CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Compression & Huffman Code

Department of Computer Science
University of Maryland, College Park
Overview

- Compression
  - Examples
  - Sources
  - Types
  - Effectiveness

- Huffman Code
  - Properties
  - Huffman tree (encoding)
  - Decoding
Compression

• **Definition**
  • Reduce size of data
    (number of bits needed to represent data)

• **Benefits**
  • Reduce storage needed
  • Reduce transmission cost / latency / bandwidth
Compression Examples

- **Formats**
  - **General**
    - .zip, .rar
  - **Images**
    - .jpg, .gif
  - **Audio**
    - .mp3, .wmv
  - **Video**
    - .mpg, .mov
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
    - Some lossless audio formats (e.g., FLAC)
- **Lossy**
  - May lose some information
  - Exploits *redundancy & human perception*
  - Applied to audio, image, video, multimedia
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: 1001110100 ⇒ ?
  • Organized data ⇒ easy
    • Example: 1111111111 ⇒ 1×10

• Corollary
  • No universally best compression algorithm
Effectiveness of Compression

- Lossless Compression is not guaranteed
  - Pigeonhole principle
    - Reduce size 1 bit $\Rightarrow$ can only store $\frac{1}{2}$ of data
  - Example
    - 000, 001, 010, 011, 100, 101, 110, 111 $\Rightarrow$ 00, 01, 10, 11
- If compression is always possible (alternative view)
  - Compress file (reduce size by 1 bit)
  - Recompress output
  - 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
- **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
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>Original Encoding Length</td>
<td>2 bits</td>
<td>2 bits</td>
<td>2 bits</td>
<td>2 bits</td>
</tr>
<tr>
<td>Huffman Encoding</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>Huffman Encoding Length</td>
<td>3 bits</td>
<td>2 bits</td>
<td>1 bit</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

• Expected size
  • Original  ⇒ 1/8×2 + 1/4×2 + 1/2×2 + 1/8×2 = 2 bits / symbol
  • Huffman  ⇒ 1/8×3 + 1/4×2 + 1/2×1 + 1/8×3 = 1.75 bits / symbol
Huffman Code Data Structures

- **Binary (Huffman) tree**
  - Represents Huffman code
  - Edge $\Rightarrow$ code (0 or 1)
  - Leaf $\Rightarrow$ symbol
  - Path to leaf $\Rightarrow$ encoding
  - Example
    - $A = \text{"11"}$, $H = \text{"10"}$, $C = \text{"0"}$

- **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 leaves)
     • 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

2 trees with lowest frequency
Huffman Tree Construction 2

2 trees with lowest frequency
Huffman Tree Construction 3

2 trees with lowest frequency
Huffman Tree Construction

2 trees with lowest frequency
Huffman Tree Construction 5

A 3
H 2
C 5
E 8
I 7
5
10
15
25

E = 01
I = 00
C = 10
A = 111
H = 110

Huffman code for each leaf
Huffman Coding Example

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

- Input
  
  - ACE  

- Output
  
  - (111)(10)(01) = 1111001
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

1111001
Huffman Decoding 2

1111001
Huffman Decoding 3

1111001
Huffman Decoding 4

A 3
H 2
C 5
E 8
I 7

1111001
Huffman Decoding 5

1111001

AC
Huffman Decoding 6

1111001

A (3)
H (2)
C (5)
E (8)
I (7)

AC
Huffman Decoding 7

A 3
H 2
C 5
E 8
I 7

1111001

ACE
Huffman Code Properties

- **Prefix code**
  - No code is a prefix of another code
  - Example
    - Huffman("dog") \(\Rightarrow 01\)
    - Huffman("cat") \(\Rightarrow 011\) // 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)