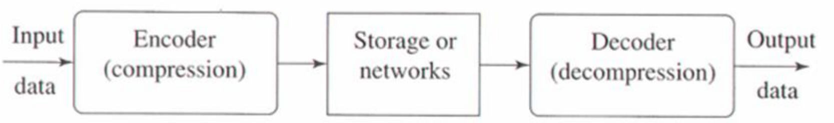
Multimedia Software Systems CS4551

Lossless Compression – Part I

Why Compression?

- Multimedia information has to be stored and transported efficiently.
- Multimedia information is bulky
  - HDTV: 1080 x 1920 x 30 x 12 =745 Mb/s!
- Use technologies with more bandwidth
  - Expensive
  - Research about Hardware
- Find ways to reduce the number of bits to transmit without compromising on the “information content” => Compression

Compression Scheme



A general data compression scheme.

Compression ratio = R/C

- **R =** the total number of bits required to represent data ***before*** compression
- **C =** the total number of bits required to represent data ***after*** compression

Types of Compression

- **Lossless**
  - Does not lose information -the original signal is *perfectly* reconstructed after decompression
  - Produces a variable bit-rate
  - Not guaranteed to actually reduce the data size
- **Lossy**
  - Loses some information -the original signal *is not perfectly* reconstructed after decompression
  - Produces any desired *constant* bit-rate

Lossless Compression

- **Lossless compression techniques ensure no loss of data after compression/decompression.**
- **Idea for lossless coding:**
  - "Translate" each symbol represented by a fixed number of bits into a "*codeword*". *Codewords* may have *different* binary lengths.
  - **Example: You have 4 symbols *(a, b, c, d*). Each in binary may be represented using 2 bits each, but coded using a different number of bits.**
    - *a*(00) -> 000
    - *b*(01) -> 001
    - *c*(10) -> 01
    - *d*(11) -> 1

Average Symbol Length

- Symbol : individual element of information
- Symbol Length *l(i)* = binary length of i^{th} symbol
- *M = total* number of symbols that the source emits and i^{th} symbol has been emitted *m(i)* times (over a certain time T):

$$M = \sum_{i=1}^{i=N} m(i)$$

- Number of bits been emitted:

$$L = \sum_{i=1}^{i=N} m(i)l(i)$$

- Average length per symbol:

$$\bar{L} = \sum_{i=1}^{i=N} \frac{m(i)l(i)}{M} = \sum_{i=1}^{i=N} p(i)l(i)$$

Consider the data "ABBACDAA"

- Symbols: A, B, C, D
- Symbol length: 2 bits per symbol
- Total number of symbols

  M = 8

  = 4As + 2Bs + 1C + 1D

- Number of bits emitted

  16 bits = 8 x 2

  = 4x2+2x2+1x2+1x2

- Average length per symbol

  2 = 2x(4/8) + 2x(2/8) +

  2x(1/8) + 2x(1/8)

Symbol	Code	Number of occurrences (probability)	Bits used by each symbol	
s_1	00	70 (0.7)	$70 \times 2 = 140$	
s_2	01	5 (0.05)	$5 \times 2 = 10$	$\boxed{200}$
s_3	10	20 (0.2)	$20 \times 2 = 40$	
s_4	11	5 (0.05)	$5 \times 2 = 10$	

Symbol	Code	Number of occurrences (probability)	Bits used by each symbol	
s_1	1	70 (0.7)	$70 \times 1 = 70$	
s_2	001	5 (0.05)	$5 \times 3 = 15$	$\boxed{140}$
s_3	01	20 (0.2)	$20 \times 2 = 40$	
s_4	000	5 (0.05)	$5 \times 3 = 15$	

Symbol	Code	Number of occurrences (probability)	Bits used by each symbol	
s_1	0	70 (0.7)	$70 \times 1 = 70$	
s_2	01	5 (0.05)	$5 \times 2 = 10$	$\boxed{110}$
s_3	1	20 (0.2)	$20 \times 1 = 20$	
s_4	10	5 (0.05)	$5 \times 2 = 10$	

Minimum Average Symbol Length

- Main goal is to minimize the average symbol length.
- Basic idea for reducing the average symbol length:
  - assign *Shorter Codewords* to symbols that appear more frequently, *Longer Codewords* to symbols that appear less frequently

Shannon's Information Theorem

- Theorem:
  - "The *Average Binary Symbol Length* of the encoded symbols is always *greater than* or *equal to* the ***entropy H*** of the source" (under the First-order Model or memory-less model)
    - **Memoryless source model:** an information source that is indepen-dently distributed. Namely, the value of the current symbol does not depend on the values of the previously appeared symbols.
- What is the *entropy* of a source of symbols and how is it computed?

Symbol Probability

- Probability *p(i)* of a symbol: number of times it can occur in the transmission (also relative frequency) and is defined as:

$$p(i)=m(i)/M$$

- From Probability Theory we know

$$0 \le p(i) \le 1 \ \& \ \sum_{i=1}^{i=N} p(i) = 1$$

- Average symbol length is defined as

$$\sum_{i=1}^{i=N} (m(i)/M)l(i) = \sum_{i=1}^{i=N} p(i)l(i)$$

Entropy Definition

Entropy is defined as

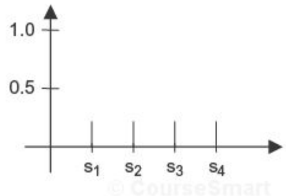
$$H = \sum_{i=1}^{i=N} P(i) \log_2 \frac{1}{P(i)} = -\sum_{i=1}^{i=N} P(i) \log_2 P(i)$$

$\log_2 \frac{1}{P(i)}$ **is the number of bits used to send** i^{th} **message**

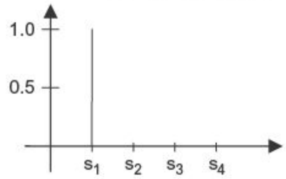
https://www.khanacademy.org/computing/computer-science/informationtheory/moderninfotheory/v/information-entropy

Entropy Examples

$$P_i = \{0.25,\ 0.25,\ 0.25,\ 0.25\}$$
$$H = -(4 \times 0.25 \times \log_2 0.25)$$
$$H = 2$$

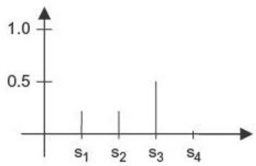


$$P_i = \{1.0,\ 0.0,\ 0.0,\ 0.0\}$$
$$H = -(1 \times \log_2 1)$$
$$H = 0$$

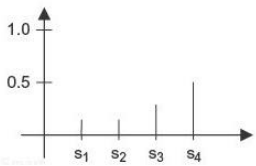


Entropy Examples

$$P_i = \{0.25\ \ 0.25\ \ 0.5\ \ 0.0\}$$
$$H = -(2 \times 0.25 \times \log_2 0.25 + 0.5 \times \log_2 0.5)$$
$$H = 1.5$$

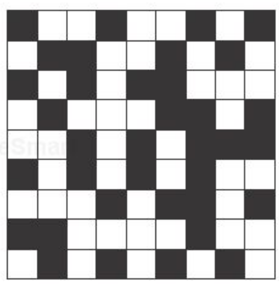


$$P_i = \{0.125\ \ 0.125\ \ 0.25\ \ 0.5\}$$
$$H = -(2 \times 0.125 \times \log_2 0.125 + 0.25 \times \log_2 0.25 + 0.5 \times \log_2 0.5)$$
$$H = 1.75$$



Entropy Examples

P(1) = 1 and P(2) = 0
H = 0



P (1) ≈ 0.5 and P(2) ≈ 0.5
H = 1
the minimum number of bits needed
to represent the two symbols

Entropy

- *Entropy* H quantifies the amount of information (uncertainty/surprises/disorder) contained in the message or the minimum number of bits to encode the symbols.
  - It is a measure of the disorder of a system – the more the entropy, the more the disorder.
- *Entropy* H depends on the probabilities of symbols.
  - Given p_i , probability that symbol s_i will occur in S, $\log_2 (1/p_i)$ indicates the amount of information contained in s_i, which corresponds to the number of bits needed to encode s_i.
  - H is the weighted sum of the information carried by each symbol. Hence, it represents the average amount of information contained per symbol in the source S.

Entropy

- *Entropy* H is always ≥ 0.
- *Entropy* H is *highest* (equal to $\log_2 N$) if all symbols are equally probable.
- *Entropy* H is *small* when some symbols that are much more likely to appear than other symbols.

- For a memory-less source S, the entropy represents the minimum average number of bits required to represent each symbol in S. In other words,
  - It specifies the lower-bound for the average of bits to code each symbol in S.
  - It provides an absolute limit on the shortest possible average length of a lossless compression encoding of the data produced by a source.

Efficiency of the Encoder

- Efficiency of the Encoder

$$\frac{H}{\overline{\overline{L_{encoder}}}}$$

H : entropy of the data

$\overline{L_{encoder}}$: the average symbol length generated by the coder for the data

  - For example, given data with N symbols, if the entropy H of the data is 2 and the average symbol length computed by a compression method C is 2.5, then the efficiency of the encoder C is 2/2.5=0.8 => 80%

Lossless Encoding Methods

- Run-length Encoding
- Repetition Suppression
- Variable Length Coding (Entropy Coding)
  - Shannon-Fano Algorithm
  - Huffman coding
  - Adaptive Huffman coding
- Pattern Substitution : A pattern, which occurs frequently, is replaced by a specific symbol
  - Dictionary based Coding LZW
- Arithmetic Coding

Run-Length Encoding (RLE)

- Sequence of elements, $c_1, c_2 \ldots c_i \ldots$, is mapped to (c_i, l_i) where c_i =symbol and l_i =length of the symbol c_i's run
- For example, given the sequence of symbols

  {1,1,1,3,3,6,6,6,2,2,2,2,3,3,1,4,4}

  the run-length encoding is

  (1,3),(3,2),(6,3),(2,4)(3,2),(1,1),(4,2).
- We can apply this run-length encoding for a bi-level image which has two symbols, 0 and 1, by simply coding the length of each run.
- Two dimensional run-length encoding is also available.

Run-Length Encoding (RLE)

- Performs best in the presence of repetitions or redundancy of symbols.
- Not practical to encode runs of length 1.
- Used as a part of compression standards such as TIFF, BMP, PCX, and JPEG.

Repetition Suppression

- Repetition Suppression
  - Repetitive occurrences of a specific character are replaced by a special flag
  - Eg. ab000000000 -> abψ

Variable Length Coding (VLC)

- VLC generates *variable length codewords* from fixed length symbol.
- VLC is one of the best known entropy coding method.
- Methods of VLC
  - Shannon-Fano Algorithm (top-down)
  - Huffman coding (bottom-up approach)
  - Adaptive Huffman coding

Shannon-Fano Algorithm

- Developed by Shannon at Bell lab and Robert Fano at MIT.

Algorithm:

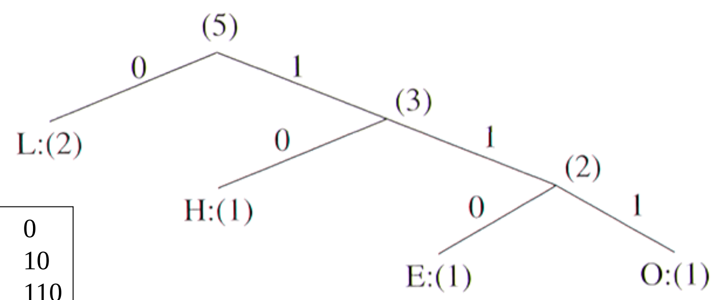
1. **Compute the frequency count of the symbols**
2. **Sort the symbols according to the frequency count of their occurrences**
3. **Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol**

– Encoding step is top-down manner.

– A natural way of implementing the algorithm is to build a binary tree assigning 0 to its left branch and 1 to its right branch

Shannon-Fano Algorithm

- Example: Consider “HELLO”
  - The frequency count: H – 1, E –1, L – 2, O-1
- A Coding tree for HELLO by Shannon-Fano Algorithm



L	0
H	10
E	110
O	111

CSULA CS451 Multimedia Software Systems by Eun-Young Kang

Shannon-Fano Algorithm

- Probabilities of each symbol: *P(H) = 1/5, P(E) = 1/5, P(L) = 2/5, P(O) = 1/5*
- Entropy H

$$H = -\sum_{i=1}^{i=4} P(i)\log_2 P(i) = 1.92$$

- Average length per symbol using fixed length coding

$$\bar{L} = \sum_{i=1}^{i=4} 2P(i) = (2\frac{1}{5})3 + (2\frac{2}{5}) = 2 \text{ bits/symbol}$$

- Symbol length by Shannon_Fano: *l*(L)=1, *l*(H)=2, *l*(E)=3, *l*(O)=3
- Average symbol length with Shannon Fano, $L_{Shannon\_Fano}$ = 1*2/5+2*1/5+3*1/5+3*1/5 = 10/5 =2 bits/symbol

Efficiency of the Encoder

- Efficiency of the Encoder

$$H / \overline{L_{encoder}}$$

- Efficiency with fixed length encoding

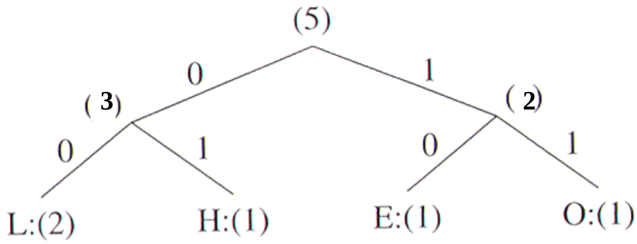
$$H / \overline{L_{fixed-length-encoding}} = 1.92/2 = 0.96 -> 96\%$$

- Efficiency of encoder of the Shanon_Fano coding

$$H / \overline{L_{Shannon\_Fano}} = 1.92/2 = 0.96 -> 96\%$$

Shannon-Fano Algorithm

- Another coding tree for HELLO by Shannon-Fano algorithm



- Shannon-Fano delivers satisfactory coding result but it was outperformed and overtaken by the Huffman coding method.
  - Eg. Consider, {A,B,C,D,E} with the frequency count 15, 7, 6, 6, and 5. Shannon-Fano algorithm needs a total of 89 bits to encode, whereas the Huffman coding needs only 87 bits.

Huffman Coding

- Huffman code assignment procedure is based on the frequency of occurrence of a symbol. It uses a ***binary tree structure*** and the algorithm is as follows.
- Algorithm:
  1. The leaves of the binary tree are associated with the list of probabilities
  2. Take the two smallest probabilities in the list and make the corresponding nodes siblings. Generate an intermediate node as their parent and label the branch from parent to one of the child nodes 1 and the branch from parent to the other child 0.
  3. Replace the probabilities and associated nodes in the list by the single new intermediate node with the sum of the two probabilities. If the list contains only one element, quit. Otherwise, go to step 2.
  4. Codeword formation: Follow the path from the root of the tree to the symbol, and accumulates the labels of all the branches.

Huffman Coding – Example 1

- Let's encode "go go gopher" using Huffman coding.
- 7 symbols = {g, o, p, h, e, r, *space*}
  - 3 bits per symbol using the fixed length coding method
  - P(g)=3/12, P(o)=3/12, P(p)=1/12, P(h)=1/12, P(e)=1/12, P(r)=1/12, P(*space*)=2/12
  - Average length per symbol using fixed length coding

$$\overline{L} = \sum_{i=1}^{i=7} 3P(i) = 3 \text{ bits/symbol}$$

  - Entropy

$$H = -\sum_{i=1}^{i=7} P(i) \log_2 P(i)$$

  - Compute the average symbol length and the coding efficiency

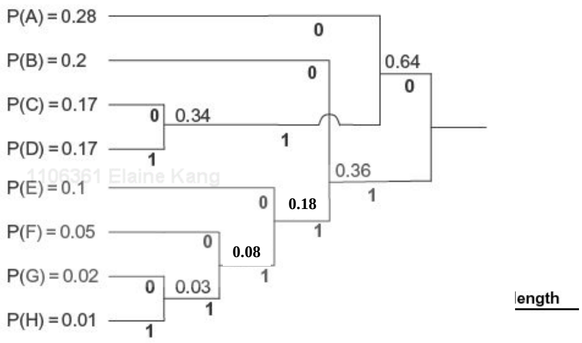
Huffman Coding – Example 1

- Huffman code
  - `g 11`
  - `o 00`
  - `p 010`
  - `h 011`
  - `e 1000`
  - `r 1001`
  - `space 101`

Average Symbol Length = 2.66
Entropy = 2.62
Coding Efficiency = 2.62/2.66 =0.98

Huffman Coding – Example 2

Symbol	Probability
A	0.28
B	0.2
C	0.17
D	0.17
E	0.1
F	0.05
G	0.02
H	0.01



	length
010	3
011	3
110	3
1110	4
11110	5
11111	5

Huffman Coding - Example

Symbol	Probability	Binary code	Code length
A	0.28	000	3
B	0.2	001	3
C	0.17	010	3
D	0.17	011	3
E	0.1	100	3
F	0.05	101	3
G	0.02	110	3
H	0.01	111	3

Average symbol length = 3

Entropy = 2.574

Efficiency = 0.858

Fixed length coding

Symbol	Probability	Huffman code	Code length
A	0.28	00	1
B	0.2	10	3
C	0.17	010	3
D	0.17	011	3
E	0.1	110	3
F	0.05	1110	4
G	0.02	11110	5
H	0.01	11111	5

Average symbol length = 2.63

Entropy = 2.574

Efficiency = 0.9792

Variable length coding

Huffman Coding - Decoding

- To decode the bit stream, the decoder needs the Huffman table
- Decoding the bit stream:
  - With fixed length coding:
    001010001110101001 = bcbgfb
  - With Huffman coding *in the previous slide*:
    11110111111110011 = ghed (variable-length)
  - Both fixed length code and the Huffman code are uniquely decodable.

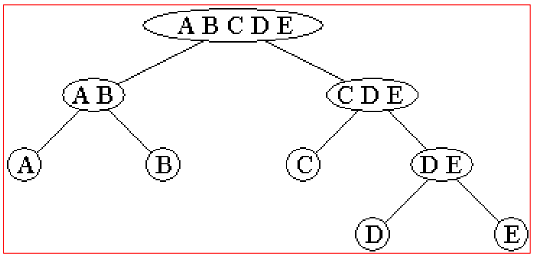
Uniquely Decodable Code

- Decoding the bit stream using the Huffman code table/tree:
  - We can decode the bit stream because it is a ***prefix code*** (no codeword is the prefix of another codeword)
  - Unique prefix property make decoding more efficient because it precludes any ambiguity in decoding and the decoder can immediately produce a symbol without waiting for any more bits to be transmitted.

* Both fixed length code and a **prefix code** are uniquely decodable.

Shanon-Fano vs Huffman

Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

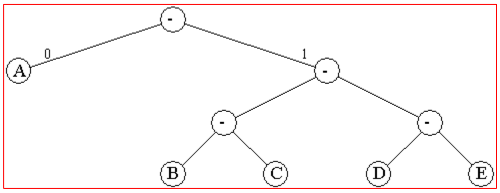


Shannon-Fano

Symbol	Code
A	00
B	01
C	10
D	110
E	111

Shanon-Fano vs Huffman

Symbol	Frequency
A	24
B	12
C	10
D	8
E	8



Huffman

Symbol	Code
A	0
B	100
C	101
D	110
E	111

The Shannon-Fano coding provides a similar result compared with Huffman coding at the best. It will never exceed Huffman coding.

Huffman Code Application - CCITT Group 3 1-D FAX Encoding

- **CCITT *(C*****omité *C*****onsultatif *I*****nternational *T*****éléphonique et *T*****élégraphique*, an organization that sets international communications standards.) – now known as ITU (International Telecommunication Union)
- Group 3 and Group 4 encodings are compression algorithms that are specifically designed for encoding 1-bit image data. Many document and FAX file formats support Group 3 compression, and several, including TIFF, also support Group 4.
- Group 3 encoding was designed specifically for bi-level, black-and-white image data telecommunications. All modern FAX machines and FAX modems support Group 3 facsimile transmissions.

Huffman Code Application - CCITT Group 3 1-D FAX Encoding

- Facsimile: *bilevel* image, 8.5” x11” page scanned (left-to-right) at 200 dpi: 3.74 Mb =(1728 pixels on each line)
- Encoding Process
  - Count each run of white/black pixels
  - Encode the run-lengths with Huffman coding

CSULA CS451 Multimedia Software Systems by Eun-Young Kang

Huffman Code Application - CCITT Group 3 1-D FAX Encoding

- Decompose each run-length n as $n = k \times 64 + m$ (with $0 \leq k < 27$ and $0 \leq m < 64$), and create Huffman tables for both k and m.
  - The standard Huffman table is available in the encoding and decoding side.
  - 92 different codes: 28 groups of pX64 pixels, and 64 short-runs of 0-63 pixels.
  - Note: Probabilities of run-lengths are not the same for black and white pixels
- Special Codewords for end-of-line, end-of-page, synchronization.
  - Each scan line ends with a special EOL (end of line) character consisting of eleven zeros and a 1 (000000000001).
- Average compression ratio on text documents: 20-to-1

Huffman Code Application - CCITT Group 3 1-D FAX Encoding

White: 500 = 7 x 64 + 52 => k = 7, m = 52

Black: 800 = 12 x64 + 32 => k=12, m =32

White: 428 = 6 x 64 + 44 => k = 6, m=44

The line will be encoded using the Huffman codes for k*s*, m*s* and end-of-line code.

CSULA CS451 Multimedia Software Systems by Eun-Young Kang

White run length	Code word	Black run length	Code word
0	00110101	0	0000110111
1	000111	1	010
2	0111	2	11
3	1000	3	10
4	1011	4	011
5	1100	5	0011
6	1110	6	0010
7	1111	7	00011
8	10011	8	000101
9	10100	9	000100
10	00111	10	0000100
11	01000	11	0000101
12	001000	12	0000111
13	000011	13	00000100
14	110100	14	00000111
15	110101	15	000011000
16	101010	16	0000010111
17	101011	17	0000011000
18	0100111	18	0000001000
19	0001100	19	00001100111
20	0001000	20	00001101000
21	0010111	21	00001101100
22	0000011	22	00000110111
23	0000100	23	00000101000
24	0101000	24	00000010111
25	0101011	25	00000011000
26	0010011	26	000011001010
27	0100100	27	000011001011
28	0011000	28	000011001100
29	00000010	29	000011001101
30	00000011	30	000001101000
31	00011010	31	000001101001

White run length	Code word	Black run length	Code word
32	00011011	32	000001101010
33	00010010	33	000001101011
34	00010011	34	000011010010
35	00010100	35	000011010011
36	00010101	36	000011010100
37	00010110	37	000011010101
38	00010111	38	000011010110
39	00101000	39	000011010111
40	00101001	40	000001101100
41	00101010	41	000001101101
42	00101011	42	000011011010
43	00101100	43	000011011011
44	00101101	44	000001010100
45	00000100	45	000001010101
46	00000101	46	000001010110
47	00001010	47	000001010111
48	00001011	48	000001100100
49	01010010	49	000001100101
50	01010011	50	000001010010
51	01010100	51	000001010011
52	01010101	52	000000100100
53	00100100	53	000000110111
54	00100101	54	000000111000
55	01011000	55	000000100111
56	01011001	56	000000101000
57	01011010	57	000001011000
58	01011011	58	000001011001
59	01001010	59	000000101011
60	01001011	60	000000101100
61	00110010	61	000001011010
62	00110011	62	000001100110
63	00110100	63	000001100111

hite run length	Code word	Black run length	Code word
64	11011	64	0000001111
128	10010	128	000011001000
192	010111	192	000011001001
256	0110111	256	000001011011
320	00110110	320	000000110011
384	00110111	384	000000110100
448	01100100	448	000000110101
512	01100101	512	0000001101100
576	01101000	576	0000001101101
640	01100111	640	0000001001010
704	011001100	704	0000001001011
768	011001101	768	0000001001100
832	011010010	832	0000001001101
896	011010011	896	0000001110010
960	011010100	960	0000001110011
1024	011010101	1024	0000001110100
1088	011010110	1088	0000001110101
1152	011010111	1152	0000001110110
1216	011011000	1216	0000001110111
1280	011011001	1280	0000001010010
1344	011011010	1344	0000001010011
1408	011011011	1408	0000001010100
1472	010011000	1472	0000001010101
1536	010011001	1536	0000001011010
1600	010011010	1600	0000001011011
1664	011000	1664	0000001100100
1728	010011011	1728	0000001100101

Huffman Code Application - CCITT Group 3 1-D FAX Encoding

e.g 500 white pixel run

500 = 448+ 52 = 7 x 64 + 52 => k = 7, m = 52

01100100**01010101**000000000001 <- 28bits

k **m** **eol**

500 bits reduced to 28bits

Huffman Coding and its Extension

- Huffman coding method has been adopted in many applications such as fax machines, JPEG, and MPEG.
- Extended Huffman Coding
  - Assign a single codeword to the group of symbols instead of assigning a codeword to each symbol.
- Adaptive Huffman Coding
  - The Huffman algorithm requires prior statistical knowledge about the information source and such information is often not available (eg. live/streaming audio and video).
  - Adaptive Huffman coding uses the probabilities that are no longer based on the prior knowledge but on the actual data received on.

Adaptive Huffman Coding

- The Huffman algorithm requires prior statistical knowledge about the information source and such information is often not available.
- Adaptive Huffman algorithm uses statistics gathered and updated based on occurrences of preceding symbols. In this approach, as the probability of the received symbols change, symbols will be given new (longer or shorter) codes.

Adaptive Huffman Coding

- Procedures for Adaptive Huffman Coding

Encoder

```
1. Initial_code();
2. while not end_of_stream
   {
      get(c);
      encode(c)
      update_tree(c);
    }
```

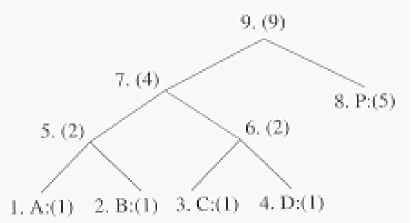
**** C is a symbol.***

Adaptive Huffman Coding

- `Initial_code(c)`: assigns codes for symbols with some initially agreed-upon. For example ASCII code.
- `encode(c)`: encodes the symbol `c` using the current Huffman tree or the initial codes.
- `update_tree(c)`: constructs an adaptive Huffman tree. It increments the frequency counts for the symbol `c` and updated the configuration of the tree.

Adaptive Huffman Coding

- `update_tree(c):`
  - The Adaptive Huffman tree must maintains is *sibling property*: all nodes (internal and leaf) are arranged *in the order of increasing counts/frequencies*. Nodes are numbered in order from left to right, bottom to top.
  - If the *sibling property* is about to be violated, a swap procedure is invoked to update the tree by rearranging the nodes.

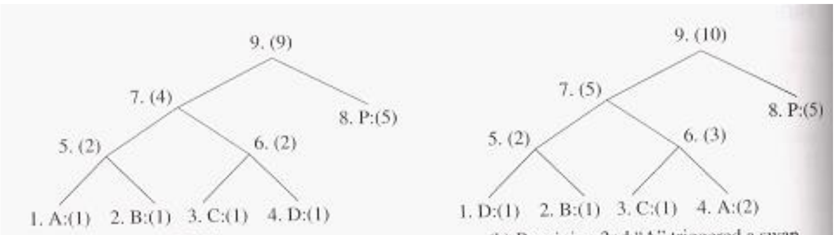


(a) Huffman tree

Adaptive Huffman Coding

- `update_tree(c):`
  - When a swap is necessary, the most distant node with count N is swapped with the node whose count has just been increased to N+1. If the node with count N is not a leaf-node (the root of a subtree), the entire subtree will go with it during the swap.
  - After the swap, propagate the counts to the parents and check if the tree maintains the sibling properties. If no, perform update_tree recursively.

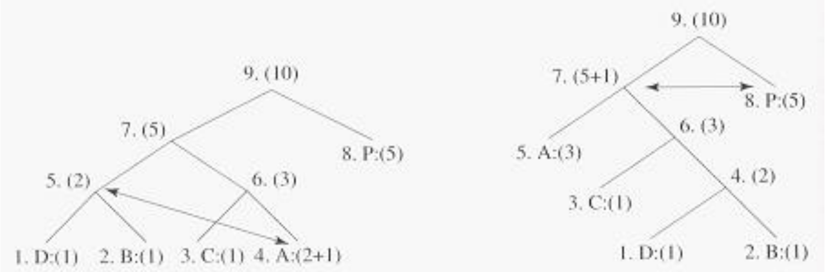
Adaptive Huffman Coding – Tree Update



(a) Huffman tree

(b) Receiving 2nd "A" triggered a swap

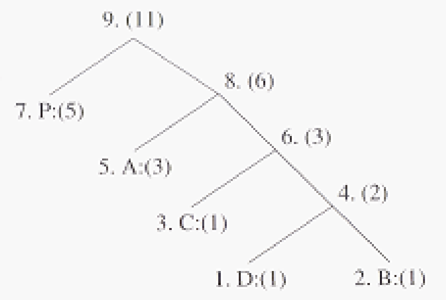
Adaptive Huffman Coding – Tree Update



(c-1) A swap is needed after receiving 3rd "A"

(c-2) Another swap is needed

Adaptive Huffman Coding – Tree Update



(c-3) The Huffman tree after receiving 3rd "A"

Adaptive Huffman Coding - Example

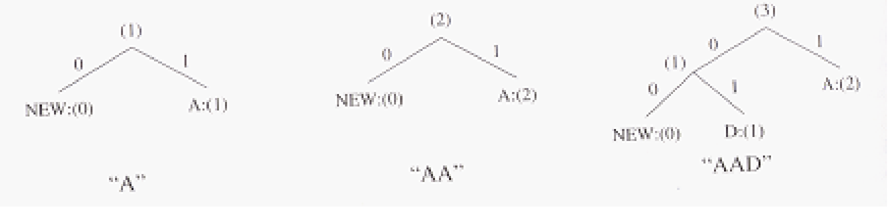
- Assume we want to encode "AADCCDD" string. We use the following codes for initial coding.

  ```
  A 00001
  B 00010
  C 00011
  D 00100
  ```

- One additional rule: if any symbol is to be encoded (sent) the first time, it must be preceded by a special symbol, NEW.
  - The initial code for NEW is 0.
  - The count for NEW is always 0.

Adaptive Huffman Coding - Example

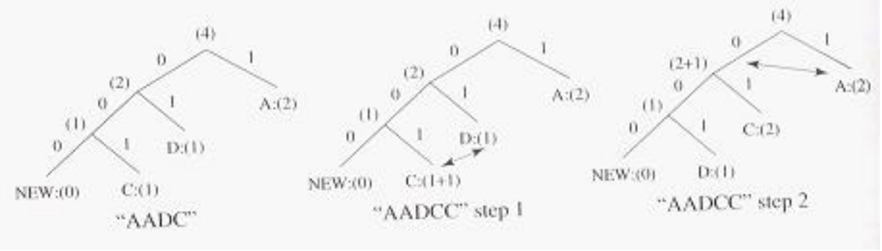
- Encoding "AADCCDD"



A - 000001
AA - 0000011
AAD - 00000110000100

Adaptive Huffman Coding - Example

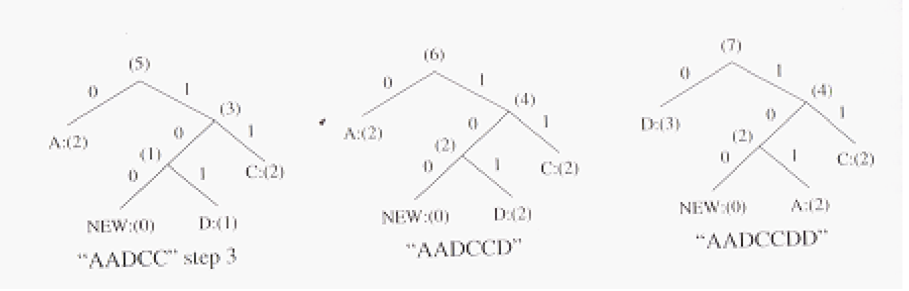
- Encoding "AADCCDD"



AADC - 0000011000100000011
AADCC - 0000011000100000011001

Adaptive Huffman Coding - Example

- Encoding “AADCCDD”



AADCC - **000001100010000000011001**
AADCCD- **000001100010000000011001101**
AADCCDD- **000001100010000000011001101101**

Adaptive Huffman Coding - Example

- **Sequence of codes sent to the decoder**

Symbol	Codes
NEW	0
A	00001
A	1
NEW	0
D	00100
NEW	00
C	00011
C	001
D	101
D	101

- **For “AADCCDD”, “000001100010000000011001101101” is sent.**

Adaptive Huffman Coding

- Procedures for Adaptive Huffman Coding

Decoder

```
1. Initial_code();
2. while not end_of_stream
  {
     decode(c);
     output(c)
     update_tree(c);
   }
```


Adaptive Huffman Coding

- **`decode(c)`:**
  - **If the previous code is `NEW`, use initial_code to decode.**
  - **If the previous code is not `NEW`, use the Huffman tree.**
- **The input to the decoder**
  `"00000110001000000011001101101"`