Level-Compressed Huffman Decoding

Kuo-Liang Chung and Jung-Gen Wu, *Member, IEEE*

***Abstract*—Based on the breadth-first search manner and the level-compression technique, this letter first presents a new array data structure to represent the classical Huffman tree. Then, the decoding algorithm is given. Both the memory and the decoding time required in the proposed method are less than those of previous methods. Some experimentations are carried out to demonstrate the advantages of the proposed method. In fact, the proposed algorithm can be applied to the canonical Huffman tree.**

***Index Terms*—Data compression, decoding algorithm, Huffman tree.**

I. Introduction

RECENTLY, some efficient array data structures were presented for representing the conventional Huffman tree (HT) [5] and the canonical Huffman tree (CHT), which is transformed from the HT using a preprocessing step [8].

Recently, based on the depth-first search (DFS) manner [7] on the HT [2], the memory required in the array data structure is $5n-4$, and the decoding time is $O(d)$, where d denotes the depth of the HT and $2n-1$ denotes the number of nodes in the HT. Some experimental results have shown that the method [2] has 34% to 39% (50% to 76%) memory utilization (decoding time) improvement over the method in [4]. In [5], the memory required in the cluster-based array data structure associated with a lookup table for representing the CHT is ranged from $O(n)$ to $O(n\log n)$. Later, instead of using three tables in [2] while still using the DFS manner, only one table is needed to represent the HT, and the memory requirement is further reduced to $2n-1$ [3].

Suppose the topmost $d'+1$ levels of the HT form a complete subtree. The motivation of this research is to use the breadth-first search (BFS) [7] to design a new array data structure for representing the HT, and the level-compression technique can be applied to improve this data structure further. In addition, the decoding time can be improved simultaneously. Note that the DFS-based data structure used in [2] and [3] is not suitable for employing the level-compression technique due to its preorder-traversal limitation.

Based on the BFS manner on the HT and the level-compression technique, this paper first presents a new array data structure to represent the HT in a more compact way. The memory required in the proposed data structure is $2n-2^{d'}+1$. Then, an efficient decoding algorithm is presented, and the decoding time is proportional to $d-d'$. Both the memory and the decoding time required in the proposed method are less than those of the previous methods [2], [3]. Some experimentations are carried out to demonstrate the advantages of the proposed method. Finally, applying the results of this paper to improve the method for the CHT [6] is discussed.

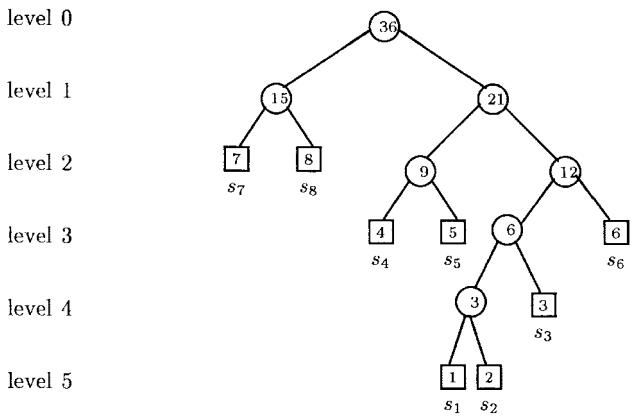

Fig. 1. A Huffman tree.

II. Proposed Data Structure for Representing HT

Given a set of source symbols $S=\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ with frequencies $W=\{1, 2, 3, 4, 5, 6, 7, 8\}$, respectively, using the construction method [1], the HT is shown in Fig. 1, where the integer inside the square box or the circle is the accumulated frequency.

We traverse the HT in a BFS manner. When traversing a leaf node, we record the associated source symbol in the array. When traversing an internal node, we record the "jump value," $2l+r+1$, where l denotes the number of internal nodes on the left-hand-side of that traversed node at the same level, and r denotes the number of nodes on the right-hand-side at the same level. This jump value will be used to slide from one entry in the array to another entry in order to emulate the pointer jumping during the decoding process in the HT. For example, the right son of the root in Fig. 1 has the jump value $3\ (=2\times 1+0+1)$. After traversing the HT, the sequence of these ordered values is saved in the array, say H_array. It is clear that the memory required in H_array is $2n-1$.

We now want to reduce H_array further by employing the level-compression technique. Suppose the topmost $d'+1$ levels of the HT form a complete subtree, i.e., all the nodes on the topmost d' levels are internal nodes. The jump values

Paper approved by E. Ayanoglu, the Editor for Communication Theory and Coding Application of the IEEE Communications Society. Manuscript received May 1, 1998; revised December 9, 1999. This research was supported by the National Science Council, R.O.C., under Contract NSC88-2213-E011-005.

K.-L. Chung is with the Department of Information Management and Institute of Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan 10672, R.O.C. (e-mail: klchung@cs.ntust.edu.tw).

J.-G. Wu is with the Department of Information and Computer Education, National Taiwan Normal University, Taipei, Taiwan 10610, R.O.C.

Publisher Item Identifier S 0090-6778(99)07785-5.

TABLE I
PERFORMANCE COMPARISON

image	$M[2]$	$M[3]$	M	R_1	R_2	$T[2]$	$T[3]$	T'	T	R_3	R_4
Lena	6092	4221	3087	49.3	26.9	1.24	0.85	0.87	0.73	41.1	14.1
Baboon	6066	4203	3069	49.4	27.0	1.24	0.92	0.86	0.73	41.1	20.7
Pepper	6144	4257	4131	32.8	3.0	1.16	0.80	0.81	0.74	36.2	7.5
F16	6066	4203	3933	35.2	6.4	1.23	0.84	0.86	0.81	34.1	3.6

of these $2^{d'}-1$ internal nodes are rather regular and are $1, 2, 3, \cdots, 2^{d'}-1$, respectively. So, we save only the value d' and discard these $2^{d'}-1$ jump values. This leads to a more compact array, say CH_array, and the total memory requirement (including the value d') is $2n-2^{d'}+1$ $(=2n-1-(2^{d'}-1)+1)$. Return to Fig. 1. The topmost three levels form a complete subtree, and the jump values of the root and its two sons are 1, 2, and 3, respectively. Therefore, the CH_array is shown below

$$\text{CH\_array}\,[0..11] = [s_7,\ s_8,\ 2,\ 3,\ s_4,\ s_5,\ 2, s_6,\ 2,\ s_3,\ s_1,\ s_2].$$

III. DECODING ALGORITHM

Our decoding algorithm is listed below. Two variables, code_ptr and array_ptr, are used to denote the current positions in the Huffman code, which is the input saved in the array, Huf_array and CH_array, respectively

code_ptr $\leftarrow d'$
array_ptr $\leftarrow$ decimal value of Huf_array $[0, 1, \cdots, d'-1]$
while (CH_array [array_ptr] is not a symbol)
array_ptr $\leftarrow$ array_ptr + CH_array [array_ptr]
    / * point to the left son * /
If (Huf_array [code_ptr] $\neq 0$) then array_ptr $\leftarrow$ array_ptr + 1
    / * point to the right son if the code bit is "1" * /
code_ptr $\leftarrow$ code_ptr + 1
    / * point to the next code bit * /
end while
output CH_array [array_ptr].

Consider the input 1101. From Fig. 1, it is known that $d'=2$, so code_ptr is set to be 2. We check the first two bits in Huf_array, i.e., Huf_array $[0..1] = 11$. Its decimal value is 3, so array_ptr is set to be 3. Next, we read Huf_array $[2] = 0$ and the jump value CH_array $[3] = 3$. Since Huf_array $[2] = 0$, array_ptr is increased by 3 (=Huf_array [2]+ CH_array $[3] = 0+3$). Then, code_ptr is increased to be $3\ (=2+1)$ and array_ptr is set to be $6\ (=3+3)$. Further, we read Huf_array $[3] = 1$ and CH_array $[6] = 2$. The array_ptr is increased by $3\ (=1+2)$, i.e., array_ptr $= 6+3 = 9$. Finally, the decoded source symbol s_3 (=CH_array [9]) is found.

In the proposed decoding algorithm, the first d' bits of the input are read at one time, and the decimal value of these d' bits is used to jump to the proper entry of the CH_array. Besides needing a decimal conversion of the first d' bits, it takes at most two integer additions and two check operations for processing each input bit in the remaining $d-d'$ bits, so the decoding-time bound is proportional to $(d-d')$.

IV. EXPERIMENTAL RESULTS

Four real gray-scale images, the Lena, Baboon, Pepper, and F16, each pixel with 256 gray levels and each image with size 512×512, are taken to evaluate the performance; the values of n's in the four images are 235, 234, 237, and 234, respectively; the values of d's (d''s) are 18, 18, 17, and 17 (7, 7, 4, and 5). The machine used is the IBM compatible personal computer with 233 MHz Pentium II microprocessor. Table I compares the memory requirement and the decoding time between our method and the methods in [2] and [3].

The memory required in the methods in [2] and [3] and our method are denoted by $M[2]$, $M[3]$, and M, where the adopted memory unit is bit. In the related array data structures, we add a ninth bit to each entry to distinguish between the jump value and the source symbol. A "1" is added to each source symbol and a "0" is added to each jump value. Therefore, each entry in the array needs nine bits. The array used in [3] needs $2n-1$ entries such that $(2n-1)\times 9$ bits are needed. Our proposed method needs $(2n-2^{d'}+1)\times 9$ bits. In [2], three tables, i.e., the preorder table, jump table, and symbol table, are needed. The preorder table has $2n-2$ entries and each entry has one bit. The jump table has $2n-2$ entries and each entry has eight bits. The symbol table has n entries and each entry has eight bits. Totally, in [2], $26n-18$ bits are needed. Let $R_1 = (M[2]-M)/(M[2])*100\%$ $(R_2 = (M[3]-M)/(M[3])*100\%)$ denote the relative ratio of the memory improvement of the proposed method when compared to the methods in [2] and [3]. It is observed that the proposed method has 32.8% to 49.4% (3.0% to 27.0%) memory utilization improvement over the methods in [2] and [3].

The decoding time includes the time needed to read compressed data from the input file and that needed to write the results into the output file. The decoding time required in [2] and [3] and our proposed method are denoted by $T[2]$, $T[3]$, and T, respectively, where the time unit is microseconds. Let $R_3 = (T[2]-T)/(T[2])*100\%$ $(R_4 = (T[3]-T)/(T[3])*100\%)$ denote the relative ratio of the decoding time improvement of the proposed method when compared to the methods in [2] and [3]. It is observed that the proposed method has 34.1% to 41.1% (3.6% to 20.7%) decoding time improvement over the methods in [2] and [3].

In Table I, the decoding time required in the proposed BFS-based data structure without employing the level-compression technique is denoted by T'. Although the decoding perfor-

mance in the DFS-based approach [3] is somewhat better than T', however, it is difficult to employ the level-compression technique to [3] in order to improve the decoding time further. Due to combining the advantages of BFS and the level-compression technique, the proposed method is better than that of [3].

V. Conclusions

For supporting faster decoding on the HT, we have presented a new, but simple, memory-efficient array data structure. Experimental results have demonstrated the advantages of the proposed method, which are dependent on the distribution of d'. In fact, the level-compression technique can be used to discard the first d' entries in the offset table and the base table in [6] for the CHT. Using the same four images, the memory requirement has improved about 1.1% to 4.1%, and the improved method needs 2100 bits on average, which is less than our result. The decoding time has improved about 6.5% to 18.2%, and the improved method needs 0.79 μs on average, which is a little more than our result.

Acknowledgment

The authors would like to thank the four reviewers and Dr. Ayanoglu for making valuable suggestions and corrections that led to the improved version of the paper.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression.* Englewood Cliffs, NJ: Prentice-Hall, 1990, pp. 105–107, sec. 5.1.2.
[2] K. L. Chung and Y. K. Lin, "A novel memory-efficient Huffman decoding algorithm and its implementation," *Signal Processing*, vol. 62, pp. 207–213, 1997.
[3] K. L. Chung, "Efficient Huffman decoding," *Inf. Process. Lett.*, vol. 61, pp. 97–99, 1997.
[4] R. Hashemian, "Memory efficient and high-speed search Huffman coding," *IEEE Trans. Commun.*, vol. 43, pp. 2576–2581, Oct. 1995.
[5] A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, pp. 1098–1101, Sept. 1952.
[6] A. Moffat and A. Turpin, "On the implementation of minimum redundancy prefix codes," *IEEE Trans. Commun.*, vol. 45, pp. 1200–1207, Oct. 1997.
[7] B. M. E. Moret and H. D. Shapiro, *Algorithms from P to NP*, vol. 1, *Design and Efficiency.* New York: Benjamin, 1991, pp. 193–196, sec. 4.1.1.
[8] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Commun. ACM*, vol. 7, pp. 166–169, 1964.