IMPROVED HUFFMAN CODING USING RECURSIVE SPLITTING.

Karl Skretting, John Håkon Husøy and Sven Ole Aase

Høgskolen i Stavanger, Department of Electrical and Computer Engineering
P. O. Box 2557 Ullandhaug, N-4004 Stavanger, Norway
E-mail: Karl.Skretting@tn.his.no

ABSTRACT

Lossless compression of a sequence of symbols is an important part of data and signal compression. Huffman coding is lossless, it is also often used in lossy compression as the final step after decomposition and quantization of a signal. In signal compression, the decomposition/quantization part seldom manages to produce a sequence of completely independent symbols. Here we present a scheme giving better results than straightforward Huffman coding by utilizing this fact. We split the original symbol sequence into two sequences in such a way that the symbol statistics are, hopefully, different for the two sequences. Individual Huffman coding for each of these sequences will reduce the average bit rate. This split is done recursively for each sub-sequence until the cost associated with the split is larger than the gain.

Experiments were done on different signals. They were decomposed with the Discrete Cosine Transform, quantized, and End Of Block coded, to give the input symbol sequence. The results using the split scheme was a bit rate reduction of usually more than 10% compared to straightforward Huffman coding, and 0-15% better than JPEG-like Huffman coding, best at low bit rates.

1. INTRODUCTION

Huffman coding creates variable-length codes, each represented by an integer number of bits. Symbols with higher probabilities get shorter codewords. Huffman coding is the best coding scheme possible when codewords are restricted to integer length, and it is not too complicated to implement [1]. It is therefore the entropy-coding scheme of choice in many applications.

The Huffman code tables usually need to be included in the compressed file as side information. To avoid this one could use a standard table derived for the relevant class of data, this is an option in the JPEG compression scheme [4]. Another alternative is adaptive Huffman coding as in [3]. While these methods do not need side information they use non-optimal codes and consequently more bits for the symbol codewords. The efficiency of Huffman coding can often be significantly improved by the use of custom made Huffman code tables. This possibility is also included in the JPEG compression scheme [4]. The methods used in this paper all use custom made Huffman code tables.

Huffman coding is effective when integer codeword lengths are suitable for the symbol sequence. Generally, this is the case when no symbols have very high probabilities, especially no symbol should have probability greater than 0.5. If the symbols probabilities are 0.5, 0.25, 0.125, 0.0625 or less than 0.05 then a scheme using integer codeword lengths will do quite well. Huffman codes do not exploit any dependencies between the symbols, so when the symbols are statistically dependent other methods may be much better.

1.1. Lossy signal compression

Lossy signal compression often has the following steps

1. Decomposition.
2. Quantization, often with threshold.
3. Run Length and End Of Block coding.
4. Huffman coding.

Here we compare three different schemes for compression: straightforward, JPEG- like, and recursive Huffman coding. The two first steps are identical for all three methods, we use DCT and uniform quantization with threshold. The results can then be represented in a matrix where the rows are the frequencies and the columns are time. The entries are the quantized values, there are as many entries as there are samples in the signal. The upper left part of this matrix may be

Block	1	2	3	4	5	6	7	8	···
LP (DC)	4	5	5	0	-4	-2	4	2	···
BP (AC)	1	0	3	0	0	-1	0	-2	···
	0	1	1	0	0	4	1	0	···
	0	0	0	0	5	0	0	-1	···
	0	0	0	0	0	0	0	0	···
HP (AC)	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	···

Since we have used a 16 points DCT, the matrix will have 16 rows (bands) and each block is 16 samples. The three different methods used here all start with this matrix of quantized values, and use different ways to form the symbol sequences.

Straightforward Huffman Coding use only End of Block coding. The End of Block symbol, (0), and the rest of the symbols are formed from the quantized values according to this table

Value	EOB	···	-2	-1	0	1	2	···
Symbol	0	···	4	2	1	3	5	···

The symbol sequence after EOB coding for the example above will then be:

9, 3, 0, 11, 1, 3, 0, 11, 7, 3, 0, 0, 8, 1, 1, 11,
0, 4, 2, 9, 0, 9, 1, 3, 0, 5, 4, 1, 2, 0, ···.

We note that there will be as many EOB symbols as there are columns in the matrix, and that the symbol sequence will be non-negative integers where the smaller ones are more probable than the larger ones, because symbols represented by small integers correspond to small magnitude of the quantized values.

JPEG-like Huffman Coding makes the symbols the same way as JPEG does, each column of the matrix correspond to the zigzag scanned sequence of a 8×8 pixel picture block in JPEG. The DC component and the AC components are coded separately. The DC component is DPCM coded and the symbols are defined by the following table

Symbol	DPCM difference	Additional bits
0	0	0
1	-1, 1	1
2	-3, -2, 2, 3	2
3	-7, …, -4, 4, …, 7	3
4	-15, …, -8, 8, …, 15	4
⋮	⋮	⋮

Each symbol is followed by some additional bits to uniquely give the DPCM difference. For the data example this gives (the two last lines are stored)

Quantized DC value	4	5	5	0	-4	…
DPCM difference	4	1	0	-5	-4	…
Symbol	3	1	0	3	3	…
Additional bits	100	1	-	010	011	…

For the AC component the zeros are run length coded. Each symbol consists of two parts, the first part is the run that tells how many zeros that precede the value (R), and the second part is the value symbol (S). The value symbols are the same as the symbols used for the DPCM differences. To completely specify the value each symbol is succeeded by additional bits the same way as for the DPCM differences. The combined symbol (represented as one integer) is $16R+S$. Symbol (0) is EOB. For the example data this gives

Quantized AC value	1	EOB	1	EOB	3	…
Value symbol (S)	1	0	1	0	2	…
Preceding zeros (R)	0	0	1	0	0	…
Symbol (16R+S)	1	0	17	0	2	…
Additional bits	1	-	1	-	11	…

Recursive Huffman Coding uses the same symbol sequence as straightforward Huffman coding. Usually these symbols are not independent, which means that the true entropy (lower limit for possible bit rate) is less than zero-order entropy (lower limit for bit rate for Huffman code). The proposed scheme takes advantage of some dependencies in the symbol sequence and exploits this in the Huffman coding procedure. The next two sections of the paper explain the details of this method. Note that the method has some limitations. If the symbol sequence is highly correlated, it will probably be better to try to improve the decomposition part rather than to hope that this Huffman coding scheme will utilize all of the correlation. Also, if integer codeword lengths are not suitable then other methods may be much better.

2. SPLITTING THE SYMBOL SEQUENCE

The basic idea is that by splitting a long sequence into several shorter ones in a way that makes the symbol probabilities (and the optimal code lengths) different for each sequence, then individual Huffman coding of each sequence will reduce the total number of bits used for the codewords. On the other hand, there will be more Huffman code tables to include. Clever splitting combined with effective coding of the side information should give an improvement in overall bit rate. We choose to use a scheme that first splits the symbol sequence after End of Block coding into three sequences.

2.1. Splitting into three sequences

When we examine the End of Block coded sequence we make the following observations

- A symbol succeeding an EOB symbol (0) is the DC component, or possibly another EOB symbol.

- An EOB symbol (0) will never succeed a (1) symbol.

This may be exploited by creating three symbol sequences from the original sequence. The first sequence contains the first symbol and the symbols following a (0) symbol, the next sequence contains the symbols following a (1) symbol, and the third sequence contains all the other symbols. The key to success is that the symbol probabilities will be different for these sequences. In fact only this splitting improve straightforward Huffman coding considerably. Using this scheme, the example sequence will be split as

Original EOB sequence: 9, 3, 0, 11, 1, 3, 0, 11, 7, 3, 0, 0, 8, 1, 1, 11, 0, 4, 2, 9, 0, 9, 1, 3, 0, 5, 4, 1, 2, 0, ...
First sequence: 9, 11, 11, 0, 8, 4, 9, 5, ...
Second sequence: 3, 1, 11, 3, 2, ...
Third sequence: 3, 0, 1, 0, 7, 3, 0, 1, 0, 2, 9, 0, 1, ...

Then each of these is dealt with independently of each other and in the same way by the recursive splitting part of the function.

2.2. Recursive splitting

The recursive splitting part either

1. splits the input sequence into two sub-sequences, this split is done either
   (a) by cutting the sequence in the middle or
   (b) by letting the previous symbol decide to which sub-sequence the following symbol should be put into,

   and then calls itself twice with each of the sub-sequences as arguments or
2. does Huffman coding of the input sequence, that is store the Huffman table information and the codewords into the output bit sequence.

The decision rules are: If the symbol sequence is long, 1.a is done. Else, we test if splitting (as in 1.b) will reduce the number of bits, and if so, we split as in 1.b, else we do point 2.

Cutting in the middle (1.a) is one obvious way to split the symbol sequence, especially if the signal is non-stationary. When sequence length is larger than 2^{15}, the sequence is split into two sequences of half the length. This ensure that no used symbol has a probability less than 2^{-15}. Then the given code word lengths always will be less or equal to 15, which is the maximum code word length that we allow when we code the Huffman tables.

Splitting by previous symbol (1.b) tries to utilize correlation between successive symbols by letting the previous symbol decide to which sub-sequence the following symbol should be put into. The first sub-sequence contains the symbols following a symbol with a value less or equal than a limit value, the second sub-sequence contains the other symbols (including the first symbol). Using this scheme with limit value equal 1, one example sequence will be split as

Ex. sequence: 3, 0, 1, 0, 7, 3, 0, 1, 0, 2, 9, 0, 1, ...
First sequence: 1, 0, 7, 1, 0, 2, 1, ...
Second sequence: 3, 0, 3, 0, 9, 0, ...

By using the median (of the numbers representing the symbols in the original sequence) as this limit value, we split into two approximately equal size sub- sequences. In addition, the split is done in such a way that we do not need the decision rule or the limit value to be included as side information.

3. INCREASED SIDE INFORMATION

The way we have done splitting, very little side information is needed to specify when and how to split a sequence. In fact, we use only one bit to tell whether a sequence is split or not. However, we need to include as many Huffman tables as we have sequences. To keep this side information small we put a bit of effort into doing a good job on compact storing of these tables. This effort pays off; our scheme often uses less than one third of the bits to store the Huffman tables, compared to what JPEG uses. Let us start by looking at how JPEG store the Huffman tables.

3.1. JPEG Huffman table specification

Usually this side information is relatively small, and consequently not much effort has been used in representing this in few bits. JPEG use a special segment, the DHT marker segment structure, to specify a Huffman table [4]. In this segment, one byte are used to tell how many symbols there are with code length i, for $i = 1 : 16$, then follows the symbols with one byte for each. This requires $(16 + N)$ bytes for N symbols.

3.2. Efficient Huffman table specification

We tried several different ad-hoc methods to store the Huffman tables. The problem was to find a method that performed well for all possible Huffman tables. We ended up with a method that performed quite well, which we will now briefly describe. This method uses 4 bits to give length of first symbol, then for each of the next symbols a code to tell its length where

Symbol	what it means
0	same length as previous symbol
10	increase length by 1
1100	reduce length by 1
1101	increase length by 2
111xxxx	set symbol length to xxxx

This way of coding the Huffman tables utilize the fact that adjacent symbols often have approximately the same probability, and thus approximately the same codeword lengths.

4. SIMULATIONS AND COMPARISON

The three signals used in the compression experiments are an AR1 ($\rho = 0.95$) signal, an ECG signal (normal sinus rhythm, MIT100 [2]), and a seismic signal (pre-stack common shot gather [5]). These signals are very different from each other, the ECG signal is quite regular and the seismic signal is quite noisy. The signal length is, 250000 samples, but when we compressed shorter signals, ex. 25000 samples, the graphs were quite similar to the graphs that we include here. All signals were decomposed by the Discrete Cosine Transform (block size 16), and then quantized, the quantizing step varied to get signals with different Signal to Noise Ratios.

The three figures show the results for the AR1 signal, the ECG signal and the seismic signal respectively.

4.1. Conclusion

Both on real world signals and a synthetic signal the proposed Huffman coding scheme does considerably better than straightforward Huffman coding, and usually better than JPEG-like Huffman coding, especially at low bit rates.

5. REFERENCES

[1] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, Boston, 1992. ISBN 0-7923-9181-0

[2] Massachusetts Institute of Technology. *The MIT-BIH Arrhythmia Database CD-ROM*, 22nd edition, 1992.

[3] Mark Nelson, Jean-Loup Gailly. *The Data Compression Book.* M&T Books, New York, USA, 1996. ISBN 1-55851-434-1

[4] William B. Pennebaker, Joan L. Mitchell. *JPEG: Still Image Data Compression Standard.* Van Nostrand Reinhold, New York, USA, 1992. ISBN: 0442012721

[5] Seismic Data Compression Reference Set. http://www.ux.his.no/~karlsk/sdata/

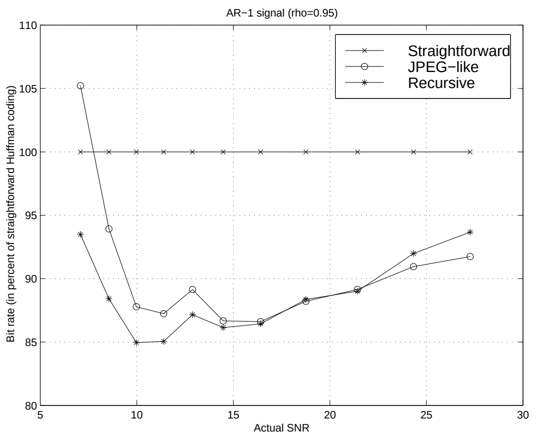

Figure 1: AR1 signal compressed at different signal to noise ratios.

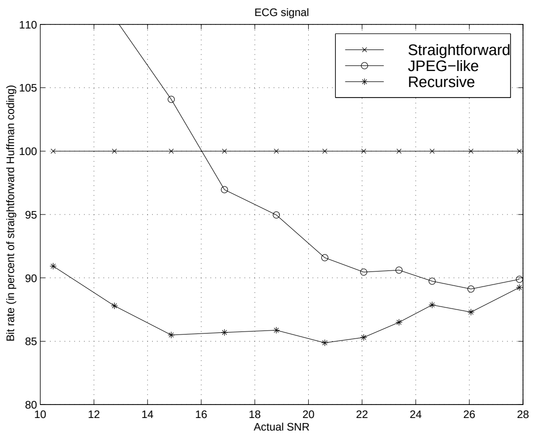

Figure 2: ECG signal compressed at different signal to noise ratios.

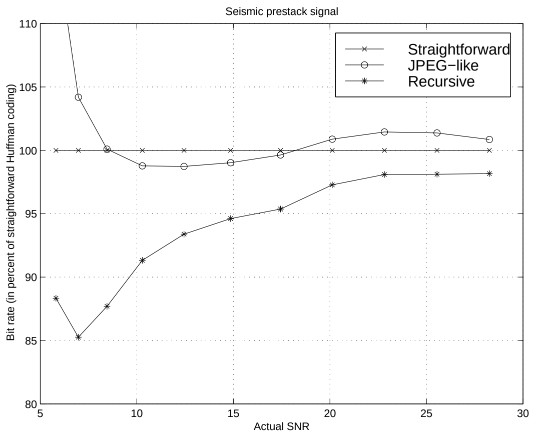

Figure 3: Seismic signal compressed at different signal to noise ratios.