Memory Efficient and High-Speed Search Huffman Coding

Reza Hashemian, *Senior Member, IEEE*

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 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 extremely efficient in memory requirement, 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 122 words, compared to $2^{13} = 8192$ words in a normal situation.**

I. Introduction

HIGH-definition television (HDTV) is a developing medium taking root both in broadcasting and high-quality commercial video applications [1], [2]. 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 [3]. Both these objectives are substantially fulfilled by coding the signals and using compression techniques to achieve higher transmission efficiency and reduced storage space [4], [5]. In general, combinations of vector quantization techniques, DCT, Run-length coding, and (semi-)entropy coding techniques is shown to provide relatively high code compression, close to 3.5 Mb/s or lower bit rates, for high-quality video signal transmissions, applied in commercial applications.

The combination of Huffman coding and run-length coding has been shown to perform efficiently in high-speed data compression [6]–[10]. 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 through a continuous stream of bits with no specific guard-bit(s) assigned to separate between two consecutive symbols. As a result, decoding procedure in this case must recognize the code length as well as the symbol itself.

In its simplest form Huffman coding [6], [12] 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 Huffman tree may cause tremendous waste of memory space, unless a properly structured technique is adopted to allocate the symbols in the memory. In addition, this sparsity may also result in a lengthy search procedure for locating a symbol. More specifically, if k-bit is the longest Huffman code assigned to a set of symbols, the memory size for the symbols may easily reach 2^k words in size, in an unstructured memory environment. This evidently becomes prohibitively large for typical video data, where k can be 13 or higher. Ideally, it is desirable to reduce the memory size from typical value of 2^k, to a size proportional to the number of the actual symbols. This is not only a substantial reduction in memory requirement, but it may even allow us to replace a sizable external RAM by a much smaller memory, possibly inside the processing chip, for quicker access.

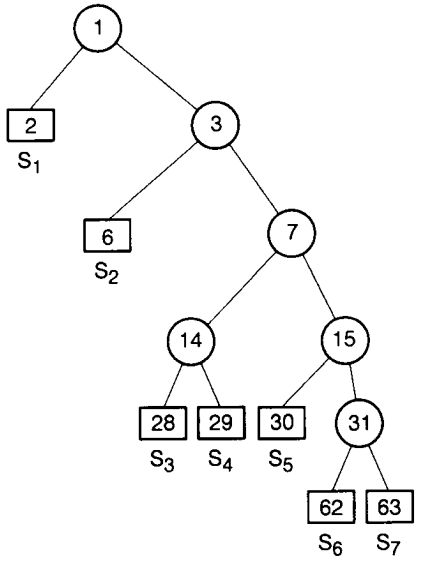


Fig. 1. A 5-level SGH-tree.

The proposed algorithm addresses both these issues, i.e., the reduction of the storage space, and achieving high-speed search for symbols. The algorithm is based on an ordering and *clustering* scheme that groups the codewords (tree nodes) within specified codeword lengths [13]. It is shown that, such ordering and clustering serves three major purposes:

1) The search time for more frequent symbols (shorter codes) is substantially reduced compare to less frequent symbols, resulting in an overall faster response.
2) For long codewords the search for the symbol is also speeded up. This is achieved through a specific partitioning technique that groups the code bits in a codeword, and the search for a symbol is conducted by jumping over the groups of bits rather than going through the bits individually.
3) The growth of the Huffman tree is directed toward one side of the tree, as shown in Fig. 1, and hence the

Paper approved by F. Marvasti, the Editor for Data Communications of the IEEE Communications Society. Manuscript received April 12, 1993; revised March 28, 1994 and June 23, 1994.

The author is with the Department of Electrical Engineering, Northern Illinois University, DeKalb, IL 60115 USA.

IEEE Log Number 9413820.

0090–6778/95$04.00 © 1995 IEEE

TABLE I
REDUCTION PROCESS IN THE SOURCE LIST

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										

sparsity of the tree is better controlled. This also results in a substantial reduction in the memory space.

In Section II we start from a source listing (Histogram) and generate a *table of codeword lengths* (TOCL) for the symbols. This TOCL then leads to the development of an special Huffman tree called *single-side growing Huffman Tree (SGH-Tree)* with certain properties. In Section III we develop a method for partitioning the tree and clustering the nodes in order to reduce the sparsity. Later in the section an addressing scheme is presented for further reduction of the memory. Huffman decoding is carried out in Section IV. For a given continuous stream of codewords (code-bits), a step by step search operation is discussed to allocate a symbol. It is shown through examples that different codeword lengths require going through different stages of search; and the proposed search is optimized for more frequent symbols. Finally the Conclusion is given in Section V.

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 $S = \{s_1, s_2, \cdots, s_n\}$ with probability distribution $P(s_i) = p_i$; for, $p_1 \geq p_2 \geq \cdots \geq p_{n-1} \geq p_n$ [12]. Next, for a given source listing H, we generate a *table of codeword lengths* (TOCL) using Algorithm 1. It is important to note that such a table is uniquely obtained within permutation of the symbols in each row. In other words, for a given source listing H the TOCL uniquely groups the symbols into blocks, where each block is specified by its codeword length (CL). And each block of symbols, so defined, occupies one *level* in the associated Huffman tree.

A. Algorithm 1: Table of Codeword Lengths

There are four steps involved in creating a TOCL.

1) A procedure similar to the initial stage in a successive Huffman table reduction and reordering [6] is carried out in this step. Initially, the symbols are listed with the probabilities in descending order (the ordering of the symbols with equal probabilities is assumed indifferent, at this point). Next, the pair of symbols at the bottom of the ordered list are merged and as a result a new symbol a_1 is created (Table I). The symbol a_1, with probability equal to sum of the probabilities of the pair, is then inserted at the proper location in the ordered list. To record this operation a *codeword length recording* (CLR) table is created which consists of three columns: columns 1 and 2 hold the last pair of symbols before being merged, and column 3, initially empty, is identified as the *codeword length* (CL) column (Table II). In order to make the size of the CLR table small and the hardware design simpler, 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 given the value of 11111110, for an 8-bit address word, and $a_2 = 11111101$, and so on. Another distinction between such a composite symbol and an original symbol is their difference in sign. This is important from design stand point, because, a dedicated sign (MSB) detector is all that is needed to distinguish between the two. In addition, it is shown that the choice of selecting a_j as the inverse of the table address further simplifies the addressing scheme, which is the subject of another discussion [14].

TABLE II
TABLE OF CL-RECORDING

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

2) We next continue applying the same procedure, developed for a single row in step i), and construct the entire CLR table, as shown in Table II. Note that Table II contains both the original symbols s_i and the composite ones a_j (carrying opposite signs).
3) The third column in Table II, designated by CL, is assigned to hold the codeword lengths. To fill up this column we start from the last row in the CLR and enter 1. This designates the codeword length for both s_1 and a_5. Next, we check for the signs of each s_1 and a_5; if positive (MSB = 0) we skip, otherwise, the symbol is a composite one, a_j (a_5 in this case), and its binary inverse $\overline{a}_j$ ($\overline{a}_5 = 00 \cdots 0101$) is a row address for Table II. We now increment the number in the CL column, and assign the new value (i.e., 2, in this example) to the CL column in row 5 ($00 \cdots 0101$), and proceed applying the same operation to other rows in the CLR table, as we move to the top, until the CL column is completely filled. Note that the arrows in the table specify the addressing directions. For example, 4th row in Table II contains a_2, a_3, and 3. Here, a_2 (or in fact $\overline{a}_2 = 00 \cdots 010$) refers to the second row in the table, and hence we assign CL $= 3 + 1 = 4$ to this row. Composite symbol a_3, on the other hand, refers to row 3 in the table, and thus we assign the same CL = 4 to row 3.
4) A close look at Table II indicates that each original symbol in the table has its codeword length (CL) specified. Hence, it only remains to order the symbols according to their CL values. Table III is the result of this ordering

TABLE III
TABLE OF CODEWORD LENGTHS (TOCL)

CL	symbols
1	S_1
2	S_2
3	
4	S_3, S_4, S_5
5	S_6, S_7

TABLE IV
SINGLE-SIDE GROWING HUFFMAN TABLE (SGHT)

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

and is identified as the *table of codeword length* (TOCL) for the source listing given in Table I.

B. Single-Side Growing Huffman Table

For a source listing H, given in Table I, *Algorithm* 1 generates a TOCL for the symbols, as described earlier. Associated with this TOCL one can actually generate a number of Huffman tables (or Huffman trees), each of which being different in codewords but identical in the codeword lengths. Among different Huffman tables we select a particular one, called *single-side growing Huffman table* (SGHT), and equivalently, *single-side growing Huffman Tree* (SGH-Tree). Table IV and Fig. 1 represent the SGHT and the SGH-Tree for the source listing in Table I, respectively.

C. Algorithm 2: Development of a SGHT

To construct a SGHT from a TOCL, such as the one shown in Table III, we start from the first row of the table and assign an "all zero" codeword $c_1 = 00 \cdots 0$ to the symbol s_1. Next we increment this codeword and assign the new value to the next symbol in the table. Similarly, we proceed creating codewords for the rest of the symbols in the same row of the TOCL. When we change rows, however, we have to expand the last codeword, after being incremented, by placing extra zeros to the right, until the codeword length matches the level (CL). In general we can write:

$$c_1 = 0\,0 \cdots 0,$$
$$c_{i+1} = (c_1 + 1) * 2^{q-p},$$

and

$$c_n = 1\,1 \cdots 1$$

TABLE V
TABLE OF MEMORY EFFICIENCIES

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	22.

where p and q are the codeword lengths for s_i and s_{i+1}, respectively, and s_n (associated with c_n) denotes the terminal symbol.

Note that the initial and the final codewords have unique forms which makes them special, and makes it easy to verify the codeword generation and implementation procedures.

III. STORAGE ALLOCATION

In a uniformly distributed (flat) source; where $p_1 = p_j$, for all i and j, the symbols have identical codeword lengths and CL = $\log_2(n)$; where n is the number of symbols. Evidently the SGH-Tree becomes *full* for this case with all its *leaf nodes* hanging at the bottom of the tree. This is defined as a zero sparse tree, in the sense that all positions (nodes) up to a particular level (CL) are filled up. In terms of memory allocation, an n-word memory (table), addressable by a CL-bit code, is sufficient to allocate each and every symbol without any *waste*.

It is evident that we do not gain any compression by coding this type of source. In addition, this type of distribution is a very rare and special case, where any symbol is equally likely. In most practical cases, however, the source is not uniformly distributed, and hence, the SGH-Tree become sparse, as illustrated in Fig. 1. Our aim here is to consider the nonuniform case, and try to 1) optimize the storage space, and 2) provide quick access to the symbols (data). Experiments with real data indicate that the SGH-Tree becomes more sparse as the tree grows in the number of levels, and as a result, the *memory efficiency* becomes a serious issue in cases of high density codewords. For some experimental video data, for example, the SGH-Tree grows up to 13th level for a source of 32 independent symbols. This indicates that of $2^{13} = 8192$ possible locations for a normal look-up table only 32 positions carry the real information. In other words, the memory efficiency has been reduced to as low as $(32/8192) * 100\%$, or 0.39%, in this case.

The method presented in this article breaks down the SGH-Tree into smaller *clusters* (subtrees) such that the memory efficiency increases. Before any further discussion we define the *memory efficiency* β for a k level binary Huffman tree (or subtree) as:

$$\beta_k = \frac{\text{Number of symbols} * 100}{2^k}\%.$$

As an example, the memory efficiency up to each level in a 5 level Huffman tree is listed in Table V. Notice that the efficiency changes only when we switch to a new level (or equivalently to a new CL), and it decreases as we proceed to the higher levels.

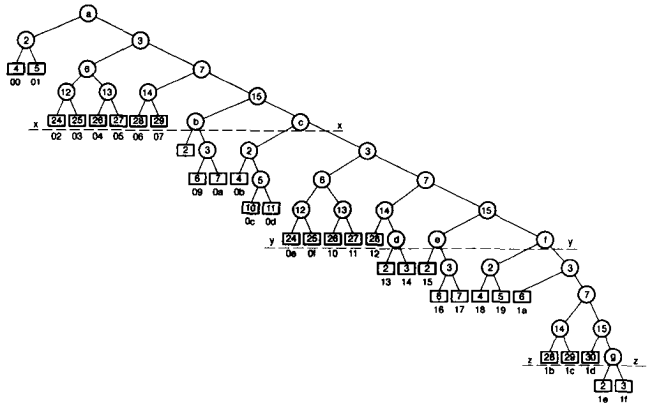


Fig. 2. A 13-level SGH-tree.

Memory efficiency can be interpreted as a measure of the performance of the system in terms of memory space requirement; and it is directly related to the sparsity of the Huffman tree. For a multi-level Huffman tree the efficiency starts from 100%, for the first level, and declines as we proceed to the higher levels. For example, for an experimental video data, as shown in Fig. 2, the memory efficiency starts from 100% for the first level and is reduced to 0.39% for the 13th level. Higher memory efficiency for the top levels (with smaller CL) is a clear indication that partitioning the tree into smaller and less sparse *clusters* will reduce the memory size. In addition, clustering also helps to reduce the search time for a symbol. We define a cluster (subtree) T_i with *minimum memory efficiency* (MME) β_i, if there is no level in T_i with memory efficiency less than β_i.

TABLE VI
SINGLE-SIDE GROWING HUFFMAN TABLE (SGHT)

CL	Symbols	Huffman Code			
2	00	00			
2	01	01			
4	02	1000			
4	03	1001			
4	04	1010			
4	05	1011			
4	06	1100			
4	07	1101			
5	08	1110	0		
6	09	1110	10		
6	0a	1110	11		
6	0b	1111	00		
7	0c	1111	010		
7	0d	1111	011		
8	0e	1111	1000		
8	0f	1111	1001		
8	10	1111	1010		
8	11	1111	1011		
8	12	1111	1100		
9	13	1111	1101	0	
9	14	1111	1101	1	
9	15	1111	1110	0	
10	16	1111	1110	10	
10	17	1111	1110	11	
10	18	1111	1111	00	
10	19	1111	1111	01	
10	1a	1111	1111	10	
12	1b	1111	1111	1100	
12	1c	1111	1111	1101	
12	1d	1111	1111	1110	
13	1e	1111	1111	1111	0
13	1f	1111	1111	1111	1

A. SGH-Tree Clustering

Given a SGH-Tree, as shown in Fig. 2 (or equivalently the SGHT, shown in Table VI), depending on the MME assigned, the tree is partitioned by a *cut-line*, x–x, at the Lth level ($L = 4$ for our choice of MME = 50%, in this example). Our first cluster (subtree), shown in Fig. 3(a), is formed by removing the remainder of the tree beyond the cut-line x–x. We define the *cluster length* to be equal to the maximum path length from the root to a node within the cluster. Here, the cluster length is 4, for the first cluster. Associated with this cluster we assign a look up table (LUT), also shown at the bottom of Fig. 3(a); where each entry in the table provides the addressing information for the corresponding terminal node (symbol) within that cluster, or beyond. We shall further explain this, later.

To identify other clusters, in the tree, we draw more cut lines y–y, and z–z, each L levels apart. And in doing so we generate more clusters, each of which starting from a single node (root of the cluster) and expanded until it is terminated either by terminal nodes, or nodes being intercepted by the next cut line. For our example a total of seven clusters are generated which are shown in Fig. 3(a)–(g), along with their LUT, shown at the bottom. The effect of this clustering can also be seen in the SGHT (Table VII), where the codewords are partitioned (cut) into groups of L-bits or less, as the result of the cuts by x–x, y–y, and z–z lines.

Next, we construct a *super-tree (S-Tree)* corresponding to a SGH-Tree. In a S-Tree each cluster is represented by a node, and the links connecting these nodes, represent the *branching nodes* in the SGH-Tree, shared between two clusters. Fig. 4 shows the S-Tree for our particular SGH-Tree (Fig. 3), and the *super-table (ST)* associated with the S-Tree is also shown at the bottom. Note that the S-Tree has seven nodes, one for each cluster, while its ST has six entries. This is because the root cluster a is left out, and the table starts from cluster b.

We now describe the entries in the ST and the LUT's. There are two numbers in each location in the ST: the first number identifies the cluster length, and the second one is the offset memory address for that cluster. For example, the entry in the ST corresponding to node f contains two numbers: 11 (binary) and 2 aH. The first number identifies the cluster length of 11 + 1 = 100, or 4; and the second number, 2a, specifies the starting address of the corresponding LUT, in the memory (see Table VII).

Each entry in a LUT is an integer in sign/magnitude format. Positive integers, with 0 sign, correspond to the nodes existed in the cluster, while the negative numbers, with 1 sign, represent the nodes being cut by the next cut-line. The magnitude of a negative integer specifies a location in the ST, for further search. For example, suppose we have encountered the 15th entry in the LUT in Fig. 3(c). We find 1/4 as sign/magnitude at this location. The negative sign (1)

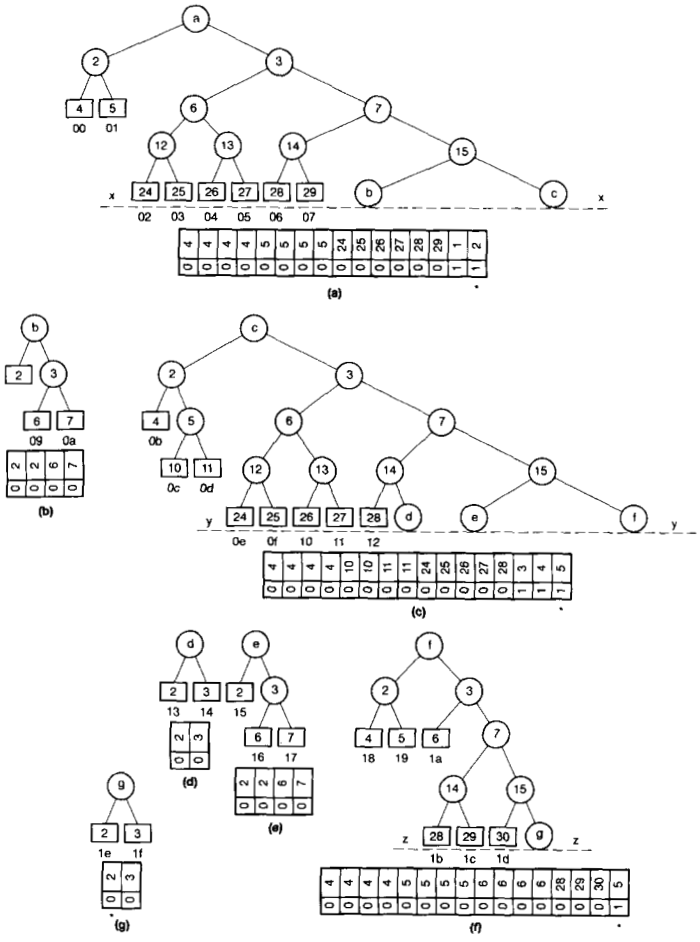


Fig. 3. The SGH-tree clusters and their corresponding look-up tables.

TABLE VII
MEMORY (RAM) SPACE ASSOCIATED WITH TABLE VI AND FIGS. 3 AND 4

00	01							02	03	04	05	06	07		
08		09	0a	0b		0c	0d					0e	0f	10	11
12				13	14	15		16	17	18	19	1a			
						1b	1c	1d		1e	1f				

indicates that we have to move to another cluster, and 4 refers to the 4th entry in the ST, which corresponds to cluster e, and contains 01 and 26H numbers. The first number, 01, indicates the cluster length of 01 + 1 = 10 or 2, and the second number, 26H, shows the starting address of cluster e in the memory.

A positive entry in a LUT, on the other hand, indicates that the symbol is already found, and the magnitude comprises three pieces of information: i) location of the symbol in the memory, ii) the pertinent codeword, and iii) the codeword length. We get all these three pieces of information by first reading the positive integer in binary form, and identifying the most significant 1 bit (MS1B). The position of the MS1B in the binary number specifies the codeword length (CL), the rest of the bits on the right side of the MS1B gives the codeword, c_j, and the magnitude of c_j is, in fact, the relative address of the symbol in the memory (Table VII). As an example, consider the entry at the 13th position in the LUT, shown in Fig. 3(a). This entry is 0/29, as the sign/magnitude. The sign 0 shows that the symbol is found, and from the magnitude

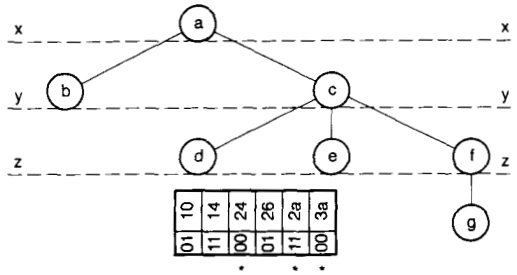


Fig. 4. The S-tree and its associated set.

29 = 000011101 we find the MS1B at location 4 indicating that CL = 4. To the immediate right of the MS1B is 1101, which identifies the codeword for the symbol, and that the symbol (07) is located at 1101 = d address (see Tables VI and VII). Before we proceed further it is interesting to see the effect of the sparsity in the memory. Although, in the process of clustering we increased the memory efficiency, still we are away from the desired 100% efficiency, and have reached close to 50% memory efficiency. This is clearly shown in Table VII, by half of the space being unused. It is, of course, possible to achieve 100% efficiency, by reducing the size of the clusters, but the price to pay may be high in terms of the search time.

We are now ready to explain the search procedure for allocating a symbol from a given string of codewords. This is the process of the Huffman decoding.

IV. HUFFMAN DECODING

Compressed codes are normally received in a stream of r-radix digital format, representing a continuous string of codewords without any separation bit(s). In our case the stream is in binary format, and the Huffman codewords are assumed to be *instantaneous* [12], i.e., each codeword in any string of codewords can be decoded as soon as it is received. The decoding procedure basically starts by receiving an L-bit code c_j, where L is the length of the top cluster in the associated SGH-Tree (or the SGHT). This L-bit code c_j is then used as the address to the associated look-up table [Fig. 3(a)].

To simplify our discussion we describe the rest of the decoding procedure through examples. We consider the case of 13-level SGH-Tree structure, previously described.

1) We take a stream of binary Huffman codes as 011001011$\cdots$, where the MSB enters first and other bits later. We use the first L = 4 bits 0110 = 6 as an address to the look-up table given in Fig. 3(a). The content of the table at this location is 0/5, as the sign/magnitude. Positive sign (0) means that the symbol is located in this cluster. We now consider the magnitude 5 = 00000101, and find the MS1B at location 2, and conclude that CL = 2. Next to the MS1B we have 01 which represents the codeword, and the symbol (01H) is also found at the address 01 in the memory (Table VII).
2) We consider a second bit stream of 110110011$\cdots$ and choose the first four bits 1101 = d. At this location in the LUT [Fig. 3(a)] we get 0/29, as the sign/magnitude. Again the symbol is located in this cluster with the magnitude, 29 = 00011101. The MS1B is apparently

at location 4, meaning that CL = 4, and immediate to the right of the MS1B is the codeword 1101. Finally, the address to the memory is 1101 = d, and the symbol is found to be 07.

3) We now assume another bit stream of 1111111111110010$\cdots$ and select the first four bits 1111. This brings us to the 15th location in the LUT for cluster a [Fig. 3(a)] where we find 1/2 as the sign/magnitude. This refers to the location 2 in the ST, assigned to cluster c. Here, we find a new pair 11 (binary) and 14H; where 11 gives the length of cluster c as 11 + 1 = 100, or 4, and 14H is the offset for the symbol memory designated for cluster c. Our next move is to select another slice of bits from the bit stream, which is 1111. We continue on and proceed to the LUT for cluster c. This again leads us to location 15 in the LUT [Fig. 3(c)] where we find data 1/5. Once again we refer to the ST, but this time to location 5, associated with cluster f. The content at this location is 11 (binary) and 2aH where 11 gives the cluster length of 11 + 1 = 100, or 4, and 2aH is the offset address for the symbols in cluster f. The symbol is not located yet, and so we have to chose another slice of bits from the bit stream. The third bit slice happens to be 1111 again, and therefore, we move to the 15th location in the LUT for cluster f. Here we find 1/6, and refer to the sixth item in the ST. The data here is read 00, and 3aH; the binary integer 00 refers to the length of cluster g which is 00 + 1 = 01, and 3aH indicates the offset address for the symbols in cluster g. We refer to the bit stream again and select one bit, which is 0. Note that only one bit of code is needed here, because the cluster length for g is 1. We proceed by referring to location 0 in the LUT for cluster g, and find 0/2. The sign 0 (positive) tells us that the symbol is here, and 2 = 00000010 indicates that; i) the MS1B is at location 1, and thus CL = 1, ii) to the right of the MS1B is a single 0 identifying the codeword, and iii) the symbol (1e) is at location $3a + 0 = 3a$ in the memory (Table VII). Finally, the overall codeword is given as 1111,1111,1111,0 with CL = 13.

In general, we conclude that for high probable symbols with short codewords (4 bits or less) the search for the symbol is very fast, and is completed in the first try. For longer codewords, however, the search time grows almost proportional to the codeword length. In particular, if CL is the codeword length, and L is the maximum level selected for each cluster ($L = 4$ in our example) then the search time is closely proportional to $1 + \mathrm{CL}/L$. Note that increasing the maximum level L should normally decrease the search time, and hence, speed up the decoding process. There are, nevertheless, some serious consequences involved here. The first problem is the growth of the memory space requirement and decay in the memory efficiency. This, of course, depends on the sparsity of the Huffman tree and may vary from case to case, as we discussed earlier. The second problem is the time needed for a single search. As the memory is growing so is the time for each search, which is mainly a memory access. Consequently, there is always a trade off between the speed and the area for the selection of the maximum level L; and it may even vary for different Huffman tree structures, if an optimal choice is in mind.

V. Conclusion

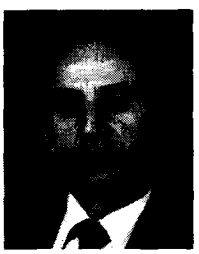
In conclusion, we claim high efficiency in memory space as well as high-speed access to the symbols in a code decompression scheme using the Huffman technique. The means to achieve this efficiency, as proposed, are to 1) avoid the sparsity of the tree structure by grouping nodes in clusters, and 2) use combined code bits to search for the symbol in smaller look-up tables.

Acknowledgment

The author wish to thank S. Eghbali and K. Golla for their contributions in simulating the algorithms.

References

[1] C. P. Sandbank and I. Childs, "The evolution toward high-definition television," *Proc. IEEE,* vol. 73, pp. 638–645, Apr. 1985.
[2] T. Fujio, "High-definition television systems," *Proc. IEEE,* vol. 73, pp. 646–655, Apr. 1985.
[3] A. K. Jain, "Image data compression: A review," *Proc. IEEE,* vol. 69, pp. 349–389, Mar. 1981.
[4] P. G. Neumann, "Efficient error-limiting variable-length codes," *IRE Trans. Inform. Theory,* vol. IT-8, pp. 292–304, July 1962.
[5] R. Hunter and A. H. Robinson, "International digital facsimile coding standards," *Proc. IEEE,* vol. 68, pp. 854–867, July 1980.
[6] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE,* vol. 40, pp. 1098–1101, Sept. 1952.
[7] T. J. Ferguson and J. H. Rabinowitz, "Self-synchronizing Huffman codes," *IEEE Trans. Inform. Theory,* vol. IT-30, pp. 687–693, July 1984.
[8] 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.
[9] K. H. Tzou, "High-order entropy coding for images," *IEEE Trans. Circuit Syst. Video Technol.,* vol. 2, pp. 87–89, Mar. 1992.
[10] M. E. Lukacs, "Variable word length coding for a high data rate DPCM video coder," in *Proc. Picture Coding Symp.,* 1986, pp. 54–56.
[11] S. M. Lei, M. T Sun, and K. 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.
[12] S. Roman, *Coding and Information Theory.* New York: Springer-Verlag, 1992.
[13] R. Hashemian, "High speed search and memory efficient Huffman coding," in *Proc. 1993 IEEE Int. Symp. Circuit Syst.,* May 3–6, 1993.
[14] ——, "Design and hardware construction of a high speed and memory efficient Huffman encoding," in *IEEE Inter. Conf. Cons. Electron.,* Chicago, IL, June 21–23, 1994.



Reza Hashemian (S'65–M'68–SM'84) was born in Ghazvin, Iran, on July 25, 1936. He received the B.S. degree from Tehran University, Tehran, Iran, in 1960, and the M.Sc. and Ph.D. degrees from the University of Wisconsin, Madison in 1965 and 1968, respectively, all in electrical engineering.

From 1968 to 1984, he was with Sharif (Area-Mehr) University of Technology, Tehran, Iran, as Assistant, Associate, and Professor. From 1984 to 1987, he was with Signetics, Inc., Sunnyvale, CA, and he joined Northern Illinois University in 1987. Presently, he is with TI, Dallas, TX, on sabbatical leave from NIU. His current interests are ASIC design, field programmable logic, data compression, and computer arithmetic.