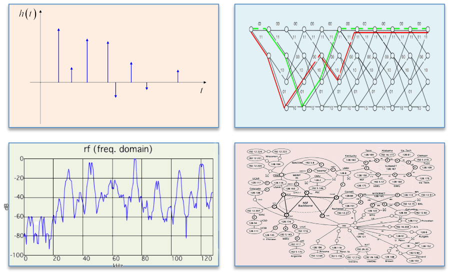
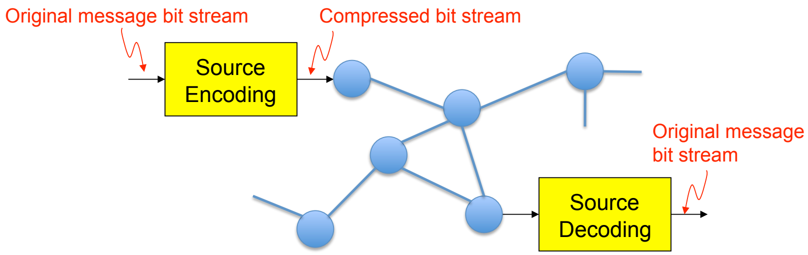


INTRODUCTION TO EECS II
DIGITAL
COMMUNICATION
SYSTEMS

6.02 Spring 2009
Lecture #24

- Information & Entropy
- Variable-length codes: Huffman's algorithm
- Adaptive variable-length codes: LZW

Efficiency via Source Coding



Original message bit stream

Compressed bit stream

Source Encoding

Original message bit stream

Source Decoding

This is an example of an end-to-end protocol – it doesn't involve intermediate nodes in the network.

Idea: Many message streams use a "natural" fixed-length encoding: 7-bit ASCII characters, 8-bit audio samples, 24-bit color pixels. But if we're willing to use variable-length encodings (message symbols of differing lengths) we could assign short encodings to common symbols and longer encodings to other symbols… this should shorten the average length of a message.

Measuring information content

Suppose you're faced with N equally probable choices, and I give you a fact that narrows it down to M choices. Claude Shannon offered the following formula for the information you've received.

$\log_2(N/M)$ *bits* of information

Information is measured in bits (binary digits) which you can interpret as the number of binary digits required to encode the choice(s)

Examples:
- information in one coin flip: $\log_2(2/1) = 1$ bit
- roll of 2 dice: $\log_2(36/1) = 5.2$ bits
- outcome of a Red Sox game: 1 bit
  (well, actually, are both outcomes equally probable?)

When choices aren't equally probable

When the choices have different probabilities (p_i), you get more information when learning of a unlikely choice than when learning of a likely choice

Information from $\text{choice}_i = \log_2(1/p_i)$ bits

We can use this to compute the average information content taking into account all possible choices:

Average information content in a choice $= \Sigma p_i \cdot \log_2(1/p_i)$

This characterization of the information content in learning of a choice is called the *information entropy* or *Shannon's entropy*.

Goal: match data rate to info rate

- Ideally we want to find a way to encode message so that the transmission data rate would match the information content of the message.
- It can be hard to come up with such a code!
  - Transmit results of 1000 flips of unfair coin: p(heads) = p
  - Avg. info in unfair coin flip: $(p)\log_2(1/p) + (1-p)\log_2(1/(1-p))$
  - For p = .999, this evaluates to .0114
  - Goal: encode 1000 flips in 11.4 bits!?
  - What's the code? Hint: can't encode each flip separately
- Morals
  - Effective codes leverage context
    - How to encode Shakespeare sonnets using just 8 bits?
  - Effective codes encode sequences, not single symbols

Example

$choice_i$	p_i	$\log_2(1/p_i)$
"A"	1/3	1.58 bits
"B"	1/2	1 bit
"C"	1/12	3.58 bits
"D"	1/12	3.58 bits

Average information content in a choice
= (.333)(1.58) + (.5)(1) + (2)(.083)(3.58)
= 1.626 bits

Can we find an encoding where transmitting 1000 choices is close to 1626 bits on the average?

The "natural" fixed-length encoding uses two bits for each choice, so transmitting the results of 1000 choices requires 2000 bits.

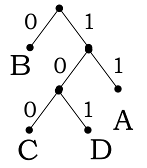
Variable-length encodings

(David Huffman, MIT 1950)

Use shorter bit sequences for high probability choices, longer sequences for less probable choices

$choice_i$	p_i	encoding
"A"	1/3	11
"B"	1/2	0
"C"	1/12	100
"D"	1/12	101

BC A BA D
010011011101



Huffman Decoding Tree

Average information
=(.333)(2)+(.5)(1)+(2)(.083)(3)
= 1.666 bits

Transmitting 1000 choices takes an average of 1666 bits… better but not optimal

To get a more efficient encoding (closer to information content) we need to encode sequences of choices, not just each choice individually.

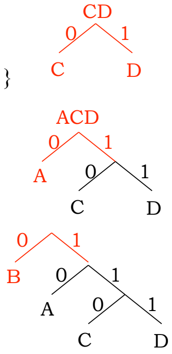
Pairs: 1.646 bits/sym, Triples: 1.637, Quads 1.633, …

Huffman's Coding Algorithm

- Begin with the set S of symbols to be encoded as binary strings, together with the probability p(s) for each symbol s in S. The probabilities sum to 1 and measure the frequencies with which each symbol appears in the input stream. In the example from the previous slide, the initial set S contains the four symbols and their associated probabilities from the table.
- Repeat the following steps until there is only 1 symbol left in S:
  - Choose the two members of S having lowest probabilities. Choose arbitrarily to resolve ties.
  - Remove the selected symbols from S, and create a new node of the decoding tree whose children (sub-nodes) are the symbols you've removed. Label the left branch with a "0", and the right branch with a "1".
  - Add to S a new symbol that represents this new node. Assign this new symbol a probability equal to the sum of the probabilities of the two nodes it replaces.

Huffman Coding Example

- Initially S = { (A, 1/3) (B, 1/2) (C, 1/12) (D, 1/12) }
- First iteration
  - Symbols in S with lowest probabilities: C and D
  - Create new node
  - Add new symbol to S = { (A, 1/3) (B, 1/2) (CD, 1/6) }
- Second iteration
  - Symbols in S with lowest probabilities: A and CD
  - Create new node
  - Add new symbol to S = { (B, 1/2) (ACD, 1/2) }
- Third iteration
  - Symbols in S with lowest probabilities: B and ACD
  - Create new node
  - Add new symbol to S = { (BACD, 1) }
- Done



Huffman Codes - the final word?

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately.
- Huffman codes have the biggest impact on average message length when some symbols are substantially more likely than other symbols.
- You can improve the results by adding encodings for symbol pairs, triples, quads, etc. But the number of possible encodings quickly becomes intractable.
- Symbol probabilities change message-to-message, or even within a single message.
- Can we do adaptive variable-length encoding?

Adaptive Variable-length Codes

- Algorithm first developed by Lempel and Ziv, later improved by Welch. Now commonly referred to as the "LZW Algorithm"
- As message is processed a "string table" is built which maps symbol sequences to a fixed-length code
  - Table size = 2 ^ (size of fixed-length code)
- Note: String table can be reconstructed by the decoder based on information in the encoded stream – the table, while central to the encoding and decoding process, is never transmitted!

LZW Encoding

```
STRING = get input symbol
WHILE there are still input symbols DO
   SYMBOL = get input symbol
   IF STRING + SYMBOL is in the string table THEN
       STRING = STRING + SYMBOL
   ELSE
       output the code for STRING
       add STRING + SYMBOL to the string table
       STRING = SYMBOL
   END
END

output the code for STRING
```

From http://marknelson.us/1989/10/01/lzw-data-compression/

```
lzw('abcabcabcabcabcabcabcabcabcabcabcabc')
```

<...> indicates current value of STRING

```
<> READ a
<a> READ b, ab not in table
  XMIT 'a'  ADD  0: ab
<b> READ c, bc not in table
  XMIT 'b'  ADD  1: bc
<c> READ a, ca not in table
  XMIT 'c'  ADD  2: ca
<a> READ b, ab in table
<ab> READ c, abc not in table
  XMIT [ 0] ADD  3: abc
<c> READ a, ca in table
<ca> READ b, cab not in table
  XMIT [ 2] ADD  4: cab
<b> READ c, bc in table
<bc> READ a, bca not in table
  XMIT [ 1] ADD  5: bca
<a> READ b, ab in table
<ab> READ c, abc in table
<abc> READ a, abca not in table
  XMIT [ 3] ADD  6: abca
<a> READ b, ab in table
<ab> READ c, abc in table
<abc> READ,a,abca in table
<abca> READ b, abcab not in table
  XMIT [ 6] ADD  7: abcab
<b> READ c, bc in table
<bc> READ a, bca in table
<bca> READ b, bcab not in table
  XMIT [ 5] ADD  8: bcab
<b> READ c, bc in table
<bc> READ a, bca in table
<bca> READ b, bcab in table
<bcab> READ c, bcabc not in table
  XMIT [ 8] ADD  9: bcabc
<c> READ a, ca in table
<ca> READ b, cab in table
<cab> READ c, cabc not in table
  XMIT [ 4] ADD 10: cabc
<c> READ a, ca in table
<ca> READ b, cab in table
<cab> READ c, cabc in table
<cabc> READ a, cabca not in table
  XMIT [10] ADD 11: cabca
<a> READ b, ab in table
<ab> READ c, abc in table
<abc> READ a, abca in table
<abca> READ b, abcab in table
<abcab> READ c, abcabc not in table c
  XMIT [ 7] ADD 12: abcabc
<c> READ -end-
  XMIT 'c'
```

LZW Decoding

```
Read CODE
output CODE
STRING = CODE

WHILE there are still codes to receive DO
    Read CODE
    IF CODE is not in the translation table THEN
        ENTRY = STRING + STRING[0]
    ELSE
        ENTRY = get translation of CODE
    END
    output ENTRY
    add STRING+ENTRY[0] to the translation table
    STRING = ENTRY
END
```

```
wzl(['a', 'b', 'c', 0, 2, 1, 3, 6, 5, 8, 4, 10, 7, 'c'])
```

```
READ 'a'  RCV 'a'
READ 'b'  RCV 'b'      ADD 0: ab
READ 'c'  RCV 'c'      ADD 1: bc
READ [0]  RCV 'ab'     ADD 2: ca
READ [2]  RCV 'ca'     ADD 3: abc
READ [1]  RCV 'bc'     ADD 4: cab
READ [3]  RCV 'abc'    ADD 5: bca
READ [6]  RCV 'abca'   ADD 6: abca
READ [5]  RCV 'bca'    ADD 7: abcab
READ [8]  RCV 'bcab'   ADD 8: bcab
READ [4]  RCV 'cab'    ADD 9: bcabc
READ [10] RCV 'cabc'   ADD 10: cabc
READ [7]  RCV 'abcab'  ADD 11: cabca
READ 'c'  RCV 'c'      ADD 12: abcabc
```

String table reconstructed from received codes

Summary

- Source coding: recode message stream to remove redundant information, aka compression. Our goal: match data rate to actual information content.
- Information content from choice$_i$ = $\log_2(1/p_i)$ bits
- Shannon's Entropy: average information content on learning a choice = $\Sigma p_i \cdot \log_2(1/p_i)$
- Huffman's encoding algorithm builds optimal variable-length codes when symbols encoded individually
- LZW algorithm implements adaptive variable-length encoding