

# DIRECT HUFFMAN CODING AND DECODING USING THE TABLE OF CODE-LENGTHS

Reza Hashemian, Life Member, IEEE  
Northern Illinois University  
DeKalb, Illinois 60115, USA  
[reza@ceet.niu.edu](mailto:reza@ceet.niu.edu)

## Abstract

*A new Huffman coding and decoding technique is presented. There is no need to construct a full size Huffman table in this technique; instead, the symbols are encoded directly from the table of code-lengths. For decoding purposes a new Condensed Huffman Table (CHT) is also introduced. It is shown that by employing this technique both encoding and decoding operations become significantly faster, and the memory consumption also becomes much smaller compare to the normal Huffman coding/decoding.*

## 1. INTRODUCTION

As a variable-length coding technique, Huffman coding [1] has been shown to be one of the most leading methods being used in various applications dealing with data compression. In fact, as it is widely practiced, the combination of Huffman coding and run-length coding provide a near optimal entropy coding technique which is easy to implement.

A typical Huffman coding, normally presented by a binary tree structure, becomes progressively *sparse* as it grows from the root. This sparsity usually causes tremendous waste of memory space, unless a well-structured procedure is adopted to efficiently allocate the symbols in the memory [2-7]. In addition, this sparsity may also result in a lengthy search procedure for locating a symbol, hence adding to the delay. There have been a number of techniques developed and reported in the literature to reduce the memory requirement and to increase the speed for the search [3-13].

To directly address the problem, we first realize that there are normally two distinct operations in Huffman coding, i) code-length determination, and ii) encoding procedure and code-words assignment. The first operation consists of ordering the symbols based on their probabilities, pairing those with the lowest probabilities and substituting them by a pseudo-symbol with the sum of probabilities of the pair. The process continues until it leads to a final destination, as a root. As an outcome of this procedure, each symbol can be represented by its unique

path to the root, called the code-word, with a path-length called the code-length. The code-length determination operation is almost identical in every technique, and it was originally established by D.A. Huffman[1]. However, some exceptions exist for the cases where some symbols/pseudo-symbols appear with identical probabilities.

The second operation, i.e. the code-word assignment, however, may differ from one technique to another; and this may severely affect the decoding effectiveness [3]. A common property in any Huffman encoding is that each code must be unique, such that it could be recognizable in the bit-stream with no guard-bit(s) attached. Therefore, the decoder must extract the symbol, and recognize the code-length simultaneously. Another important property of the Huffman coding is that, assigning any valid code-word to a symbol does not alter the coding efficiency (entropy), as long as the code-length remain unchanged. Nevertheless, the difference may appear in terms of the memory requirement as well as the time for search, which leads to change in the decoding speed. In other words, although code-length determination is almost unique in every technique the different encoding procedure may consume different memory space, as well as different timing needed to access a symbol.

The method presented in this article deals mainly with encoding/decoding procedure, and it is based on the technique reported in [3], as recorded in the table of code-lengths TOCL (Table III). Due to some unique features in this representation, it is shown that both encoding and decoding procedures may avoid using a full-size Huffman table. In case of encoding it is shown that the TOCL is, in fact, sufficient to extract the code-words, and similarly, a condensed table of code-words, that is much smaller than the normal Huffman table, is all needed to decode the code-words.

The remaining part of this article is organized in the following sections. In Section 2, we introduce the encoding procedure, and discuss how to produce the code-words from a given Table of Code-lengths. In Section 3, the construction of a Condensed Huffman Table is discussed, and the generation of a CHT from a TOCL is formulated. In Section 4, the decoding procedure using

CHT is explained and an example is given to clarify the procedure. Finally, Section 5 contains the conclusion.

## 2. ENCODING TECHNIQUE

Consider a set of symbols  $s_i$  with the probability of occurrences  $p_i$ , for  $i = 1, 2, \dots, N$ . Similar to Huffman coding [1], we first produce a listing of the symbols in descending order of the probabilities – the ordering of symbols with equal probabilities is assumed indifferent. Again, a procedure similar to the initial stage in a successive Huffman table reduction and reordering [1] is carried out. This procedure results in the construction of the Table of Code-lengths (TOCL) for a given set of symbols, as shown in Table III, in [3]. Evidently, a TOCL contains the entire information for the construction of a Huffman table, and the ordered symbol listing is no longer necessary to keep. In fact, a TOCL is the sole departure point for any Huffman coding assigned to a set of pairs  $(s_i, p_i)$ , for  $i = 1, 2, \dots, N$ . To see this, and to fully demonstrate the procedure for encoding -- and decoding -- symbols we start with the following example.

Consider a set of 18 symbols with their probability of occurrences already ordered in descending order, as shown in Table 1.

Table 1- an example of symbols and their occurrences

Symbols $s_i$	Occurrences	Probabilities $p_i$
$s_1$	1517	0.256
$s_2$	1512	0.255
$s_3$	1459	0.246
$s_4$	731	0.123
$s_5$	107	0.018
$s_6$	103	0.0173
$s_7$	100	0.0168
$s_8$	51	0.0085
$s_9$	48	0.0082
$s_{10}$	47	0.0079
$s_{11}$	46	0.0077
$s_{12}$	42	0.0071
$s_{13}$	41	0.0069
$s_{14}$	38	0.0064
$s_{15}$	35	0.0059
$s_{16}$	33	0.0056
$s_{17}$	15	0.0025
$s_{18}$	13	0.0022

Next, we follow the procedure discussed in [3] and construct the TOCL corresponding to Table 1, as shown in Table 2. It is also important to note that having a TOCL (Table 2) is enough to generate the symbol listing exactly in the order of descending probabilities, as shown in Table 1. This is, in fact, the basis for producing the code-words (Huffman) table from the TOCL that is much smaller in

size. To proceed, we first need to generate another table, which represents the differential code-lengths, TDCL. This table consists of two columns, the first column is the differential code-lengths (rather than the actual ones, as they appear in the TOCL), and the second column records the number of symbols (not the symbols themselves), in each level. Table 3 shows the TDCL for our example, generated from Table 2.

Table 2 – The Table of Code-lengths TOCL

Code-length L	Symbol
2	$s_1, s_2, s_3$
3	$s_4$
6	$s_5, s_6, s_7$
7	$s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}$
8	$s_{17}, s_{18}$

Table 3 – The Table of Differential Code-lengths TDCL

$i$	Differential Code-length $DCL$	Number of Symbols $NS$
0	2	3
1	1	1
2	3	3
3	1	9
4	1	2

Now, given a TDCL,  $T$  (Table 3), with  $n$  number of rows, and an ordered symbol list, (Table 1), containing  $N$  number of symbols, we proceed to develop an algorithm that leads to the generation of the code-words for the given symbols. The procedure generates a code-word for each symbol in the symbol list which then leads to creation of a Huffman table. That is, for each symbol  $s_j$  in the symbol list we record a unique code-word  $CW_j$ . Note that  $DCL(i)$  and  $NS(i)$  respectively denote the differential code length and the number of symbols associated with the  $i$ th row in the TDCL (Table 3).

**Algorithm 1:** - {

Start with the first row in the listing, i.e.,  $j = 1$ , and assume no value (no digit) for  $k$ ;

For ( $i = 0; i < n; i = i+1$ ) {

Tap  $k$  with  $DCL(i)$  number of zero to the rights, and let the code-word  $CW(j) = k$ ;

$j = j+1$ ;

If ( $NS(i) > 1$ ) {

For ( $m = 1; m < NS(i); m = m+1$ ) {

$k = k+1$ ;

$CW(j) = k$ ;

$j = j+1$ ;

}

}

incrément  $k$  :

}  
}

This completes the generation of the code-words  $CW(j)$ , for  $j = 1$  to  $N$ , and hence the generation of the Huffman table, as shown in Table 4. Notice that Tables 1 and 4 have the same order of symbols, except that the value of the code-words in Table 4 are in ascending order, which is quite in contrast to the order of the symbol probabilities. This is another property of the Huffman table, called Single-Side Growing Huffman Table [3], where it makes it possible for reordering the table for encoding purposes.

Table 4 – The Huffman Table for the Example

Symbols	Code-words <i>CW</i>
$s_1$	00
$s_2$	01
$s_3$	10
$s_4$	110
$s_5$	111000
$s_6$	111001
$s_7$	111010
$s_8$	1110110
$s_9$	1110111
$s_{10}$	1111000
$s_{11}$	1111001
$s_{12}$	1111010
$s_{13}$	1111011
$s_{14}$	1111100
$s_{15}$	1111101
$s_{16}$	1111110
$s_{17}$	11111110
$s_{18}$	11111111

### Encoding procedure

As it stands, encoding a symbol by employing the Huffman Table (Table 4) requires a long and blind search to find the corresponding code-word. One method to speed up the process is to start the search from the symbols with smallest code-word values. This is more efficient procedure because in this case the symbols with higher probabilities are searched first. Another efficient method, suitable for larger Huffman tables, is to take advantage of the inversely ordered code-words in the Huffman table and use it to reorder a section of the table with larger code-words (lower probabilities). This reordering is according to the symbol values and makes the search quicker. In fact, this method combined with a direct search for the section of higher probability symbols (the top section in the table) prove to be even more efficient.

*Example 1:* Suppose we need to encode the sequence of symbols  $s_2, s_6, s_1, s_{18}$ , and  $s_4$ . After performing a series of searches through Table 4 we get the coded stream as  $CS = 01111001001111111110$ .

### 3. CONDENSED HUFFMAN TABLE

Now, we are ready to prove that a TOCL is, in dead, sufficient to generate a Condensed Huffman Table (CHT), which is all we need to decode the incoming bit stream code-words.

To generate a CHT from a TOCL we first notice that each row in a TOCL represents symbols with identical code-lengths, as listed in the L column in Table 2. And according to *Algorithm 1*, if we know the code-word for the first symbol in a row the other code-words in that row are simply found by incrementing the previous code-word by one. Hence, we can take advantage of this unique feature and create a separate table, called Condensed Huffman Table CHT. A CHT, shown in Table 5, has three columns. One column shows the code-lengths ( $L$ ), which is identical to that in the TOCL (Table 2), except for the first row, which is removed here. Another column specifies the address in the symbol list (Table 1) for the first symbols in each row of Table 2, i.e.,  $s_1, s_4, s_5, s_8$ , and  $s_{17}$ . Finally, the third column,  $CW$ , provides the *augmented* code-words for those first symbols. An augmented code-word is the code-word that is tapped with enough zeros, on the right, in order to gain the maximum code-length in that Huffman table. For example, consider the code-length 6 in Table 2. The smallest code-word in this category is 111000 for  $s_5$ . However, since the largest code-length in our example is 8 we need to tap two more zeros and get the *augmented* code-word for  $s_5$  as 11100000, as indicated in Table 5.

Table 5 – The Condensed Huffman Table

$i$	Code-word <i>CW</i>	Code-length $L$	Address $a$
1	11000000	3	3
2	11100000	6	4
3	11101100	7	7
4	11111110	8	16

Algorithm 2 provides an alternative procedure to directly generate the CHT from a TDCL.

**Algorithm 2:** - {

Let  $CW(0) = 00, \dots, 0$ , representing  $DCL(0)$  number of zeros (see Table 3);

For ( $i = 1; i < n; i = i+1$ ) {

$CW(i) = CW(i-1) + NS(i-1)$ ;

Tap  $CW(i)$  with  $DCL(i)$  number of zeros to the right;

}  
 Augment all the entries in the column  $CW$  to length  $L_m$ ,  
 where  $L_m$  is the maximum word-length.  
 }

Figure 1 is a graphical representation (characteristic curve) of the Huffman Table 4 with augmented code-words. In Fig. 1 the horizontal axis shows the symbols as listed in descending probabilities, and the vertical axis represents the value of the augmented Huffman codes. It is important to note that this characteristic curve is piecewise linear, with darker dots marking the break points, and associated with the entries in the CHT (Table 5). As noticed, the break points separate the symbols in one level from those in the next. This means that, to precisely draw the entire characteristic curve all we need to do is to record the starting point for each segment plus its slope, indicating the corresponding code-length. This is, in fact, the basis for the construction of the CHT, as discussed earlier.

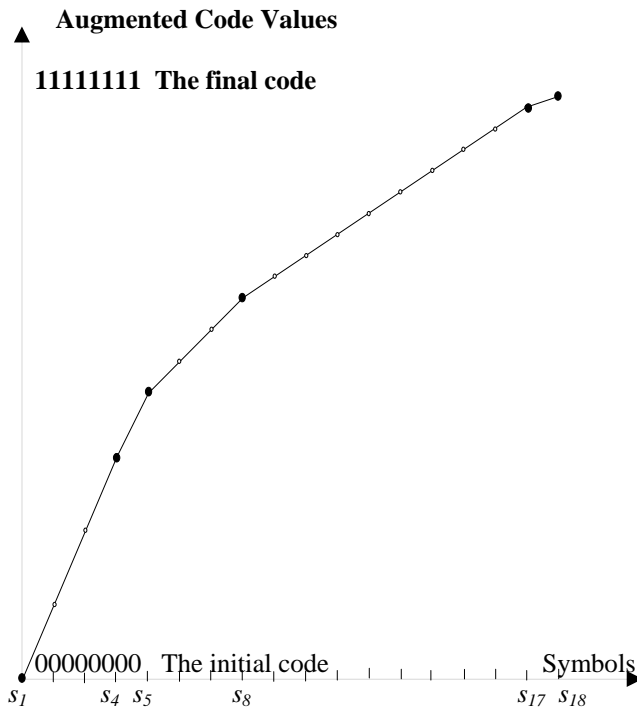


Fig. 1- The characteristic curve for the symbol-list (Huffman code), Table 3

#### 4. DECODING PROCEDURE

Given a CHT, Algorithm 3 presents a stepwise procedure to decode an input bit-stream. The procedure starts by taking a chunk of  $L_m$  number of bits from the input bit-stream, where  $L_m$  is the maximum code-length. The value of this piece of data must tell us the location of the line segment in Fig. 1, or specify the associated row in the CHT table (Table 5), where we can find the intended

symbol. The rest of the process is just to conduct a linear search.

#### Algorithm 3: –

1. Take  $L_m$  number of bits from the bit-stream and call it  $P$ . Compare  $P$  to  $CW_1$ , the first (smallest) augmented code-word in the CHT.
2. If  $P$  is less than or equal to  $CW_1$  then the first  $L_0$  (the smallest code-length) bits in  $P$  represent the code-word  $cw_{0j}$  for the intended symbol. Also  $a_{0,j} = cw_{0j}$  is the address of the desire symbol in the symbol-list (Table 4) (in case the symbol-list starts from address 1, rather than 0, then we need to write  $a_{0,j} = cw_{0j} + 1$ ). Finally, return the excess,  $L_m - L_0$  bits into the bit-stream.
3. If, on the other hand,  $P$  is greater than  $CW_1$ , continue searching for larger code-words in the CHT, until  $CW_{i+1}$  is found to be the first one greater than  $P$ . The pertinent code-length is then going to be  $L_i$ , and the code-word is the first  $L_i$  bits in  $P$ . To find the address of the symbol in the symbol-list,  $a_{i,j}$ , find  $P - CW_i$ , and keep the first  $L_i$  bits, and call it  $m_j$ . The pertinent address is now found as  $a_{i,j} = a_i + m_j$ . Finally, return the excess,  $L_m - L_i$  bits into the bit-stream.
4. If  $P$  is greater than  $CW_1$  and also greater than all other augmented code-words in the CHT, including the last one  $CW_m$ , then  $P$  itself must be the intended code-word. Find  $m_j = P - CW_m$ , and calculate  $a_{m,j} = a_m + m_j$ , which is the address of the symbol in the symbol-list. This concludes the algorithm.

Notice that any bit stream, although it may not generate the correct output, is decodable by using Algorithm 3. In other words, for any  $P$  there is always a code-word in the original symbol-list that is identical to the first  $L_i$  bits in  $P$ , where  $L_i$  can take any value from  $L_0$  to  $L_m$ . This is, of course, a serious concern in any Huffman coding, and it requires an effective error checking technique, which is beyond the scope of this study.

To illustrate the decoding procedure, consider the symbol-list in Table 4, and run the following example.

*Example 2:* Suppose the input stream is given as  $CS = 01111001001111111110$ , generated in Example 1. First take  $L_m = 8$  number of bits from the bit-stream and get  $P = 01111001 = 79$ . This code is smaller than  $CW_1 = c0$ , hence take only  $L_0 = 2$  MSBs from  $P$  as the code-word for the symbol  $cw_{0j} = 01$ , and drop the rest of the bits in  $P$  into the bit-stream. Next, use the code  $01$  to allocate the symbol  $s_2$  in the symbol list (Table 1).

For the next search, again take  $L_m = 8$  number of bits from the bit-stream ( $111001001111111110$ ) and get  $P = 11100100 = e4$ . This code is larger than  $CW_1 = c0$ , hence a search follows until  $ec$ , in row 3, is found. Then turn back to row 2 in the CHT and find  $e0$  as the code-word. Next find the difference between the two code-words,  $c4$  and  $c0$ , i.e.,  $c4 - c0 = 4$ , and take the first 6 bits which

results in 1. Therefore, the symbol must be in address 1, in Table 2 row 2, which is  $s_6$ . Or alternatively, the symbol address can be calculated from  $a_{i,j} = a_i + m_j = 4 + 1 = 5$  and  $s_6$  can be allocated at address 5 in Table 1.

Next, continue with another 8-bit code from the bit-stream and find  $P = 00111111 = 3f$ . This code is smaller than  $CW_l = c0$ , and similar to an earlier step the code-word is found to be  $cw_{0l} = 00$  and the symbol is  $s_l$ . The next 8-bit code from the bit-stream is  $P = 11111111 = ff$ . This code is greater the  $CW_m = fe$ , and hence it is indeed the intended code-word, and  $m_j = ff - fe = 1$ . The address of the symbol is found from  $a_{m,j} = 16 + 1 = 17$ , which is the address for  $s_{18}$ . Finally,  $110$  is the remainder bit-stream, and after augmentation  $P = 11000000 = c0$ . This is equal to the first augmented code-word in Table 5, and hence the symbol must be  $s_4$ .

### Memory requirement and Search

As shown, the process is quite efficient in both memory requirement and access time. This is specially true in decoding and extracting the symbol. In general, for  $n$  symbols, and  $l$  active levels (code-lengths) in the TOCL the total memory space required in the decoder side is about

$$M = 2n + 3l - 2 \cong 2.4n$$

Due to high probabilities of occurrence of symbols with smaller code-words typically more that 50% to 80% of time the search to find the symbol will resolve in the first step, and there is no need to make any addition search. Therefore, although in the worse case (the longest code-word) the search may take about  $l-1$  steps plus one addition and one subtraction of data to conclude, but if we neglect the time for shifting the average search time for decoding comes close to:

$$D = 0.5add + one \text{ step search}$$

## 5. CONCLUSION

A memory efficient and high-speed search technique is developed for encoding and decoding symbols using Huffman coding. This technique is based on a Condensed Huffman Table (CHT) for decoding purposes, and it is shown that a CHT is significantly smaller than the ordinary Huffman Table. In addition, the procedure is shown to be faster in searching for a code-word and its corresponding symbol in the symbol-list. An efficient technique is also proposed for encoding symbols by using the code-word properties in a Single-Side Growing Huffman Table (SGHT), where code-word values are ordered in the ascending order exactly in contrast with the probabilities that are in descending order.

## 6. REFERENCES

- [1] A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, pp. 1098-1101, Sept. 1952.
- [2] R. Hunter and A. H. Robinson, "International Digital Facsimile Standard," *Proc. IEEE*, vol. 68, no. 7, pp. 854-867, 1980.
- [3] R. Hashemian, "Memory Efficient and High Speed Search Huffman Coding", *IEEE Transactions on Communications*, Vol. 43, No. 10, pp. 2576-2581, October, 1995.
- [4] R. Hashemian, "Design and Hardware Implementation of a Memory Efficient Huffman Decoding," *IEEE Trans, on. Consumer Electro*, vol. 40, No. 3, pp 345-351, Aug. 1994.
- [5] M. K. Rudberg and L. Wanhammar, "Implementation of a Fast MPEG-2 Compliant Huffman Decoder", *Proc. EUSIPCO '96*, Trieste, Italy, September 1996.
- [6] S. F. Chang and D. G. Messerschmitt, "Designing High-Throughput VLC Decoder Part I Concurrent VLSI Architectures", *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2, No. 2, pp. 187-196, June 1992.
- [7] H. D. Lin and D. G. Messerschmitt, "Designing High-Throughput VLC Decoder Part II - Parallel Decoding Methods", *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2, No. 2, pp. 197-206, June 1992,
- [8] S. Ho and P. Law, "Efficient Hardware Decoding Method for Modified Huffman code", *Electronics Letters*, Vol. 27, No 10, pp. 855-856, May 1991.
- [9] A. IC Jain, "Image data compression: A review," *Proc. IEEE*, vol. 69, pp. 349-389, Mar, 1981.
- [10] T. J. Fexguson and J. H. Rabinowitz, "Self-synchronizing Huffman codes," *IEEE Trans. InformL Theory*, vol. IT-30, pp. 687-693, July 1984.
- [11] S. M. Lei and M. T. Sun, "An entropy coding system for digital HDTV applications," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 1, pp. 147-155, Mar. 1991.
- [12] S. M. Lzi, M. T Sun, and & H. Tzou, "Design and hardware architecture of high-order conditional entropy coding for image," *IEEE Trans, Circuit Syst. Video Technol.*, vol, 2, pp. 176-186, June 1992.
- [13] S. Roman, *Coding and Information Theory*. New York: SpringerVerlag, 1992.