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

Domenico Cantone    Simone Faro    Emanuele Giaquinta

Department of Mathematics and Computer Science, University of Catania, Italy
String matching in compressed texts

Compressed matching problem (Amir & Benson, 1992):

- Alphabet $\Sigma$
- Pattern $P$
- Compression system $(\mathcal{E}, \mathcal{D})$
- Encoded Text $\mathcal{E}(T)$

Find all the shifts of $P$ in $T$ using $\mathcal{E}(P)$ and $\mathcal{E}(T)$
A static compression method is characterized by a system \((\mathcal{E}, \mathcal{D})\) of two complementary functions:

- \(\mathcal{E} : \Sigma \rightarrow \{0, 1\}^+\)
  - \(\mathcal{E}(\varepsilon) = \varepsilon\)
  - \(\mathcal{E}(T[1..\ell]) = \mathcal{E}(T[1..\ell - 1]) \cdot \mathcal{E}(T[\ell]), \forall \ell : 1 \leq \ell \leq |T|\)
- \(\mathcal{D}(\mathcal{E}(c)) = c, \forall c \in \Sigma\)
String matching in compressed texts

A static compression method is characterized by a system \((\mathcal{E}, \mathcal{D})\) of two complementary functions:

- \(\mathcal{E} : \Sigma \rightarrow \{0, 1\}^+\)
  - \(\mathcal{E}(\varepsilon) = \varepsilon\)
  - \(\mathcal{E}(T[1 \ldots \ell]) = \mathcal{E}(T[1 \ldots \ell - 1]).\mathcal{E}(T[\ell]), \forall \ell : 1 \leq \ell \leq |T|\)
- \(\mathcal{D}(\mathcal{E}(c)) = c, \forall c \in \Sigma\)

- Prefix property - \(\not\exists c_1, c_2 : \mathcal{E}(c_1) \sqsubseteq \mathcal{E}(c_2)\)
- Canonical Huffman coding
String matching in compressed texts

t : 00

w : 100  

a : 101  

b : 1111  

e : 01  

n : 110

y : 1110  

Problem: false positives, occurrences of $E(P)$ in $E(T)$ which do not correspond to occurrences of $P$ in $T$.

An occurrence of $E(P)$ which does not start on a codeword boundary is a false positive.
String matching in compressed texts

t: 00
e: 01
ten: 0001110
w: 100
twenty: 0010001110001110
a: 101
ten: 0001110
n: 110
ten: 0001110
y: 1110
ten: 0001110
b: 1111
String matching in compressed texts

<table>
<thead>
<tr>
<th>t</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>01</td>
</tr>
<tr>
<td>w</td>
<td>100</td>
</tr>
<tr>
<td>a</td>
<td>101</td>
</tr>
<tr>
<td>n</td>
<td>110</td>
</tr>
<tr>
<td>y</td>
<td>1110</td>
</tr>
<tr>
<td>b</td>
<td>1111</td>
</tr>
</tbody>
</table>

An occurrence of $\mathcal{E}(P)$ which does not start on a codeword boundary is a false positive.

**Problem:** false positives, occurrences of $\mathcal{E}(P)$ in $\mathcal{E}(T)$ which do not correspond to occurrences of $P$ in $T$. 
minimum redundancy codes

A binary prefix code can be represented with an ordered binary tree, whose leaves are labeled with characters in $\Sigma$ and whose edges are labeled by 0 (left) and 1 (right).

The codeword of a given character is the word labeling the branch from the root to the leaf labeled by the same character.
The minimal prefix of a codeword which allows one to unambiguously determine the codeword length corresponds to the minimum depth node in the path from the root induced by the codeword, which is the root of a complete subtree.

The skeleton tree (Klein, 2000) is a pruned canonical Huffman tree whose leaves correspond to minimum 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.
skeleton tree

b: 00
i: 01
d: 1000
t: 1001
a: 1010
r: 1011
l: 1100
c: 1101
g: 11100
k: 11101
u: 11110
e: 11111
Previous work

- **sk-KMP** (Daptardar & Shapira, 2006): modified KMP, prefix alignments always respect codeword boundaries. Decoding uses the skeleton tree. If no prefix matches the border of the current window, the algorithm can skip the remaining bits for the current codeword.
Filtering/verification paradigm

The filtering phase searches the occurrences of $\mathcal{E}(P)$ in $\mathcal{E}(T)$

The verification phase uses an algorithm based on the skeleton tree to verify candidate matches
Suppose that we have found a candidate shift $s$. We need to know if $s$ is codeword aligned.

We maintain an offset $\rho$ pointing to the start of the last window where a verification has taken place.

We update - using the skeleton tree - $\rho$ to a minimal position $\rho^* \geq s$ which is codeword aligned. If $\rho^* = s$, $s$ is a valid shift.
skeleton tree verification

```
Sk-Align(root, t, ρ, s)
1  x ← root, ℓ ← 0
2  while TRUE
3      do 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
6              then ρ ← ρ + KEY(x) − ℓ, ℓ ← 0, x ← root
7                  if ρ ≥ s then break
8              else ρ ← ρ + 1, ℓ ← ℓ + 1
9  return ρ
```
skeleton tree verification

t : 00
e : 01
w : 100  ten  000110
a : 101  twenty  010011100110
n : 110
y : 1110
b : 1111
skeleton tree verification

\[
\begin{align*}
t : & \quad 00 \\
e : & \quad 01 \\
w : & \quad 100 \\
a : & \quad 101 \\
n : & \quad 110 \\
y : & \quad 1110 \\
b : & \quad 1111 \\
\end{align*}
\]

\[
\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{0}1\bar{1}10 j = 3
\]

\[
\text{SK-ALIGN}(\text{root}, t, 0, 3) = 5 \neq 3
\]
skeleton tree verification

\[
\begin{align*}
t & : 00 \\
e & : 01 \\
w & : 100 \\
a & : 101 \\
n & : 110 \\
y & : 1110 \\
b & : 1111 \\
\end{align*}
\]

\[
\begin{align*}
\overline{00100110001110} & \quad j = 3 \\
SK-ALIGN(root, t, 0, 3) & = 5 \neq 3 \\
\overline{00100110001110} & \quad j = 9 \\
SK-ALIGN(root, t, 5, 9) & = 10 \neq 9 \\
\end{align*}
\]
skeleton tree verification

Pros

- Lazy decoding: in the best case (the pattern does not occur) no decoding is performed
- For every codeword the algorithm reads the minimum number only of bits necessary to infer its length

Cons

- The number of bits processed depends on the position of candidate matches
skeleton tree verification

HUFFMAN-MATCHER($P, m, T, n$)

1. Precompute $\text{Globals}(P)$
2. $n \leftarrow |T|$
3. $m \leftarrow |P|$
4. $s \leftarrow 0$
5. $\rho \leftarrow 0$
6. $\text{root} \leftarrow \text{Build-Sk-Tree}(\phi)$
7. for $i \leftarrow 0$ to $n - 1$
8. \hspace{1em} do $j \leftarrow \text{Check_Shift}(s, P, T)$
9. \hspace{2em} if $j = s + m - 1$
10. \hspace{3em} then $\rho \leftarrow \text{Sk-Align}(\text{root}, T, \rho, j)$
11. \hspace{3em} if $\rho = j$ then $\text{Print}(j)$
12. \hspace{1em} $s \leftarrow s + \text{Shift_Increment}(s, P, T, j)$
string matching in binary strings

Adaptation of existing algorithms for binary strings

- **BINARY-HASH-MATCHING** (Lecroq & Faro, 2009)
- **FED** (J. Kim & E. Kim & Park, 2007)
string matching in binary strings

- Scanning $T$ with bit granularity is slow
- $T[0, \ldots, n - 1] \rightarrow B_T[0, \ldots, \lceil n/k \rceil - 1]$, $B_T[i]$ is a block of $k$ bits
- An occurrence of $P$ can start at bit 1, 2, \ldots $k$ of a block
- $P_i \leftarrow P \gg i$
- $P_i[0, \ldots, m + i - 1] \rightarrow Patt_i[0, \ldots, \lceil (m + i)/k \rceil - 1]$
string matching in binary strings

- Scanning $T$ with bit granularity is slow
- $T[0, \ldots, n−1] \rightarrow B_T[0, \ldots, \lceil n/k \rceil − 1]$, $B_T[i]$ is a block of $k$ bits
- An occurrence of $P$ can start at bit $1, 2, \ldots k$ of a block
- $P_i \leftarrow P \gg i$
- $P_i[0, \ldots, m + i − 1] \rightarrow Patt_i[0, \ldots, \lceil (m + i)/k \rceil − 1]$

ent 10001110
twenty 00100011–10001110
String matching in binary strings

- Scanning $T$ with bit granularity is slow
- $T[0, \ldots, n-1] \rightarrow B_T[0, \ldots, \lceil n/k \rceil - 1]$, $B_T[i]$ is a block of $k$ bits
- An occurrence of $P$ can start at bit 1, 2, $\ldots$ $k$ of a block
- $P_i \leftarrow P \gg i$
- $P_i[0, \ldots, m+i-1] \rightarrow Patt_i[0, \ldots, \lceil (m+i)/k \rceil - 1]$

ent 10001110

twenty 00100011–10001110

$P_2$ 00100011–10000000
string matching in binary strings

\[ \mathcal{E}(P) = 110010110010110010110 \]

(A) \( \textit{Patt} \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\textcolor{gray}{11001011}</td>
<td>\textcolor{gray}{00101100}</td>
<td>\textcolor{gray}{10110000}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01100101</td>
<td>10010110</td>
<td>01011000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00110010</td>
<td>11001011</td>
<td>00101100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00011001</td>
<td>01100101</td>
<td>\textcolor{gray}{10010110}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00001100</td>
<td>10110010</td>
<td>11001011</td>
<td>00000000</td>
</tr>
<tr>
<td>5</td>
<td>00000110</td>
<td>01010011</td>
<td>01100101</td>
<td>10000000</td>
</tr>
<tr>
<td>6</td>
<td>00000011</td>
<td>00101100</td>
<td>10110010</td>
<td>11000000</td>
</tr>
<tr>
<td>7</td>
<td>00000001</td>
<td>10010110</td>
<td>01011001</td>
<td>01100000</td>
</tr>
</tbody>
</table>

(B) \( \textit{Mask} \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\textcolor{gray}{11111111}</td>
<td>\textcolor{gray}{11111111}</td>
<td>\textcolor{gray}{11111000}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01111111</td>
<td>11111111</td>
<td>11111100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00111111</td>
<td>11111111</td>
<td>11111100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00011111</td>
<td>11111111</td>
<td>\textcolor{gray}{11111111}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00001111</td>
<td>11111111</td>
<td>11111111</td>
<td>10000000</td>
</tr>
<tr>
<td>5</td>
<td>00000111</td>
<td>11111111</td>
<td>11111111</td>
<td>11000000</td>
</tr>
<tr>
<td>6</td>
<td>00000011</td>
<td>11111111</td>
<td>11111111</td>
<td>11100000</td>
</tr>
<tr>
<td>7</td>
<td>00000001</td>
<td>11111111</td>
<td>11111111</td>
<td>11110000</td>
</tr>
</tbody>
</table>
The pattern $\mathcal{E}(P)$ is aligned with the $s$-th bit of the text if

$$Patt_i[h] = B_T[j + h] \& Mask_i[h], \text{ for } h = 0, 1, ..., m_i - 1$$

where $j = \lfloor s/k \rfloor$, $i = s \mod k$, $m_i = \lceil (m + i)/k \rceil$
Bad character rule does not work well with binary alphabets,
\[ bc(c) \rightarrow 1, \forall c \in \Sigma \]

Super alphabet: \( 2 \rightarrow 2^k \)

\( P \) is logically divided into \( m - k \) overlapping grams
Shift with bit granularity

\[ H_s(B) = \min \left( \{ 0 \leq u < m \mid p[m - u - k \ldots m - u - 1] \supseteq B \} \cup \{ m \} \right), \quad 0 \leq B < 2^k \]
Shift with bit granularity

\[ H_s(B) = \min \left( \{0 \leq u < m \mid p[m - u - k .. m - u - 1] \supseteq B\} \cup \{m\} \right), \quad 0 \leq B < 2^k \]

- if \( H_s[B] = 0 \), the alignment \( (s - m + 1) \mod k \) is checked, where \( s \) is the current position in \( T \) in bits
Shift with byte granularity

\[ \delta_i(c) = \min(\{m_i - 2 + 1\} \cup \{m_i - 2 + 1 - k \mid Patt_i[k] = c \text{ and } 1 \leq k \leq m_i - 2\}) \]

\[ \delta(c) = \min\{\delta_i(c), 0 \leq i < k\} \]
Shift with byte granularity

\[ \delta_i(c) = \min(\{m_i - 2 + 1\} \cup \{m_i - 2 + 1 - k \mid Patt_i[k] = c \text{ and } 1 \leq k \leq m_i - 2\}) \]

\[ \delta(c) = \min\{\delta_i(c), 0 \leq i < k\} \]

- hash table \( \lambda \) to find candidate alignments, each entry in the table pointing to a linked list of patterns
- pattern \( Patt_i \) is inserted into the slot with hash value \( Patt_i[m_i - 2] \)
Complexity

- $O(\lceil m/k \rceil n)$ worst case time complexity
- $O(m + 2^k)$ space complexity

$k \leq 8$ works well for a single pattern, space overhead negligible
Experimental results

- Implementation in C, compiled with gcc 4, run on PowerPC G4 1.5 GHz
- Natural language texts
- Set of 100 patterns of fixed length $m \in \{4, 8, 16, 32, 64, 128, 256\}$, randomly extracted from the text
- Comparison between the following algorithms:
  - Huffman-Kmp
  - Huffman-Hash-Matching
  - Huffman-Fed
  - Decompress and Search with 3-Hash algorithm
Experimental results - running times

![Graph showing experimental results for running times with HKMP, HHM, HFED, and D&S algorithms. The x-axis represents the variable m ranging from 0 to 250, and the y-axis represents time ranging from 0 to 600. The graph compares the performance of each algorithm over the range of m values.]
Experimental results - processed bits

![Graph showing processed bits vs. m for HKMP, HHM, HFED]
Our generic algorithm can skip many bits when decoding

Sublinear on average when used with Boyer-Moore like algorithms

Fast, especially when the pattern frequency is low