Adapting Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | faro | giaquinta}@dmi.unict.it

Abstract. In this paper we propose an efficient approach to the compressed string matching problem on Huffman encoded texts, based on the Boyer-Moore strategy. Once a candidate valid shift has been located, a subsequent verification phase checks whether the shift is codeword aligned by taking advantage of the skeleton tree data structure. Our approach leads to algorithms with a sublinear behavior on the average, as shown by extensive experimentation.

Keywords: string matching, compression algorithms, Huffman coding, Boyer-Moore algorithm, text processing, information retrieval

1 Introduction

The *compressed string matching problem* is a variant of the classical string matching problem. It consists in searching for all the occurrences of a given pattern P in a text T stored in compressed form.

A straightforward solution is the so-called *decompress-and-search* strategy, which consists in decompressing the text and then using any classical string matching algorithm for searching. However, recent results show that in many cases searching directly in compressed texts can be more efficient.

Here we are interested in the string matching problem on Huffman compressed texts. The Huffman data compression method [7] is an optimal statistical coding. More precisely, the Huffman algorithm computes an optimal *prefix code*, relative to given frequencies of the alphabet characters. A prefix code is a set of (binary) words containing no word which is a prefix of another word in the set. Thanks to such a property, decoding is particularly simple. Indeed, a binary prefix code can be represented by an ordered binary tree, whose leaves are labeled with the alphabet characters and whose edges are labeled by 0 (left edges) and 1 (right edges) in such a way that the codeword of an alphabet character is the word labeling the branch from the root to the leaf labeled by the same character.

Prefix code trees, as computed by the Huffman algorithm, are called *Huffman trees*. These are not unique, by any means. The usually preferred tree for a given set of frequencies, out of the various possible Huffman trees, is the one induced by *canonical Huffman codes* [14]. This tree has the property that, when scanning its leaves from left to right, the sequence of depths is nondecreasing.

When performing a search on the bitstream of a Huffman encoded text by a classical string matching algorithm, one faces the problem of *false matches*, i.e., occurrences of the encoded pattern in the encoded text which do not correspond to occurrences of the pattern in the original text. Indeed, the only valid occurrences of the pattern are those correctly aligned with codeword boundaries, or, otherwise said, valid matches

Domenico Cantone, Simone Faro, Emanuele Giaquinta: Adapting Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts, pp. 29–39.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2 © Czech Technical University in Prague, Czech Republic

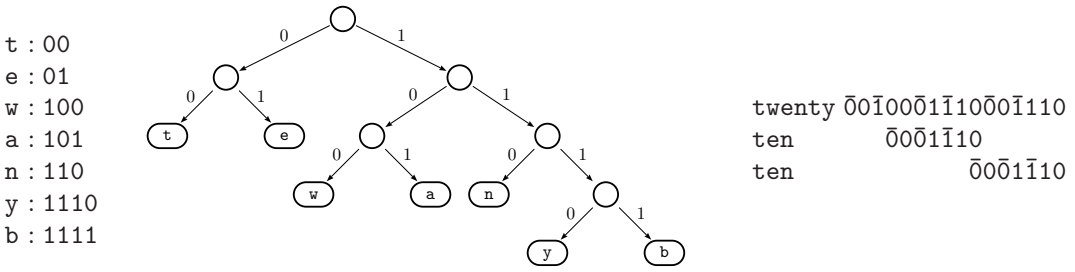

Figure 1. A Huffman code for the set of symbols {t, e, w, a, n, y, b}. The binary string 0̄0̄1̄0̄0̄1̄1̄1̄0̄0̄1̄1̄1̄0 is the encoding of the string twenty, where a "bar" indicates the starting bit of each codeword. Two occurrences of the binary string ten start at the 4-th and 10-th bit of the encoded version of the string twenty. Both of them are false matches.

must start on the first bit of a codeword. Consider, for example, the Huffman code presented in Figure 1. Note that there are two false occurrences of the string ten starting at the 4-th and at the 10-th bit, respectively, of the encoded string twenty. Thus a verification that the occurrences detected by the pattern matching algorithm are aligned on codeword boundaries is in order.

False matches could be avoided by using codes in which no codeword is a prefix or a suffix of any other codeword. However, such codes, which are called *affix* or *fix-free*, are extremely infrequent [5].

Klein and Shapira [11] showed that, for long enough patterns, the probability of finding a false match is often very low, independently of the algorithm. They then proposed a probabilistic algorithm which works on the assumption that Huffman codes tend to realign quickly after an error.

More recently, Shapira and Daptardar [15] proposed a modification of the Knuth-Morris-Pratt algorithm [12], in this paper referred to as Huffman-Kmp, which makes use of a data structure, called *skeleton tree* [9], suitably designed for efficient decoding of Huffman encoded sequences. The resulting algorithm is characterized by fast search times, if compared with the *decompress-and-search* method.

Algorithms based on the Boyer-Moore algorithm [2] have been considered unsuitable for searching Huffman encoded texts because the right to left scan does not allow one to determine the codeword boundaries in the compressed text, unless the text is decoded from left to right. In addition, Boyer-Moore-like algorithms are generally considered unsuitable for binary alphabets.

In this paper we present a new way to exploit skeleton trees for adapting Boyer-Moore-like algorithms to the compressed string matching problem in Huffman encoded texts. Specifically, we use skeleton trees to verify codeword alignments rather than for decoding. This allows us to skip up to 70% of bits during the processing of the encoded text. Futhermore, we make use of algorithms based on the Boyer-Moore strategy, suitably adapted for searching on binary strings by regarding texts and patterns as sequences of q-grams rather than as sequences of bits.

The paper is organized as follows. In Section 2 we introduce basic definitions and notations. In Section 3 we describe a strategy based on skeleton trees which is not specific to any algorithm and then in Section 4 we apply it to two string matching

algorithms for binary strings. In Section 5 we present some experimental results and finally we draw our conclusions in Section 6.

2 Some Basic Definitions and Preliminaries

A string P of length $|P| = m \geq 0$ 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 ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < m$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the $(i+1)$-st 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

$$P[i\,..\,j] = \begin{cases} \varepsilon & \text{if } i > j \\ P[\max(i,0)\,..\,\min(j, m-1)] & \text{if } i \leq j. \end{cases}$$

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 P and Q, we write $Q \sqsupseteq P$ to indicate that Q is a suffix of P. Similarly, we write $Q \sqsubseteq P$ to indicate that Q is a prefix of P. In addition, we write $Q.P$ to denote the concatenation of Q and P. Also, if P is a string of length m and $P[i] = b$, for $i = 0, \ldots, m-1$, then we write $P = b^m$.

A *compression method* for a given text T over an alphabet Σ is characterized by a system $(\mathcal{E}, \mathcal{D})$ of two complementary functions,

- an *encoding function* $\mathcal{E} : \Sigma \to \{0,1\}^+$, and
- an inverse *decoding function* $\mathcal{D}$,

such that $\mathcal{D}(\mathcal{E}(c)) = c$, for each $c \in \Sigma$. The encoding function $\mathcal{E}$ is then recursively extended over strings of characters by putting

$$\begin{aligned} \mathcal{E}(\varepsilon) &= \varepsilon \\ \mathcal{E}(T[0\,..\,\ell]) &= \mathcal{E}(T[0\,..\,\ell-1]).\mathcal{E}(T[\ell]), \quad \text{for } 0 \leq \ell < |T|, \end{aligned}$$

so that $\mathcal{E}(T) = \mathcal{E}(T[0\,..\,|T|-1])$ is just a binary string, i.e., a string over the alphabet $\{0,1\}$.

For ease of notation, we usually write t in place of $\mathcal{E}(T)$ and, more generally, denote binary strings by lowercase letters.

Binary strings are conveniently stored in blocks of k bits, typically bytes ($k = 8$), half-words ($k = 16$), or words ($k = 32$), which can be processed at the cost of a single operation. If p is any binary string, we denote by B_p the vector of blocks whose concatenation gives p, for a given block size k, so that

$$p[i] = B_p[\lfloor i/k \rfloor][i \bmod k], \quad \text{for } i = 0, \ldots, |p| - 1$$

(we assume that the last block, if not complete, is padded with 0's).

Thus, a genuine solution to the *compressed string matching problem* consists in finding *all* occurrences of a pattern P in a text T, over a common alphabet Σ, by operating directly on the block vectors B_t and B_p, representing respectively the binary strings $t = \mathcal{E}(T)$ and $p = \mathcal{E}(P)$ (again relative to a fixed block size k).

The algorithms for the compressed string matching problem in Huffman encoded texts, to be presented in Section 3, are based on a high-level model to process binary strings, adopted in [10,8,4], which we review next.

(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

Figure 2. Let P =110010110010110010110. **(A)** The matrix *Patt*. **(B)** The matrix *Mask*. **(C)** The array *Last*. In the tables *Patt* and *Mask*, bits belonging to P are underlined. Blocks containing a factor of P of length 8 have a shaded background.

2.1 A High-Level Model for Matching on Binary Strings

Let us assume 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.

We first define a vector *Patt*, of size $k \times (\lceil m/k \rceil + 1)$, consisting in several copies of the pattern P stored in the form of a matrix B_p of bytes, where $p = \mathcal{E}(P)$ and $m = |p|$. More precisely, the i-th row of the matrix *Patt*, for $i = 1, \ldots, k$, contains a copy of p shifted by i position to the right, whose length in bytes is $m_i = \lceil (m+i)/k \rceil$. 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.

Observe that each factor of p of length k appears exactly once in the table *Patt*. For instance, the factor of length k starting at position j of p is stored in $Patt[k - (j \bmod k), \lceil j/k \rceil]$.

The vector *Patt* is paired with a matrix of bytes, *Mask*, of size $k \times (\lceil m/k \rceil + 1)$, containing binary masks of length k, to distinguish between significant and padding bits in *Patt*. 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.

Finally, we define a vector *Last*, of size k, where $Last[i]$ is the index of the last byte in the row $Patt[i]$, i.e., $Last[i] = m_i$, for $0 \leq i < k$.

The procedure Preprocess used to precompute the above vectors requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space. Figure 2 shows the tables *Patt*, *Mask*, and *Last* relative to the pattern P =110010110010110010110, for a block size $k = 8$.

When the pattern is aligned with the s-th bit of the text, a match is reported if

$$Patt[i, h] = B_t[j + h] \ \&\ Mask[i, h]\,,$$

for $h = 0, 1, \ldots, Last[i]$, where

- B_t is the block representation of the text encoding $t = \mathcal{E}(T)$,
- $j = \lfloor s/k \rfloor$ is the starting byte position in t,
- $i = (s \bmod k)$, and
- "&" is the bitwise logic AND.

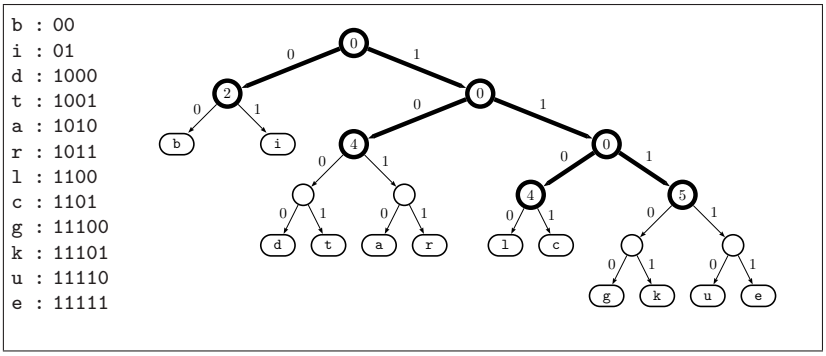

Figure 3. The Huffman tree induced by a Huffman code for the set of symbols $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{g}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{u}\}$. The skeleton tree is identified by bold lines.

3 Skeleton Tree Based Verification

The *skeleton tree* [9] is a pruned canonical Huffman tree, whose leaves correspond to minimal depth nodes in the Huffman tree which are roots of complete subtrees. It is useful to maintain at each leaf of a skeleton tree the common length of the codeword(s) sharing the prefix which labels the path from the root to it. A fast algorithm for building skeleton trees is described in [9]. Figure 3 shows a canonical Huffman tree and its corresponding skeleton tree, for the set of symbols $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{g}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{u}\}$, relative to suitable character frequencies.

Skeleton trees allow a faster Huffman decoding because, once the codeword length has been retrieved at its leaves, it is possible to read a burst of bits to complete the codeword, or just to skip them, if one is only interested in finding codeword boundaries.

Our approach consists in searching for the candidate occurrences of B_p in B_t, where we recall that B_p and B_t are respectively the block vectors associated to given Huffman encoded pattern and text, using Boyer-Moore-like algorithms and then taking advantage of the skeleton tree to verify whether the candidate matches are codeword aligned. In this way we obtain a substantial speedup, especially when the frequency of the pattern in the text is low or when the length of the pattern increases.

For every candidate valid shift s found by the binary pattern matching algorithm, one must verify whether s is codeword aligned. For this purpose, we maintain an offset ρ pointing at the start of the last window where a skeleton tree verification took place. The offset ρ is then updated, with the aid of the skeleton tree, to a minimal position $\rho^* \geq s$ which is codeword aligned. Only if $\rho^* = s$ the current window is codeword aligned and s is a valid shift. Plainly, the performance of the algorithm depends on the number of skeleton tree verifications and on the relative distance between candidate valid shifts.

Figure 4 shows the pseudocode for the procedure Sk-Align used to update ρ. In the pseudocode we assume that the starting value of ρ is codeword aligned and that a node x in the skeleton tree is a leaf if the corresponding key is nonzero, i.e., if $\mathtt{Key}(x) > 0$. If $\mathtt{Key}(x) = \ell > 0$ and c_x is the bit code which labels the path from the root to x, then all codewords c such that $c_x \sqsubseteq c$ have a length equal to ℓ. Thus, if we are interested only in the codeword boundaries, we can skip the $\ell - |c_x|$ following bits and restore the skeleton-tree verification from the first bit of the next codeword.

```
SK-ALIGN (root, t, ρ, b)
 1. x ← root, ℓ ← 0
 2. while TRUE do
 3.       B = B_t[⌊ρ / k⌋] ≪ (ρ mod k)
 4.       if B < 2^(k−1) then x ← LEFT(x) else x ← RIGHT(x)
 5.       if KEY(x) ≠ 0 then
 6.            ρ ← ρ + KEY(x) − ℓ, ℓ ← 0, x ← root
 7.            if ρ ≥ b then break
 8.       else ρ ← ρ + 1,   ℓ ← ℓ + 1
 9. return ρ
```

Figure 4. Procedure SK-ALIGN(*root*, t, ρ, b) which computes the next codeword alignment starting from position ρ, where *root* is the root of the skeleton tree, t is the encoded text in binary form, b is a codeword boundary, and k is the block size ($\ll$ denotes the left shift operator).

Consider as an example the search of the pattern $P =$ "bit" in the text $T =$ "abigblackbugbitabigblackbear". Suppose moreover that codewords are defined by the Huffman tree of Figure 3, so that $p = \mathcal{E}(P) =$"00011001".
A first candidate valid shift is encountered at position 12 in t, as shown below

```
   t  10100001111000011001010110111101001111011100000110011010000 1[···]
   p              0̄0̄1̄1̄001
verif. 1̄0--0̄-0̄-1̄11--0̄
```

The skeleton tree verification starts at position 0 and stops at position 13, skipping 6 bits over 14 (unprocessed bits are represented by the symbol "-"), showing that the occurrence at position 12 is not codeword aligned.

A second occurrence is found at the 45-th bit of t, as shown below

```
   t  [···]00011001010110111101001111011100 00011001 10100001111000[···]
   p                                        0̄0̄1̄1̄001
verif.      0̄-1̄10-1̄0--1̄10-1̄11--0̄-1̄11--1̄11--0̄
```

The skeleton tree verification restarts from position 14 and finds a codeword alignment at position 45. Thus the occurrence is codeword aligned and the shift is valid. The verification skips 12 bits over 32.

Finally, a third candidate valid shift is found at the 65-th bit of t. This time, the skeleton tree verification skips 10 bits over 22.

```
   t  [···]00011001 0101000011110 00011001 0101101111010011111110101011
   p                              0̄0̄1̄1̄001
verif.      0̄-0̄-1̄0--1̄0--0̄-0̄-1̄11--0̄
```

The strategy presented above for verifying codeword alignment is general and not specific to any algorithm.

4 Adapting Two Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts

Next we deal with the problem of searching for all candidate valid shifts. For this purpose, we present two algorithms which are adaptations to the case of Huffman encoded texts, along the lines of the high-level model outlined in Section 2.1, of the FED algorithm [8] and the BINARY-HASH-MATCHING algorithm [4].

4.1 The Huffman-Hash-Matching Algorithm

Algorithms in the q-HASH family for exact pattern matching have been introduced in [13], by adapting the Wu and Manber multiple string matching algorithm [17] to the single string matching problem. Recently, variants of the q-HASH algorithms have been proposed for searching on binary strings [4].

The first algorithm which we present, called HUFFMAN-HASH-MATCHING, associates directly each binary substring of length q with its numeric value in the range $[0, 2^q - 1]$, without using any *hash* function. To exploit the block structure of the text, the algorithm considers substrings of length $q = k$.

To begin with, a function $Hs : \{0, 1, \ldots, 2^k - 1\}, \rightarrow \{0, 1, \ldots, m\}$, defined by

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

for each byte $0 \le B < 2^k$, is computed during the preprocessing phase. Observe that if $B = p[m-k\,..\,m-1]$, then $Hs[B] = 0$.

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

In contrast with algorithms in the q-HASH family, where the maximum shift is $m - q$, in this case maximum shifts can reach the value m. Since we do not use a hash function but rather map directly the binary substrings of the pattern, the shift table can be modified by taking into account the prefixes of the patterns $Patt[i]$ of length $k - i$, with $1 \le i \le k - 1$. Thus Hs can be conveniently computed by setting $Hs[B] = m-k+i$, where i is the minimum index such that $Patt[i][0] \sqsupseteq B$, if it exists; otherwise $Hs[B]$ is set to m.

The code of the HUFFMAN-HASH-MATCHING algorithm is presented in Figure 5.

The preprocessing phase of the algorithm just 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 $B = B_t[s + m - k\,..\,s + m - 1]$ of k bits (line `9`). If $Hs(B) > 0$ then a shift of length $Hs(B)$ takes place (line `21`). Otherwise, if $Hs(B) = 0$, the pattern p is naively checked in the text block by block (lines `11-15`). The verification step is performed using the procedure SK-ALIGN described before (lines `16-19`).

After the test, an advancement of length *shift* takes place (line `20`), 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 the j-th block of t and the $(k - s\ell)$ leftmost bits in the block $B_t[j-1]$, then it is computed by performing the following bitwise operations (line `9`)

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

The HUFFMAN-HASH-MATCHING algorithm has an overall $\mathcal{O}(\lfloor m/k \rfloor n)$-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

```
HUFFMAN-HASH-MATCHING (p, m, t, n)
 1. root ← BUILD-SK-TREE(φ)
 2. (Patt, Last, Mask) ← PREPROCESS (p, m)
 3. Hs ← COMPUTE-HASH(Patt, Last, Mask, m)
 4. ρ ← 0
 5. i ← (k − (m mod k)) mod k
 6. B ← Patt[i][Last[i]]
 7. shift ← Hs[B], Hs[B] ← 0
 8. gap ← i + 1,    j ← m − 1
 9. while j < n do
10.      s ← j ≫ 3,    sℓ ← j & 7
11.      B ← (Bt[s] ≫ (k − sℓ))| (Bt[s − 1] ≪ (sℓ + 1))
12.    if Hs[B] = 0 then
13.       i ← (sℓ + gap) mod k
14.       h ← Last[i],    q ← s
15.       while h ≥ 0 and
16.             Patt[i, h] = (Bt[q] & Mask[i, h]) do
17.          h ← h − 1,    q ← q − 1
18.       if h < 0 then
19.          b ← (q + 1) × k + i
20.          ρ ← SK-ALIGN(root, t, ρ, b)
21.          if ρ = b then PRINT(b)
22.       j ← j + shift
23.    else j ← j + Hs[B]
```

```
HUFFMAN-FED (p, m, t, n)
 1. root ← BUILD-SK-TREE(φ)
 2. (Patt, Last, Mask) ← PREPROCESS (p, m)
 3. (δ, λ) ← COMPUTE-FED(Patt, Last, m)
 4. ρ ← 0
 5. s = m/8
 6. while s < n do
 7.    for each i in λ[Bt[s]] do
 8.       h ← Last[i]
 9.       q ← s + 1
10.       while h ≥ 0 and
11.          Patt[i][h] = Bt[q] &Mask[i][h] do
12.             h ← h − 1
13.             q ← q − 1
14.       if h < 0 then
15.          b ← (q + 1) × 8 + i
16.          ρ ← SK-ALIGN(root, t, ρ, b)
17.          if ρ = b then PRINT(b)
18.    do
19.       s ← s + δ[Bt[s + 1]]
20.    while s < n and δ[Bt[s]] ≠ 1
```

Figure 5. The HUFFMAN-HASH-MATCHING algorithm and the HUFFMAN-FED algorithm for the compressed string matching problem on Huffman encoded texts. Parameters p and t stand for the Huffman compressed version of the pattern and text, respectively.

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 size of the tables to 2^K, at the price of possibly getting shorter shift advancements of the pattern than the ones that would have been obtained if the full length of blocks had been taken into consideration.

4.2 The Huffman-Fed Algorithm

The FED algorithm [8] (Fast matching with Encoded DNA sequences) is a string matching algorithm specifically designed for matching DNA sequences compressed with a fixed-length encoding, requiring two bits for each character of the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. It combines a multi-pattern version of the QUICK-SEARCH algorithm [16] and a simplified version of the COMMENTZ-WALTER algorithm [3]. However, its strategy is general enough to be adapted to different encodings, including the Huffman one.

The resulting algorithm, which we call HUFFMAN-FED, makes use of a shift table δ and a hash table λ, both of size 2^k.

More specifically, the shift table δ is defined as follows. For $0 \leq i < k$ and $c \in \Sigma$, we first define the QUICK-SEARCH shift table $qs[i][c]$, by putting

$$qs[i][c] = \min\Big(\{m_i - 2 + 1\} \cup \{m_i - 2 + 1 - k \mid Patt[i][k] = c \text{ and } 1 \leq k \leq m_i - 2\}\Big).$$

Then, we put $\delta[c] = \min\{qs[i][c], 0 \leq i < k\}$, for $c \in \Sigma$.

The algorithm maintains also, 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

$$\lambda[B] = \{0 \leq i < k \mid P[i][Last[i] - 1] = B\}.$$

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

The code of the HUFFMAN-FED algorithm is presented in Figure 5.

The preprocessing phase of the algorithm consists in computing the shift table δ and the hash table λ defined above and, as in the HUFFMAN-HASH-MATCHING algorithm, it requires $\mathcal{O}(m + k2^{k+1})$-time complexity and $\mathcal{O}(m + 2^k)$ extra space.

During the searching phase, the algorithm performs a fast loop using the shift table δ to locate a candidate alignment of the pattern (lines `18-20`). In particular, the algorithm checks whether $\delta[B_t[s]] \neq 1$ and, if this is the case, it advances the shift by $\delta[B_t[s+1]]$ positions to the right.

If $\delta[B_t[s]] = 1$ then, by definition of δ, we have $B_t[s] = P[i, Last[i] - 1]$, for some $0 \leq i < k$. In such a case the last byte of the current window is used as an index in the hash table and all patterns $Patt[i]$, such that $i \in \lambda[B_t[s]]$, are checked naively against the window (line `7`). For each alignment i found, the pattern $Patt[i]$ is compared block by block with the text.

As in the HUFFMAN-HASH-MATCHING algorithm, one has also to verify that the window is codeword aligned (line `14-17`).

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

5 Experimental Results

In this section we present experimental results which allow to compare, in terms of running times and percentage of processed bits, the following algorithms:

- the HUFFMAN-KMP algorithm (HKMP) [15];
- the HUFFMAN-HASH-MATCHING algorithm (HHM), presented in Section 4.1;
- the HUFFMAN-FED algorithm (HFED), presented in Section 4.2.

In addition, we also tested an algorithm based on the *decompress-and-search* method (D&S for short) that makes use of the 3-*Hash* algorithm [13] for classical exact pattern matching, which is considered among the most efficient algorithms for the problem.

All algorithms have been implemented in the `C` programming language and have been compiled with the `GNU C Compiler`, using the optimization options `-O2 -fno-guess-branch-probability`. The tests have been performed on a 1.5 GHz PowerPC G4 and running times have been measured with a hardware cycle counter, available on modern CPUs.

We used the following input files:

- the English King James version of the "Bible" (3 Mb),
- the English "CIA World Fact Book" (2 Mb), and
- the Spanish novel "Don Quixote" by Cervantes (2 Mb).

The first two files are from the Canterbury Corpus [1], whereas the third one is from the Project Gutenberg [6].

For each input file, we have generated sets of 100 patterns of fixed length m, for m ranging in the set {4, 8, 16, 32, 64, 128, 256}, randomly extracted from the text. For each set of patterns we reported the mean over the running times of the 100 runs. The tables also show the minimum ($l_{\min}$) and maximum ($l_{\max}$) length in bits of the compressed patterns. For each set of patterns we have also computed the average number of processed bits.

In the following tables, running times are expressed in milliseconds whereas the number of processed bits is expressed as percentage of the total number of bits.

Running times						Processed bits			
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S	m	HKMP	HHM	HFED
4	[17, 31]	188.64	134.79	146.34	502.82	4	0.75	0.81	0.95
8	[36, 53]	185.77	105.59	112.98	491.87	8	0.76	0.68	0.77
16	[79, 102]	185.99	76.04	81.25	489.03	16	0.75	0.45	0.53
32	[164, 204]	184.23	65.78	70.31	487.74	32	0.75	0.42	0.48
64	[336, 378]	185.36	64.71	68.91	489.27	64	0.75	0.38	0.42
128	[694, 768]	187.73	72.00	77.11	487.31	128	0.75	0.34	0.37
256	[1383, 1545]	184.09	61.45	65.77	488.46	256	0.76	0.34	0.36

Experimental results on the Huffman encoded version of the King James version of the Bible

Running times						Processed bits			
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S	m	HKMP	HHM	HFED
4	[18, 29]	96.35	64.13	74.23	296.69	4	0.67	0.74	0.98
8	[38, 53]	95.50	49.38	56.47	289.82	8	0.66	0.55	0.68
16	[77, 108]	95.23	39.26	45.03	287.78	16	0.66	0.43	0.50
32	[162, 207]	94.74	33.55	38.53	287.34	32	0.65	0.35	0.40
64	[327, 392]	94.99	34.21	39.39	287.85	64	0.65	0.35	0.38
128	[662, 761]	94.42	28.54	32.92	287.51	128	0.64	0.29	0.31
256	[1347, 1610]	94.39	29.67	34.18	287.21	256	0.65	0.30	0.32

Experimental results on the Huffman encoded version of the CIA World Fact Book

Running times						Processed bits			
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S	m	HKMP	HHM	HFED
4	[18, 35]	122.25	87.44	95.44	308.56	4	0.75	0.81	0.95
8	[37, 60]	119.33	73.69	79.74	302.38	8	0.76	0.68	0.77
16	[83, 140]	120.35	45.99	49.67	300.53	16	0.75	0.45	0.53
32	[171, 216]	120.12	45.97	49.70	299.81	32	0.75	0.42	0.48
64	[348, 525]	119.20	41.86	45.26	300.90	64	0.75	0.38	0.42
128	[712, 1068]	117.55	37.80	40.83	299.78	128	0.75	0.34	0.37
256	[1439, 1773]	124.10	38.43	41.45	300.36	256	0.76	0.34	0.36

Experimental results on the Huffman encoded version of "Don Quixote"

The experimental results show that the HUFFMAN-HASH-MATCHING and HUFFMAN-FED algorithms always achieve the best running times. In addition, the HUFFMAN-HASH-MATCHING algorithm always obtains better results than the HUFFMAN-FED algorithm. In particular the running time of both algorithms decreases as the length of the pattern increases, since, as is reasonable to expect, the frequency of the patterns, and thus the number of skeleton tree verifications, is inversely proportional to m.

As expected, the HUFFMAN-KMP algorithm maintains the same performance independently of the pattern frequency. The gain of our algorithms compared to HUFFMAN-KMP is at least around 20 % and grows as the pattern frequency decreases and the pattern length increases.

Observe that, with the exception of very short patterns, the percentage of bits processed by our newly presented algorithms is always lower than that of the HUFFMAN-KMP algorithm and, in many cases, the gain is almost 50 %.

6 Conclusions

We have presented a new efficient approach to the compressed string matching problem on Huffman encoded texts, based on the BOYER-MOORE strategy. Codeword alignment takes advantage of the skeleton tree data structure, which allows to skip over a significant percentage of the bits. In particular, we have presented adaptations of the BINARY-HASH-MATCHING and FED algorithms for searching Huffman encoded texts. The experimental results show that our algorithms exhibit a sublinear behavior on the average and in most cases are able to skip more than 50 % of the total number of bits in the encoded text.

References

1. R. ARNOLD AND T. BELL: *A corpus for the evaluation of lossless compression algorithms*, in DCC '97: Proceedings of the Conference on Data Compression, Washington, DC, USA, 1997, IEEE Computer Society, p. 201, `http://corpus.canterbury.ac.nz/`.
2. R. S. BOYER AND J. S. MOORE: *A fast string searching algorithm.* 20(10) 1977, pp. 762–772.
3. B. COMMENTZ-WALTER: *A string matching algorithm fast on the average*, in Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings, vol. 71 of Lecture Notes in Computer Science, 1979, pp. 118–132.
4. S. FARO AND T. LECROQ: *Efficient pattern matching on binary strings*, in Current Trends in Theory and Practice of Computer Science (SOFSEM 09), 2009.
5. A. S. FRAENKEL AND S. T. KLEIN: *Bidirectional Huffman coding.* Comput. J., 33(4) 1990, pp. 296–307.
6. M. HART: *Project Gutenberg*, `http://www.gutenberg.org/`.
7. D. A. HUFFMAN: *A method for the construction of minimum redundancy codes.* Proc. I.R.E., 40 1951, pp. 1098–1101.
8. J. W. KIM, E. KIM, AND K. PARK: *Fast matching method for DNA sequences*, in Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, no. 4614 in Lecture Notes in Computer Science, 2007, pp. 271–281.
9. S. T. KLEIN: *Skeleton trees for the efficient decoding of Huffman encoded texts.* Inf. Retr., 3(1) 2000, pp. 7–23.
10. S. T. KLEIN AND M. K. BEN-NISSAN: *Accelerating Boyer Moore searches on binary texts*, in Implementation and Application of Automata, 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers, J. Holub and J. Žďárek, eds., vol. 4783 of Lecture Notes in Computer Science, Springer-Verlag Berlin, 2007, pp. 130–143.
11. S. T. KLEIN AND D. SHAPIRA: *Pattern matching in Huffman encoded texts.* Inf. Process. Manage., 41(4) 2005, pp. 829–841.
12. D. E. KNUTH, J. H. MORRIS, JR, AND V. R. PRATT: *Fast pattern matching in strings.* 6(1) 1977, pp. 323–350.
13. T. LECROQ: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
14. E. S. SCHWARTZ AND B. KALLICK: *Generating a canonical prefix encoding.* Commun. ACM, 7(3) 1964, pp. 166–169.
15. D. SHAPIRA AND A. DAPTARDAR: *Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts.* Inf. Process. Manage., 42(2) 2006, pp. 429–439.
16. D. SUNDAY: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
17. 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.