Multimedia Systems and Applications

Data Compression

James Wang

An overview of Compression

- **Compression becomes necessary in multimedia because it requires large amounts of storage space and bandwidth**
- **Types of Compression**
  - Lossless compression = data is not altered or lost in the process
  - Lossy = some info is lost but can be reasonably reproduced using the data.

2

Binary Image Compression

- **RLE (Run Length Encoding)**
  - Also called Packed Bits encoding
  - E.g. aaaaaaaaaaaaaaaaaaaa111110000000
  - Can be coded as:
  - Byte1 Byte 2 Byte3 Byte4 Byte5 Byte6
  - 20 a 05 1 07 0
  - This is a one dimensional scheme. Some schemes will also use a *flag* to separate the data bytes

3

Binary Image Compression

- **Disadvantage of RLE scheme:**
  - When groups of adjacent pixels change rapidly, the run length will be shorter. It could take more bits for the code to represent the run length that the uncompressed data → negative compression.
  - It is a generalization of *zero suppression*, which assumes that just one symbol appears particularly often in sequences.

4

Lossless Compression Algorithms (Entropy Encoding)

Adapted from:
http://www.cs.cf.ac.uk/Dave/Multimedia/node207.html

5

Basics of Information Theory

- **According to Shannon, the entropy of an information source S is defined as:**

$$H(S) = \eta = \sum_i p_i \log_2 \frac{1}{p_i}$$

where p_i is the probability that symbol S_i in S will occur.

- $\log_2 \frac{1}{p_i}$ **indicates the amount of information contained in S_i, i.e., the number of bits needed to code S_i.**

For example, in an image with uniform distribution of grey-level intensity, i.e. p_i = 1/256, then the number of bits needed to code each grey level is 8 bits. The entropy of this image is 8.

6

The Shannon-Fano Algorithm

- **A simple example will be used to illustrate the algorithm:**

Symbol	A	B	C	D	E
Count	15	7	6	6	5

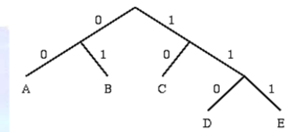
7

The Shannon-Fano Algorithm

- Encoding for the Shannon-Fano Algorithm:

A top-down approach

1. Sort symbols according to their frequencies/probabilities, e.g., ABCDE.
2. Recursively divide into two parts, each with approx. same number of counts.



MORE FREQUENT → LESS BITS,
LESS FREQUENT → MORE BITS

8

The Shannon-Fano Algorithm

Symbol	Count	$\log_2(1/p_i)$	Code	Subtotal (# of bits) =Count X Code Size
A	15	1.38	**00**	30
B	7	2.48	**01**	14
C	6	2.70	**10**	12
D	6	2.70	**110**	18
E	5	2.96	**111**	15

TOTAL (# of bits): 89

9

Huffman Coding

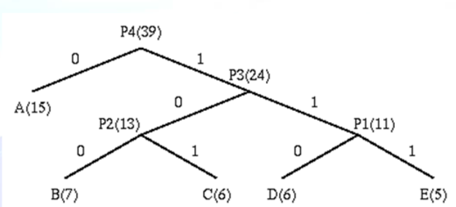
Encoding for Huffman Algorithm:

- **A bottom-up approach**
- **1. Initialization: Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE).**
- **2. Repeat until the OPEN list has only one node left:**

(a) From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them.
(b) Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN.
(c) Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN.

10

Huffman Coding



11

Huffman Coding

Symbol	Count	$\log_2(1/p_i)$	Code	Subtotal (# of bits) =Count X Code Size
A	15	1.38	**0**	15
B	7	2.48	**100**	21
C	6	2.70	**101**	18
D	6	2.70	**110**	18
E	5	2.96	**111**	15

TOTAL (# of bits): 87

12

Huffman Coding

Discussions:

- **Decoding for the above two algorithms is trivial as long as the coding table (the statistics) is sent before the data. (There is a bit overhead for sending this, negligible if the data file is big.)**
- Unique Prefix Property: **no code is a prefix to any other code (all symbols are at the leaf nodes) --> great for decoder, unambiguous.**
- **If prior statistics are available and accurate, then Huffman coding is very good.**

In the above example:

entropy **= (15 x 1.38 + 7 x 2.48 + 6 x 2.7 + 6 x 2.7 + 5 x 2.96) / 39 = 85.26 / 39 = 2.19**

Number of bits needed **for Human Coding is: 87 / 39 = 2.23**

13

Adaptive Huffman Coding

Motivations:

(a) The previous algorithms require the statistical knowledge which is often not available (e.g., live audio, video).
(b) Even when it is available, it could be a heavy overhead especially when many tables had to be sent when a non-order0 model is used, i.e. taking into account the impact of the previous symbol to the probability of the current symbol (e.g., "qu" often come together, ...).

- **The solution is to use adaptive algorithms. As an example, the Adaptive Huffman Coding is examined below. The idea is however applicable to other adaptive compression algorithms.**

14

Adaptive Huffman Coding

- **ENCODER**

```
Initialize_model();
while ((c = getc (input)) != eof)
{
        encode (c, output);
        update_model (c);
}
```

15

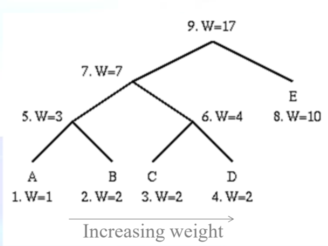
Adaptive Huffman Coding

- **DECODER**

```
Initialize_model();
while ((c = decode (input)) != eof)
{
        encode (c, output);
        update_model (c);
}
```

16

Adaptive Huffman Coding



Adaptive Huffman Tree

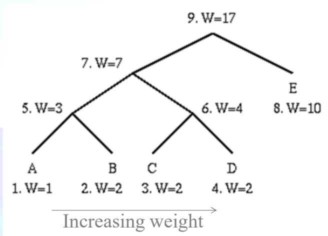
17

Adaptive Huffman Coding

- **The key is to have both encoder and decoder to use exactly the same** *initialization* **and** *update_model* **routines.**
- *update_model* **does two things: (a) increment the count, (b) update the Huffman tree.**
- **During the updates, the Huffman tree will be maintained its** *sibling property*, **i.e. the nodes (internal and leaf) are** ***arranged in order of increasing weights*** **(see figure).**
- **When** *swapping* **is necessary, the farthest node with weight W is swapped with the node whose weight has just been increased to W+1.**
  Note: **If the node with weight W has a sub tree beneath it, then the sub tree will go with it.**
- **The Huffman tree could look very different after node swapping, e.g., in the third tree, node A is again swapped and becomes the #5 node. It is now encoded using only 2 bits.**

18

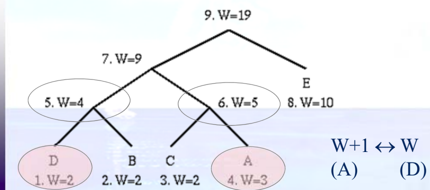
Adaptive Huffman Coding



Adaptive Huffman Tree

19

Adaptive Huffman Coding

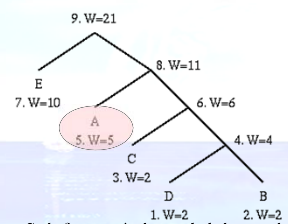


Note: Code for a particular symbol changes during the adaptive coding process.

After a node switch (A was incremented twice)

20

Adaptive Huffman Coding



Note: Code for a particular symbol changes during the adaptive coding process.

After A was incremented two more times

21

Lempel-Ziv-Welch Algorithm

Motivation:

- **Suppose we want to encode the Webster's English dictionary which contains about 159,000 entries. Why not just transmit each word as an 18 bit number?**

Problems: **(a) Too many bits, (b) everyone needs a dictionary, (c) only works for English text.**

- **Solution: Find a way to build the dictionary adaptively.**
- **Original methods due to** Ziv and Lempel **in 1977 and 1978. Terry** Welch **improved the scheme in 1984 (called LZW compression). It is used in e.g., UNIX** *compress*, **GIF, V.42 bis.**
- ***Reference:*** **Terry A. Welch, "A Technique for High Performance Data Compression", IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.**

22

LZW Compression Algorithm

LZW Compression Algorithm:

```
w = NIL;
while ( read a character k )
{
    if wk exists in the dictionary
        w = wk;
    else
        { add wk to the dictionary (so wk is stored);
          output the code for w;
          w = k; }
}
```

23

LZW Compression Algorithm

- **Original LZW used dictionary with 4K entries, first 256 (0-255) are ASCII codes.**
- Example: **Input string is** "^WED^WE^WEE^WEB^WET".
- Steps:

	w	k	Output	Index	Symbol
1	NIL	^			
2	^	W	^	256	^W
3 4	W	E	W	257	WE
5	E	D	E	258	ED
6	D	^	D	259	D^

24

LZW Compression Algorithm

- Steps:-

6	^	W			
7	^W	E	256	260	^WE
8	E	^	E	261	E^
9	^	W			
10	^W	E			
11	^WE	E	260	262	^WEE

25

LZW Compression Algorithm

- Steps:-

12	E	^			
13	E^	W	261	263	E^W
14	W	E			
15	WE	B	257	264	WEB
16	B	^	B	265	B^

26

LZW Compression Algorithm

- Steps:-

17	^	W			
18	^W	E			
19	^WE	T	260	266	^WET
20	T	EOF	T		

27

LZW Compression Algorithm

- A 19-symbol input has been reduced to 7-symbol plus 5-code output. Each code/symbol will need more than 8 bits, say 9 bits.
- Usually, compression doesn't start until a large number of bytes (e.g., > 100) are read in

28

LZW Compression Algorithm

LZW Decompression Algorithm:

```
read a character k;
output k;
w = k;
while ( read a character k ) /* k could be a character or a code. */
{
    entry = dictionary entry for k;
    output entry;
    add w + entry[0] to dictionary;
    w = entry;
}
```

29

LZW Compression Algorithm

Example:

Input string is "^WED<256>E<260><261><257>B<260>T".

Steps:

	w	*k*	*Output*	*Index*	*Symbol*
1		^	^		
2	^	W	W	256	^W
3	W	E	E	257	WE
4	E	D	D	258	ED
5	D	<256>	^W	259	D^

30

LZW Compression Algorithm

- Steps:-

6	**<256>**	E	E	260	^WE
7	**E**	<260>	^WE	261	E^
8	**<260>**	<261>	E^	262	^WEE
9	**<261>**	<257>	WE	263	E^W
10	**<257>**	B	B	264	WEB
11	**B**	<260>	^WE	265	B^
12	**<260>**	T	T	266	^WET

31

LZW Compression Algorithm

- Problem: What if we run out of dictionary space?
- Solution 1: Keep track of unused entries and use LRU (Least Recently Used)
- Solution 2: Monitor compression performance and flush dictionary when performance is poor.
- Implementation Note: LZW can be made *really* fast; it grabs a fixed number of bits from input stream, so bit parsing is very easy. Table lookup is automatic.

32

Huffman vs. Arithmetic Code

- **Lowest L_{ave} for Huffman codes is 1. Suppose $H << 1$?**
  - One option: use one code symbol for several source symbols
  - Another option: Arithmetic code.
- **Idea behind arithmetic code:**
  - Represent the probability of a sequence by a binary number.

33

Arithmetic Encoding

- **Assume source alphabet has values 0 and 1, $p_0 = p$, $p_1 = 1 – p$.**
- **A sequence of symbols $s_1, s_2, \ldots s_m$ is represented by a *probability* interval found as follows:**

```
Initialize, lo = 0; hi = 1
For i = 0 to m
    if s_i = 0
        hi = lo + (hi-lo)* p_0
    else
        lo = lo + (hi-lo)* p_0
    end
end
```

- Send binary fraction x such that lo ≤ x < hi. This will require

$\lceil x \rceil$ bits, where $x = -\sum_{i=1}^{m} \log_2 P(s_i)$

34

Arithmetic Encoding

- **Assume source alphabet has values 0 and 1, $p_0 = p$, $p_1 = 1 – p$.**
- **A sequence of symbols $s_1, s_2, \ldots s_m$ is represented by a *probability* interval found as follows:**

```
Initialize, lo = 0; range = 1
For i = 0 to m
    if s_i = 0
        range = range*p
    else // s_i = 1
        lo = lo + range*p
        range = range*(1-p)
    end
end
```

- Send binary fraction x such that lo ≤ x < hi. This will require

$\lceil \log_2 range \rceil$ bits

35

Arithmetic coding: example

- **p_0 = 0.2, source sequence is 1101**

bit	low	high
	0	1
1	0.2	1
1	0.36	1
0	0.36	0.488
1	0.3856	0.488

→ 0 + (1-0)*0.2 = 0.2

→ 0.36 + (1-0.36)*0.2
= 0.36 + 0.128

Number of bits = ceiling(-$\log_2$(0.1024)) = 4   Bits sent: 0111

- **p_0 = 0.2, source sequence is 1101**

symbol	low	range
	0	1
1	0.2000	0.8000
1	0.3600	0.6400
0	0.3600	0.1280
1	0.3856	0.1024

Number of bits = ceiling(-$\log_2$(0.1024)) = 4 low_2 =.01100010, $(low+range)_2$ = .01111100   Bits sent: 0111

36

Arithmetic Decoding

- **We receive *x*, a binary fraction**

```
lo = 0; hi = 1
for i = 1 to m
    if (x - lo) < p*(hi-lo)
        si = 0
        hi = lo + (hi-lo)*p
    else
        si = 1
        lo  = lo + (hi-lo)*p
    end
end
```

- ***m* and *p* are sent to the decoder in the header.**

Arithmetic Decoding

- **We receive *x*, a binary fraction**

```
lo = 0; range = 1;
for i = 1 to m
    if (x - lo) < p*range
        si = 0
        range = p*range
    else
        si = 1
        lo  = lo +range*p
        range = range*(1 - p)
    end
end
```


Arithmetic Decoding

- **We receive *x*, a binary fraction**

```
for i = 1 to m
    if x  < p
        si = 0
        x = x/p
    Else // x > p
        si = 1
        x = (x - p)/(1 - p)
    end
end
```

Receive x = 0111 = 0.4375
p = 0.2

symbol	x	range
	0.4375	1
1	0.2969	0.8000
1	0.1211	0.6400
0	0.6055	0.1280
1	0.5068	0.1024

Arithmetic decoding example

- **Receive 0111 (0.4375), decode 4 bits, p0 = 0.2**

x= 0.4375

low	high	bit
0	1	1
0.2	1	1
0.36	1	0
0.36	0.488	1

symbol	low	range
	0	1
1	0.2000	0.8000
1	0.3600	0.6400
0	0.3600	0.1280
1	0.3856	0.1024

Magic Features of Arithmetic Coding

- **Remember I (information) = - $\log_2 p$**
  - p = 0.5, I = 1
  - p = 0.125, I = 3
  - p = 0.99, I = 0.0145 (wow!)
- **High p symbol, less than 1 code bit per symbol!**
- **In encoder, hi - lo = ∑ I(symbols)**

Discussion on Arithmetic Coding

- **When the length of the original can not be predetermined, how does the decoder know where to stop?**
  - A special ending symbol, similar to EOF.
  - Sending fixed length chunks.
- **How do we determine the value for *P*?**
  - Based on estimating the probabilities of symbols to appear in the sequence.
- **Can we encode data with symbols other than 0 and 1?**

Conclusions

- **Huffman maps fixed length symbols to variable length codes. Optimal only when symbol probabilities are powers of 2.**
- **Lempel-Ziv-Welch (LZW) is a dictionary-based compression method. It maps a variable number of symbols to a fixed length code.**
- **Adaptive algorithms do not need a priori estimation of probabilities, they are more useful in real applications.**
- **Arithmetic algorithms: Complexity: requires arithmetic (multiplications, divisions), rather than just table lookups**
  - Algorithms are complex, accuracy (significant bits) is tricky
  - Can be made to operate incrementally
    - Both encoder and decoder can output symbols with limited internal memory
  - Provides important compression savings in certain settings

43

References

- ***The Data Compression Book*, Mark Nelson,M&T Books, 1995.**
- ***Introduction to Data Compression*, Khalid Sayood, Morgan Kaufmann, 1996.**

44