Sub-linear Decoding of Huffman Codes Almost In-Place

Andrej Brodnik* Svante Carlsson†

May 6, 1998

Abstract

We present a succinct data structure storing the Huffman encoding that permits sub-linear decoding in the number of transmitted bits. The size of the extra storage except for the storage of the symbols in the alphabet for the new data structure is $O(l \log N)$ bits, where l is the longest Huffman code and N is the number of symbols in the alphabet. We present a solution that typically decodes texts of sizes ranging from a few hundreds up to 68 000 with only one third to one fifth of the number of memory accesses of that of regular Huffman implementations. In our solution, the overhead structure where we do all but one memory access to, is never more than 342 bytes. This will with a very high probability reside in cache, which means that the actual decoding time compares even better.

1 Introduction

If you have an alphabet of N symbols that you would like to encode the typical solution would be to use $\lceil \log n \rceil$ bits to encode N different numbers, each number corresponding to a symbol. This is done, for example, in ASCII-coding. If all symbols in a text that should be coded are equally frequent this will also give a minimal size encoding in the number of bits used. The encoding of a symbol is done by a standard dictionary look-up and the decoding is done by a constant-time table look-up.

If the frequency between the symbols varies one can construct a smaller encoding of a given text. Let us arrange N symbols with normalized frequencies ν_i $(0 \leq i < N)$ at leaves of a binary tree, where l_i is the depth of the leaf. The binary tree with the minimal weighted external path length

$$\overline{l} = \min \sum_{i=0}^{N-1} \nu_i l_i \tag{1}$$

is called the *Huffman tree* (cf. [5, 13]).

A path from the root to the leaf in a binary tree can be encoded as a binary string where descending to the left (right) is represented with 0 (1) in the string. Using such a technique we can uniquely encode each symbol in the text. Moreover, these encodings are *prefix-free*. The encoding constructed using a Huffman tree, is called *Huffman encoding*.

*Institute of Mathematics, Physics and Mechanics, Ljubljana, Slovenia and Department of Computer Science, Luleå University, Sweden,

†Department of Computer Science, Luleå University, Sweden

It is easy to see that the Huffman encoding, because of the property from eq. (1), gives the best possible compression of a finite text if one has to use one static code for each symbol of the alphabet. Therefore the encoding is extensively used in various fields of computer science, such as picture compression ([1]), HDTV ([4]), data transmission ([4, 5]), etc. We gain this decrease of the size of the encoded text at the cost of increased decoding time, which is linear in the number of bits of the text.

In this paper we are considering the size of the data structure to store the Huffman tree and the time necessary to decode a given Huffman code of a symbol. The best solution known so far requires $D+2N-3$ space for the data structure (D is the size of dictionary storing the symbols) and $O(l)$ time to decode the symbol ([1]). Moreover, the data structure, and hence, the accompanying algorithm, is very involved. In this paper we improve the space bound to $D + O(l \log N)$ bits for the data structure with a hidden constant at most 4 (l is the longest Huffman code); and the time bound to $\Theta(\log l')$ worst-case (l' is the number of different lengths of Huffman codes). The average decoding time for any Huffman tree is bounded by $\Theta(\log \log n)$. In fact, if the symbols have equal probabilities we have an in-place, constant time decoding solution.

The paper has a usual structure: first we do the homework by browsing through the literature searching for similar and related solutions; then we present general ideas how to change the Huffman tree not changing the cost of encoding described by the optimization function in eq. (1). In the core of the paper we present the basic structure and the decoding algorithm. This solution is further improved to the best possible on the average time-wise and almost optimal space-wise. Finally, we present some practical results of our algorithms as they are used in the *Phone Book of Slovenia*, showing a remarkable speed-up and space reduction.

NOTATION: In this paper we consider symbols with normalized frequencies ν_i, $0 < i \leq N$ and $\sum_{i=0}^{N-1} \nu_i = 1$. The length of the Huffman code of the i^{th} symbol is l_i bits, the longest code is l bits long, the number of different code lengths is l' and the weighted length of all codes is $\overline{l} = \sum_{i=0}^{N-1} \nu_i l_i$. To write down all the symbols we need D space.

2 Homework

Construction of the Huffman codes minimizes eq. (1) over all possible trees (cf. [5]). There exists a simple greedy algorithm which, from the list of symbols with frequencies, constructs Huffman codes in $O(N \log N)$ time (cf. [2, pp. 339–343]). The algorithm first sorts symbols in descending order of frequencies ($\Theta(N \log N)$ time), and then constructs the Huffman tree ($\Theta(N)$ time). The whole construction takes $O(N)$ registers of space, but can be brought down to $N + O(1)$ registers ([9]).

There are numerous variations to the basic problem. In the basic version we have as an input the list of symbols with their frequencies. However, we do not always have a possibility to construct such a list. In this case we use dynamic Huffman coding (cf. [13, 14]). Even more generalized version of the problem is when we do not have the list of symbols either (cf. [7]). Yet another version of the problem is to construct Huffman codes that must be shorter than some predefined constant (cf. [8]). The solution presented in this paper can be applied to all these different versions of the problem. However, our solution only substantially speeds up their decoding process, while (in general) it does not decrease the size of the data structure.

When the Huffman encoding is constructed appears the problem of its decoding. The decoding

algorithm has to be very fast and simple. The main reason is that it usually runs on a cheap hardware without too much memory (e.g. when used for HDTV etc.). Therefore a number of different decoding algorithms were constructed which try to minimize the size of the data structure and simultaneously decrease the decoding time (cf. [11, 4, 1, 6]). All of the mentioned algorithms try to traverse the Huffman tree constructed by the common "merging algorithm" (cf. [2, pp. 339–340]). They apply at least one of the following two techniques: they "wrap" the Huffman tree in such way that the leaves are replaced by the root of the tree; and/or they layer the tree (graph) into a small number of layers (preferably constant number) that are inspected quickly. Technically, the first technique gives a directed graph which needs to be traversed and when the root is reached, the code is decoded (cf. finite automaton); while the second technique gives an r-ary trie (graph). Applying only the first technique can give us $O(l)$ time and $D+O(N)$ space solution, while applying only the second technique can give us $O(1)$ time and $O(D + 2^l)$ space solution. Obviously, there is a wide spectrum of solutions between both techniques which, however, employ involved pointer passing data structures.

3 Mutation of the Huffman Tree and the Basic Data Structure

Vitter ([13]) states the following important so-called characterization of Huffman trees (cf. Figure 1):

Lemma 1 (Sibling Property) *A binary tree with N leaves is a Huffman tree iff:*

1. *the N leaves have nonnegative weights ν_i $(0 < i \leq N)$ and the weight of each internal node is the sum of the weights of its children; and*

2. *the nodes can be numbered by weight, so that nodes $2j + 1$ and $2j$ are* siblings *$(0 < j < N)$ and their common parent node is higher in the numbering.*