ICICS-PCM 2003
15-18 December 2003
Singapore

An SGH-Tree Based Efficient Huffman Decoding

Yuh-Jue Chuang, Ja-Ling Wu Senior Member IEEE

Communication and Multimedia Laboratory
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan, R.O.C.
E-mail:{chuangyj, wjl }@cmlab.csie.ntu.edu.tw

Abstract

Huffman code [1] is the most well-known and widely used variable-length codes (VLCs) for compressing various kinds of data. Conventional table lookup and/or bit-serial decoding techniques for realizing VLCs are inefficient in memory usage and/or computation time. In this paper, we investigated some existing memory efficient VLC decoding methods [2, 3, 4, 5] and gave some comments and comparisons among them. We also proposed an SGH-Tree based data structure to implement *Aggarwal*'s approach [5] such that a symbol can be decoded in constant time; moreover, the required memory is much smaller than that of the original approach.

1. Introduction

Since its discovery in 1952, Huffman code [1] has become one of the most widely used VLCs and been included in almost all image and video coding standards (such as JPEG, MPEG-1, MPEG-2 and MPEG-4). The advantages of Huffman code are its effective compression ratio and implementation simplicity. The simplest and most well-known data structure used in a Huffman coding scheme is the Huffman tree. The Huffman coding usually consists of a binary code tree, in which each leaf node represents a source symbol and the path from the root to the leaf defines the variable length codeword for that symbol.

Two intuitive Huffman decoding approaches are the Bit-serial decoding and the Lookup-table-based decoding. The Bit-serial decoding process starts with the root node of the code tree and recursively traverses the tree according to the bits taken from the compressed input data stream, until it reaches a leaf. Thus, the computational complexity of the Bit-serial decoding for a symbol is $O(h)$, where h is the height of the Huffman tree. In Lookup-table-based decoding, each codeword can be quickly decoded by one table referencing. More specifically, if the longest Huffman codeword assigned to a set of symbols is of length h-bit, the memory size for storing the symbols may easily reach 2^h bits. The major disadvantage of this lookup-table-based approach is its memory cost, in the order of $O(2^h)$, spent on storing such a binary tree based on an unstructured array.

Without doubt, memory and complexity issues are of importance in most of real applications, such as complexity in real time video and audio applications and memory and complexity in realizing multimedia applications on low cost hand held devices. The Huffman codes are usually decoded one bit at a time. Due to its variable-length nature, the Huffman tree is getting sparser progressively as it growing from the root. This sparsity of the Huffman tree may cause tremendous waste of memory space and requires a lengthy search procedure for locating a symbol.

In recent literatures [2-7] a number of effective schemes have been proposed to decode Huffman codes such that the memory requirement and decoding complexity can be reduced to $O(n)$ and $O(h)$, respectively, where n denotes the total number of codewords in the Huffman tree. For a sparse Huffman tree, n is quite small as compared to 2^h. In this paper, some comparisons and analyses among existing memory efficient Huffman decoding methods [2-5] are made. Most VLC decoding methods proposed specific data structure for reducing memory size effectively, as will be discribed and commented in Section 2. Our proposed method and experimental results are addressed in Section 3. Finally, the conclusions are given in Section 4.

2. Performance analyses of some existing memory efficient Huffman decoding methods

In this section, some existing memory efficient Huffman decoding methods and their corresponding performance are investigated, according to their published order.

2.1 Hashemian's Method (HM) – An memory efficient and high-speed search Huffman coding algorithm [2]

Hashemian presented an efficient Huffman decoding algorithm based on alleviating the affection of sparsity of the Huffman tree, and therefore, reduced the search time required for looking up the code table. The main idea of HM is that by effectively grouping the codewords within specified codeword length, the search for a code symbol can be conducted by jumping over groups of bits rather

0-7803-8185-8/03/$17.00 © 2003 IEEE

than going through the bits step-by-step. To achieve the goal, HM needs to construct a special splay Huffman tree called the single-side growing Huffman tree (SGH-Tree) for representing the source symbols. SGH-Tree is a kind of Huffman trees whose growth is directed toward one side of the tree.

To avoid efficiency decline when Huffman tree growth, a clustering technique has been adopted in [2]. With clustering, an SGH-tree is partitioned every r levels to reduce the memory size. Fig.1. shows the SGH-tree T_1 clustered by HM every 2 levels, and T_1 is divided into three subtrees rooted at nodes t_1, t_2, and t_3, respectively. Using HM, r bits of the coded streams are read each time, and this explains why HM is efficient. However, how to partition the Huffman tree into many smaller subtrees such that the memory requirement is minimized is still an open problem.

Though the time complexity for decoding a symbol is no more than O(h), the memory requirement in HM is ranged from O(n) to O(2^h). In the worst case, this method behaves the same as a brute-force decoding process.

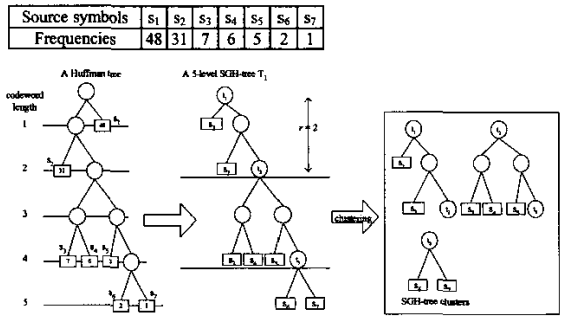


Fig1. An example Huffman tree, in which the corresponding SGH-tree T_1 is of height 5 and the 3 subtrees rooted at t_1, t_2 and t_3 for HM

2.2 Chung's Method (CM1) – An efficient Huffman decoding algorithm [3]

Chung also presented a memory-efficient data structure, the H_array, to represent the Huffman tree. For each left edge, it recorded the "jump value" which is one plus the number of edges and the number of leaf nodes in the subtree of that edge. CM1 traverses the Huffman tree in preorder, and saves the "jump value" in the H_array when a left edge is encountered, a "1" when a right edge is encountered, or the source symbol when a leaf node is encountered. The memory requirement in this data structure is $3n$-2. Moreover, this value can be further reduced to $2n$-3, as shown in [3].

Decoding process starts from the first element of the H_array and recursively increases the position according to the bits reading from the input data stream and the values of the H_array entries until a symbol is reached. Clearly, in this approach, the memory requirement is O(n) while the decoding complexity is $O(h)$.

2.3 Chen's Method (CM2) – A memory-efficient and fast Huffman decoding algorithm [4]

To reduce the memory size and speed up the process of searching for a symbol in a Huffman tree, Chen proposed a memory-efficient array data structure. CM2 declares that this algorithm takes O(h) time and uses $\lceil 3n/2 \rceil + \lceil n/2\log n \rceil + 1$ memory space, but we will illustrate that it's time complexity would be more than O(h), later.

Chen's paper defined the parameters, weight(w_i) and count($count_i$) for each symbol to conduct the decoding. The weight of symbol s_i at level l is 2^{h-l}, which is the number of leaves of the subtree s_i (the subtree rooted at s_i) where s_i is an internal node of a full binary tree with height h. Define $count_0 = w_0$ and $count_i = count_{i-1} + w_i$. CM2 can determine if the input bitstream, of any length, can be decoded or not just by w_i and $count_i$. Since CM2 needs to search count value of each symbol, the spending time depends on the length of the symbol. Decoding process also needs to examine bitstreams of different lengths, from 1 bit to the length that a symbol is decoded. The number of checking times depends on codeword lengths of all symbols, so it is in the order of O(h).

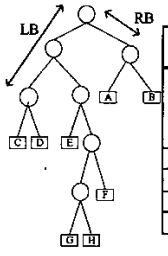
From above discussions, the complexity for decoding a symbol may be higher than O(h). But CM2 has tried some tricks to save memory, successfully. Comparing with CM1, though the decoding complexity is of the same order, CM2 saves more space.

2.4. Aggarwal's Method (AM) – Another efficient Huffman decoding algorithm [5]

Most existing efficient Huffman decoding algorithms tried to reduce the memory requirement. However, Aggarwal proposed another decoding scheme, which requires only a few computations per codeword, independent of the number of codewords n, the height of the Huffman tree h, and the length of a codeword. Although the memory requirement depends on the Huffman tree, for Huffman tables used in image and video coding standards (such as JPEG, H.263, MPEG-1 and MPEG-2), the size of additional memory is reasonably small.

Consider a Huffman tree shown in Fig.2 (a), which can be viewed as subtrees attached to the left branch (LB) or right branch (RB) of a node at different levels of the tree. Consider any subtree s and let the node at which the subtree is attached to the main tree be in level t. The codewords in the subtree s have the same first (t+1) bits. In addition, the position of the first-bit-change in any codeword belonging to subtree s should be t, and this value is unique among the left and the right subtrees. Thus the first-bit-change position

along with the first bit uniquely determines the subtree of the leading codeword in the bitstream. The codewords in a particular partition, as shown in Fig.2(b), do not have the same length, and are not consecutive binary numbers. For reducing the decoding complexity, AM modified the original Huffman table. The new decoding table is derived as follows. For every symbol x of each partition p, AM adds all possible words with prefix x and length *max_length*(p), the length of the largest codeword in the partition p, to the new decoding table, and then assign them to x. Clearly, the new decoding table is not one-to-one. Then, AM adds the missing words to all applicable partitions and assigns all of them to a new symbol **Null**. Thus, the modified table may increase memory requirement.



Subtree	first-bit-change position	Symbol	Codeword
s_1^L	1	E G H F	010 01100 01101 0111
s_2^L	2	D	001
s_3^L	≥3	C	000
s_1^R	1	A	10
s_2^R	≥2	B	11

(a) (b)

Fig.2 (a) An Example Huffman tree and (b) the corresponding Huffman Table of Fig.2 (a) for AM.

The new decoding table is shown in Fig.3(a). The "actlen" column contains the actual codeword lengths of every symbol x in the original Huffman table. Furthermore, AM constructed two 2-D arrays Leftbase and Rightbase of sizes $3\times M^L$ and $3\times R^L$, where M^L and R^L are the numbers of left and right subtrees, respectively (c.f. Fig.3(b)). The modified decoding table with Leftbase and Rightbase allows a very efficient decoding. Roughly, AM requires constant computational complexity to decode a codeword.

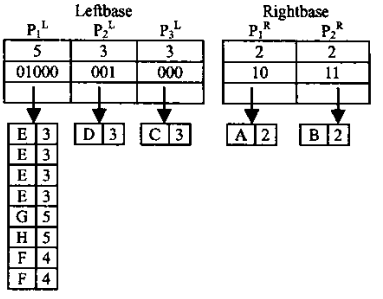
Table1 summarizes the computational complexities and memory requirements of the prescribed algorithms.

Methods	Computation Complexity	Memory requirement
HM	Less than O(h)	O(n) ~ O(2^h)
CM1	O(h)	O(n)
CM2	More than O(h)	O(n)
AM	O(1)	Depends on Huffman tree, no less than O(n)
Bit-Serial decoding	O(h)	O(n)
Lookup-table-Based decoding	O(1)	O(2^h)

Table1. The complexity and memory requirement comparisons among six different Huffman decoding methods.

Partition	first-bit-change position	Symbol	Codeword	actlen
P_1^L	1	E	01000	3
		E	01001	3
		E	01010	3
		E	01011	3
		G	01100	5
		H	01101	5
		F	01110	4
		F	01111	4
P_2^L	2	D	001	3
P_3^L	≥3	C	000	3
P_1^R	1	A	10	2
P_2^R	≥2	B	11	2

(a)



(b)

Fig.3 (a) The modified Huffman table of Fig.2(b) and (b) The decoding data structures.

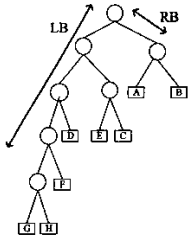
3. The proposed approach and experimental results

An SGH-tree based AM algorithm achieving efficient Huffman decoding is proposed in this section. According to Table1, we can choose a suitable coding method depends on different implementation environments. Moreover, Table1 also provides us some ideas about how to increase the efficiency for realizing a VLC. Among prescribed methods, we are most interesting in Aggarwal's approach (AM), because it can decode a symbol in constant time. But the disadvantage of AM is that the memory requirement depends on the corresponding Huffman tree. When the difference between the lengths of codewords within the same partition gets larger, the waste of memory gets mare serious, too. Just like P_1^L in Fig.3.(a), AM uses 8 entries to store 4 symbols E, F, G and H. If we can choice an efficient tree structure before Huffman coding such that the length difference within the partition is as small as possible, then we can save more space. Since we didn't change the Aggarwal's algorithm, the decoding time for a symbol is still a constant.

The proposed approach starts from an SGH-tree structure. Hashemian had proposed an algorithm to construct the SGH-tree in [2]. Without loss of generality, we assume that the SGH-tree is a left splay tree, that is, the growth of the SGH-tree is directed toward the left side of the Huffman tree. In an SGH-tree structure, the symbols with equal codeword length are treated as consecutive binary numbers

and the symbols with longer codeword lengths are put to deeper left sides of the tree. These steps produce more subtrees which attached to the LB or RB are complete. In other words, by embedding an SGH-tree structure into a Huffman tree difference between the codeword lengths within the same subtree can be reduced effectively, so that the waste of memory in AM is alleviated.

We illustrate the effectiveness of the proposed approach by the following example. After we changing the Huffman tree, as given in Fig.2(a), into an SGH-tree, as shown in Fig.4(a). We observed from the corresponding Huffman table (c.f. Fig.4 (b)) that each codeword in the same partition is of the same length. Thus, as shown in this example, there is not any waste any memory waste during the decoding process.

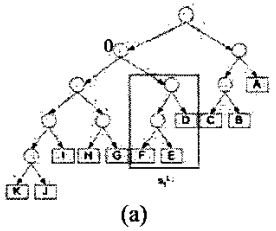


(a)

Subtree	first-bit-change position	Symbol	Codeword
s_1^L	1	E	010
		C	011
s_2^L	2	D	001
s_3^L	3	F	0001
s_4^L	4	H	00001
s_5^L	≥5	G	000000
s_1^R	1	A	10
s_2^R	≥2	B	11

(b)

Fig.4 (a) An SGH-tree embedded Huffman tree, whose codeword assignment differ from the original ones, as Fig.2(a), but with identical average codeword length. (b) The corresponding Huffman table of Fig.4(a).



(a)

Partition	first-bit-change position	Symbol	Codeword	actlen
P_2^R	≥2	A	11	2
P_1^R	1	B	100	3
		C	101	3
P_1^L	1	D	0111	3
		D	0110	3
		E	0101	4
		F	0100	4
P_2^L	2	G	0011	4
		H	0010	4
P_3^L	3	I	0001	4
P_4^L	4	J	00001	5
P_5^L	≥5	K	000000	6

(b)

Fig.5 (a) An SGH-tree and s_1^L is a incomplete subtree attached to the node '0' of LB, and (b) The corresponding modified Huffman table of (a) and we assign two codewords, 0111 and 0110 to represent.

However, SGH-tree structure cannot ensure all subtrees attached to the LB or RB are complete. For example, Fig.5(a) shows an SGH-tree and s_1^L is an incomplete subtree attached to node '0' of the LB. Fig.5(b) shows the corresponding modified Huffman table of Fig.5(a), in which two codewords, 0111 and 0110, are assigned to represent symbol D. For this reason, we propose some processing steps to improve this shortage as follows. Before explaining the details, we give some properties of the Huffman tree, first.

Property 1: Consider any subtrees s_1, s_2, which are rooted from the main tree at the same level. The tree after swapping subtrees s_1 and s_2 will still be a Huffman tree with the same average codeword length.

From *Property 1*, we can swap subtrees s_1 and s_2 for producing more complete subtrees attached to LB or RB, and therefore, reducing memory size.

Property 2: The codewords in a subtree rooted from the main tree at the level t have the same first $(t+1)$ bits.

From *Properties 1* and *2*, we can derive a way to save memory size. We denote the codeword corresponding to the symbol x of a Huffman code as c_x, the actual length of c_x as $actlen(c_x)$, the partition containing x as p_x and the longest codeword length in p_x as $l(p_x)$. If $actlen(c_x) < l(p_x)$ it implies p_x is not complete, and we must use more than one codeword to represent a symbol x in the modified Huffman Table. The proposed approach involves the following processing steps:

Step1: Construct an SGH-tree and the corresponding modified Huffman table which is sorted according to "actlen" for representing the source symbols.
Step2: Search for two subtrees s_1 and s_2 and swap them such that the required memory size is decreased.
Step3: If the memory can not be reduced then Stop else go to Step2.

The experimental results of applying the above approach to AM are shown in Table2. In Table2, we chosen some Huffman codeword tables from the MPEG4 standard [8] as the bench mark and calculated the waste of memory size of the three types of Huffman tree: MPEG-4 original, SGH-based and our proposed Huffman tree. From the experimental results, it is easy to find that, our approach certainly reduces memory waste. Thus, in cooperating the Aggarwal's algorithm with the proposed data structure is not only effective in keeping the decoding time for a symbol be a constant but also reducing the memory requirement for decoding the Huffman code.

4. Conclusions

In this paper, we gave some performance analyses among existing efficient VLC decoding methods, and proposed an

MPEG4 Tables	Number of Codewords	Memory-waste of Huffman tables among three Huffman Trees		
		MPEG-4	SGH-based	Proposed approach
B6	9	0	0	0
B7	21	3	3	2
B12	65	22	22	15
B13	13	0	0	0
B14	13	0	0	0
B16	205	155	77	40
B17	205	155	77	40
B29	33	0	0	0
B30	33	0	0	0
B35	29	0	0	0
B36	13	0	0	0
B37	127	32773	22	14
B38	16	1	1	0
B39	512	16005	67	65
B40	512	15874	140	10

Table2. The comparison of the memory waste of Huffman tables among MPEG-4, SGH-based and our proposed Huffman tree.

SGH-based data structure to obtain a more memory efficient decoding algorithm. In cooperating with the Aggarwal's algorithm, the proposed data structure spends constant time on decoding each symbol and needs the least memory size among all existing approaches. Huffman code has been included in almost all image and video coding standards, therefore, we believed that the proposed approach is useful for various multimedia coding applications, especially when the applications are conducted a memory bound devices.

References

[1]. D.A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE, vol. 40, pp. 1098-1101*, Sept. 1952.

[2]. Reza Hashemian, "Memory Efficient and High-Speed Search Huffman Code," *IEEE Transactions On Communications*, vol. 43, no. 10, pp. 2576-2581, 1995

[3]. K.L. Chung, "Efficient Huffman decoding," Information Processing Letters 61, pp. 97-99, 1997.

[4]. H. C. Chen, Y. L. Wang and Y. F: Lan, "A memory-efficient and fast Huffman deciding algorithm," Information Processing Letters 69, pp. 119-122, 1999.

[5]. M. Aggarwal and A. Narayan, "Efficient Huffman Decoding," Proc. Internation Conference on Image Processing, pp. 936-939, 2000.

[6]. H. Tanka, "Data Structure of Huffman Codes and Its Application to Efficient Encoding and Decoding," *IEEE Transactions On Inform. Theory*, vol. 33, pp. 154-156, 1987

[7]. Y.S. Lee, B.J. Shieh, C.Y. Lee "A Generalized Prediction Method for Modified Memory-Based High Throughput VLC Decoder Design," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 46, No. 6, pp. 742 –754, 1999

[8]. ISO/IEC 11496-2, "Information technology – Coding of audio-visual objects - Part 2: Visual" 2001