2. Data Compression Methods

References

David Salomon, "A Guide to Data Compression Methods", Springer 2001, Chapters 1-2
Fred Halsall, "Multimedia Communications", Addison Wesley 2001, Chapter 3

Introduction

The most widely used types are based on the statistics of the input to be coded. Therefore there is no one code, or type of code, that is optimal, since text , speech, image, video have all very different characteristics and statistics. The compression itself can be lossless or lossy, depending on the application. Text compression is always lossless. While words have redundancy, numbers do not, and there is no algorithm that can decide what text can be dropped and what cannot. In contrast both speech and image can make use of lossy algorithms for compression. Such algorithms delete much image or speech information. The test for the performance of such an algorithm is for a person to compare the original image or speech to the compressed (lossy) version and find whether it looks or sounds practically the same.

It is important to look at generic requirements and methods of compression:

(i) variable size codes – short codes to the common symbols; long codes to the rare symbols
(ii) the sequence of bits making up the symbols generated by the coder can be uniquely decoded back into the respective symbols by the decoder
(iii) the decoder must know the (prefix) code of the symbol to be able to decode it. This can be done in one of three ways:
    (a) a set of (prefix) codes is determined once and used by all encoders and decoders; (eg fax compression)
    (b) encoder performs a two-pass job. In the first pass it reads the data and collects the statistical information. This is used to compute the set of (prefix) codes suitable for the file, which are then first transmitted to the decoder and used for compression
    (c) adaptive compression used by the system. The encoder starts with no knowledge of the statistical properties of the data. First part of data is therefore poorly compressed, but while compressing, the encoder collects statistical information, and improves the prefix codes assigned and therefore improve the performance. The algorithm must allow the decoder to gather the same statistical information and improve the prefix codes in the same way as the encoder.

Three types of compression algorithms will be considered. Huffman Coding, Arithmetic Coding based on statistics. LZ* compression based on dictionary methods.

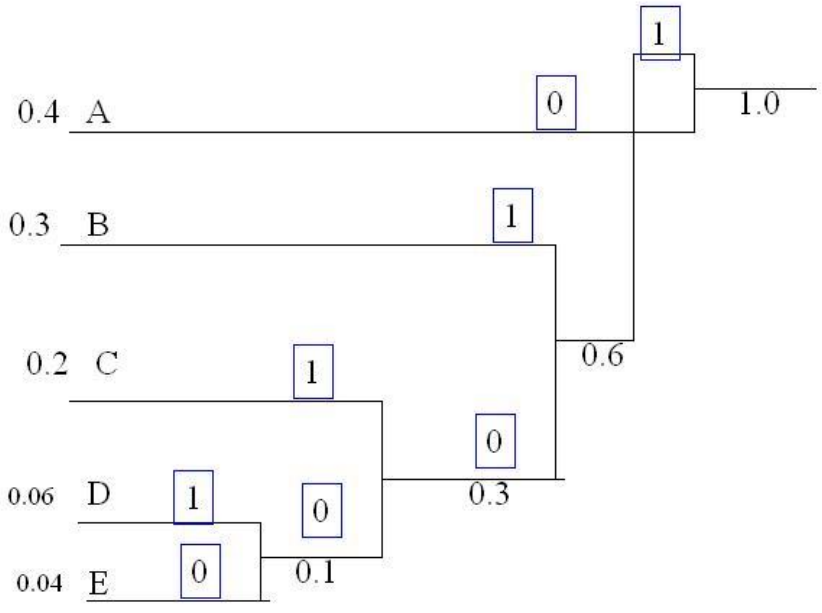
Huffman Coding

The method starts by ordering the symbols according to their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up.
Step 1: The two symbols with the smallest probabilities are selected and added to the partial tree at a branch node generating a new branch that represents both.
Step 2 Repeat step 1 always combining together the two symbols with smallest probability
Step 3 When the list is reduced to a final branch the tree is complete

Comments: Since in step 2 there can be more than one alternative of pairs that have the smallest probability, there can be more than one Huffman code representation. The Huffman code resulting in the minimum variance is considered to be the best, since when generating the code bits it has the minimum generating rate variability, when directly transmitting. (Otherwise it is immaterial.)
When building the tree the bit 0 and 1 are assigned to the branches that make up a node. For consistency the branch that has the higher probability is attached from the top, to the new node, and assigned the bit value 1. Again if the two branches that are forming the new node have equal probability any one of them can be attached from the top.
The code of a given symbol is obtained by traversing the tree backward along the branches to the original symbol, picking up the bit value along the corresponding branch.

Ex: Build a Huffman Code for an alphabet with five symbols, A,B,C,D,E, having a probability of 0.4, 0.3, 0.2, 0.06, 0.04, respectively.

What is the entropy of this code? What is the average size of the code? What is the variance of the code? Can another Huffman code be generated for this alphabet?



Example1

Entropy using (1.4) gives 1.9434 bits

For the system above the Huffman codes are
A = 0; B = 11; C = 101; D = 1001; E = 1000;

The average bits is found from $0.4 \times 1 + 0.3 \times 2 + 0.2 \times 3 + 0.1 \times 4 = 2.0$

Variance
$0.4\,(1-2)^2 + 0.3(2-2)^2 + 0.2(3-2)^2 + 0.1(4-2)^2 = 1.0$

In this case, no other Huffman code structure is possible. It is possible to invert the 1 and 0 and have a reflected code.

Huffman Decoding

First, the decoder must generate the tree, from information it receives as a header (such as the probabilities or the frequencies of occurrence of the alphabet symbols). The algorithm consists of
Step 1 Start at the root, read the first bit and move in using the branch of the bit. Continue until the sequence of bits moving in the tree come to a symbol.
Step 2 Emit the symbol and move back to the root of the tree
Step 3 Repeat Steps 1 and 2 till the bits received are all used.

Example 2:

For the alphabet in Example 1, what are the received symbols for a bit sequence 10111010011110000

Answer C B A D B E A

The Average Code Size of a Huffman code can also be obtained directly by summing up the values of all the internal nodes of a tree, (without the need of explicit multiplication).

Adaptive Huffman Codes

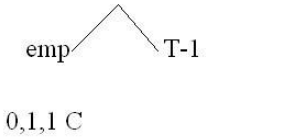
Since in practice the statistics are not known, this method is used in practice in applications involving file compression, (zip, UNIX compact). Note that for short files this method will not produce compression.

Both transmitter and receiver start with a tree that has the root node and a single empty leaf node – (a leaf having occurrence 0) This is used as the escape code, followed by the uncompressed code of the character whenever a new character is introduced. The uncompressed code can be the ASCII character, or if there are general symbols some table built up for the symbols being used that uses the minimum number of bits necessary. For n symbols $a_1, a_2, a_3, \ldots, a_n$, select integers m and r such that $2^m \leq n < 2^{m+1}$ and $r = n - 2^m$. The first 2^m symbols are encoded as the

(m+1) bit numbers 0 through. The remaining symbols are encoded as m-bit numbers such that the code a_k is k - 2^m – 1. This type of code is called a phased-in binary code.

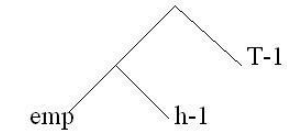
The encoder reads in the first character. Since tree is empty it sends the code for the empty leaf followed by the uncompressed code for the first character.
The tree is then updated at transmitter and receiver. Initially all that happens is that the first character is assigned to the right branch, its frequency is updated to 1, and the root node also updated to 1. Updating is as follows:
Step 1: Get in the next character. If it is a new character then go to the empty leaf node send the current code of the empty leaf node followed by the uncompressed character.
If character already exists then both encoder and decoder proceed to update their copy of the tree.
Step 2: The frequency of all nodes and branch nodes are updated, wrt current node.
Step 3 Starting from the empty leaf node write down the frequency of successive nodes and branch nodes from left to right and from bottom to top.
Step 4 If the nodes are not in proper weighting (frequency order) the out of order node is swapped with the node in the highest position that has a lower frequency then itself. The swapping moves all the hanging branches and nodes from one position to another.
Step 5 The ordered weighting is again recalculated to check whether the frequency is out of order.
Step 6 The system iterates back to step 1 till all the characters (symbols) are input
.
In this way both transmitter and receiver update the tree using the same procedure so that both trees are a mirror of each other at the same point in the character (symbol) sequence.

Example 3
Build up an adaptive Huffman tree using the phrase – The cat sat on the mat. Assume characters are represented by the ASCII equivalent in the uncompressed form.



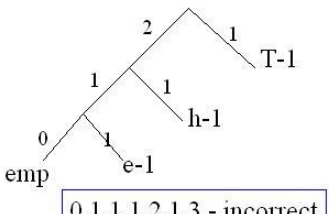
0,1,1 C

Fig 1



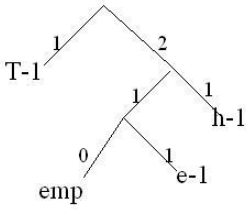
0,1,1,1,2 - correct

Fig 2



0,1,1,1,1,2,1,3 - incorrect

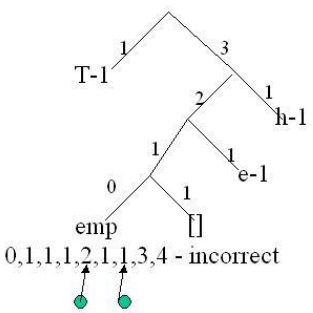
Fig 3a



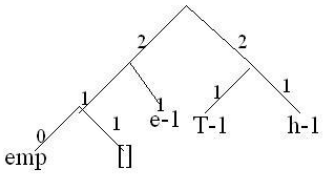
0,1,1,1,1,2,3 - correct

Fig 3b

Start of tree building process involving the first three letters.



0,1,1,1,2,1,1,3,4 - incorrect



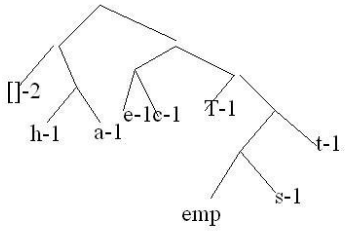
0,1,1,1,1,1,2,2,4,- correct

The tree hanging from node with value 2 is exchanged with the tree Hanging from the farthest node 1.

Figure4a

Figure 4b

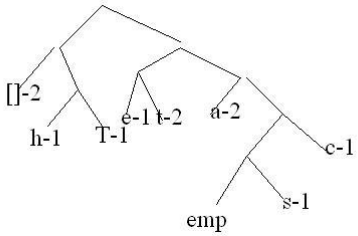
Addition of fourth letter [] to tree.



The[]cat[]s

0,1,1,1,1,1,1,1,1,2,2,2,2,3,4,5,9 - correct

Figure7



The[]cat[]sat

0,1,1,1,1,1,1,1,2,2,2,2,2,3,4,4,7,11 - correct

Figure8

Example 4. Work out the bit pattern sent from the transmitter to the decoder for the example 3.

Problems: There can be overflows in the count (frequency) register if the symbol frequency exceeds register length. Can be solved by doing a division by 2 over all the counts. However there must be a check on the order after this.
There can be errors if the size of the tree is deeper than the size of the register that works out the code for a symbol, as it traverses the tree.

Variants on Huffman Code

One major variant uses run length codes on dots making up a fax and then giving a Huffman variable length code to these groups, based on the statistical distribution of the white or black groups, standardized by the CCITT as Group3 and Group 4 codes (see Table 1.20 p 35, Salomon or Fig 3.11, p145, Halsall)
For current digital, (ISDN or ADSL) based fax machines there is additionally a two dimensional coding named modified modified Read (MMR) to account for the gray scale possible on such fax machines. (Salomon pp37-40; Halsall pp146-150)

Arithmetic Coding

This type of coding works out one code over the input, instead of assigning codes to individual symbols. The code number is built up, becoming longer and needing a higher number of digits as the input becomes longer. As for Huffman coding, the statistics of the symbols is required to operate properly. However, an adaptive technique is also possible. The algorithm, in theory, works as follows
Step 1 Find the probabilities of the symbols
Step2 The interval [0, 1) is defined as the Current interval
Step3 Divide the current interval into subintervals whose sizes are proportional to the symbols' probabilities
Step4 Start with the first symbol, using its subinterval as the current interval
Step5 Given the new symbol X calculate the new subinterval as

$$NewHigh = OldLow + (OldHigh - OldLow)*HighRange(X)$$
$$NewLow = OldLow + (OldHigh - OldLow)*LowRange(X)$$

Step6 Iterate Step5 till all the symbols are used.

Example 5

A system uses 5 symbols A,B,C,D,E with probability 0.5, 0.2, 0.1, 0.1, 0.1. The symbols transmitted are C,B,A,A,E,A,D,B,A,A. Calculate the resultant Arithmetic Code.

Initially

C H 0.3
L 0.2

B H $0.2 + (0.3 - 0.2)*0.5 = 0.25$
L $0.2 + (0.3 - 0.2)*0.3 = 0.23$

A H $0.23 + (0.25-0.23)*1 = 0.25$

L 0.23 + (0.25-0.23)*0.5 = 0.24

A H 0.24 + (0.25-0.24)*1 = 0.25
L 0.24 + (0.25-0.24)*0.5 = 0.245

E H 0.245 + (0.25 - 0.245) * 0.1 = 0.2455
L 0.245 + (0.25 - 0.245) * 0 = 0.245

Example 5

The algorithm is not practical, since it assumes that numbers of unlimited precision are available for Low and High. A practical implementation should use just integers, and they should not be very long.
This is done by shifting out the leftmost digits of Low and High when they become identical. This way the two variables have to keep only the recent part of the code and not the entire code.

The system is initialized with High holding 99999… and Low 0000… The following shows the encoding of the system in Example 6

Step1 Initialise with the first symbol using the symbol range within the overall range 0000 to 9999
Step 2 For the second symbol calculate the range. If the leftmost digit is the same for Low and High, save the digit, and remove it from Low and High. Shift to the left adding 0 to the low and a 9 to the high
Step3 Work out the new L and H by dividing the integer L and H to the range 0 – 1.
Step4 Iterate step2 each time that the leftmost digit resulting is the same output it and remove it from working number.

Using the previous example: Initially

C	L	2000		
	H	2999		
B	L 0.2 +0.1*0.3=0.23	2300	2	3000
	H 0.2 +0.1*0.5=0.25	2499		4999
A	L 0.3+0.2*0.5=0.4	4000	4	0000
	H 0.3+0.2*1.0=0.5	4999		9999
A	L 0 + 1.0*0.5=0.5	5000		5000
	H 0 + 1.0*1.0=1.0	9999		9999
E	L 0.5 + 0.5*0.1 = 0.55	5500	5	5000
	H 0.5 + 0.5*0.2 = 0.6	5999		9999

Arithmetic Decoding

This is the opposite of the encoding system. It needs the symbols and the range of each. It then operates on the codeword starting from the leftmost digit.
Step1: Assign the symbol of the range of the leftmost digit
Step2: Reduce its effect by computing newcode = (oldcode-lowrange(X))/OldRange
Step3: Repeat steps1 and 2 till the remainder of the codeword is 0

Using the coded value of 0.245:

0.2 --→ C
(0.245-0.2)/ 0.1 = 0.45
0.4 -→ B
(0.45 – 0.3)/0.2 = 0.75

0.7 -→ A
(0.75 – 0.5)/0.5 = 0.5
0.5 -→ A
(0.5 –0.5)/0.5 = 0

This has a problem when the last symbol, (in this case E), is in the range starting from 0. Therefore an endoffile symbol is added with a small probability to the list, and should be encoded at the end of the input file.

Because of the implementation issues, the decoding in practice is based on the range 0000 to 9999 making use also of code digits that are sent by the encoder together with the final 4 digits..
Step1 calculate an index that gives the current symbol to be decoded using
Index = ((code – low+1)x10 –1)/ (High – Low +1) truncate to nearest integer, and obtain the symbol.. Initially Low is 0000 and High is 9999.
Step2 Update Low and High using
Low = Low+(High – Low +1)LowCumFreq(X)/10;
High= Low+(High – Low =!)HighCumFreq(X)/10;
Where lowcumfreq(X) and highcumfreq(X) are the cumulative frequencies of symbol X decoded in Step1, obtained from the original table of ranges, and expressed between 0 and 10.
Step3: If the leftmost digits of Low and High are identical, shift Low, High and Code, one position to the left. Low gets a 0 entered on the right, High gets 9 entered on the right, and Code gets the next input digit from the compressed code received.
Using the same example, the decoder has received
2455000

2 -→ C 2455-2000 = 455/100 = 4.55 -→ 4

4 -→ B 4550 – 3000 = 1550/200 = 7.75 -→ 7

7 -→ A 7750 – 5000 = 1750/500 = 5.50 -→ 5

5 -→ A 5500 – 5000 = 0500/500 = 1.0 -→ 1

There can be problems unless the endoffile symbol is also included in trying to recognize the final symbol sent.

Adaptive Arithmetic Coding

In this case the frequency of the symbols is not fixed, but changes with the input, changing as well the range of the symbol depending on the frequency.
The encoding algorithm has two parts. The probability model and the arithmetic encoder.
Step1 Read next symbol from input file
Step2 Use the encoder to work out

Low = Low+(High – Low+1)LowCumFreq[X}/10;
High = Low+(High – Low +1)HighCumFreq[X]/10;
Where the values for X are based on the old count (frequency)

Step3 Increment the count of the symbol and update the cumulative frequencies.

In this way, the decoder can mirror the operation of the encoder, by first decoding the symbol based on its current knowledge, and then updating its cumulative frequency, to be ready for the next symbol with the updated frequency.

As a corollary, the data structure for the symbol frequency count should be kept in sorted order of the counts implying that the order of symbols may change. The best data structure for this is the balanced binary tree.