An SGH-Tree Based Memory Efficient and Constant Decoding Algorithm for Huffman Codes

Yuh-Jue Chuang¹, Ja-Ling Wu^{1, 2} Senior Member IEEE

Communication and Multimedia Laboratory

¹Department of Computer Science and Information Engineering,

²Graduate Institute of Networking and Multimedia

National Taiwan University, Taipei, Taiwan, R.O.C.

E-mail: {chuangyj, wjl}@cmlab.csie.ntu.edu.tw

ABSTRACT

We proposed an SGH-based data structure to obtain a memory efficient and constant Huffman decoding algorithm. In cooperating with the Aggarwal and Narayan's algorithm, the proposed data structure spends constant time on decoding each symbol and needs less memory size. In addition, we derive some properties to reduce time for searching proposed codewords. We believe that the proposed approach is useful for various multimedia coding applications, especially when the applications are conducted on memory bound devices.

1. INTRODUCTION

Since its discovery in 1952, Huffman code [1] has become one of the most widely used variable length codes (VLCs) and been included in almost all image and video coding standards (such as JPEG, MPEG-1, MPEG-2 and MPEG-4). Two most intuitive Huffman decoding approaches are the Bit-serial decoding and the Lookup-table-based decoding. The Bit-serial decoding process starts from the root node and recursively traverses the tree according to the bits taken from the compressed data stream, until a leaf node it reached. 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 the Lookup-table-based decoding, each codeword can be instantaneously decoded by one table referencing. 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 if an unstructured array is used. There is no doubt, memory and complexity issues are of importance in most of practical 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. Due to its variable-length nature, the Huffman tree is getting sparser progressively as it growing from the root. This sparsity of Huffman tree may cause tremendous waste of memory space and lengthy search procedure for locating a symbol.

In recent literatures [2-9] a number of effective schemes have been proposed to decode Huffman codes such that the memory requirement and decoding complexity can be reduced. Among the schemes, Aggarwal and Narayan [8] used an additional table to represent the Huffman tree for decoding a symbol in constant time. Constant time decoding is useful for some cases like circuit design, time estimation and process scheduling. However, the main disadvantage of this approach is that the memory requirement depends on the corresponding Huffman tree heavily. In this paper, we design a set of single-side growing Huffman tree (SGH-Tree) based codewords for enhancing the

Fig.1 (a) An Example Huffman tree and (b) the corresponding ANM-based Huffman Table.

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

Aggarwal and Narayan's decoding algorithm. Comparing with the MPEG-4 Huffman codewords, we show that the proposed codewords not only can be decoded in constant time but also the memory requirement is much smaller than that of the Aggarwal and Narayan's Method (ANM). Furthermore, the proposed codewords will not change the ANM's original average codeword length per source symbol.

The organization of this paper is as follows. In Section II, a brief review of ANM is presented. Our proposed method, derived properties and the experimental results are addressed and listed in Section III. Finally, Section IV concludes this write-up.

2. AGGARWAL AND NARAYAN'S EFFICIENT HUFFMAN DECODING METHOD

Consider a Huffman tree shown in Fig.1(a), which can be viewed as the compositions of subtrees attached to the left branch(LB) or the right branch(RB) of a node at different levels of the tree. Let any subtree s and the node at which the subtree is attached to the main tree be in level t . Then, 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 which belong to the same subtree in a particular partition, as shown in Fig.1(b), do not have the same length, and therefore, can not be represented as consecutive binary numbers. For reducing the decoding complexity, ANM modified the corresponding Huffman table. The new decoding table is derived as follows.

For every symbol x of each partition p , ANM adds all possible words with prefix x and length $\max_length(p)$, to the new decoding table, and then assign them to x . Clearly, the new decoding table is not one-to-one. Then, ANM adds the missing words to all applicable partitions and assigns all of them to a new symbol **Null**. The new

decoding table is shown in Fig.2(a). The “actlen” column contains the actual codeword lengths of every symbol x in the original Huffman table. Furthermore, ANM constructed two 2-D arrays ‘Leftbase’ and ‘Rightbase’ of sizes $2 \times M^L$ and $2 \times M^R$, where M^L and M^R are the numbers of left and right subtrees, respectively. Each column i of Leftbase corresponds to the left partition P_i^L of the new decoding table, with the first element numerically equals to the least codeword in P_i^L , and the second element is a pointer to another 2-D array O_i^L . The array O_i^L is a $N_i^L \times 2$ matrix, where N_i^L is the number of codewords in P_i^L . Each row of O_i^L corresponds to a codeword of P_i^L , with the first element as the corresponding symbol and the second being the length of the original codeword associated with that symbol. The Rightbase contains the corresponding values of the right subtrees. The modified decoding table, with Leftbase and Rightbase, provides a very efficient decoding algorithm Algorithm 1 formulates a procedure that leads to an efficient decoding of ANM. We explain some related terms in the following, first.

Let B be a pointer pointed to the start of the bitstream to be decoded. The function $fkB(B, k)$ returns the first k bits of B , the function $lmbc()$, which takes a sequence of bits as input, returns the position of the first-bit-change (from 0 to 1 or 1 to 0) starting from the leading bit, and the function $lsh(B, k)$ left shifts B by k bits.

Algorithm 1:

Partition Step: if $fkB(B, 1) = 0$ then Basearray = LeftBase
 else Basearray = RightBase
 $i = lmbc(B)$; $FC = Basearray[1][i]$
 $Oarray = Basearray[2][i]$
 Decoding Step: $C = fkB(B, Length(FC))$
 $index = C - FC + 1$
 $Symbol = Oarray[index][1]$
 $actlen = Oarray[index][2]$
 Next Iteration: $B = lsh(B, actlen)$

If ($B \neq \text{END}$) then go to Partition Step

The number of computations required for decoding a codeword are constant, which include: four indexed memory accesses, two bit extraction and shift operations and three computations of $C-FC$, $lmbc()$ and $length()$ where the function $lmbc()$ can be performed in a single clock cycle[10, 11]. However, the modified table may increase the memory requirement, for example, if there are six codewords $\{0, 11, 101, 1001, 10001, 10000\}$ in the original Huffman table, the new decoding table possesses 10 entries (partition P_1^L includes 0, partition P_1^R includes 10000 ~ 10111 and partition P_2^R includes 11). Since the size of 2-D array O_i^L (or O_i^R) is twice as large as that of P_i^L (or P_i^R), the Memory-waste is $2 \times (10-6) = 8$. That is, ANM adds extra codewords in the partition which introducing Memory-waste where Memory-waste is defined as the double of the sum of the all additional codewords in different partitions. That is,

Memory-waste = $2 \times$ (The total number of codewords in all partitions - The number of symbols),

note that Memory-waste is a non-negative integer. In the previous case, if we choose $\{0, 10, 110, 1110, 11110, 11111\}$ as codewords instead of the previous one, then the Memory-waste reduce to 0. However, in this situation, the number of partitions increases. One more partition needs two extra entries for ANM and may increase the Memory-requirement defined as follows:

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

Memory-requirement = $2 \times (\text{Memory-waste} + \text{The number of symbols} + \text{The number of partitions})$
 and the number of partitions equals $(M^L + M^R)$.

By the definitions, the solution to the problem: ‘How to find low Memory-requirement codewords to enhance the ANM decoding algorithm?’ will be addressed in the next section.

3. THE PROPOSED APPROACH AND EXPERIMENT RESULTS

All legal Huffman codewords can cooperate with ANM well to achieve a good decoding performance. However, the memory requirement of ANM grows exponentially in the worst case. In this section, we try to design a set of codewords that reduces the memory requirement and keeps the same average codeword length, of ANM, at the same time. Our design motivated by the observation that when the difference between the codeword lengths within the same partition gets larger, the waste of memory gets more serious. Just like P_1^L in Fig.2(a), ANM uses 8 entries to store 4 symbols E, F, G and H. If we can embed an efficient tree structure into the Huffman coding such that the difference between codeword lengths within the same partition is as small as possible, then we can save more memory space. Since we do not change the Aggarwal and Narayan’s algorithm, the decoding time for a symbol is still a constant.

A bit length vector V is defined as an ordered pairs $V = (L, CN)$, where L represents a set of codeword lengths (i.e. $L = \{1, 2, \dots, n\}$) and $CN(i)$ denotes the number of codewords with length i . Given a bit length vector $V = (L, CN)$, the number of distinct Huffman codes will be

$$\binom{2^1}{CN(1)} \binom{2^2 - 2^1 CN(1)}{CN(2)} \dots \binom{2^n - 2^{n-1} CN(1) - 2^{n-2} CN(2) - \dots - 2^1 CN(n-1)}{CN(n)}$$

For example, the bit length vector of Table B.12 from the MPEG4 standard is (1, 0, 2, 2, 2, 0, 2, 6, 0, 28, 12, 8). In other words, there are

Fig.3 (a)The SGHT constructed from Table I. (b)The 5-Level SGH-Tree constructed from Table I.

Codeword-length	Symbols
1	s_1
2	s_2
3	s_3, s_4, s_5
4	s_6, s_7
5	

$$\left(\begin{matrix} 2 \\ 1 \end{matrix} \right) \left(\begin{matrix} 2 \\ 0 \end{matrix} \right) \left(\begin{matrix} 4 \\ 2 \end{matrix} \right) \left(\begin{matrix} 4 \\ 2 \end{matrix} \right) \left(\begin{matrix} 4 \\ 2 \end{matrix} \right) \left(\begin{matrix} 8 \\ 6 \end{matrix} \right) \left(\begin{matrix} 12 \\ 6 \end{matrix} \right) \left(\begin{matrix} 12 \\ 6 \end{matrix} \right) \left(\begin{matrix} 24 \\ 6 \end{matrix} \right) \left(\begin{matrix} 36 \\ 28 \end{matrix} \right) \left(\begin{matrix} 16 \\ 12 \end{matrix} \right) \left(\begin{matrix} 8 \\ 8 \end{matrix} \right) \approx 8.28497 \times 10^{22}$$

distinct codewords. Since this number is too large to be investigated to find the prescribed proper low memory requirement codewords directly. We try to find a set of good codewords, without changing the average codeword length, for improving the performance of ANM by using a heuristic approach.

The proposed approach starts from a single-side-growing Huffman tree (an SGH-Tree). Hashemian proposed an efficient algorithm to build the SGH-Tree in [4]. An SGH-Tree associated with an SGH table (SGHT) can be constructed from a table of codeword lengths (TOCL). As shown by the example in Table I, a TOCL tabulates the relations among the codeword lengths and the symbols, $s_1, s_2, \dots, s_7$, where the symbols are listed in descending order of their probabilities, as described in [1]. Associated with the TOCL one can effectively generate a number of Huffman tables (or Huffman trees), each of which being different in codewords but identical in the average codeword length. Among different Huffman tables, we conduct our design based on the SGHT or equivalently, the SGH-Tree. Fig.3 illustrates the SGHT and the SGH-Tree of the example given in Table I. **Algorithm 2** shows the derivation of an SGHT.

Algorithm 2: To construct an SGHT from a TOCL, we start from the first row of the table and assign an "all zero" codeword $c_1 = 00\dots 0$ to the symbol s_1 . Next, we add one to this codeword and assign the new codeword to the next symbol in the table. When we move to the next row in the table, however, we have to expand the last codeword, after adding one to the last codeword, by placing zeros to the right of the modified codeword so that the codeword length matches the associated level.

Without loss of generality, we assume that the SGH-Tree is a left splay tree. In an SGH-tree, 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. The prescribed steps produce more complete subtrees that are attached to LB (or RB). In other words, by embedding an SGH-tree structure into a Huffman tree the difference between the codeword lengths within the same subtree can be reduced effectively. And as prescribed, the waste

of memory in ANM can be controlled better. However, the SGH-Tree structure cannot ensure all subtrees attached to the LB (or RB) are complete. Before explaining the details of how to reduce the memory waste, some properties of the Huffman tree are addressed below.

Property 1: Consider a Huffman tree S whose **Memory-waste** is W . Assume two subtrees S_i, S_j of S attached to LB (or RB) from the main tree S at levels l_i and l_j , respectively. Let $H(T)$ represents the height of a Huffman tree T . If there is an l satisfying the condition that $\max\{l_i, l_j\} \leq l \leq \min\{l_i + H(S_i), l_j + H(S_j)\}$, where s_i and s_j respectively are subtrees of S_i and S_j (rooted at the same level l of S), then, the tree, S' obtained by swapping subtrees s_i and s_j , will still be a Huffman tree with the same average codeword length as S . The **Memory-waste** W' of S' is $W + 2 \times (2^{H(S_i)} + 2^{H(S_j)} - 2^{H(S_i)} - 2^{H(S_j)})$, where S_i' and S_j' respectively are the corresponding subtrees of S_i and S_j after s_i and s_j being swapped. Moreover, if **Memory-waste** is reduced after swapping, i.e. $W' < W$, then only one of S_i and S_j whose height will become shorter than its origin version while the other one will increase or not change.

Property 2: Consider two complete subtrees S_i and S_j of S attached to LB (or RB) from the main tree S at levels l_i and l_j , respectively. If there is an l satisfying the condition that $\max\{l_i, l_j\} \leq l \leq \min\{l_i + H(S_i), l_j + H(S_j)\}$, the tree, S' obtained by swapping subtrees s_i and s_j , where s_i and s_j respectively are subtrees of S_i and S_j (rooted at the same level l of S), then, the **Memory-waste** W' of S' is equal to W , the **Memory-waste** of S .

Property 3: Consider a complete subtree S_i and an incomplete one S_j attached to LB (or RB) from the main tree S at levels l_i and l_j , respectively. If there is an l satisfying the condition that $\max\{l_i, l_j\} \leq l \leq \min\{l_i + H(S_i), l_j + H(S_j)\}$, the tree, S' obtained by swapping subtrees s_i and s_j , where s_i and s_j respectively are subtrees of S_i and S_j (rooted at the same level l of S): (a) If $l_i \leq l_j$, or (b) If $l_i > l_j$ and $l_i + H(S_i) \geq l_j + H(S_j)$ then the **Memory-waste** W' of S' will not be reduced. Otherwise if $l_i > l_j$ and $l_i + H(S_i) < l_j + H(S_j)$ (i.e. $H(S_i) < H(S_j) + l_j - l_i$), there is a possibility to reduce the **Memory-waste** after swapping subtrees s_i and s_j .

Based on **Properties 1-3**, we derive an efficient way to reduce the memory requirement, in the following.

Algorithm 3:

Step1: Construct the SGH-tree, S , from TOCL and then calculate the **Memory-requirement** M .

Step2: S_{in} is a set that contains each incomplete subtree $\hat{S}_j$ attached to LB (or RB) at the level l_j of S .

If $S_{in} = \emptyset$ then go to Step4

Step3: For $\{S_i = \arg \min_{S_j \in S_{in}} l_j\}$ do *begin*

- (i) Consider another incomplete subtree S_j attached to LB (or RB) at level l_j , where $l_i < l_j$ or a complete subtree S_j where $l_i < l_j$ and $H(S_i) < H(S_j) + l_j - l_i$, s_i and s_j are subtrees of the S_i and S_j rooted at the same level l of S where $l_j \leq l \leq \min\{l_i + H(S_i), l_j + H(S_j)\}$.
- (ii) Calculate the required memory size for swapping different s_i and s_j .
- (iii) Swap a specific couple $\{s_i, s_j\}$ which makes the **Memory-requirement** M' is minimum and smaller than M .

- (iv) If there is a swap in (iii) then go to Step2 to re-search other incomplete subtrees else delete $\hat{s}$ from S_m . If $S_m \neq \emptyset$ then go to Step3.

end

Step4: Output whole codewords associated with the path from root to a specific leaf

The experimental result of applying the proposed approach to ANM is shown in Table II. In Table II, we chosen all the Huffman codeword tables from the MPEG-4 standard [12] as the testing benchmark and calculated the *Memory-requirements* for different types of Huffman codewords using ANM decoding algorithm: MPEG-4 original and our proposed Huffman codewords. For simplifying our expression, we use R to represent the number of codewords. From the experimental results, it is clear that our approach does reduce the *Memory-requirement*. Thus, in cooperating the ANM algorithm with the proposed data structure is not only effective in keeping the decoding time for a symbol is a constant but also reducing the memory requirement for decoding the Huffman code.

Next, we compare our approach with another efficient and fast Huffman decoding method, called CHT presented in [9]. CHT is also an SGH-based approach and defining a condensed Huffman table which is smaller than the ordinary Huffman table. Since the time for bit operations are relatively small for indexed memory access, we neglect this time spend in the decoding time analysis. When CHT decodes a codeword c_i , it must search a greater augmented codeword, a , and then take two fields, Codeword-length and Symbol Position. Finally, the symbol will be decoded by one memory access. Thus at least four memory accesses are needed for CHT. We also take the Huffman table of MPEG-4 standard as the testing benchmark and make two different Huffman codewords by CHT and our proposed approach for comparison. In Table III, we use $2^{L(c)}$ to substitute the probability of codeword c , where $L(c)$ represents the length of c , and count the *Memory-requirement* and average decoding time for both methods. Although the *Memory-requirement* of CHT is smaller than that of the proposed one, the memory access time for the proposed Huffman codeword is a constant and better than that of CHT.

4. CONCLUSIONS

In this paper, we proposed an SGH-based data structure to obtain a memory efficient and constant decoding algorithm for Huffman codes. Furthermore, we derive some properties for reducing the computation complexity. In cooperating with Aggarwal and Narayan's algorithm, the proposed data structure spends constant time on decoding each symbol and needs less memory size. Huffman codes have been included in almost all image and video coding standards, therefore, we believe that the proposed approach is useful for various multimedia coding applications, especially when the applications are realized on memory bound devices.

5. REFERENCES

- [1] D.A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, pp. 1098-1101, Sept. 1952.
- [2] H. Tanka, "Data Structure of Huffman Codes and Its Application to Efficient Encoding and Decoding," *IEEE*

Table II. Using MPEG-4 Huffman Tables as the testing benchmark. The comparison of the *Memory-requirements* of Huffman codewords between MPEG-4 and our proposed approach based on ANM decoding algorithm, where R represents the number of codewords in the table.

MPEG4 Tables	Number of Codewords	Longest codeword length	Memory-Requirement of different Huffman codewords using ANM	
			MPEG-4	Proposed approach
B6	16	9	2R + 20	2R + 18
B7	22	9	2R + 23	2R + 18
B8	18	6	2R + 17	2R + 14
B9	9	6	2R + 14	2R + 12
B10	5	4	2R + 10	2R + 8
B12	69	13	2R + 50	2R + 43
B13	14	11	2R + 26	2R + 24
B14	13	12	2R + 28	2R + 26
B16, 17	221	13	2R + 191	2R + 69
B29, 30	35	18	2R + 38	2R + 36
B35	29	6	2R + 18	2R + 12
B36	13	5	2R + 10	2R + 10
B37	127	17	2R + 32785	2R + 50
B38	16	9	2R + 19	2R + 18
B39	512	17	2R + 16017	2R + 99
B40	512	16	2R + 15880	2R + 42

Table III. Using MPEG-4 Huffman Tables as the testing benchmark. The comparison of the *Memory-requirements* and average decoding time of a codeword for both new Huffman codewords made by CHT, and our proposed approach based on ANM decoding algorithm, where R represents the number of codewords in the table.

MPEG4 Tables	Memory-Requirement of two Huffman decoding algorithms		Average memory access time of two Huffman decoding algorithms	
	CHT[9]	Proposed approach	CHT[9]	Proposed approach
B6	R + 16	2R + 18	4.69	4
B7	R + 28	2R + 18	5.06	4
B8	R + 12	2R + 14	4.94	4
B9	R + 12	2R + 12	4.56	4
B10	R + 12	2R + 8	4.75	4
B12	R + 36	2R + 43	5.03	4
B13	R + 36	2R + 24	4.75	4
B14	R + 40	2R + 26	4.5	4
B16, 17	R + 40	2R + 69	6.56	4
B29, 30	R + 64	2R + 36	5.25	4
B35	R + 8	2R + 12	4.5	4
B36	R + 8	2R + 10	4.5	4
B37	R + 40	2R + 50	5.1	4
B38	R + 24	2R + 18	4.97	4
B39	R + 40	2R + 99	5.17	4
B40	R + 32	2R + 42	5.01	4

- Transactions On Inform. Theory*, vol. 33, pp. 154-156, 1987.
- [3] Reza Hashemian, "Design and Hardware Construction of a High Speed and Memory Efficient Huffman Decoding," *IEEE International Conference on Consumer Electronics*, 1994.
- [4] Reza Hashemian, "Memory Efficient and High-Speed Search Huffman Code," *IEEE Transactions On Communications*, vol. 43, no. 10, pp. 2576-2581, 1995.
- [5] K.L. Chung, "Efficient Huffman decoding," *Information Processing Letters* 61, pp. 97-99, 1997.
- [6] H. C. Chen, Y. L. Wang and Y. F. Lan, "A memory-efficient and fast Huffman decoding algorithm," *Information Processing Letters* 69, pp. 119-122, 1999.
- [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] M. Aggarwal and A. Narayan, "Efficient Huffman Decoding," *Proc. International Conference on Image Processing*, pp. 936-939, 2000.
- [9] Reza Hashemian, "Condensed Table of Huffman Coding, a New Approach to Efficient Decoding," *IEEE Transactions On Communications*, vol. 52, no. 1, pp. 6-8, 2004.
- [10] Texas Instruments, TMS320C80(MVP) Parallel Processor, User Guide, 1995.
- [11] Texas Instruments, TMS320C6000, CPU and Instruction set and Reference Guide, May 1999.
- [12] ISO/IEC 11496-2, "Information technology - Coding of audio-visual objects - Part 2: Visual" 2001.