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 Σ
- 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 $\mathcal{E}(T)$

String matching in compressed texts

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

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

String matching in compressed texts

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

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

- Prefix property - $\nexists c_1, c_2 : \mathcal{E}(c_1) \sqsubseteq \mathcal{E}(c_2)$
- Canonical Huffman coding

String matching in compressed texts

t :	00
e :	01
w :	100
a :	101
n :	110
y :	1110
b :	1111

`ten`	$\bar{0}0\bar{0}1\bar{1}10$
`twenty`	$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$

String matching in compressed texts

t :	00
e :	01
w :	100
a :	101
n :	110
y :	1110
b :	1111

$$
\begin{array}{ll}
\texttt{ten} & \bar{0}0\bar{0}1\bar{1}10 \\
\texttt{twenty} & \bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110 \\
\texttt{ten} & \bar{0}0\bar{0}1\bar{1}10 \\
\texttt{ten} & \bar{0}0\bar{0}1\bar{1}10
\end{array}
$$

String matching in compressed texts

t :	00
e :	01
w :	100
a :	101
n :	110
y :	1110
b :	1111

ten	$\bar{0}0\bar{0}1\bar{1}10$
twenty	$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$
ten	$\bar{0}0\bar{0}1\bar{1}10$
ten	$\bar{0}0\bar{0}1\bar{1}10$

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

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

minimum redundancy codes

A binary prefix code can be represented with an ordered binary tree, whose leaves are labeled with characters in Σ 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.

skeleton tree

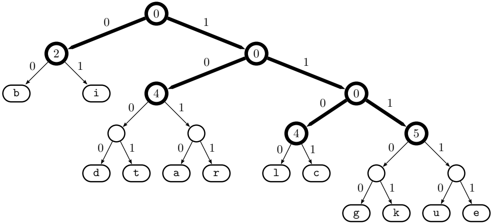
The minimal prefix of a codeword which allows one to unambigously determine the codeword length corresponds to the minimum depth node in the path from the root induced by the codeword, which is the root of a complete subtree.

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.

skeleton tree verification

- 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

skeleton tree verification

Suppose that we have found a candidate shift s. We need to know if s is codeword aligned.

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

We update - using the skeleton tree - ρ 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)
x ← root, ℓ ← 0
while TRUE
      do B ← B_t[⌊ρ / k⌋] ≪ (ρ mod k)
         if B < 2^(k−1) then x ← Left(x) else x ← Right(x)
         if Key(x) ≠ 0
            then ρ ← ρ + Key(x) − ℓ, ℓ ← 0, x ← root
                 if ρ ≥ s then break
            else ρ ← ρ + 1, ℓ ← ℓ + 1
return ρ
```

skeleton tree verification

t :	00
e :	01
w :	100
a :	101
n :	110
y :	1110
b :	1111

ten	$\bar{0}0\bar{0}1\bar{1}10$
twenty	$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$

skeleton tree verification

t :	00		
e :	01		
w :	100		
a :	101	ten	$\bar{0}0\bar{0}1\bar{1}10$
n :	110	twenty	$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$
y :	1110		
b :	1111		

$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$ $j = 3$

$\mathrm{SK\text{-}ALIGN}(root, t, 0, 3) = 5 \neq 3$

skeleton tree verification

t :	00
e :	01
w :	100
a :	101
n :	110
y :	1110
b :	1111

ten	$\bar{0}0\bar{0}1\bar{1}10$
twenty	$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$

$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$ $j = 3$

$\mathrm{SK\text{-}ALIGN}(root, t, 0, 3) = 5 \neq 3$

$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$ $j = 9$

$\mathrm{SK\text{-}ALIGN}(root, t, 5, 9) = 10 \neq 9$

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)
PRECOMPUTE_GLOBALS(P)
n ← |T|
m ← |P|
s ← 0
ρ ← 0
root ← BUILD-SK-TREE(ϕ)
for i ← 0 to n − 1
        do j ← CHECK_SHIFT(s, P, T)
            if j = s + m − 1
               then ρ ← SK-ALIGN(root, T, ρ, j)
                       if ρ = j then PRINT(j)
            s ← s + 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
- $T[0,\dots,n-1] \rightarrow B_T[0,\dots,\lceil n/k \rceil - 1]$, $B_T[i]$ is a block of k bits
- An occurrence of P can start at bit $1, 2, \dots k$ of a block
- $P_i \leftarrow P \gg i$
- $P_i[0,\dots,m+i-1] \rightarrow Patt_i[0,\dots,\lceil (m+i)/k \rceil - 1]$

```
ent       10001110
twenty    00100011-10001110
P_2       00100011-10000000
```

string matching in binary strings

$\mathcal{E}(P) = 110010110010110010110$

(A) *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

(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

string matching in binary strings

The pattern $\mathcal{E}(P)$ is aligned with the s-th bit of the text if

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

where $j = \lfloor s/k \rfloor$, $i = s \mod k, m_i = \lceil (m + i)/k \rceil$

string matching in binary strings

- 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

BINARY-HASH-MATCHING

Shift with bit granularity

$$Hs(B) = \min\left(\{0 \leq u < m \mid p[m-u-k \mathinner{..} m-u-1] \sqsupseteq B\} \cup \{m\}\right),\ 0 \leq B < 2^k$$

BINARY-HASH-MATCHING

Shift with bit granularity

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

- if $Hs[B] = 0$, 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 λ 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

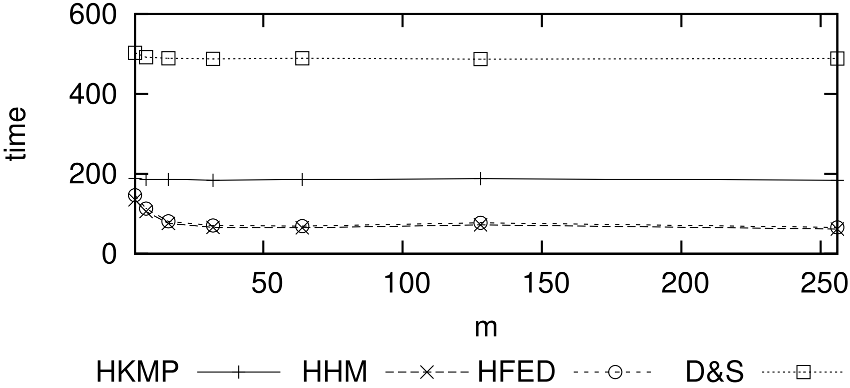
- $\mathcal{O}(\lceil m/k \rceil n)$ worst case time complexity
- $\mathcal{O}(m + 2^k)$ space complexity

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

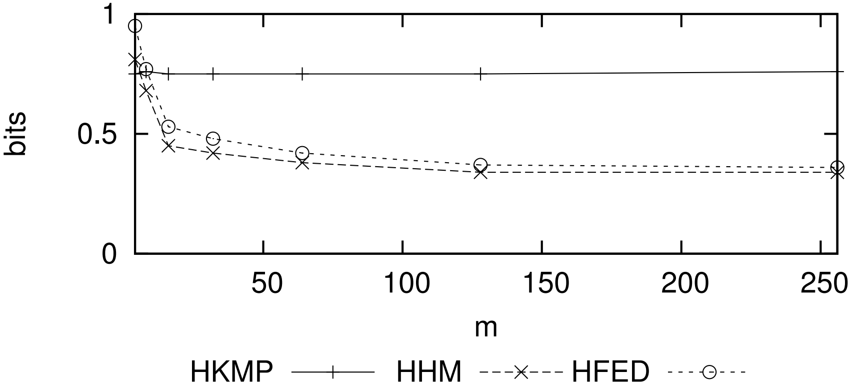
Experimental results

- 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



Experimental results - processed bits



Conclusions

- 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