Compression & Huffman Codes

Nelson Padua-Perez
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Compression

**Definition**

- Reduce size of data
  (number of bits needed to represent data)

**Benefits**

- Reduce storage needed
- Reduce transmission cost / latency / bandwidth
Compression Examples

- **Tools**
  - winzip, pkzip, compress, gzip

- **Formats**
  - **Images**
    - .jpg, .gif
  - **Audio**
    - .mp3, .wav
  - **Video**
    - mpeg1 (VCD), mpeg2 (DVD), mpeg4 (Divx)
  - **General**
    - .zip, .gz

Sources of Compressibility

- **Redundancy**
  - Recognize repeating patterns
  - Exploit using
    - Dictionary
    - Variable length encoding

- **Human perception**
  - Less sensitive to some information
  - Can discard less important data
Types of Compression

- **Lossless**
  - Preserves all information
  - Exploits redundancy in data
  - Applied to general data

- **Lossy**
  - May lose some information
  - Exploits redundancy & human perception
  - Applied to audio, image, video

Effectiveness of Compression

- **Metrics**
  - Bits per byte (8 bits)
    - 2 bits / byte ⇒ ¼ original size
    - 8 bits / byte ⇒ no compression
  - Percentage
    - 75% compression ⇒ ¼ original size
Effectiveness of Compression

- Depends on data
  - Random data ⇒ hard
    - Example: 100110100 ⇒ ?
  - Organized data ⇒ easy
    - Example: 111111111 ⇒ 1×10

- Corollary
  - No universally best compression algorithm

Effectiveness of Compression

- Compression is not guaranteed
  - Pigeonhole principle
    - Reduce size 1 bit ⇒ can only store ½ of data
    - Example
      - 000, 001, 010, 011, 100, 101, 110, 111 ⇒ 00, 01, 10, 11
  - If compression is always possible (alternative view)
    1. Compress file (reduce size by 1 bit)
    2. Recompress output
    3. Repeat (until we can store data with 0 bits)
Lossless Compression Techniques

- **LZW (Lempel-Ziv-Welch) compression**
  - Build pattern dictionary
  - Replace patterns with index into dictionary

- **Burrows-Wheeler transform**
  - Block sort data to improve compression

- **Run length encoding**
  - Find & compress repetitive sequences

- **Huffman code**
  - Use variable length codes based on frequency

---

**Huffman Code**

- **Approach**
  - Variable length encoding of symbols
  - Exploit statistical frequency of symbols
  - Efficient when symbol probabilities vary widely

- **Principle**
  - Use fewer bits to represent frequent symbols
  - Use more bits to represent infrequent symbols

\[ \text{A A B A} \]

\[ \text{A A B A} \]
Huffman Code Example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Dog</th>
<th>Cat</th>
<th>Bird</th>
<th>Fish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1/8</td>
<td>1/4</td>
<td>1/2</td>
<td>1/8</td>
</tr>
<tr>
<td>Original Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Huffman Encoding</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>

**Symbol**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Original Encoding</th>
<th>Huffman Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dog</td>
<td>1/8</td>
<td>00</td>
<td>110</td>
</tr>
<tr>
<td>Cat</td>
<td>1/4</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>Bird</td>
<td>1/2</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Fish</td>
<td>1/8</td>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

**Expected size**

- Original ⇒ \(\frac{1}{8} \times 2 + \frac{1}{4} \times 2 + \frac{1}{2} \times 2 + \frac{1}{8} \times 2 = 2 \text{ bits / symbol}\)
- Huffman ⇒ \(\frac{1}{8} \times 3 + \frac{1}{4} \times 2 + \frac{1}{2} \times 1 + \frac{1}{8} \times 3 = 1.75 \text{ bits / symbol}\)

Huffman Code Data Structures

- **Binary (Huffman) tree**
  - Represents Huffman code
  - Edge ⇒ code (0 or 1)
  - Leaf ⇒ symbol
  - Path to leaf ⇒ encoding
  - Example
    - A = “11”, H = “10”, C = “0”

- **Priority queue**
  - To efficiently build binary tree
Huffman Code Algorithm Overview

Encoding
1. Calculate frequency of symbols in file
2. Create binary tree representing “best” encoding
3. Use binary tree to encode compressed file
   - For each symbol, output path from root to leaf
   - Size of encoding = length of path
4. Save binary tree

Huffman Code – Creating Tree

Algorithm
1. Place each symbol in leaf
   - Weight of leaf = symbol frequency
2. Select two trees L and R (initially leafs)
   - Such that L, R have lowest frequencies in tree
3. Create new (internal) node
   - Left child ⇒ L
   - Right child ⇒ R
   - New frequency ⇒ frequency( L ) + frequency( R )
4. Repeat until all nodes merged into one tree
Huffman Tree Construction 1

A 3
C 5
E 8
H 2
I 7

Huffman Tree Construction 2

A 3
H 2
C 5
E 8
I 7
Huffman Tree Construction 3

Huffman Tree Construction 4
Huffman Tree Construction 5

Huffman Coding Example

- Huffman code
  - E = 01
  - I = 00
  - C = 10
  - A = 111
  - H = 110

- Input
  - ACE

- Output
  - (111)(10)(01) = 111001
Huffman Code Algorithm Overview

Decoding
1. Read compressed file & binary tree
2. Use binary tree to decode file
   - Follow path from root to leaf

Huffman Decoding 1
Huffman Decoding 2

Huffman Decoding 3
Huffman Decoding 4

Huffman Decoding 5
Huffman Decoding 6

Huffman Decoding 7
Huffman Code Properties

- **Prefix code**
  - No code is a prefix of another code
  - Example
    - $\text{Huffman(“dog”) } \Rightarrow ab$
    - $\text{Huffman(“cat”) } \Rightarrow abc$  // not legal prefix code
  - Can stop as soon as complete code found
  - No need for end-of-code marker

- **Nondeterministic**
  - Multiple Huffman coding possible for same input
  - If more than two trees with same minimal weight

Huffman Code Properties

- **Greedy algorithm**
  - Chooses best local solution at each step
  - Combines 2 trees with lowest frequency

- **Still yields overall best solution**
  - Optimal prefix code
  - Based on statistical frequency

- **Better compression possible (depends on data)**
  - Using other approaches (e.g., pattern dictionary)