High Speed Search and Memory Efficient Huffman Coding

Reza Hashemian, *Senior Member IEEE*
Northern Illinois University
Electrical Engineering Department
DeKalb, Illinois 60115-2854

Abstract- Code compression is a key element in high speed digital data transport. A major compression is performed by converting the fixed-length codes to variable-length codes through a (semi-)entropy coding scheme. Huffman coding combined with run-length coding is shown to be a very efficient coding scheme. To speed up the process of search for a symbol in a Huffman tree and to reduce the memory size we have proposed a tree clustering algorithm to avoid high sparsity of the tree. The method is shown to be very efficient in memory size, and fast in searching for the symbol. For an experimental video data with Huffman codes extended up to 13 bits in length, the entire memory space is shown to be 126 words, compared to normal $2^{13} = 8192$ words.

I. INTRODUCTION

High definition television (HDTV) is a progressing medium taking speed both in broadcasting and high quality commercial video applications. The effective use of HDTV depends much on the efficiency of the digital video signal transport as well as the storage requirement for such data[1]. Both these objectives are majorly fulfilled by coding the signals and using compression techniques to reduce the storage space and obtain higher transmission efficiency. With combined Vector Quantization techniques, DCT, Run-length coding, and (semi-)entropy coding techniques a relatively high code compression, close to 3.5 Mbps or lower bit rates, for high quality video signal transmission has been achieved in commercial applications.

The combination of Huffman coding and run-length coding has been shown to perform very efficiently in high speed data compression applications[1,4]. In fact, with some variations, this combined technique has been widely used as a near optimal entropy coding technique. For maximum compression the coded data is normally sent in a continuous stream of bits with no separation between the codes of two consecutive symbols. Thus, decoding in this case must also recognize the code length as well as the symbol itself.

In its simplest form Huffman coding[2,3] may structurally be represented by a binary tree. Due to variable-length coding, however, the Huffman tree gets progressively sparse as it grows from the root. This sparsity in the tree structure may also result in a lengthy search procedure for locating a symbol. In addition, if *p-bit* is the longest Huffman code assigned, then the memory would be 2^p words in size, which becomes very large for typical video applications such as $p=13$ or higher.

To achieve a high speed search for a symbol, and to reduce the memory requirement for the Huffman tree structure we propose a technique that allows an ordering of the variable-length codes based on their code length. It is shown that, such ordering serves two major purposes: i) the searching time for more frequent symbols (shorter codes) is substantially reduced compare to less frequent symbols, resulting in reduction in the overall search delay. ii) The growth of the Huffman tree is directed toward one side only, and hence the sparsity of the tree is better controlled, and as a consequence, a drastic reduction in the memory space is resulted.

In Section II we start from a source listing (Histogram) and generate a *Table of Code-word Lengths* (TOCL). This TOCL then leads to the development of an special Huffman tree called *Single-side Growing Huffman Tree (SGH-Tree)*. In Section III we develop a method for clustering the SGH-Tree and later an addressing scheme is presented for reduced memory allocation. Huffman decoding is carried out in Section IV through several examples, and finally the Conclusion is given in Section V.

0-7803-1254-6/93$03.00 © 1993 IEEE

II. VARIABLE-LENGTH CODE STRUCTURING

We first consider a fixed-length source code. A source H is defined as an ordered pair H = (S,P); where, S represents a set of source symbols $S = \{s_1, s_2, ..., s_n\}$ with probability distribution $P(s_i) = p_i$; for, $p_1 >= p_2 >= ... >= p_{n-1} >= p_n$. Given a source listing H, the following Algorithm is applied to generate the TOCL for H:

A. *Algorithm 1: Table of Code-word Length*

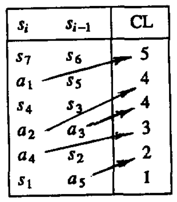
i) A procedure similar to the initial stage in a successive Huffman table reduction and reordering [2] is carried out. First the pair of symbols at the bottom of the ordered list are merged (ensembled) and instead a new symbol a_1 is created (see Fig.1). The symbol a_1, with the probability q_1, is then located at the proper location in the ordered list. To make a progressive record of this operation a *Code-word Length Recording* (CLR) table is created which consists of three columns: columns 1 and 2 hold the initial pair of symbols before merge and column 3, initially empty, will contain the *Code-word Length* (CL) for the symbols in that row, as will be discussed shortly (see Fig.2). In order to reduce the memory requirement for the CLR table, and also to speed up the process, the new symbol a_1 (in general a_j) is selected such that its inverse $\overline{a_1}$ (or $\overline{a_j}$) represents the associated table address. For example, a_1, referring to the first row in the CLR table, is designated by 11111110, for an 8-bit address word, and a_2 = 11111101, and so on. Another property of this selection of composite symbol is the sign difference between the composite symbol (a_j) and that of the original symbol (s_i); where, the later is assumed to be positive integer. In addition, it is shown that the choice of selecting a_j as the inverse of the table address significantly simplifies the hardware, which is the subject of another discussion [5].

Table 1

s_1	48	s_1	48	s_1	48	s_1	48	s_1	48	a_5	52
s_2	31	s_2	31	s_2	31	s_2	31	s_2	31	s_1	48
s_3	7	s_3	7	a_2	8	a_3	13	a_4	21		
s_4	6	s_4	6	s_3	7	a_2	8				
s_5	5	s_5	5	s_4	6						
s_6	2	a_1	3								
s_7	1										

Fig.1- Reduction process in the source list.

Table 2



s_i	s_{i-1}	CL
s_7	s_6	5
a_1	s_5	4
s_4	s_3	4
a_2	a_3	3
a_4	s_2	2
s_1	a_5	1

Fig.2- Table of CL-Recording.

Table 3

CL	symbols	Efficiency %
1	S_1	100.
2	S_2	75.
3		50.
4	S_3 , S_4 , S_5	37.
5	S_6 , S_7	12.

Fig.3- Table of Code-word Lengths.

ii) In step i) we succeeded in reducing the source listing (Table 1) by one row and instead we added one row to the, newly created, CLR table. Here we continue the same procedure, reducing Table 1, one row at a time, and growing the CLR table accordingly, until no symbol (row) is left in Table 1. To simplify our discussion we continue the remaider of the procedure through a simple example.

Let Table 1, in Fig.1, represent an ordered source with the probability distribution (in terms of per cent) shown. Through steps i and ii we shrink Table 1 until it becomes empty and, instead, we generate the corresponding CLR table (Table 2), as shown in Fig.2. Note that Table 2 contains both; the original symbols s_i, and the composite symbols a_j. However, due to the specific method of selecting a_j, the later symbols are different from the former ones either in the sign (for sign-integer numbers) or in the MSB (for unsigned-integer numbers). Also notice that the third column in Table 2 is initially empty, lefted to be filled up as we proceed.

iii) The third column in Table 2 designates the CL for the symbols in that row. We start from the last row in Table 2 and assign 1 to the third column, as the CL for the pair of symbols in that row. Next, we check the sign (or the MSB) of each symbol in this row; if positive (or the MSB is 0) we skip; otherwise, the symbol a_j is of the composite type and its binary inverse $\overline{a_j}$ is the address to a row in Table 2. With row $\overline{a_j}$ allocated in Table 2, we then increment CL and assign the new value to the CL in row $\overline{a_j}$. This operation is continued until the third column of Table 2 is completely filled up with CL values, as shown in Fig.2. Note that the arrows, shown in the figure, specify the addressing direction in the table. For example row 4, in this table, contains a_2, a_3, and 3 as the CL for the (composite) symbols. Here a_2 (or in fact $\overline{a_2}$) refers to the second row in the table and assigns CL=3+1=4 to this row, and a_3 refers to row 3 in the table and also assigns CL=4 to this row.

vi) A close look at Table 2, in Fig.2, indicates that each (original) symbol in the table has its code-word length (CL) specified. Thus, the only task remains is to order the symbols according to their CL. Table 3, in Fig.3, is the result of this ordering and it is designated as the Table of Code-word Length (TOCL) for the source listing given in Table 1, Fig.1.

B. Single-side Growing Huffman Table

For a given source listing H Algorithm 1 generates a TOCL for the symbols in the source. Associated with this TOCL there are evidently a number of Huffman tables (Huffman trees); differing in code-words for a symbol but identical in the code-word lengths. Among these Huffman tables, however, we have selected a specific table, called *Single-side Growing Huffman Table* (SGHT), or equivalently *Single-side Growing Huffman Tree* (SGH-Tree). A SGHT has always a starting code of 00...0 for the lowest CL and it grows to a final code 11...1 for the highest value of CL. The SGHT for our example, with TOCL given in Fig.3, is shown in Fig.4, and its corresponding SGH-Tree is given in Fig.5.

In what follows we try to develop a simple algorithm to construct a SGHT from a given TOCL. Again, to simplify the procedure we follow an example as we proceed.

C. Algorithm 2: Development of SGHT

For a given TOCL, such as the one shown in Fig.3, we start from the first row and assign the code-word c_1 = 00...0 to the symbol s_1; where the number of zeros in c_1 is equal to the value of CL for s_1 (in our example c_1 = 0). We next increment the code-word and assign the new value to the next symbol in the table, and keep creating code-words. However, if the symbols in each row are exhausted the procedure still continues to the next row down, except that an extra 0 will be positioned to the right of the code-word, whenever a change of row is encountered. For example, to find the code-word for s_2, in Fig.3, we first increment c_1 and then position one 0 to its right to get c_2 = 10 (see Fig.4). The same is true for s_3; in this case c_2 is first incremented and then two zeros are positioned to the right (one for each level down) to get c_3 = 1100, and so on. It is interesting to note that the final code-word, in this procedure, is always in the form of 11...1 (with no 0s), and no other code-word has this code structure. This property allows us to verify the code-word generation procedure, error checking, and to recognize the final code-word in the Huffman table.

Table 4

CL	symbols	Huffman Code
1	s_1	0
2	s_2	10
4	s_3	1100
4	s_4	1101
4	s_5	1110
5	s_6	11110
5	s_7	11111

Fig.4- Single-side Growing Huffman Table.

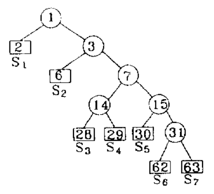


Fig.5 A 5-level SGH-Tree.

III. STORAGE ALLOCATION

In a uniformly (flat) distributed source; where $p_i = p_j$, for all i and j, all symbols have identical code-word length and CL = $\log_2$ (n+e); where n is the number of symbols in the source, and e is the smallest positive integer to make CL an integer. Evidently the SGH-Tree in this specific case becomes *full* with all its *leaf nodes* hanging at the bottom of the tree (we assume e = 0, for simplicity). This is a zero sparse tree, in the sense that all positions (nodes) at the CLth level are filled. Or, in terms of memory allocation, an n-word memory (table) addressable by CL-bit word is sufficient to allocate any symbol without any *waste*.

This, as we know, is a very special case and in fact this is not a proper source for coding. In almost all other cases the source distribution is non-uniform (no flat histogram). In this situation depending on the probabilities the code-word length for different symbols may differ, and hence, the leaf nodes in the SGH-Tree are distributed within different levels in the tree, as shown in Fig.5. An optimal storage allocation as well as a quick access to the symbol (data) is by no means a strait forward operation in this case []. A simple and rather quick access to the data (symbol) is to allocate 2^p number of memory locations to Huffman table; where p is the highest code-word length in the table. In this environment the code-word (extended with zeros, if necessary) is used as an address to the Huffman table to access the symbol. This evidently is not an efficient method of storing the Huffman table, not at least from memory requirement view point. Experiments with real data indicate that the SGH-Tree becomes very sparse as the tree grows in number of levels (number of bits in SGHT), and hence, the *memory efficiency* becomes poor. For example, for some experimental video data (see the example below) the SGH-Tree grows up to 13th level for a source of 32 independent symbols. This indicates that of 2^{13} = 8192 possible locations only 32 positions carry information. In other words, the memory efficiency has reduced to 0.39 per cent.

The method presented here tries to break down the SGH-Tree into smaller *clusters* (sub-trees) such that the memory efficiency for each cluster is substantially increased compared to that of the SGH-Tree itself. Before any further discussion we define the *memory efficiency k* for a p level tree as:

$$k = \frac{Existing\ number\ of\ symbols\ *\ 100}{2^p}\ \%$$

As an example, the memory efficiency for a 5 levels Huffman tree is listed in Table 3 (Fig.3). Notice that the efficiency changes only when we switch to a new level (or equivalently to a new CL), and it decreases as we go to the higher levels, down in the tree structure. As another example, we have taken an experimental video data and have created the SGH-Tree, shown in Fig.6 and its corresponding SGHT shown in Fig.7, by using Algorithms 1 and 2 discussed earlier in this article. The memory efficiency, in this example, starts from 100% for the first level and it is reduced to 0.39% for the 13th level. Higher memory efficiency for the lower

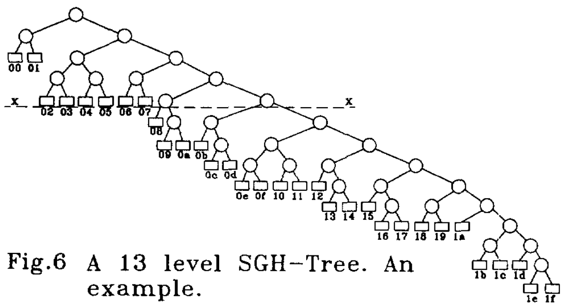


Fig.6 A 13 level SGH-Tree. An example.

Table 4

CL	Symbols	Huffman code
2	0	00
2	1	01
4	2	1000
4	3	1001
4	4	1010
4	5	1011
4	6	1100
4	7	1101
5	8	11100
6	9	111010
6	a	111011
6	b	111100
7	c	1111010
7	d	1111011
8	e	11111000
8	f	11111001
8	10	11111010
8	11	11111011
8	12	11111100
9	13	111111010
9	14	111111011
9	15	111111100
10	16	1111111010
10	17	1111111011
10	18	1111111100
10	19	1111111101
10	1a	1111111110
12	1b	111111111100
12	1c	111111111101
12	1d	111111111110
13	1e	1111111111110
13	1f	1111111111111

Fig.7 Single-side Growing Huffman Table.

levels in a SGH-Tree structure is a clear indication that partitioning the tree into smaller and less sparse clusters will reduce the memory requirement in a Huffman coding procedure. Here in this article, we explain a new method for tree clustering leading to an efficient memory management for Huffman coding.

A. SGH-Tree clustering

Given a SGH-Tree, as shown in Fig.6 (or equivalently a SGHT, as shown in Fig.7), depending on the *memory efficiency* requirement, the tree is cut by a *cut-line*, x-x, at the ith level (i = 4 in our example). The first cluster (subtree) is formed by removing the remainder of the tree beyond the cut-line x-x, as shown in Fig.8(a). Associated with this cluster we assign a look-up table, also shown in Fig.8(a); where, the entries in the table reveal the status of the terminal nodes (*leaf nodes*) within the cluster. The procedure for searching a symbol is as follows: a) an i-bit size Huffman code represents the address to the look-up table, and b) the stored data in the table indicates whether the corresponding symbol is already found, or a further search has to be followed in order to allocate the symbol. More clearly, a positive integer (MSB = 0) in the table indicates that the node is a leaf node, either on the cut-line or within the cluster, terminated before being cut by the cut-line. on the other hand, a negative integer (MSB = 1) in the table corresponds to a *branching node*, meaning that the Huffman code is to be found not in this cluster but in the extended part. A close look at this table reveals the followings:

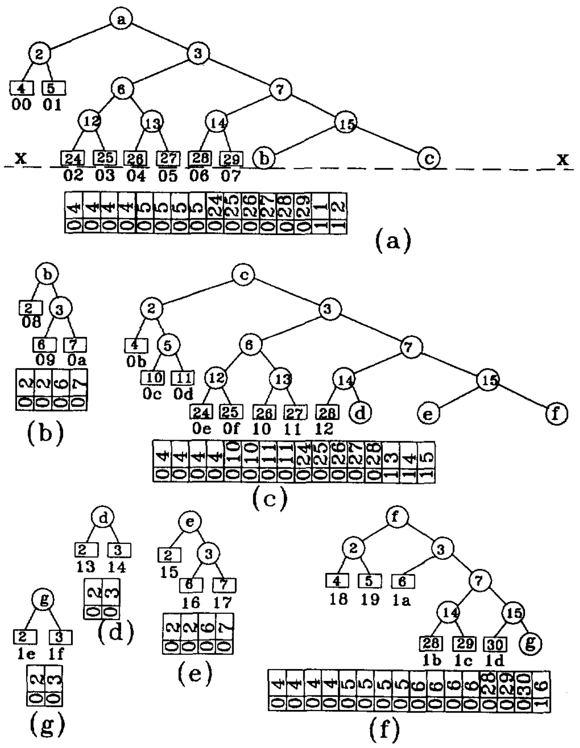


Fig.8 The SGH-Tree clusters and their corresponding look-up tables.

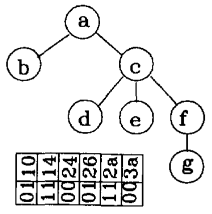


Fig.9 The S-Tree and its associated ST.

i) A positive integer in the table carry three pieces of information. 1) It identifies the leaf node number in the tree, 2) it provides the address of the symbol in the memory, which is obtained by removing the most significant bit 1 (excluding the sign bit) from the data, and 3) it provides the associated code-word length (by counting the significant bits following the most significant bit 1 in the data).

ii) A negative number, on the other hand, provides the address to another look-up table assigned to the higher order tree (called super tree), as will be discussed shortly (see Fig.9).

Evidently, this clustering technique could also be applied to the rest of the SGH-Tree each starting with a *root node.* With similar procedures we generate other clusters and their look-up tables from the original tree until the tree is exhausted. Figures 8(a) to (g) provide such clusters for the entire 13 level SGH-Tree, shown in Fig.6. Finally, a higher order tree, called *Super Tree* (S-Tree), is developed to represent the connectivity among the clusters. In a S-Tree each node represents a cluster, and the branches connecting the nodes indicate a sharing node between the two clusters. Here we also associate to a S-Tree a look-up table, called *Super Table* (ST). Figure 9 shows both, the S-Tree and its ST, for our example of 13 level tree structure. Each location within the ST caries the information about one cluster, excluding the first cluster. This information is in two parts: the first two (or more) bits gives the maximum number of levels existed in that cluster, and the rest of the data gives the beginning address of the memory, containing the symbols associated with that cluster. We shall clarify this as we discuss it in the *Huffman decoding* section.

A simple calculation reveals that by applying the method just described the total memory allocation for Huffman coding is drastically reduced. For example, given the 13-level Huffman tree example, this memory, including the look-up tables and the memory to store the symbols, is reduced to 126 words compared to $2^{13} = 8192$ words, in a normal (un-processed) situation.

In addition to the memory space reduction, as discussed earlier, the time to search for a symbol is also reduced in this method by two main factors; first, her we deal with very small look-up tables and memory blocks, and this substantially reduces the memory access time (one may use an internal RAM rather than an external one). Second, through an exact grouping of the bits in a code-word, it has been possible to sweep a quick path to the symbol without being delayed by some feed-backs, being used in some other methods. We shall discuss more about the access time as we proceed into the Huffman decoding procedure.

IV. HUFFMAN DECODING

We describe the decoding procedure through some examples going through the SGHT structure.

1) Consider a steam of binary Huffman code 011001011..., where the MSB enters first and other bits later. We use the first four bits 0110 = 6 as an address to the look-up table given in Fig.8(a). The content of the table shows 5 = 101b. This indicates that the symbol, 01H, is found and it is located in memory (RAM) address 01b (with leading 1 in 101b removed). Next, we place a 1 bit to the left of the code, 0110, to get 10110, and we match this code (from left to right) with the content of the table, i.e. 5 = 101b, which results in selecting 101b from 10110. This indicates that the Huffman code for the symbol is 01b, with CL=2.

2) Next assume a second bit stream as 111110001.... As usual, we take the first four bits 1111 and refer to the table in Fig 8(a). At address 1111b = fH, we find the code 1, meaning that the symbol is to be found outside this table. In the second part (the same location in the table) we find a pointer value 2 referring to: i) the cluster c (as the second cluster, excluding the main one), shown in Fig.8(c), and ii) the second location in the ST table, given in Fig.9. Referring to location 2 in the ST (Fig.9) we get two numbers: one 11b which indicates that the next cluster (c) has 11b+1b=100b levels, and thus we need 4 more code-bits from the stream, i.e., 1000b=8H, to search for the symbol. The second portion of data, i.e., 14H denotes the beginning address for the cluster c. In conclusion, following the same procedure explained in 1), we obtain the symbol in the cluster c as 0eH, located at 14H+8H=1cH in the memory and the Huffman code in this case is 11111000b=f8H.

In general, we conclude that for high probable symbols with short code-words (4 bits or less) the search for the symbol is very fast. For longer code-words, however, the search time grows almost proportional to the code-word length. In specific, if i is the maximum level for the clusters (i=4 in our example) then the search time is closely proportional to *1+CL/i.*

V. CONCLUSION

In conclusion, we claim high efficiency in memory space as well as high speed access to the symbols in a code de-compression scheme using the Huffman technique. The means to achieve this efficiency, as proposed, are to; i) avoid the sparsity of the tree structure by grouping nodes in clusters, and ii) use combined code bits to search for the symbol in smaller look-up tables.

ACKNOWLEDGEMENT

The author wish to thank Shahrokh. Eghbali, and Kumar. Golla for their contributions in simulating the algorithms.

REFERENCES

[1]. A.K. Jain, "Image Data Compression: A Review," *Proc. IEEE,* vol. 69, no.3, pp. 349-389, Mar. 1981.

[2]. D.A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE,* vol. 40, no. 10, pp. 1098-1101, Sept. 1952.

[3]. S. Roman, *Coding and Information Theory,* Springer-Verlag, New York, 1992.

[4]. S.M. Sun, M.T Sun, and T.H. Tzou, "Design and Hardware Architecture of High-Order Conditional Entropy Coding for Image," *IEEE Trans. Circuit Syst. Video Tech.,* vol. 2, no. 2, pp. 176-186, June 1992.

[5]. R. Hashemian, "Algorithm Development and Hardware Design of a Memory Efficient Huffman Encoding," to be submitted for publication.