Guaranteed Synchronization of Huffman Codes with Known Position of Decoder

Marek Biskup, Wojciech Plandowski

University of Warsaw, Poland

Data Compression Conference '09, 2009-03-16
Agenda

- Synchronization of Huffman Codes
- Guaranteed synchronization
- New method: Guaranteed synchronization of Huffman codes with known start position of decoder
- Performance evaluation + applications
Huffman Codes

- Each letter has a corresponding binary string (its codeword)
- The codewords form a proper binary tree
- The depth of a letter depends on its probability
- The code is uniquely decodable

\[
\begin{align*}
a & \rightarrow \ 00 \\
b & \rightarrow \ 01 \\
c & \rightarrow \ 10 \\
d & \rightarrow \ 110 \\
e & \rightarrow \ 111 \\
\end{align*}
\]
Codeword alignment

- Encoded message
Codeword alignment

- Encoded message

- Decoder's perspective:
Codeword alignment

- Encoded message

- Decoder's perspective:

Decode from here
Codeword alignment

- Encoded message

- Decoded message

- Decoder's perspective:

  Decode from here
Codeword alignment

- Encoded message
- Decoded message
- Decoder's perspective:

Resynchronization

Decode from here
Codeword alignment

- Encoded message
- Decoded message
- Decoder's perspective:

Synchronization delay

Resynchronization

Decode from here
Misalignment

- Reasons:
  - Decoding a fragment
  - Parallel decoding
  - Bit corruption

- Consequences:
  - (part of) message is decoded incorrectly

- Good news
  - Resynchronization occurs quite quickly
  - After resynchronization the rest is decoded correctly
Problems

- Synchronization is only statistical
  - no upper bound on the synchronization delay

- Example:
Guaranteed synchronization

- **Goal:**
  - limit the synchronization delay to $< L$ bits
Guaranteed synchronization

- **Goal:**
  - limit the synchronization delay to \(< L\) bits

- **Simple approaches:**
  - Divide the data into **fixed-size** blocks
Guaranteed synchronization

- **Goal:**
  - limit the synchronization delay to $< L$ bits

- **Simple approaches:**
  - Divide the data into *fixed-size* blocks
  - Insert a synchronizing string in *regular* intervals
Guaranteed synchronization

- **Goal:**
  - limit the synchronization delay to $< L$ bits

- **Simple approaches:**
  - Divide the data into **fixed-size** blocks
  - Insert a synchronizing string in **regular** intervals
Guaranteed synchronization

- Additional goals:
  - keep the code **optimal**
  - exploit **statistical synchronization**

- Realization: M.Biskup, DCC'08: 
  *Guaranteed synchronization of Huffman codes*
  - Second type: arbitrary start position
  - Little redundancy
  - Large computational overhead
Guaranteed synchronization with known start position

- First type:
  - Divide data into blocks
    - The decoder may start only at position $K_i, i \in \mathbb{N}$
  - Equivalently, the decoder always knows the position where it starts
  - But insert the marker only if necessary

- In comparison to Biskup '08:
  - Much faster
  - Even less redundancy
  - But doesn't handle bit insertion/deletion errors
Encoder

- Blocks of length K; encode the first block normally, then encode the **second** block:

E

[Diagram of encoder process]
Encoder

- Blocks of length K; encode the first block normally, then encode the **second** block:

- Check the result of decoding this block

  - Resynchronization:

  - No resynchronization:

  ![Encoder Diagram](image-url)
Encoder

- Blocks of length K; encode the first block normally, then encode the **second** block:

- Check the result of decoding this block

- Resynchronization:

- No resynchronization:
Encoder

- Blocks of length K; encode the first block normally, then encode the second block:

- Check the result of decoding this block

  - Resynchronization:

- No resynchronization:

Leaf string
Encoder

- Blocks of length $K$; encode the first block normally, then encode the **second** block:

- Check the result of decoding this block
  - Resynchronization:

  ![Encoder Diagram](image)

  ![Resynchronization Diagram](image)

- No resynchronization:

  ![No Resynchronization Diagram](image)

  ![Leaf String Diagram](image)
Encoder algorithm (simple!)

Algorithm 1: Encoder

1. $i \leftarrow 0$; \hfill \textlangle \ i \text{ is used only for the description of the algorithm} \\
2. $p \leftarrow 0$; \hfill \textlangle \ i \text{ the current position relative to the bit } iK \text{ (the position of } D_{iK}) \\
3. $q \leftarrow \varepsilon$; \hfill \textlangle \ i \text{ the state of decoder } D_{iK} \text{ at the current position} \\
4. \textbf{while } (a \leftarrow M.\textsc{nextSymbol}()) \neq \textsc{null} \textbf{do} \\
5. \quad \text{output } c(a); \hfill \textlangle \ i \text{ encode the symbol normally} \\
6. \quad q \leftarrow \delta^*(q, c(a)); \hfill \textlangle \ i \text{ update the state of } D_{iK} \\
7. \quad p \leftarrow p + |c(a)|; \hfill \textlangle \ i \text{ update the position of } D_{iK} \\
8. \quad \textbf{if } p \geq K \textbf{ then} \hfill \textlangle \ i \text{ make sure } D_{iK} \text{ is in sync and start with } D_{(i+1)K} \\
9. \quad \quad s \leftarrow \text{suffix of } c(a) \text{ of length } p - K; \hfill \textlangle \ i \text{ the overflow from } i^{\text{th}} \text{ K-bit block} \\
10. \quad \quad \text{output } L(q); \hfill \textlangle \ i \text{ synchronize } D_{iK}; \text{ recall that } L(\varepsilon) = \varepsilon \\
11. \quad \quad q \leftarrow \delta^*(\varepsilon, sL(q)); \hfill \textlangle \ i \text{ the state of } D_{(i+1)K} \text{ at the current position} \\
12. \quad \quad p \leftarrow |sL(q)|; \hfill \textlangle \ i \text{ the number of bits processed by } D_{(i+1)K} \\
13. \quad \quad i++; \hfill \textlangle \ i \text{ proceed to the next K-bit block}
Decoder

- Decoder the first block (K bits) normally
- Decode the second block
- Track the decoder of the second block

Is D' synchronized? **NO**
Decoder

- Decoder the first block (K bits) normally

  \[ \text{E} \]
  \[ \text{D} \]

- Decode the second block

  \[ \text{E} \]
  \[ \text{D} \]

- Track the decoder of the second block

  \[ \text{E} \]
  \[ \text{D} \]
  \[ \text{D}' \]

Leaf string detected by decoder
Evaluation – time

- Time overhead:
  - Silesia Corpus files (~10MB)
  - Two possible implementations

<table>
<thead>
<tr>
<th></th>
<th>Linear</th>
<th>Sublinear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing:</td>
<td>(O(M))</td>
<td>(o(M))</td>
</tr>
<tr>
<td>Preprocessing:</td>
<td>(O(N))</td>
<td>(O(N^2))</td>
</tr>
<tr>
<td>Small code (256)</td>
<td>60%</td>
<td>16%</td>
</tr>
<tr>
<td>Large code (4096)</td>
<td>65%</td>
<td>300%</td>
</tr>
</tbody>
</table>

- Legend: \(M\) – message length \(N\) – code size
Evaluation – redundancy (%)

- Tests on Silesia Corpus (large files)
- Small codes (256 elem.)

<table>
<thead>
<tr>
<th></th>
<th>K=50</th>
<th>K=100</th>
<th>K=200</th>
<th>K=500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple code</td>
<td>0,097</td>
<td>0,018</td>
<td>0,002</td>
<td>3,9E-5</td>
</tr>
<tr>
<td>Canonical code</td>
<td>0,106</td>
<td>0,019</td>
<td>0,002</td>
<td>4,0E-5</td>
</tr>
<tr>
<td>Fixed-order code</td>
<td>0,014</td>
<td>0,002</td>
<td>2,6E-4</td>
<td>6,7E-6</td>
</tr>
</tbody>
</table>

- Large codes (4096 elem.)

<table>
<thead>
<tr>
<th></th>
<th>K=50</th>
<th>K=100</th>
<th>K=200</th>
<th>K=500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple code</td>
<td>0,321</td>
<td>0,094</td>
<td>0,017</td>
<td>4,3E-4</td>
</tr>
<tr>
<td>Canonical code</td>
<td>0,371</td>
<td>0,089</td>
<td>0,011</td>
<td>1,7E-4</td>
</tr>
<tr>
<td>Fixed-order code</td>
<td>0,038</td>
<td>0,003</td>
<td>9,4E-5</td>
<td>5,0E-6</td>
</tr>
</tbody>
</table>
Application: Parallel Huffman decompression

- Requirements:
  - divide data into blocks
  - more blocks than processors
  - Increase granularity at the end

- Encoder support (to find codeword alignment)
  - But see Klein and Wiseman, 2003

- Methods:
  - WC (Whole Codewords)
  - LogH
  - KSP (Guaranteed Synchronization with Known Start Position)
Division into blocks

- Whole Codewords, – padding

- LogH, – pos. of the first complete codeword

- KSP: – (optional) resynchronization marker
Tests on Jpeg

- Redundancy [%]

<table>
<thead>
<tr>
<th></th>
<th>Q = 1</th>
<th></th>
<th>Q = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>WC</td>
<td>LogH</td>
<td>KSP</td>
</tr>
<tr>
<td>50</td>
<td>10,72</td>
<td>11,07</td>
<td>0,91</td>
</tr>
<tr>
<td>100</td>
<td>4,78</td>
<td>5,24</td>
<td>0,19</td>
</tr>
<tr>
<td>200</td>
<td>2,52</td>
<td>2,56</td>
<td>0,03</td>
</tr>
<tr>
<td>500</td>
<td>0,92</td>
<td>1,01</td>
<td>5E-3</td>
</tr>
<tr>
<td>10000</td>
<td>0,04</td>
<td>0,05</td>
<td>0 bits</td>
</tr>
</tbody>
</table>

- KSP is much better
  - For large blocks any method is good.
Summary

- Modification of Huffman coding
  - the decoder always resynchronizes in L bits
  - start position of the decoder must be known

- Application
  - division of Huffman data into blocks
  - limiting error propagation

- Advantages:
  - little redundancy,
  - reasonable processing overhead
  - simple algorithm