

CONDENSED TABLE OF HUFFMAN CODING, A NEW APPROACH TO EFFICIENT DECODING

Reza Hashemian, Senior Member, IEEE
Northern Illinois University
DeKalb, Illinois 60115, USA
reza@ceet.niu.edu

Abstract:- This paper introduces a new technique for designing and decoding Huffman codes. The key idea is to define a Condensed Huffman Table (CHT) that is smaller than the ordinary Huffman Table and which leads to fast decoding. For example, the new approach has been shown to reduce the memory consumption by a factor of eight compared to the Single-Side Grown Huffman table [3].

1. Introduction: Huffman coding [1] has been shown to be one of the most efficient and simple variable-length coding techniques used in high speed data compression applications. Normally, the binary tree associated with a Huffman code becomes progressively *sparse* as it grows from the root. This sparsity usually causes tremendous waste of memory space, unless a properly structured technique 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 [3-14]. For example, Kuo-Liang Chung, and Yih-Kai Lin [13] use an array data structure to represent a Huffman tree. They report as low as $5n-4$ memory space, and $O(d)$ time for an average search, where n is the number of symbols and d is the dept of the tree.

To directly address the problem, one realizes that there are normally two distinct operations involved in Huffman coding: code-length determination, and code representation. The code-length determination is almost unique (except for some special cases) in every technique, and it was originally established by D.A. Huffman[1]. However, it is the coding representation that differs from one technique to another. And it is, in fact, this feature that affects both space requirement and the time for search [3]. A common property in any Huffman encoding is that the decoding procedure must recognize the code-

length as well as the symbol itself with no guard-bit(s) assigned to separate between the two consecutive symbols. Another property in a Huffman coding is that, within certain codes that have the same code-length, any reordering of the codes in the Huffman table does not change the coding effectiveness (entropy). Nevertheless, the difference may appear in terms of the memory requirement as well as the time for search.

The method presented here is based upon a variant of the standard Huffman code representation called ‘Single-Side Grown Huffman Table (SGHT)’ [3]. The unique features of this representation allows for significant lossless compression of the Huffman table. We first introduce the SGHT concept. The tree associated with a SGHT is called SGH-Tree.

2. Single-Side Growing Huffman Table: A SGHT represents a special Huffman table that is constructed from a Table of Code-lengths (TOCL). As shown by the example in Table 1, a TOCL provides the code-lengths of the ordered symbols, s_1, s_2, \dots, s_{18} , where the symbols are listed in descending order of their probabilities, as described in [1]. Algorithm 1 formulates a procedure that leads to the construction of a SGHT.

Table 1 - The Table of Code-lengths

| Code-length | 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} |

Algorithm 1: We start from a symbol, say s_1 , with shortest code-length L_0 . We assign a code-word of L_0 zeros to s_1 . Next, we increment this code-word by one and assign the new code to the next symbol in the list, still with L_0 code-length. We continue likewise to cover all symbols with identical code-lengths, L_0 . Next, we first increment the last code-word by one, and then append enough zeros to the right of the code-word until its size become equal to the code-length of the next symbol in the list. The rest of the process is to repeat these two operations until we get to the last symbol (s_{18}), which always contains “all ones”. In mathematical terms:

$$c_1 = 00 \dots 0,$$

$$c_{i+1} = (c_i + 1) * 2^{q-p},$$

And

$$c_n = 11 \dots 1,$$

Where, p and q are the code-lengths for the symbols s_i and s_{i+1} , respectively. Note that the initial and the final code-words have unique forms, which might help in error checking. Table 2 shows the SGHT developed from the TOCL (Table 1).

Table 2 - The Single-Side Grown Huffman Table

| Symbols | Code-words |
|-----------------|------------|
| 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 |

A SGHT has several properties that are essential in achieving our objectives. i) The value of the code-words in a SGHT are continuously increasing as we move down in the *symbol-list*, where, the symbol-list is the ordered list of symbols with descending probabilities (Table 2). This is also true even if the code-words are *augmented*, where each code-word is appending by enough zeros to the right until its code-length becomes equal to the maximum code-length L_m . ii) Given a code-length (level), the values of the code-words are only increasing linearly. This is also true even if the code-words in that level are augmented. In general, the entire augmented code-words in a SGHT form a piecewise linear characteristic. For example, Fig. 1 shows the characteristic curve for the augmented SGHT, shown in Table 2. iii) Any sub-tree of an SGH-Tree is an SGH-Tree as long as it terminates with leaf-nodes (symbols). iv) Finally, the first and the last code-words in a SGHT are trivially 00...0, and 11...1, respectively. In addition, it is shown, as depicted in Fig. 1, that the piecewise linear representation of a SGHT is always a concave graph, leading to some interesting properties that are outside the scope of this article.

Property ii) is particularly useful in this study, and it provides the basis for the construction of a *Condensed Huffman Table*.

3. Condensed Huffman Table: It is shown that a Condensed Huffman Table (CHT), generated from a SGHT, is nearly an order of magnitude smaller than the size of the original SGHT, yet it carries the entire property and the capability of the SGHT. In an

experiment conducted on fifteen full size images with different patterns, the average size of the SGHTs resulted in 96 rows, whereas the average size of the CHTs was 11.4 rows.

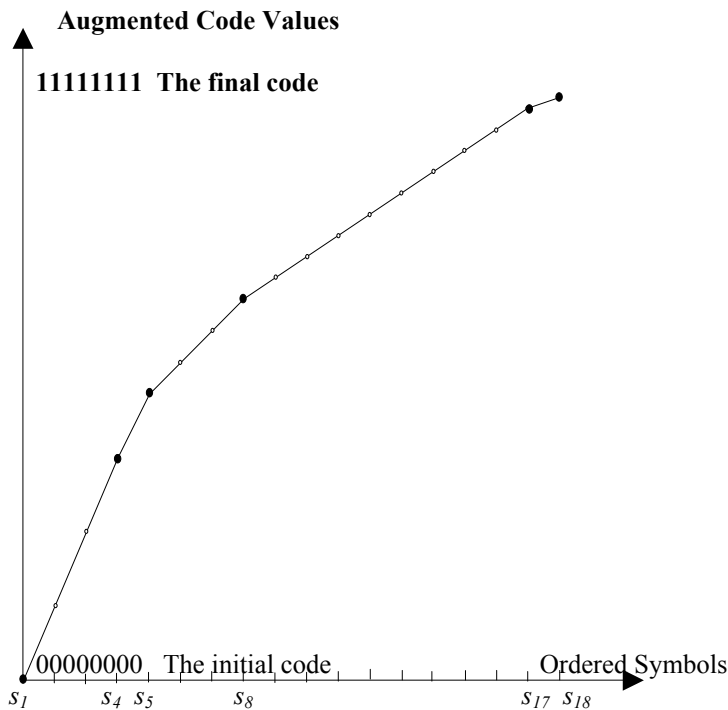


Fig. 1- The characteristic curve for the augmented code-words in Table 2

Theoretically, a CHT can have from 0 up to $n-1$ rows. A CHT with no row (empty) happens when the probabilities in the symbol-list change within less than 50%, resulting in a single code-length for every symbol. In general, for medium to larger symbol-lists, an estimated $l = 2 * \log_2(n) - 1$ rows can be expected for a CHT.

To construct a CHT we start from the second smallest code-length (there is no need to represent the code-words with smallest code-length in a CHT), and if there are multiple code-words in the SGHT with the same code-length, only the one with the smallest code-word value is recorded. In each row of the CHT we need to record i) the code-length L_i , ii) the position of the corresponding symbol in the *symbol-list*, n_i , and iii) the augmented code-word, C_i (see Table 3).

Table 3 is also directly related to the characteristic curve, Fig.1, where each row represents a line segment in Fig.1, except for the first segment that is missing. The coordinates for the starting point of each line segment appear in the second and forth columns, while the third column represents the slop.

Now, we are ready to decode an input bit-stream using a CHT.

Algorithm 2 – Take L_m bits from the bit-stream and call it W (in case L_m number of bits are not available in the stream, the procedure takes the entire stream and appends enough zeros to the right to have a code-length of L_m). Next, compare W to C_1 , the first (smallest) augmented code-word in the CHT. Note that the bit-stream is read from left (MSB) to right, and the operations always start from the MSB, as well.

1. If W is less than or equal to C_1 then the first L_0 bits in W represents the code-word, c_{0j} , for the intended symbol, where c_{0j} represents the code-word of the j th symbol in the first (0) level. Evidently, c_{0j} is also the address of the intended symbol, in the symbol-list (see *Algorithm 1*). Note that, in case the symbol-list starts from address 1, rather than 0, then we need to get $c_{0j} + 1$ as the correct address. Finally, return the excess, $L_m - L_0$ bits into the bit-stream.

2. If, on the other hand, W is greater than C_1 , continue searching for greater augmented code-words in the CHT. Let C_{i+1} be the first such augmented code-word that is greater than W , then the pertinent code-length is L_i and the code-word is the first L_i bits in W . Return the remainder $L_m - L_i$ bits of W into the bit-stream for the next process. To find the address of the symbol in the symbol-list, first find $D = W - C_i$, and take the first L_i bits in D , denoted by m_i . Then the address is given by $n_{ij} = m_i + n_i$, where n_{ij} refers to the j th symbol in the $i+1$ st level, and n_i is specified in Table 3.

3. Finally, if W is greater than C_1 and also greater than all other augmented code-words in the CHT, including the last one, C_m , then W itself must be the intended code-word. Find $m_m = W - C_m$, and then find $m_m + n_m$, which is the address of the symbol in the symbol-list.

Example: Take the CHT in Table 3 for the SGHT shown in Table 2, where $L_0 = 2$ and $L_m = 8$. Note that the Augmented Code-words are reported in Hexadecimal numbers.

Table 3 - The CHT for the SGHT shown in Table 2

| | Augmented Code-word C | Code-length L | Symbol Position n |
|---|-----------------------|---------------|-------------------|
| 1 | c0 | 3 | 4 |
| 2 | e0 | 6 | 5 |
| 3 | ec | 7 | 8 |
| 4 | fe | 8 | 17 |

Consider an input stream $CS = 0111110101111111$. First take $L_m = 8$ bits from the bit-stream and get $W = 01111101 = 7d$. This code is smaller than $C_1 = c0$, hence take $L_0 = 2$

MSBs from W as the code-word for the symbol $c_{0l} = 01$, and return the rest of the bits in W to the bit-stream. Next, use $c_{0l} + 1 = 2$ as the address to identify the symbol s_2 in the symbol list. Next take another $L_m = 8$ bits from the bit-stream (11110101111111) and get $W = 11110101 = f5$. This code is larger than $C_l = c0$, and the next larger augmented code in Table 3 is fe , in row 4. We take ec in row 3, where $L_3 = 7$. Next, we take 7 MSBs of W which gives the code-word $c_{3j} = 1111010$. To find the address for this symbol we need to find $D = f5 - ec = 09$, drop its 8th bit to get 04. Then add this number to the symbol position $n_3 = 8$ to get 12, and the symbol is found to be s_{12} . Finally, the last 8-bit code from the bit-stream is $W = 11111111 = ff$. This code is greater the $C_m = fe$, and hence it is indeed the intended code-word with $m_j = ff - fe = 1$. The address of the symbol is given as $n_{m,j} = 17 + 1 = 18$, which is the address for s_{18} .

4. Memory Requirement And Access Time – A Comparison: For a performance evaluation we compare the proposed method with the decoding method, reported in [3]. We only take two criteria, the memory requirement, and the symbol access time, into consideration. Other criteria such as processing hardware, and error protection are not included in this study. For this comparison we have experimented with fifteen regular size images with different patterns. Through a series of DCT transformations, quantizations, Huffman/Run-length coding an average of $n = 96$ symbols are generated for this set of experiments, while the average size (rows) of CHTs is reported as 11.4. The results are as follows:

| <u>Method</u> | <u>Memory Consumption</u> | <u>Symbol access time</u> |
|---------------|-----------------------------|---------------------------------|
| SGHT: | $12 * 96 = 1152$ Bytes | 1.6 add/subt. |
| CHT: | $96 + 4 * 11.4 = 142$ Bytes | 0.5 add/subt. + one step search |

As expected, the proposed procedure is quite efficient in memory usage. The delay can also be reduced with appropriately designed hardware. In general, for n symbols, and l active levels (code-lengths) in the TOCL the total memory space required for decoding is

$$M = n + 4l \cong 1.6n$$

Typically for more than 50% to 80% of time the search for finding a symbol will be resolved in the first step, in Algorithm 2, and there is no need to perform any addition or subtraction of data. For example, symbols s_1 , s_2 , and s_3 in our example, cover about 75% of

the total search, and no add/subtract is needed to allocate these symbols. So, if we neglect the time for shifting, the average search time for decoding a symbol is approximately

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

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 original Huffman Table, and hence the memory is significantly reduced. In addition, the decoding is also faster because of shorter searches.

- [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 11 - 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] Kuo-Liang Chung, and Yih-Kai Lin, "A Novel Memory-Efficient Huffman Decoding Algorithm And Its Implementation," *Signal Processing.*, vol, 62, ELSEVIER, pp. 207-213, 1997.
- [14] S. Roman, *Coding and Information Theory*. New York: SpringerVerlag, 1992.