Digital Communications III (ECE 154C)
Introduction to Coding and Information Theory

Tara Javidi
These lecture notes were originally developed by late Prof. J. K. Wolf.
UC San Diego

Spring 2014
Noiseless Source Coding Continued
Non-binary Huffman Coding

- The objective is to create a *Huffman Code* where the code words are from an alphabet with $n$ letter is to:

  1. Order probabilities high to low (perhaps with an extra symbol with probability 0)
  2. Combine $n$ least likely probabilities. Add them and re-order.
  3. End up with $n$ symbols (i.e. probabilities)!!

Example 1: A source with alphabet \{A, B, C, D, E\} and probabilities (.5, .3, .1, .08, .02) coded into ternary stream $n = 3$:
Non-binary Huffman Coding

Example 2: \( n = 3 \) \( \{A, B, C, D\} \)
\[ (p_1, p_2, p_3, p_4) = (.5, .3, .1, .1) \]
Example 2: $n = 3$ \{A, B, C, D\}

$(p_1, p_2, p_3, p_4) = (.5, .3, .1, .1)$

$\overline{L}_1 = 1.5$ \text{ SUBOPTIMAL}

$\overline{L}_1 = 1.2$ \text{ OPTIMAL}
Non-binary Huffman Coding

Example 2: \( n = 3 \) \( \{ A, B, C, D \} \)
\[ (p_1, p_2, p_3, p_4) = (0.5, 0.3, 0.1, 0.1) \]

- Sometimes one has to add *Phantom Source Symbols* with 0 probability in order to make a *Non-Binary Huffman Code*.
- How many?
  - If one starts with \( M \) source symbols and one combines the \( n \) least likely into one symbol, one is left with \( M - (n - 1) \) symbols.
  - After doing this \( \alpha \) times, one is left with \( M - \alpha (n - 1) \) symbols.
  - But at the end we must be left with \( n \) symbols. If this is not the case, we must add *Phantom Symbols*.

- Add \( D \) *Phantom Symbols* to insure that
\[ M + D - \alpha (n - 1) = n \text{ or } (M + D) = \alpha' (n - 1) + 1 \]
### Non-binary Huffman Coding

#### Examples:

<table>
<thead>
<tr>
<th>$n = 3$</th>
<th>$n = 4$</th>
<th>$n = 5$</th>
<th>$n = 6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>D</td>
<td>M</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Note:

- $M + D - 1$ must be divisible by $n - 1$.

  **Ex:** $n = 3 \Rightarrow M + D - 1$ must be even

- $D \leq n - 2$
Beyond Huffman Codes
## Run Length Codes for Fax (B/W)

### Source Coding

**Beyond Huffman**
- Fax
- Tunstall Code
- Tunstall Code
- Tunstall Code
- Tunstall Code
- Tunstall Code

**Lempel-Ziv**
Previously we only considered the situation where we encoded $N$ source symbols into variable length code sequences for a fixed value of $N$. We could call this “fixed length to variable length” encoding. But another possibility exists. We could encode variable length source sequences into fixed or variable length code words.

**Example 1:** Consider the DMS source $\{A, B\}$ with probabilities $(.9, .1)$ and the following code book

<table>
<thead>
<tr>
<th>Source Sequences</th>
<th>Codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>00</td>
</tr>
<tr>
<td>$AB$</td>
<td>01</td>
</tr>
<tr>
<td>$AAB$</td>
<td>10</td>
</tr>
<tr>
<td>$AAA$</td>
<td>11</td>
</tr>
</tbody>
</table>

Average length of source phrase

$$= 1 \times .1 + 2 \times .09 + 3 \times (.081 + .70) = 2.75$$

Average # of code symbols/source symbols $= \frac{2}{2.71} = 0.738$
Tunstall Codes

**Tunstall Codes** are U.D. **Variable**- to **Fixed**- length codes with binary code words

**Basic Idea** – Encode into binary code words of fixed length $L$, make $2^L$ source phrases that are as nearly equally probable as we can. We do this by making the source phrases as leaves of a tree and always splitting the leaf with the highest probability.
## Tunstall Codes

### Example 2: \((A, B, C, D)\) with \((p_1, p_2, p_3, p_4) = (0.5, 0.3, 0.1, 0.1)\)

<table>
<thead>
<tr>
<th>Source Symbol \ Source Symbol Code word</th>
<th>Source Symbol \ Code word</th>
</tr>
</thead>
<tbody>
<tr>
<td>D \ 0000</td>
<td>C \ 0001</td>
</tr>
<tr>
<td>BB \ 0010</td>
<td>AB \ 1001</td>
</tr>
<tr>
<td>BC \ 0011</td>
<td>AC \ 1010</td>
</tr>
<tr>
<td>BD \ 0100</td>
<td>AD \ 1011</td>
</tr>
<tr>
<td>BAA \ 0101</td>
<td>AAA \ 1100</td>
</tr>
<tr>
<td>BAB \ 0110</td>
<td>AAB \ 1101</td>
</tr>
<tr>
<td>BAC \ 0111</td>
<td>AAC \ 1110</td>
</tr>
<tr>
<td>BAD \ 1000</td>
<td>AAD \ 1111</td>
</tr>
</tbody>
</table>

Average length of source phrase = Sum of probabilities of internal nodes = \(1 + 0.5 + 0.3 + 0.25 + 0.15 = 2.2\)

Average number of code symbols/source symbols = \(4/2.2 = 1.82\)
Improved Tunstall Coding

- Since the phrases are not equally probable, one can use a *Huffman Code* on the phrases.
- The result is encoding a variable number of source symbols into a variable number of code symbols.

**Example 3:** Back to Example 1 with $(A, B)$ with $(.9, .1)$

We have seen that Tunstall alone $I = 2.71$ $Av = \frac{2}{2.71} = .738$

<table>
<thead>
<tr>
<th>Source Phrases</th>
<th>Tunstall Code</th>
<th>Probability</th>
<th>Improved Tunstall (Huffman)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>11</td>
<td>0.729</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
<td>0.1</td>
<td>11</td>
</tr>
<tr>
<td>AB</td>
<td>01</td>
<td>0.09</td>
<td>100</td>
</tr>
<tr>
<td>AAB</td>
<td>10</td>
<td>0.081</td>
<td>101</td>
</tr>
</tbody>
</table>

Average # of code symbols per source symbol

$$= \frac{1+.271+.171}{1+.9+.81} \cdot \frac{1.442}{2.71} = .532$$
Summary of Results for \((A, B) = (0.9, 0.1)\)

All of the following use 4 code words in coding table:

1. Huffman Code, \(N = 2\)
   
   \[
   \begin{align*}
   AA & \rightarrow 0 \\
   AB & \rightarrow 11 \\
   BA & \rightarrow 100 \\
   BB & \rightarrow 101
   \end{align*}
   \]

2. Shannon-Fano Code \(N = 2\)
   
   \[
   \begin{align*}
   AA & \rightarrow 0 \\
   AB & \rightarrow 10 \\
   BA & \rightarrow 110 \\
   BB & \rightarrow 111
   \end{align*}
   \]
Summary of Results for \((A, B) = (.9, .1)\)

1. Tunstall Code

   \[\begin{align*}
   B & \rightarrow 00 \\
   AB & \rightarrow 01 \\
   AAB & \rightarrow 10 \\
   AAA & \rightarrow 11 \\
   \end{align*}\]

2. Tunstall/Huffman

   \[\begin{align*}
   B & \rightarrow 11 \\
   AB & \rightarrow 100 \\
   AAA & \rightarrow 0 \\
   AAB & \rightarrow 101 \\
   \end{align*}\]
Lempel-Ziv Source Coding
Lempel-Ziv Source Coding

- The basic idea is that if we have a dictionary of $2^A$ source phrases (Available at both the encoder and the decoder) in order to encode one of these phrases one needs only $A$ binary digits.
- Normally, a computer stores each symbol as an ASCII character of 8 binary digits. (Actually only 7 are needed)
- Using L-Z encoding, far less than 7 binary digits per symbol are needed. Typically the compression is about 2:1 or 3:1.
- Variants of L-Z codes was the algorithm of the widely used Unix file compression utility `compress` as well as `gzip`. Several other popular compression utilities also used L-Z, or closely related encoding.
- LZ became very widely used when it became part of the GIF image format in 1987. It may also (optionally) be used in TIFF and PDF files.
- There are two versions of L-Z codes. We will only discuss the “window” version.
Lempel-Ziv Source Coding

- In (the window version of) Lempel Ziv, symbols that have already been encoded are stored in a window.
- The encoder then looks at the next symbols to be encoded to find the longest string that is in the window that matches the source symbols to be encoded.
  - If it can’t find the next symbol in the window, it sends a ‘0’ followed by the 8 (or 7) bits of the ASCII character.
  - If it finds a sequence of one or more symbols in the window, it sends a ‘1’ followed by the bit position of the first symbol in the match followed by the length of the match. These latter two quantities are encoded into binary.
  - Then the sequence that was just encoded is put into the window.
Lempel-Ziv Source Coding

Example: Suppose the content of the window is given as

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
T H E _ T H R E E E _ A R E _ I N
```

The next word, assuming it is “THE_”, will be encoded as

```
(1, “15”, ”4”)
```

```
4bits
```

And then the windows content will be updated as

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
T H R E E E _ A R E _ I N T H E _
**Lempel-Ziv Source Coding**

**Example:** Encode the text

"MY_MY_WHAT_A_HAT_IS_THAT"

16 BIT WINDOW

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- M (0, M) 9
- M (0, Y) 9
- M (0, A) 9
- M (0, T) 9
- M (0, H) 9
- M (0, I) 9
- M (0, S) 9
- M (0, 1) 6
- M (0, 2) 6
- M (0, 3) 6
- M (0, 4) 6
- M (0, 5) 6
- M (0, 6) 6
- M (0, 7) 6
- M (0, 8) 6
- M (0, 9) 6
- M (0, 10) 6
- M (0, 11) 6
- M (0, 12) 6
- M (0, 13) 6
- M (0, 14) 6
- M (0, 15) 6

# of 1 bits: 144
Lempel-Ziv Source Coding

No match \((0, M)\) \rightarrow 1 bit more than needed for a symbol \((1 + 8 = 9)\)

Match \((1, \_, \_, \_)\)

\[ \begin{array}{c}
1 \text{ bits} \\
\downarrow \\
\text{depends on window size}
\end{array} \quad \begin{array}{c}
\downarrow \\
\text{depends on the code used to encode lengths}
\end{array} \]

One really simple code for this purpose might be

\[
\begin{align*}
1 & \rightarrow 0 \\
2 & \rightarrow 10 \\
3 & \rightarrow 110 \\
\vdots &
\end{align*}
\]

What are the advantages/disadvantages of this code?