In-Place Calculation of Minimum-Redundancy Codes

Alistair Moffat[1] Jyrki Katajainen[2]

[1] Department of Computer Science, The University of Melbourne, Parkville 3052, Australia
alistair@cs.mu.oz.au
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk

Abstract. The *optimal prefix-free code problem* is to determine, for a given array $p = [p_i \mid i \in \{1 \ldots n\}]$ of n weights, an integer array $l = [l_i \mid i \in \{1 \ldots n\}]$ of n codeword lengths such that $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ and $\sum_{i=1}^{n} p_i l_i$ is minimized. Huffman's famous greedy algorithm solves this problem in $O(n \log n)$ time, if p is unsorted; and can be implemented to execute in $O(n)$ time, if the input array p is sorted. Here we consider the space requirements of the greedy method. We show that if p is sorted then it is possible to calculate the array l in-place, with l_i overwriting p_i, in $O(n)$ time and using $O(1)$ additional space. The new implementation leads directly to an $O(n \log n)$-time and $n + O(1)$ words of extra space implementation for the case when p is not sorted. The proposed method is simple to implement and executes quickly.

Keywords. Prefix-free code, Huffman code, in-place algorithm, data compression.

1 Introduction

The algorithm introduced by Huffman [4, 7] for devising minimum-redundancy prefix-free codes is well known and continues to enjoy widespread use in data compression programs. Huffman's method is also a good illustration of the greedy paradigm of algorithm design and, at the implementation level, provides a useful motivation for the priority queue abstract data type. For these reasons Huffman's algorithm enjoys a prominence enjoyed by only a relatively small number of fundamental methods.

In this paper we examine the space-efficiency of this greedy algorithm for constructing optimal prefix-free codes. Textbooks describing the technique often provide pseudo-code rather than a complete implementation and draw figures showing forests of binary trees. These descriptions create the impression that the implementation of the greedy algorithm should be pointer-based and reliant upon a linear amount of auxiliary memory for node addresses and for internal tree nodes. This is, as we shall show, an erroneous impression. We describe an implementation of the greedy algorithm that, in addition to an input array

storing the weights of the symbols to be coded, requires just $O(1)$ words of extra space if the input array is sorted and $n + O(1)$ words of extra space if the input array is not sorted, where n is the number of symbols for which the code is to be constructed. As for pointer-based implementations, the algorithms require $O(n)$ time for sorted input, and $O(n \log n)$ time for unsorted input. Moreover, implementation of the algorithms is straightforward, and they are suitable for practical use.

The main motivation for this study is our algorithmic curiosity. The best previous implementation of the greedy method for optimal prefix-free coding requires $n + O(1)$ words of extra memory and $O(n \log n)$ time for unsorted input arrays [10], so it was natural to ask whether these bounds could be improved if the input array is sorted. In particular, we were interested to know whether an in-place calculation was possible, since, for practical computation on large alphabets, the space constant is of overriding concern. For example, a typical textbook implementation of the greedy method requires around 20 megabytes of memory to calculate a code for a collection of one million symbols, whereas our implementation requires just 4 megabytes (one million rather than five million 4-byte words). Furthermore, recent research papers report (see, for example, [2]) that in-place algorithms can be faster in practice than their space-inefficient counterparts when run on a modern computer system with a hierarchical memory. Speed is one of the important characteristics of our implementation, too. We have calculated an optimal code for a set of over one million symbols in just a few seconds of CPU time.

2 Prefix Codes

Suppose that in some token stream there are n distinct symbols and that the ith least frequent symbol appears p_i times. That is, we suppose that $p = [p_i \mid i \in \{1 \ldots n\}]$ is a non-decreasing array of n positive integer *weights*, $p_1 \leq p_2 \leq p_3 \cdots \leq p_n$. A *code* is an array $l = [l_i \mid i \in \{1 \ldots n\}]$ of n integers, where the presumption is that the ith symbol is to be represented by an l_i-bit long binary *codeword* over the alphabet $\{0, 1\}$. A *prefix-free code* is a code for which $\sum_{i=1}^{n} 2^{-l_i} \leq 1$. For example, assigning $l_i = \lceil \log_2 n \rceil$ is a prefix-free code, since $n \cdot 2^{-\lceil \log_2 n \rceil} \leq 1$. Given a prefix-free code l, it is straightforward to determine a set of n codewords, one per distinct symbol, with the property that the codeword for symbol i is l_i bits long, and such that no codeword in the set is a proper prefix of any other.

An *optimal prefix-free code* is a set of codeword lengths l_i such that not only is $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ satisfied, but also such that $B = \sum_{i=1}^{n} l_i p_i$ is minimized over all prefix-free codes. Quantity B is the number of output bits used by the code to represent the token stream in question; a code is optimal if there is no other code that results in an output representation requiring fewer than B bits. For any given array p there can be more than one optimal code; for the assignment $p = [1, 1, 2, 2]$ both $l = [2, 2, 2, 2]$ and $l = [3, 3, 1, 2]$ (and one other) result in compressed representations that require $B = 12$ bits. Note, however, that there