ∈ S_k[T[j]] do
 7.      h ← 0
 8.      while h < Last[i] and
           P[i, h] = (T[j − pos + h] & Mask[i, h])
 9.        do h ← h + 1
10.      if h = Last[i] then Output(j × k + i)
11.    j ← j + shift
```

Fig. 4. The BINARY-SKIP-SEARCH algorithm for the binary string matching problem.

The BINARY-SKIP-SEARCH algorithm is shown in Figure 4. Its preprocessing phase consists in computing the buckets for all possible blocks of k bits. The space and time complexity of this preprocessing phase is $\mathcal{O}(m + 2^k)$. The main loop of the search phase consists in examining every (m'/k)th text block. For each block $T[j]$ examined in the main loop, the algorithm inspects each pair (i, pos) in the bucket $S_k[T[j]]$ to obtain a possible alignment of the pattern against the text (line 6). For each pair (i, pos) the algorithm checks whether p occurs in t by comparing $Patt[i, h]$ and $T[j - pos + h]$, for $h = 0, \ldots, Last[i]$ (lines 7-10). The BINARY-SKIP-SEARCH algorithm has a $\mathcal{O}(\lceil m/k \rceil n)$ quadratic worst case time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

In practice, if the block size is k, the BINARY-SKIP-SEARCH algorithm requires a table of size 2^k to compute the function S_k. This is just 256 for $k = 8$, but for $k = 16$ or even 32, such a table might be too large. In particular for growing values of k, there will be many cache misses, with strong impact on the performance of the algorithm. Thus for values of k greater than 8 it may be suitable to compute the function on the fly, still using a table for single bytes. Suppose for example that $k = 32$ and suppose B is a block of k bits. Let B_j be the j-th byte of B, with $j = 1, \ldots, 4$. The set of all possible pairs associated to the block B can be computed as

$$S_k[B] \;=\; S_k[B_1] \;\cap\; S_k^1[B_2] \;\cap\; S_k^2[B_3] \;\cap\; S_k^3[B_4]$$

where we have set $S_k^q[B_j] = \{(i, h) \mid (i, h + q) \;\in\; S_k[B_j]\}$.

If we suppose that the distribution of zeros and ones in the input string is like in a randomly generated one, then the probability of occurrence in the text of any binary string of length k is 2^{-k}. This is a reasonable assumption for compressed text [KBD89]. Then the expected cardinality of set $S_k[B]$, for a pattern p of length m, is $(m - 7) \times 2^{-8}$, that is less than 2 if $m < 500$. Thus in practical cases the set $S_k[B]$ can be computed in constant expected time.

5 Experimental Results

Here we present experimental data which allow to compare, in terms of running time and number of text character inspections, the following string matching algorithms under various conditions: the Binary-Naive algorithm (BNAIVE) of Figure 2, the Binary-Boyer-Moore algorithm by Klein (BBM) presented in [KBN07], the Binary-Hash-Matching algorithm (BHM) of Figure 3, and the Binary-Skip-Search algorithm (BSKS) of Figure 4.

For the sake of completeness, for experimental results on running times we have also tested the following algorithms for standard pattern matching: the q-Hash algorithm [Lec07] with $q = 8$ (HASH8) and the Extended-BOM algorithm [FL08] (EBOM). These are among the most efficient in practical cases. The q-Hash and Extended-BOM algorithms have been tested on the same texts and patterns but in their standard form, i.e. each character is an ASCII value of 8-bit, thus obtaining a comparison between methods on standard and binary strings.

To simulate the different conditions which can arise when processing binary data we have performed our tests on texts 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 [KBD89]. In contrast when processing binary images we aspect 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.

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. In particular, the algorithms have been tested on three $\mathsf{Rand}(1/0)_\gamma$ problems, for $\gamma = 50, 70$ and 90. Searching have been performed for binary patterns, of length m from 20 to 500, which have been taken as substring of the text at random starting positions.

In particular each $\mathsf{Rand}(1/0)_\gamma$ 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. In particular bit 0 appears with a percentage equal to $\gamma\%$.

Moreover, for each test, the average number of character inspections has been computed by taking the total number of times a text byte is accessed (either to perform a comparison with the pattern, or to perform a shift) and dividing it by the length of the text buffer.

In the following tables, running times (on the left) are expressed in hundredths of seconds. Tables with the number of text character inspections (on the right) are presented in light-gray background color. Best results are bold faced.

m	BNAIVE	BBM	BSKS	BHM	HASH8	EBOM	BNAIVE	BBM	BSKS	BHM
20	41.53	13.53	3.66	**3.40**	5.12	8.89	9.00	1.82	1.04	**0.90**
60	41.72	7.77	**1.16**	1.60	1.72	3.85	9.00	0.85	**0.20**	0.31
100	41.68	6.80	**0.70**	1.44	1.64	3.06	9.00	0.63	**0.13**	0.20
140	42.11	6.21	**0.89**	1.24	1.54	2.67	9.00	0.54	**0.10**	0.15
180	41.95	5.76	**0.66**	1.10	1.80	2.25	9.00	0.47	**0.08**	0.13
220	41.93	5.36	**0.74**	1.24	1.79	1.87	9.00	0.44	**0.07**	0.11
260	41.95	5.08	**0.54**	1.05	1.47	2.09	9.00	0.41	**0.07**	0.10
300	41.74	5.07	**0.54**	1.11	1.82	1.48	9.00	0.39	**0.06**	0.09
340	41.93	4.86	**0.39**	1.07	1.56	1.56	9.00	0.38	**0.06**	0.09
380	41.97	4.59	**0.46**	0.97	1.87	1.43	9.00	0.37	**0.06**	0.08
420	42.07	4.52	**0.31**	1.23	1.59	1.23	9.00	0.36	**0.05**	0.08
460	41.99	4.68	**0.23**	1.04	1.52	1.19	9.00	0.35	**0.05**	0.08
500	42.06	4.61	**0.37**	0.81	1.53	1.32	9.00	0.35	**0.05**	0.07

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

m	BNAIVE	BBM	BSKS	BHM	HASH8	EBOM	BNAIVE	BBM	BSKS	BHM
20	43.26	17.25	**4.01**	4.21	4.86	10.92	9.41	2.27	1.12	**1.01**
60	43.15	10.26	**1.66**	2.09	2.03	4.27	9.40	1.14	**0.29**	0.38
100	43.80	8.44	**1.60**	2.26	1.95	2.54	9.38	0.89	**0.21**	0.26
140	43.70	8.13	**1.28**	1.61	1.52	2.68	9.38	0.77	**0.18**	0.21
180	43.22	7.37	**1.02**	1.67	2.08	2.33	9.37	0.71	**0.17**	0.18
220	43.29	6.82	**1.08**	1.34	1.94	2.50	9.39	0.65	**0.16**	**0.16**
260	42.93	6.67	**1.07**	1.53	1.79	1.94	9.38	0.61	**0.15**	**0.15**
300	43.66	6.46	**0.89**	1.22	1.59	1.94	9.39	0.59	0.15	**0.14**
340	43.53	6.35	**0.97**	1.23	1.28	1.86	9.39	0.57	0.15	**0.13**
380	43.76	6.15	**0.70**	1.42	1.31	1.65	9.38	0.55	0.14	**0.12**
420	43.29	6.03	**0.85**	1.34	1.67	1.48	9.38	0.54	0.14	**0.12**
460	43.45	6.00	**0.92**	1.27	1.37	1.43	9.38	0.53	0.14	**0.11**
500	43.31	6.00	**0.70**	1.28	1.41	1.48	9.37	0.51	0.14	**0.11**

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

m	BNAIVE	BBM	BSKS	BHM	HASH8	EBOM	BNAIVE	BBM	BSKS	BHM
20	50.61	41.19	**18.51**	21.00	24.30	24.95	12.46	6.88	**3.79**	4.87
60	51.68	30.62	**13.61**	14.32	13.65	7.23	12.53	5.14	**2.82**	3.28
100	53.00	28.44	**12.22**	12.64	11.64	5.15	12.72	4.70	2.78	**2.76**
140	51.78	27.16	11.86	**11.44**	10.31	4.09	12.46	4.47	2.63	**2.53**
180	51.51	24.78	11.80	**10.21**	9.83	3.21	12.45	4.11	2.59	**2.22**
220	52.54	24.60	11.50	**9.63**	9.13	3.12	12.69	4.02	2.65	**2.09**
260	52.38	23.59	11.85	**8.74**	8.61	2.31	12.55	3.87	2.58	**1.97**
300	52.00	22.68	11.15	**8.73**	8.11	2.63	12.59	3.67	2.64	**1.80**
340	51.98	21.72	11.30	**8.02**	7.24	2.29	12.53	3.53	2.60	**1.70**
380	52.33	21.79	11.39	**7.66**	7.57	2.17	12.56	3.53	2.64	**1.69**
420	52.35	21.16	10.94	**7.58**	7.43	1.82	12.60	3.46	2.56	**1.60**
460	52.29	20.54	11.09	**6.75**	6.45	2.12	12.51	3.28	2.59	**1.48**
500	51.68	20.68	11.12	**7.42**	6.99	1.79	12.34	3.39	2.57	**1.55**

Experimental results for a Rand$(0/1)_{90}$ problem

Experimental results show that the Binary-Skip-Search and the Binary-Hash-Matching algorithms obtain the best run-time performance in all cases. In particular it turns out that the Binary-Skip-Search algorithm is the best choice when the distribution of character is uniform. In this case the algorithm is 10 times faster than Binary-Boyer-Moore, and 100 times faster than Binary-Naive. Moreover performs less than 50% of inspections performed by the Binary-Boyer-Moore algorithm, especially for long patterns.

For non-uniform distribution of characters the Binary-Hash-Matching algorithm obtains the best results in terms of both running time and number of character inspections. It turns out to be at least two times faster than Binary-Boyer-Moore algorithm and to perform a number of text character inspections which is less than 50% of that performed by the Binary-Boyer-Moore algorithm.

6 Conclusion

Efficient variants of the q-Hash and Skip-Search pattern matching algorithms have been presented for the case in which both text and pattern are over a binary alphabet. The algorithm exploit the block structure of the binary strings and process text and pattern with no use of any bit manipulations. Both algorithms have a $\mathcal{O}(n/m)$ time complexity. However, from our experimental results it turns out that the presented algorithms are the most effective in practical cases.

References

[CKP85] Y. Choueka, S. T. Klein, and Y. Perl. Efficient variants of Huffman codes in high level languages. In *Proc. 8th ACMSIGIR Conf., Montreal*, pages 122–130, 1985.

[CLP98] C. Charras, T. Lecroq, and J. D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, volume 1448 of *LNCS*, pages 55–64, Piscataway, NJ, 1998. Springer-Verlag.

[FG06] K. Fredriksson and S. Grabowski. Efficient algorithms for pattern matching with general gaps and character classes. In F. Crestani, P. Ferragina, and M. Sanderson, editors, *Proc. Conference on String Processing and Information Retrieval SPIRE06*, volume 4209 of *LNCS*, pages 267–278. Springer-Verlag, 2006.

[FL08] S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. In J. Holub and J. Žďárek, editors, *Proc. of the Prague Stringology Conference*, pages 146–160, 2008.

[Fre02] K. Fredriksson. String matching with super-alphabets. In A. H. F. Laender and A. L. Oliveira, editors, *Proc. Symposium on String Processing and Information Retrieval SPIRE02*, volume 2476 of *LNCS*, pages 44–57. Springer-Verlag, 2002.

[KBD89] 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.

[KBN07] S. T. Klein and M. K. Ben-Nissan. Accelerating Boyer Moore searches on binary texts. In J. Holub and J. Žďárek, editors, *Proc. of the 12th International Conference on Implementation and Application of Automata, CIAA07*, volume 4783 of *LNCS*, pages 130–143, Prague, Czech Republic, 2007. Springer-Verlag.

[KS05] S. T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. *Inf. Process. Manage.*, 41(4):829–841, 2005.

[Lec07] T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.

[SD06] D. Shapira and A. H. Daptardar. Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts. *Inf. Process. Manage.*, 42(2):429–439, 2006.

[WM94] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.