The Journal of Systems and Software 85 (2012) 974–980

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A fast algorithm for Huffman decoding based on a recursion Huffman tree

Yih-Kai Lin*, Shu-Chien Huang, Cheng-Hsing Yang

Department of Computer Science, National Pingtung University of Education, No. 4-18 Minsheng Rd., Pingtung City 90003, Taiwan

ARTICLE INFO

Article history:
Received 25 July 2011
Received in revised form
10 November 2011
Accepted 17 November 2011
Available online 1 December 2011

Keywords:
Data compression
Huffman code
Decoding

ABSTRACT

This paper focuses on the time efficiency of Huffman decoding. In this paper, we utilize numerical interpretation to speed up the decoding process. The proposed algorithm firstly transforms the given Huffman tree into a recursion Huffman tree. Then, with the help of the recursion Huffman tree, the algorithm has the possibility to decode more than one symbol at a time if the minimum code length is less than or equal to half of the width of the processing unit. When the minimum code length is larger than the half of the width of the processing unit, the proposed method can still increase the average symbols decoded in one table access (thus speeding up the decoding time). In fact, the experimental results of the test files show that the average number of decoded symbols at one time for the proposed method ranges from 1.91 to 2.13 when the processing unit is 10. The experimental comparisons show that, compared to the conventional binary tree search method and the level-compressed Huffman decoding method, the decoding time of the proposed method is a great improvement.

1. Introduction

A Huffman code is a minimum redundancy code (Huffman, 1952; Bell et al., 1990; Roman, 1992) used in lossless compression. For example, Huffman coding is one of the major processing stages of entropy coding in JPEG (Pennebaker and Mitchell, 1993). Recently, Huffman code has been used to improve MFCVQ-based reversible data hiding (Yang et al., 2011). In practice, the efficiency of Huffman decoding is a major issue in the design of the Huffman decoder. The space efficiency of Huffman decoding has been studied exhaustively, e.g. Huffman (1952), Hashemian (1995), Chung and Lin (1997), Chung and Wu (1999) and Lin and Chung (2000). Hashemian (1995) presented a decoding algorithm which decodes a symbol in the worst case $\Theta(d)$ time in which the required memory space ranges from $\Theta((n+d)\lceil \log_2 n \rceil)$-bits to $\Theta(2^d \lceil \log_2 n \rceil)$-bits, where n is the number of source symbols and d is the depth of the Huffman tree. Later, Lin and Chung (2000) presented a $\Theta(d)$-time Huffman decoding algorithm with a memory-efficient data structure, which requires $\Theta((n+d)\lceil \log_2 n \rceil)$-bits of memory space. Hashemian (2004) later presented an $\Theta(t)$-time decoding algorithm in which the required memory space is $\Theta((n+4t))\lceil \log_2 n \rceil)$-bits where t is the number of distinct code lengths. Theoretically, t can be 1 up to n.

Since the time efficiency of Huffman decoding has become more and more important as memory is getting cheaper, and the requirements of decoding time are getting stricter, this paper focuses on the time efficiency of Huffman decoding. The traditional decoding algorithms traverse the path from the root to the leaf node while scanning the input Huffman stream bit by bit where the left edge corresponds to '1' and the right edge corresponds to '0'. To improve the decoding time, we utilize numerical interpretation to speed up the decoding process. In the proposed method, the leaves of the Huffman tree are rearranged into an array structure. Then the numerical interpretation of the incoming bit stream is used to calculate the address of the corresponding source symbols. The proposed algorithm firstly transforms the given Huffman tree to a recursion Huffman tree. Then, with the help of the recursion Huffman tree, the algorithm has the possibility to decode more than one symbol at a time if the minimum code length is less than or equal to half of the width of the processing unit, which is denoted as z. When the minimum code length is larger than the width of the processing unit, the proposed method can still speed up the decoding time. In the proposed method, the width of processing unit z is a parameter. A large z will make the tree too large for available memory, but will result in high speed decoding. But, in practice, the requirement to access large memory usually causes cache misses. That is, the time of one memory access (including the time to deal with cache misses) and the number of total memory accesses shows trade-off characteristics. The experimental work shows that the fast decoding time of our test files occurs when z is about 8 among $z = 5, 6, 7, 8, 9, 10, 11, 12, 13$, and 14. One problem of reading a block of Huffman stream is that when the length of concatenation of some codewords is not a multiple of z, the decoding algorithm needs to reread some bits from the Huffman stream and to restart the decoding process. To avoid this problem, a heuristic strategy is proposed to partially deal with this problem, namely,

* Corresponding author. Tel.: +886 8 722 6141x33559; fax: +886 8 721 5034.
E-mail address: yklin@mail.npue.edu.tw (Y.-K. Lin).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.11.1019

for each internal node of the top most z level, a special subtree is created.

Chung and Wu (1999) also utilize the numerical interpretation of the first d' bit of a codeword to speed up the decoding, where d' is the minimum code length. Hashemian (2004) used a Condensed Huffman Table (CHT) to represent a single side grown Huffman tree. The algorithm in Hashemian (2004) reads d bits at a time where d is the maximum code length. Then, at most one symbol can be decoded after searching at most $n-1$ rows of CHT. The comparisons of the proposed method with Chung and Wu (1999) and Hashemian (2004) are presented in Section 4.

The major drawback of our approach is the large memory requirement. However, there are hardware applications where the block size as well as the number of encoded symbols remain small since our method performs well for small block sizes. For example, Kavousianos et al. (2007) use optimal selective Huffman coding to alleviate the problems of large hardware overhead of the required decoder for test-data compression in automatic test equipment. The approach of selective Huffman coding encodes only a few of the symbols. And each unencoded symbol is preceded by the 'unencoded' Huffman codeword. In Kavousianos et al. (2007), the occurrence frequency of the longest codeword of encoded symbol is equal to the sum of the occurrence frequencies of all the unencoded symbols. Thus, the number of Huffman codewords keeps small in selective Huffman coding. Therefore, our method suits to test-data compression problems with optimal selective Huffman coding for decoding time reduction.

2. Preliminaries

Consider the source symbols $\{s_1, s_2, \ldots, s_n\}$ with frequencies $\{w_1, w_2, \ldots, w_n\}$ for $w_1 \geq w_2 \geq \ldots \geq w_n$, where the symbol s_i has frequency w_i. Using the Huffman's algorithm (Huffman, 1952), the codeword c_i for $1 \leq i \leq n$, which is a binary string, for symbol s_i can be obtained. Let us denote by $\mathcal{C} = \{c_i, \ldots, c_n\}$ the Huffman code. Let the level of the root of the Huffman tree (Huffman, 1952) be zero, and the level of the other node be equal to adding one and its parent's level. Codeword length l_i for s_i can be known as the level of s_i. For example, a Huffman tree corresponding to the source symbols $\{s_1, s_2, \ldots, s_7\}$ with the frequencies $\{17, 8, 7, 6, 3, 1, 1\}$ is shown in Fig. 1,

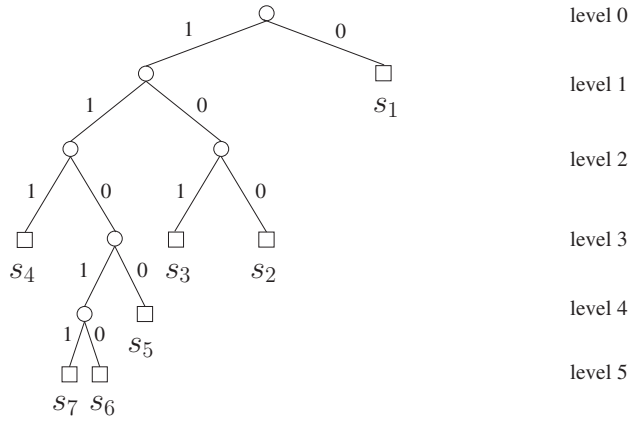

Fig. 1. An example of a Huffman tree.

and we have the two ordered sets $<c_1, c_2, c_3, c_4, c_5, c_6, c_7> = <0, 100, 101, 111, 1100, 11010, 11011>$ and $<l_1, l_2, l_3, l_4, l_5, l_6, l_7> = <1, 3, 3, 3, 4, 5, 5>$.

Assume the right edge corresponds to '0' and the left edge corresponds to '1'. The codeword of a node i, denoted $c(i)$, is defined as the bit sequence corresponding to the path from the root to node i. The codeword of a subtree T_i, denoted $c(T_i)$, is defined as the codeword of T_i's root. The level of a subtree T_i, denoted $l(T_i)$, is defined as the level of T_i's root. Given a string $x = x_1x_2\cdots x_m$, we define the ith prefix of x, for $i = 1, \ldots, m$, as $\text{prefix}_i(x) = x_1x_2\cdots x_i$ and $\text{prefix}_0(x) = \epsilon$ is an empty string. For example, if $x = 001011$, then $\text{prefix}_5(x) = 00101$.

In this paper, we assume that the computing architecture can manipulate one z-bits computer word at a time. In other words, we can compare the numerical interpretation of two codewords in constant time.

3. Decoding multiple symbols at once

In this section, we present the proposed decoding algorithm. We first transform the Huffman tree into a recursion Huffman tree, then present a decoding algorithm benefiting from the recursion Huffman tree. Given a Huffman tree called initial Huffman tree T, a

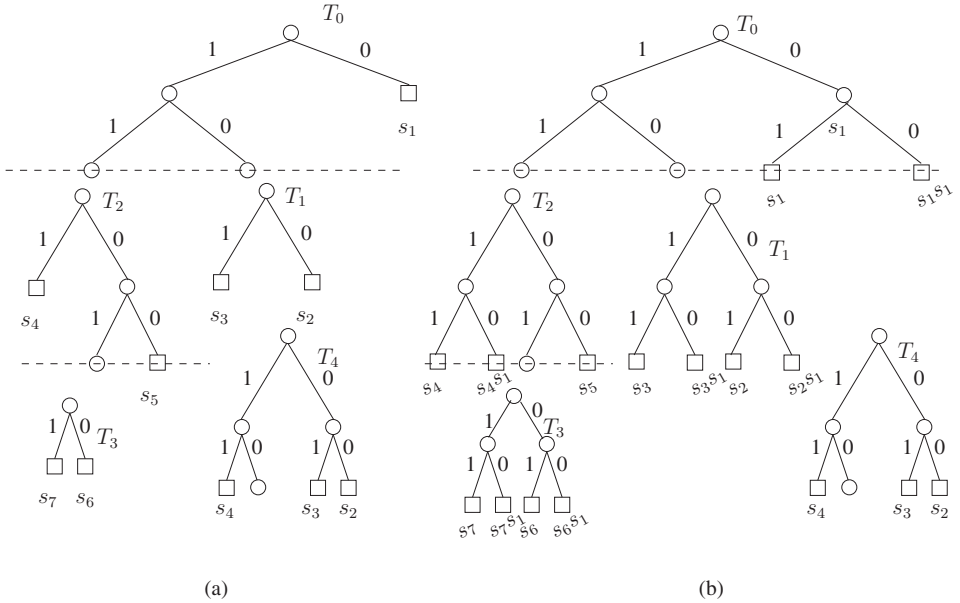

Fig. 2. Procedure for forming a recursion Huffman tree. (a) Five subtrees cut from the Huffman tree are shown in Fig. 1. (b) The initial Huffman tree is recursively appended onto the leaves of each subtree and nodes with a level higher than z are deleted from each subtree.