A memory-efficient and fast Huffman decoding algorithm

Hong-Chung Chen, Yue-Li Wang *, Yu-Feng Lan

Department of Information Management, National Taiwan University of Science and Technology, 43 Section 4, Kee-Lung Road, Taipei 106, Taiwan

Received 1 September 1998; received in revised form 7 December 1998
Communicated by K. Iwama

Abstract

To reduce the memory size and speed up the process of searching for a symbol in a Huffman tree, we propose a memory-efficient array data structure to represent the Huffman tree. Then, we present a fast Huffman decoding algorithm, which takes O(\log n) time and uses \( \lfloor 3n/2 \rfloor + \lceil n/2 \log n \rceil + 1 \) memory space, where \( n \) is the number of symbols in a Huffman tree. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Data structures; Decoding algorithm; Huffman code

1. Introduction

Huffman codes are a widely used and very effective technique for compressing data [2,4–7]. Huffman’s algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string (i.e., a codeword). The running time of a Huffman algorithm on a set of \( n \) characters is \( O(n \log n) \).

In [3], Hashemian presented an algorithm to speed up the search process for a symbol in a Huffman tree and to reduce the memory size. He used a tree clustering algorithm to avoid high sparsity of the Huffman tree. However, finding the optimal solution of the clustering problem is still open. Moreover, the codewords of a single-side growing Huffman tree are different from the codewords of the original Huffman tree. Later, Chung gave a memory-efficient data structure, which needs the memory size \( 2n - 3 \), to represent the Huffman tree, where \( n \) is the number of symbols in a Huffman tree [1]. In this paper, we shall propose a more efficient algorithm to save memory space.

The remaining part of this paper is organized as follows. In Section 2, for easy understanding, we introduce our basic concept without considering the memory-efficient problem. In Section 3, a memory-efficient version of our algorithm is presented. Section 4 contains our concluding remarks.

2. The main idea of our algorithm

In this section, we introduce our algorithm without saving any memory space in order to present our idea simply. Then, in the next section, we shall describe how to implement our algorithm so that the memory requirement is extremely efficient.

Let \( T \) be a Huffman tree which contains \( n \) symbols. The symbols (i.e., the leaves of \( T \)) are labeled from left to right as \( s_0, s_1, \ldots, s_{n-1} \). The level of a node
with respect to $T$ is defined by saying that the root has level 0, and other nodes have a level that is one higher than they have with respect to the subtree of the root which contains them. The largest level is the height of the Huffman tree. The weight of a symbol is defined to be $2^{h_l}$, where $h$ is the height of the Huffman tree and $l$ is the level of the symbol. Let $w_i$ be the weight of symbol $s_i$ for $i = 0, 1, \ldots, n - 1$. For example, see Fig. 1. The values of $w_i$, $\text{count}_i$ and $s_i$; $i = 0, 1, \ldots, n - 1$, in the Huffman tree are shown in Table 1. Notice that the height $h$ of the Huffman tree is 5.

Now, we describe our decoding algorithm as follows.

**Algorithm A**

**Input:** The values of $s_i$, $w_i$ and $\text{count}_i$, $i = 0, 1, \ldots, n - 1$, of a Huffman tree $T$ with height $h$ and a binary codeword $c$.

**Output:** The corresponding symbol $s_k$ of $c$.

**Method:**

**Step 1.** Compute $t = (c + 1) \times 2^{h-d}$, where $d$ is the number of binary digits in $c$.

**Step 2.** Search $t$ from array $\text{count}$, if $t$ is not in the array $\text{count}$, then $c$ is not a codeword of $T$; otherwise assume that $\text{count}_k = t$.

**Step 3.** If $w_k \neq 2^{h-d}$, then $c$ is not a codeword of $T$; otherwise $s_k$ is the corresponding symbol of codeword $c$.

**End of Algorithm A**

Clearly, Algorithm A can be done in $O(\log n)$ time. The memory required in Algorithm A is $3n + 1$. That is, each of the three arrays $s$, $w$ and $\text{count}$ needs $n$ elements and one for the height of $T$. We use three examples to illustrate Algorithm A.

Let $c = 0111$. In Step 1, $t = (7 + 1) \times 2^{5-4} = 16$. In Step 2, $\text{count}_5 = 16$. In Step 3, $w_5 = 2^{5-4} = 2$. Therefore, $s_5$ is the corresponding symbol of codeword 0111.

Let $c = 011$. In Step 1, $t = (3 + 1) \times 2^{5-3} = 16$. In Step 2, $\text{count}_5 = 16$. However, in Step 3, $w_5 \neq 2^{5-3}$. Thus, 011 is not a codeword of $T$.

Let $c = 100$. In Step 1, $t = (4 + 1) \times 2^{5-3} = 20$. We cannot find 20 from array $\text{count}$ and 100 is not a codeword of $T$.

The basic concept of Algorithm A is described as follows. Imagine a full binary Huffman tree. A full binary tree of height $h$ is a binary tree having $2^{h+1} - 1$ nodes in which there are $2^h$ leaves. Thus, the symbols in a full binary Huffman tree are $s_0, s_1, \ldots, s_{2^h-1}$. It means that $w_i = 1$ and $\text{count}_i = i + 1$ for $i = 0, 1, \ldots, n - 1$. That is, given a codeword $c$, the value

![Fig. 1. An example of a Huffman tree.](image-url)
of c is the index of symbol s_c. Assume that the given Huffman tree T having height h is not a full binary tree. Then, the weight of symbol s_i at level l is 2^h−l which is the number of leaves of subtree s_i when s_i is the corresponding internal node of a full binary tree with height h. In searching a codeword c, if the length (i.e., the number of binary digits) of c is less than h, then append enough 1’s to get a codeword c’ with length h. Obviously, the weight of c, denoted X, is c + 1 if not appended (i.e., the length of c is h) or c’ + 1 if appended. Obviously, if X is not in array count, then c is not a codeword of T. Since we append enough 1’s to get a binary string with length h, there may be more than one binary string having X as its weight. This is the reason why the binary string with level l is the correct codeword.

3. A memory efficient version of Algorithm A

In this section, we describe how to save memory in the implementation of Algorithm A. At first, since w_i can be obtained from the equation w_i = count_i − count_i−1 for i = 1, 2, . . . , n − 1 and w_0 = count_0, the array w can be omitted. The needed memory space becomes 2n + 1. Now, we consider how to decrease the memory space needed by array count. Let W_i = w_2i + w_2i+1 for i = 0, 1, . . . , [(n − 1)/2]. Note that W_{[(n−1)/2]} = w_{n−1} if n is an odd number. Let COUNT_0 = W_0 and COUNT_i = COUNT_{i−1} + W_i, i = 1, 2, . . . , [(n − 1)/2]. Moreover, a bit b_i is equal to 0 (respectively, 1) to indicate w_{2i} ≤ w_{2i+1} (respectively, w_{2i} > w_{2i+1}) for i = 0, 1, . . . , [(n − 1)/2]. Since we can obtain W_i, i = 1, 2, . . . , [(n − 1)/2] from array COUNT, it is not necessary to store array W, either. Now, we describe the memory efficient algorithm as follows.

Algorithm B

Input: The arrays s, b and COUNT of a Huffman tree T with height h and a binary codeword c.

Output: The corresponding symbol s_k of c.

Method:

Step 1. Compute t = (c + 1) × 2^{h−d}, where d is the number of binary digits in c.

Step 2. Find COUNT_k such that COUNT_{k−1} < t ≤ COUNT_k.

Step 3. Compute x = COUNT_k − COUNT_{k−1}.

Step 4. Decompose x into x_1 and x_2 such that x = x_1 + x_2, x_1 = 2^e_1t, i = 1, 2, for some nonnegative integer e_1 and assume, without loss of generality that e_1 ≤ e_2.

Step 5. Use b_e, x_1 and x_2 to determine w_a and w_b which are the weights of s_{2k} and s_{2k+1}, respectively.

Step 6. If t = COUNT_k and w_b = 2^{h−d}, then s_{2k+1} is the corresponding symbol of c. Let k = 2k + 1 and stop.

Step 7. If t = COUNT_k − w_b and w_a = 2^{h−d}, then s_{2k} is the corresponding symbol of c. Let k = 2k and stop; otherwise c is not a codeword of T.

End of Algorithm B

Step 2 of Algorithm B takes O(log n) time. In Step 4, the decomposition of x can be done as follows. Determine whether 2^{log_2 t} is equal to x or not. If they are not equal, then x_2 = 2^{log_2 t} and x_1 = x − x_2; otherwise x_1 = x_2 = x/2. Thus, Step 4 takes O(1) time. The other steps can be done in O(1) time. Therefore, the time complexity of Algorithm B is O(log n). The needed memory space for the arrays s, COUNT, and b and the height of T are n, [n/2], [n/2 log n] and 1, respectively.

We also give three examples on the Huffman tree of Fig. 1 to illustrate Algorithm B. The arrays COUNT and b are shown in Table 2.

Let c = 0111. In Step 1, t = (7 + 1) × 2^5−4 = 16. In Step 2, COUNT_1 < 16 ≤ COUNT_2 and k = 2. In Step 3, x = 3. In Step 4, x is decomposed to x_1 = 1 and x_2 = 2. In Step 5, wa = 1 and wb = 2 since b_2 = 0. In Step 6, since t = COUNT_2 = 16 and wb = 2^{5−4} = 2, s_5 is the corresponding symbol of 0111 and stop.

Let c = 011. All the results are the same as the previous example except Step 6. In Step 6, wb = 2, but 2^{h−d} = 2^5−3 = 4. Thus, 011 is not a codeword of T.

<table>
<thead>
<tr>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>The values of COUNT_i and b_i</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>COUNT_i</td>
</tr>
<tr>
<td>b_i</td>
</tr>
</tbody>
</table>
Let \( c = 100 \). In Step 1, \( t = (4 + 1) \times 2^{5-3} = 20 \). In Step 2, \( COUNT_2 < 20 \leq COUNT_3 \) and \( k = 3 \). In Step 3, \( x = 12 \). In Step 4, \( x \) is decomposed to \( x_1 = 4 \) and \( x_2 = 8 \). In Step 5, \( wa = 8 \) and \( wb = 4 \) since \( b_3 = 1 \). However, \( t = 20 \) is not equal to either \( COUNT_3 \) or \( COUNT_3 - wb \) in Steps 6 and 7. Therefore, 100 is not a codeword of \( T \).

The comparison of our algorithm and previous work is summarized as Table 3.

### Table 3

The comparison table

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Memory space</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashemian’s algorithm [3]</td>
<td>( O(n) \sim O(2^b) )</td>
<td>( h )</td>
</tr>
<tr>
<td>Chung’s algorithm [1]</td>
<td>( 2n - 3 )</td>
<td>( h )</td>
</tr>
<tr>
<td>Our algorithm</td>
<td>( \lceil 3n/2 \rceil + \lceil n/2 \log n \rceil + 1 )</td>
<td>( \log(n) )</td>
</tr>
</tbody>
</table>

4. Concluding remarks

We conclude that our algorithm can be done in \( O(\log n) \) time and needs memory space \( \lceil 3n/2 \rceil + \lceil n/2 \log n \rceil + 1 \). Moreover, our algorithm can also be parallelized easily. Since Step 2 of Algorithm B can be done in \( O(1) \) time by using \( O(n) \) processors in EREW PRAM model. Therefore, the running time of the parallelized implementation of Algorithm B is \( O(1) \) by using \( O(n) \) processors in EREW PRAM model.

**References**