Hamming, Huffman, Shanon-Fano Encoding

Compression

- Compression makes it possible for efficient storage and communication of media.
- Compression is essentially the elimination of redundancy which is inherent in most representations of text audio or video.
- Redundancy arises from the fact that most representations of media are not optimized in terms of space.
- For example, using 8-bit ASCII codes to represent characters is convenient but clearly not optimal:
  - It would make sense to give smaller codes to more frequently occurring characters, like vowels and letters, r, t, s, n, etc., and longer codes to less frequently occurring characters like z, x, v, k etc.
  - – This questions is tackled by information theory.

2

Encoding and Compression of Data

- Fax Machines
- ASCII
- Variations on ASCII
  - – min number of bits needed
  - – cost of savings
  - – patterns
  - – modifications

Purpose of Huffman Coding

- Proposed by Dr. David A. Huffman in 1952
  - – *"A Method for the Construction of Minimum Redundancy Codes"*
- Applicable to many forms of data transmission
  - – Our example: text files

The Basic Algorithm

- Huffman coding is a form of statistical coding
- Not all characters occur with the same frequency
- Yet all characters are allocated the same amount of space
  - 1 char = 1 byte, be it A or a

The Basic Algorithm

- Any savings in tailoring codes to frequency of character?
- Code word lengths are no longer fixed like ASCII.
- Code word lengths vary and will be shorter for the more frequently used characters.

The Basic Algorithm

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

Fixed and variable bit widths

- To encode English text, we need 26 lower case letters, 26 upper case letters, and a handful of punctuation
- We can get by with 64 characters (6 bits) in all
- Each character is therefore 6 bits wide
- We can do better, provided:
  - Some characters are more frequent than others
  - Characters may be different bit widths, so that for example, we use only one or two bits, while x uses several
  - We have a way of decoding the bit stream
    - Must tell where each character begins and ends

Example Huffman encoding

- A = 0
  B = 100
  C = 1010
  D = 1011
  R = 11
- ABRACADABRA = 01001101010010110100110
- This is eleven letters in 23 bits
- A fixed-width encoding would require 3 bits for five different letters, or 33 bits for 11 letters

Why it works

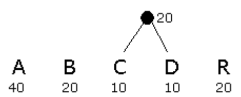
- In this example, A was the most common letter
- In ABRACADABRA:
  - 5 As code for A is 1 bit long
  - 2 Rs code for R is 2 bits long
  - 2 Bs code for B is 3 bits long
  - 1 C code for C is 4 bits long
  - 1 D code for D is 4 bits long

Creating a Huffman encoding

- For each encoding unit (letter, in this example), associate a frequency (number of times it occurs)
- Create a binary tree whose children are the encoding units with the smallest frequencies
  - The frequency of the root is the sum of the frequencies of the leaves
- Repeat this procedure until all the encoding units are in the binary tree

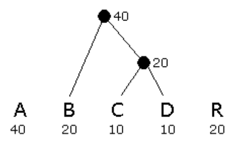
Example, step I

- Assume that relative frequencies are:
  - A: 40
  - B: 20
  - C: 10
  - D: 10
  - R: 20
- Smallest number are 10 and 10 (C and D), so connect those



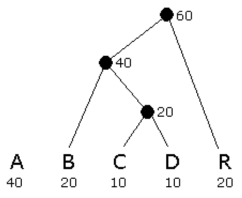
Example, step II

- C and D have already been used, and the new node above them (call it C+D) has value 20
- The smallest values are B, C+D, and R, all of which have value 20
  - – Connect any two of these



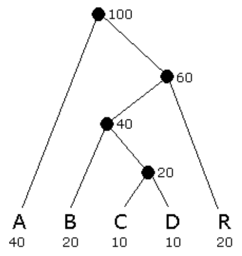
Example, step III

- The smallest values is R, while A and B+C+D all have value 40
- Connect R to either of the others



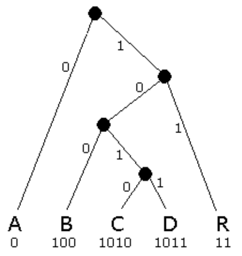
Example, step IV

- Connect the final two nodes



Example, step V

- Assign 0 to left branches, 1 to right branches
- Each encoding is a path from the root



- A = 0
  B = 100
  C = 1010
  D = 1011
  R = 11
- Each path terminates at a leaf

Unique prefix property

- A = 0
  B = 100
  C = 1010
  D = 1011
  R = 11
- No bit string is a prefix of any other bit string
- For example, if we added E=01, then A (0) would be a prefix of E
- Similarly, if we added F=10, then it would be a prefix of three other encodings (B=100, C=1010, and D=1011)

Practical considerations

- It is not practical to create a Huffman encoding for a single short string, such as **ABRACADABRA**
  – To decode it, you would need the code table
  – If you include the code table in the entire message, the whole thing is bigger than just the ASCII message
- Huffman encoding is practical if:
  – The encoded string is large relative to the code table, OR
  – We agree on the code table beforehand
    - For example, it's easy to find a table of letter frequencies for English

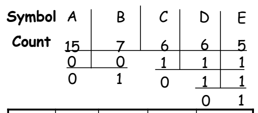
Data compression

- Huffman encoding is a simple example of data compression: representing data in fewer bits than it would otherwise need

The Shannon-Fano Encoding Algorithm

1. **Calculate the frequencies of the list of symbols (organize as a list).**
2. **Sort the list in (decreasing) order of frequencies.**
3. **Divide list into two halfs, with the total freq. *Counts of each half being as close as possible to the other*.**
4. **The upper half is assigned a code of 0 and lower a code of 1.**
5. **Recursively apply steps 3 and 4 to each of the halves, until each symbol has become a corresponding code leaf on a tree.**

<u>Example</u>



Symbol	Count		Info. -log$_2$(p$_i$)	Code	Subtotal # of Bits
A	15	x	1.38	00	30
B	7	x	2.48	01	14
C	6	x	2.70	10	12
D	6	x	2.70	110	18
E	5	x	2.96	111	15
	85.25				89

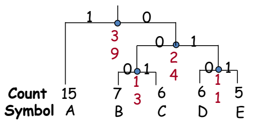
It takes a total of 89 bits to encode 85.25 bits of information

20

The Huffman Algorithm

1. **Initialization: Put all nodes in an OPEN list *L*, keep it sorted at all times (e.g., ABCDE).**
2. **Repeat the following steps until the list *L* has only one node left:**
    1. **From *L* pick two nodes having the lowest frequencies, create a parent node of them.**
    2. **Assign the sum of the children's frequencies to the parent node and insert it into L.**
    3. **Assign code 0, 1 to the two branches of the tree, and delete the children from L.**

Example



Symbol	Count	Info. $-\log_2(p_i)$	Code	Subtotal # of Bits
A	15	x 1.38	1	15
B	7	x 2.48	000	21
C	6	x 2.70	001	18
D	6	x 2.70	010	18
E	5	x 2.96	011	15
	85.25			87

Huffman Alg.: Discussion

Decoding for the above two algorithms is trivial as long as the coding table (the statistics) is sent before the data. There is an overhead for sending this, negligible if the data file is big.

If prior statistics are available and accurate, then Huffman coding is very good.

Number of bits (per symbol) needed for Huffman Coding is:

87 / 39 = 2.23

Number of bits (per symbol)needed for Shannon-Fano Coding is:

89 / 39 = 2.28

Hamming Codes

Historical Background

- Early computer input would frequently contain errors
- 1950 Richard Hamming invented the error correcting algorithm that now bears his name

Uses

- Hamming Codes are still widely used in computing, telecommunication, and other applications.
- Hamming Codes also applied in
  - Data compression
  - Some solutions to the popular puzzle The Hat Game
  - Block Turbo Codes

How Hamming Codes Work

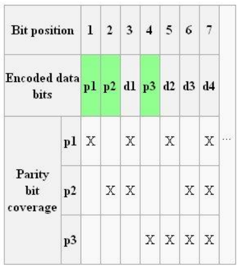
- One parity bit can tell us an error occurred
- Multiple parity bits can also tell us where it occurred
- O(lg(n)) bits needed to detect and correct one bit errors

Hamming (7, 4)

- 7 bits total
  - 4 data bits
  - 3 parity bits
- Can find and correct 1 bit errors

  or
- Can find but not correct 2 bit errors

Hamming Bits Table



Bit position		1	2	3	4	5	6	7	
Encoded data bits		p1	p2	d1	p3	d2	d3	d4	
Parity bit coverage	p1	X		X		X		X	...
	p2		X	X			X	X	
	p3				X	X	X	X	

Example

Byte **1011 0001**

Two data blocks, **1011** and **0001**.

Expand the first block to 7 bits: _ _ **1** _ **0 1 1**.

Bit 1 is 0, because b3+b5+b7 is even.

Bit 2 is 1, b3+b6+b7 is odd.

bit 4 is 0, because b5+b6+b7 is even.

Our 7 bit block is: **0 1 1 0 0 1 1**

Repeat for right block giving **1 1 0 1 0 0 1**

Detecting Errors

0 1 1 0 1 1 1

Re-Check each parity bit

Bits 1 and 4 are incorrect

1 + 4 = 5, so the error occurred in bit 5