The Computer Journal **36** (1993) 668–678

Bounding the Depth of Search Trees

Aviezri S. Fraenkel and **Shmuel T. Klein**

Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

May 1987

ABSTRACT

For an ordered sequence of n weights, Huffman's algorithm constructs in time and space $O(n)$ a search tree with minimum average path length, or, which is equivalent, a minimum redundancy code. However, if an upper bound B is imposed on the length of the codewords, the best known algorithms for the construction of an optimal code have time and space complexities $O(Bn^2)$. A new algorithm is presented, which yields sub-optimal codes, but in time $O(n \log n)$ and space $O(n)$. Under certain conditions, these codes are shown to be close to optimal, and extensive experiments suggest that in many practical applications, the deviation from the optimum is negligible.

1. Motivation and Introduction

We consider the set $\mathcal{B}(n,b)$ of *extended binary trees* with n leaves, labelled 1 to n, and with depth $\leq b$, henceforth called *b-restricted* trees. An extended binary tree is a binary tree in which every internal node has two sons (here, and in what follows, we use the terminology of Knuth [16, pp. 399–405]). For a given set of *weights* w_i, $1 \leq i \leq n$, and a given bound $B \geq \lceil \log_2 n \rceil$, the problem is to find a tree in $\mathcal{B}(n,B)$ which minimizes the *weighted path length* $\sum_{i=1}^{n} w_i l_i$, where l_i is the length (number of edges) of the path from the root to leaf i.

A possible application is the construction of a binary *prefix-code* with minimal average codeword length and subject to the additional constraint that no codeword has length exceeding B. Here w_i is the frequency of the element which will be encoded by the i-th codeword. Another application is the organization of a file of n records, which are stored at the leaves of a binary search tree; w_i is the probability of record i being requested, and the problem is to minimize the average search time such that no search takes more than B comparisons.

The approach is recommended by Gilbert [8] for the case of inaccurately known probabilities w_i: if some of the w_i are significantly underestimated, Huffman's well-known procedure [13] would assign long codewords to the corresponding elements and the code thus obtained may be fairly inefficient. Another possible application of bounding the depth of a tree is to reduce the *external path length* $L = \sum_{i=1}^{n} l_i$, a quantity which appears in the complexity function of many algorithms. In the worst case, L is $O(n^2)$ and on the average (with all trees equally likely) $O(n\sqrt{n})$ (see [16]), but imposing a bound $B = O(\log n)$ on the depth reduces L to be $O(n \log n)$. In [3] this approach is suggested to improve the space requirements of a method which allows efficient decoding of Huffman codes without bit-manipulations.

When there is no bound imposed, or equivalently, when $B \geq n-1$, our problem is solved by Huffman's algorithm, which can be implemented in time $O(n \log n)$ (see for example Van Leeuwen [21]) and space $O(n)$. In fact, the dominating part of the time complexity is sorting the weights w_i, requiring time $\Omega(n \log n)$. If the weights are already given in order, the algorithm can be implemented in time linear in n. However, no simple procedure is known which extends Huffman's algorithm to the problem with bounded depth.

The solution proposed by Gilbert [8] is an exhaustive search through all the possible trees in $\mathcal{B}(n,B)$, which is not feasible for even moderately large values of n and B. Hu and Tan [12] provide a nonenumerative algorithm, in which, however, both time and space complexities grow exponentially with the bound B. A similar idea is used by Van Voorhis [22], but using dynamic programming he solves the problem in $O\big((B - \log_2 n)\, n^2\big)$; this bound applies for both time and space. A completely different dynamic programming solution is given by Garey [7] with $O(Bn^2)$ time and space complexity. Garey's algorithm is based on a procedure proposed by Gilbert & Moore [9] for *alphabetical* encodings, using time $O(n^3)$. The latter pro-

cedure was improved by Knuth [15] to $O(n^2)$ in an application to optimum binary search trees, for which records can be stored also in internal nodes, but with no restriction on the depth of the tree. Garey shows how to extend Knuth's method to the depth-restricted case.

The following reformulation of the problem will be useful. We are given an ordered sequence of n weights $w_1 \geq \cdots \geq w_n$, and a bound $B \geq \lceil \log_2 n \rceil$; the problem is to find a sequence of integers l_i, which minimizes $\sum_{i=1}^{n} w_i l_i$ subject to the constraints $l_i \leq B$ and

$$\sum_{i=1}^{n} 2^{-l_i} = 1. \tag{1}$$

McMillan [18] has shown that the lengths l_i of the binary codewords of any uniquely decipherable (UD) code $\mathcal{C}$ must satisfy $\sum 2^{-l_i} \leq 1$; the equality (1) is a sufficient condition for the *completeness* of the code $\mathcal{C}$, which means that adjoining any binary string $c \notin \mathcal{C}$ yields a code $\mathcal{C} \cup \{c\}$ which is not UD. In an application to binary search trees, l_i is the level of the leaf with weight w_i in a tree T, and (1) is equivalent to T being an extended binary tree (see [16, Exercise 2.3.4.5–3]).

The difficult part of the construction of an optimal code or tree is to find the integers l_i. Once they are given, the i-th codeword of an optimal code can be chosen as the l_i first bits to the right of the "binary point" in the binary representation of $\sum_{j=1}^{i-1} 2^{-l_j}$ (see [9, Theorem 11]). We shall use throughout the languages of codes (codewords and their lengths) and trees (leaves and their levels) interchangeably.

Our interest in this problem was stimulated by the following reflections:

(1) The construction of an optimal B-restricted tree requires $O(Bn^2)$ time and space using the methods of either [22] or [7] (actually, the *space* complexity for Garey's method can be lowered to $O(n^2)$). On the other hand, if B is large enough, Huffman's algorithm solves the problem in time $O(n \log n)$ and space $O(n)$. This discontinuity in the complexities of two problems at points where they should coincide, suggests that before applying dynamic programming, the optimal unrestricted Huffman tree should be constructed. If the depth K of the latter is $\leq B$, this tree is optimal also for the restricted case and we have a significant improvement; if however $B < K$, we can still apply the methods of [22] or [7], with no change in the order of magnitude of their complexities.

(2) For the case $B < K$, we would expect that the closer B is to K, the greater is the similarity between the restricted and the unrestricted trees. For example the Huffman tree, based on the distribution of the characters of the English alphabet, as given by Heaps [10], has depth $K = 10$ with the lengths of the paths corresponding to the four least frequent characters being 8, 9, 10 and 10. If we choose $B = 9$, each of these four characters will be on the lowest level of the optimal 9-restricted tree, and the other characters will remain on the same level as in the unrestricted tree. Hence in this case, the restricted tree is obtained from Huffman's tree by the

rearrangement of a small subtree. For $B = 8$, there are already 6 elements which must be rearranged, and for $B = 7$, there are 11. Therefore, we would intuitively find it more natural if an algorithm for the construction of an optimal B-restricted tree would require time proportional to $K - B$, rather than to B.

These thoughts suggest the following type of procedures for our problem:

Step 1: Apply Huffman's algorithm for the given sequence of weights; let K be the depth of the Huffman tree H.

Step 2: If $K \leq B$, we are done. Otherwise

Step 3: Reduce the depth of the tree to B by local rearrangements in the Huffman tree.

Note that the main problem with the optimal algorithms is their space complexity. While it can sometimes be justified to spend $O(Bn^2)$ time to get an optimal code, a quadratic space complexity for an application with large n may often be prohibitive. We thus feel that a *sub-optimal* algorithm with considerably lower space requirements can be justified, especially when it is also fast and easy to implement. In the next section, we present such an algorithm, based on the above reflections, producing sub-optimal trees, but in *linear* time and space, provided the frequencies are already ordered. In Section 3, some refinements of the method are suggested, which have time complexity $O(n \log n)$ with no change in the space complexity. Tests run on a large variety of "real-life" weight-distributions, described in Section 4, show that often the optimum is actually achieved, or that the deviation from the optimum is very small.

2. Sub-optimal trees with bounded depth

After the Huffman tree has been constructed, the rearrangements proposed in Step 3 must be applied to every branch of the tree extending below level B. For example, the subtree in Figure 1(a) could be replaced by that in Figure 1(b). The root of the subtree which is rearranged in this example is on level $B - 2$, which is the lowest possible level at which the rearrangement may be started. For other examples, the subtree will be rooted on a higher level, as for example in Figure 2.

(a) (b)

Figure 1: *Minimal Rearrangement*

(a) (b)

Figure 2: *Larger Example*

It would help to have all the branches extending below level B concentrated in the same area of the tree, so as to minimize the size of the subtree which includes all these branches. Huffman's original algorithm, however, does not assure this property. We therefore replace Step 1 by:

Step 1a: Evaluate the optimal lengths l_i using Huffman's algorithm (recall that since $w_1 \geq \cdots \geq w_n$, we may assume $l_1 \leq \cdots \leq l_n$).

Step 1b: Construct an extended binary tree in which the leaves are, *in order* from left to right, on levels $l_1, \ldots, l_n$.

An algorithm for Step 1b can be found in Schwartz & Kallik [20]. Alternatively, the tree can be generated in linear time by the procedure BUILD, which will be useful later. BUILD passes sequentially over the vector of lengths l_i and simulates a depth first traversal of a binary tree, which is built by the procedure itself, i.e., when passing to a left or right son which was not yet defined, a new node is generated and linked into the tree. During this traversal, every time a level is reached which equals the current value of l_i, the procedure passes to l_{i+1} and considers the current node v as a leaf (thus the next node to be visited will be the father of v). The procedure stops after having generated the node corresponding to l_n. For a formal description of BUILD refer to the Appendix. Henceforth we will assume that every Huffman tree and all the extended binary trees mentioned in the sequel satisfy the order requirement of Step 1b.

Algorithm ROT

Our algorithm for the construction of a B-restricted tree consists of Steps 1a, 1b, 2 and 3. The following procedure is used for Step 3. Starting at the rightmost leaf r of the Huffman tree (corresponding to the lowest weight, hence being at level K), we climb upwards in order to find the root of the subtree which will be rearranged.

For any node x of the tree, let $D(x)$ denote its level, let $T(x)$ denote the subtree rooted at x and $N(x)$ the number of leaves in $T(x)$. During the construction of the Huffman tree, D and N can recursively be defined by:

$$D(x) = \begin{cases} 0 & \text{if } x \text{ is the root;} \\ 1 + D\big(\mathit{father}(x)\big) & \text{otherwise.} \end{cases}$$

$$N(x) = \begin{cases} 1 & \text{if } x \text{ is a leaf;} \\ N\big(\mathit{left}(x)\big) + N\big(\mathit{right}(x)\big) & \text{otherwise.} \end{cases}$$

We seek an ancestor q of the leaf r, such that $T(q)$ can be rearranged into a subtree of depth $B - D(q)$, and such that q is as close to r as possible. This is obtained by considering sequentially the father of r, then the father's father, etc., until an ancestor q is found for which

$$N(q) \;\leq\; 2^{B-D(q)}. \tag{2}$$

In the worst case we climb all the way to the root, which satisfies (2) since $B \geq \lceil \log_2 n \rceil$. After having found the node q, $T(q)$ is transformed into a *complete binary tree*. First, the lengths l_i are updated to $B-1$ or B, for $n - N(q) < i \leq n$, so that $l_{i-1} \leq l_i$ and so as to satisfy (1).

Then we again apply the procedure BUILD which builds a binary tree when the levels of its leaves are given. In this case, when q is not the root, there is even no need to restart the construction from scratch, since the structure of the Huffman tree is altered only in the subtree $T(q)$. The construction passes sequentially from l_1 to l_n, so one can use the previously built Huffman tree and apply the procedure BUILD only for $l_{n-N(q)+1}, \ldots, l_n$.

Because the **R**eorganization **O**f the sub-**T**ree resembles the ROTations in AVL-trees, we call this the ROT Algorithm. If the weights w_i are already given in order, the time and space complexities of the ROT Algorithm are obviously $O(n)$.

(a) Huffman tree **(b)** 3-bounded tree **(c)** better 3-bounded tree

Figure 3: *Example for non-optimality of the algorithm*

Let $T^*(q)$ be the sub-tree which replaces $T(q)$ after this transformation, and let H^* denote the tree obtained from the Huffman tree H by replacing $T(q)$ by $T^*(q)$. Since all the leaves, whose level exceeded B in the original tree H, belong to

$T(q)$, it follows that H^* is a B-restricted tree. However, H^* is not always optimal. For example, suppose the weights are $(w_1, \ldots, w_5) = (9, 6, 4, 2, 2)$. They correspond to a degenerate Huffman tree with $(l_1, \ldots, l_5) = (1, 2, 3, 4, 4)$ which is depicted in Figure 3(a) and has path length $\sum w_i l_i = 49$. If we want to use the above algorithm to bound the depth of the tree to $B = 3$, the resulting length-vector is $(1, 3, 3, 3, 3)$ with path length 51 (Figure 3(b)), whereas there exists a better solution $(2, 2, 2, 3, 3)$ with path length 50 (Figure 3(c)).

But for small values of n, like in this example, there is no problem to use Garey's algorithm or even an exhaustive search. We now show that also for large n, there is, under certain circumstances, only a small deviation in the performance of ROT from the possible optimum.

The idea is to look at the size of the subtree $T(q)$ which will be rearranged. Obviously this size depends on the imposed bound B, but it depends also on the skewness of the distribution. If the smallest probabilities differ only slightly, then there are possibly many nodes on the lowest levels of the Huffman tree (and thus of $T(q)$). Hence $N(q)$, the number of leaves of $T(q)$, may be large, even if the bound B is close to K, the natural depth of the tree. On the other hand, great differences in the smallest probabilities tend to produce Huffman trees with only a few nodes on each of the lowest levels. The following theorem gives a sufficient condition for the algorithm to be close to optimal, in terms of a relation between the *size* of the rearranged subtree and its *shape*. Let us express the number of leaves $N(q)$ of $T(q)$ by n^α, for some $0 < \alpha \leq 1$. Let R be the level of the leftmost leaf of $T(q)$. The Huffman tree being fixed, the rightmost leaf of $T(q)$ is on level K, thus R can serve as measure for the proximity of the shape of $T(q)$ to that of a full binary tree. Define s by $R = s \log_2 n$.

Theorem 1. *If $s > 1.44\alpha$, then the difference between the average path length of the optimal tree and that of the tree obtained from algorithm ROT tends to 0 as $n \to \infty$.*

Proof: For technical reasons, we shall use probabilities p_i instead of weights w_i ($p_i = w_i / \sum_{j=1}^{n} w_j$), so that $\sum p_i l_i$ will be the average path length. Let L_H, L_O and L_R respectively denote the average path length in the (unrestricted) Huffman tree, the B-restricted optimal tree and the B-restricted tree obtained by Algorithm ROT. Clearly

$$L_H \leq L_O \leq L_R. \tag{3}$$

The quantity we wish to bound is $L_R - L_O$, but for the given conditions, we show that even $L_R - L_H$ is small. After the rearrangement of $T(q)$, some leaves are on a different level than before. Let C_i denote this difference (level in $T^*(q) -$ level in $T(q)$) for the leaf labelled i. Then

$$L_R - L_H \leq \sum_{\{i\,:\,C_i > 0\}} p_i C_i. \tag{4}$$

The subtree $T(q)$ is transformed into a complete binary tree, thus

$$C_i \leq \log_2 N(q) = \alpha \log_2 n \tag{5}$$

holds for any i. On the other hand, since we assume that all the leaves in $T(q)$ were in the Huffman tree on levels $\geq R = s \log_2 n$, it follows from Katona & Nemetz [14, Theorem 1], that $p_i < 1/F_{R+1}$, where F_j is the j-th Fibonacci number. By [16, Exercise 1.2.1–4], we get

$$p_i < \left(\frac{1}{\phi}\right)^{R-1} = \frac{\phi}{\phi^{s \log_2 n}} = \frac{\phi}{n^{0.694s}}, \tag{6}$$

where $\phi = (1+\sqrt{5})/2$ is the golden ratio. The number of summands in (4) is clearly bounded by $N(q) = n^\alpha$. Putting this together, we have

$$L_R - L_H < \frac{\phi\, \alpha\, n^\alpha \log_2 n}{n^{0.694s}} < \frac{2 \log_2 n}{n^{0.694s-\alpha}},$$

which tends to zero when $s > 1.44\alpha$ and $n \to \infty$. ∎

In particular, suppose that $T(q)$ is almost the entire tree, say $N(q) = n - C \log n$ for some constant C (for example if $l_i = i$ for $1 \leq i \leq \log n$ and $B = O(\log n)$), the change in the average path length will still be small if $R > 1.44 \log_2(n - C \log n)$. If $T(q)$ is only a small subtree, say $N(q) = C \log n$, then the difference will tend to zero even for $R > 1.44\,(\log\log n + \log C)$.

The bounds in Theorem 1 are not very tight. The negative part on the right hand side of (4), $\sum_{\{i\,:\,C_i<0\}} p_i C_i$, was omitted, and the number of positive summands can be shown to be at most $N(q)/3$. In (5), at most 2^{j-1} of the C_i can be $\log_2 N(q) - j$, for $j \geq 1$, moreover if one of the C_i is $\log_2 N(q) - 1$, then at most one can be $\log_2 N(q) - 2$, etc. The upper bound on p_i in (6) is an extreme case; generally, p_i will be much smaller. It should also be noted that the imposed new depth B appears implicitly in the Theorem, since for a fixed Huffman tree of depth K, $N(q)$ is an increasing function and R a non-increasing function of $K - B$.

When the conditions of Theorem 1 are not satisfied, this is often due to a severe restriction on the depth, which causes extended changes in the structure of the Huffman tree. In such cases, there may be a significant difference between L_H and L_R. However, our experimental results (see Section 4) suggest that the major part of this difference must be attributed to $L_O - L_H$, whereas $L_R - L_O$ is still very small.

3. Refinements

We shall specify only how to change the l_i, since once they are fixed, the corresponding tree is defined. Obviously, we are restricted to changes in l_i which do not violate (1).

There are many possibilities to improve the algorithm presented in the previous section. For example, one could further climb upwards in the tree, and not stop at the first node q, which satisfies (2). The subtree $T(q)$ to be rearranged would then be larger, but if it is still small enough, one could choose the optimal among all the possible rearrangements. The number of possible rearrangements can be found in Table VII of [8]. But even when the smallest subtree $T(q)$ is chosen, there may be other possible rearrangements than the complete binary tree. For example, suppose $N(q) = 9$; then the complete binary tree of depth 4 would have its leaves on levels $3, \ldots, 3, 4, 4$. However, for certain weight distributions, the tree with leaves on levels $1, 4, \ldots, 4$ may be preferable. There is a natural trade-off between the amount of additional work one is willing to invest and the proximity to the optimum, but our experiments suggest that mostly, too large an effort cannot be justified. Since our main concern is the simplicity of the algorithm, we propose only the following two refinements.

3.1 Smoothing the transition point

Consider the Huffman code corresponding to the unrestricted tree which was constructed in Step 1b in §2, and write the codewords one below the other, sequentially from the shortest to the longest. Schematically, this column of codewords will have a more or less trapezoidal form (Figure 4(a)). While rearranging the subtree $T(q)$, some codewords become longer and others are shortened, so the rearrangement can be interpreted as changing a lower part of the trapezoid into a "rectangle" (Figure 4(b)).

(a) Huffman tree **(b)** after bounding **(c)** after smoothing

Figure 4: *Schematic representation of the changes in the tree*

Since the codewords which do not belong to $T(q)$ are not changed, there may be a great difference between $l_{n-N(q)}$ and $l_{n-N(q)+1}$, although the corresponding

probabilities differ perhaps only slightly. In Figure 4(b) this is symbolized by the "discontinuity" at the transition point between the rectangle and the trapezoid above it. Our first refinement will be to try to "smooth the edge", as depicted in Figure 4(c), of course only if such a transformation reduces the average codeword length.

Let $m = n - N(q)$ be the index of the last codeword which is not in $T(q)$ (the "lower base" of the trapezoid). The smoothing action will in practice be achieved by incrementing l_m and decrementing the lengths of the first few codewords of the rectangle. Two cases can occur: $l_{m+1} = l_{m+2}$, i.e., the two leftmost leaves in $T^*(q)$ are on the same level, or $l_{m+2} = l_{m+1} + 1$, because $T^*(q)$ is a full binary tree. In the former case we perform:

$$\begin{aligned} l_m &\leftarrow l_m + 1 \\ l_{m+1} &\leftarrow l_m \\ l_{m+2} &\leftarrow l_{m+2} - 1. \end{aligned}$$

Figures 5(a) and 5(b) show a part of the tree, resp. before and after these changes. In the latter case, a fourth statement must be added, so as to reestablish the equality in (1):

$$l_{m+3} \leftarrow l_{m+3} - 1,$$

which is well-defined, since $T^*(q)$ contains at least four leaves (Figures 5(c) and 5(d)).

(a) **(b)** **(c)** **(d)**

Figure 5: *Changes in the tree for "smoothing" the transition point*

These changes are justified only when

$$w_m < (l_{m+1} - l_m - 1)w_{m+1} + w_{m+2} + (l_{m+2} - l_{m+1})w_{m+3},$$

where w_{m+1} is multiplied by the difference of levels of the leaf $m+1$ before and after the change, and w_{m+3} is only added if $l_{m+2} \neq l_{m+1}$.

3.2 Transfer to adjacent blocks

The following refinement is based on the ideas of [8, Theorem 4]. As in the previous section, we consider that the codewords are written one below the other in order of non-increasing weights. Let E_r denote the block of codewords of length r, and let $t(r)$ and $b(r)$ be resp. the indices of the top and bottom element in E_r. Suppose there are at least two elements in E_r and E_{r-2} is not empty. If

$$w_{b(r-2)} \;<\; w_{t(r)} + w_{t(r)+1},$$

then we can reduce the average codeword length by executing:

$$\begin{aligned} l_{b(r-2)} &\leftarrow l_{b(r-2)} + 1 \\ l_{t(r)} &\leftarrow l_{t(r)} - 1 \\ l_{t(r)+1} &\leftarrow l_{t(r)+1} - 1. \end{aligned}$$

Let us call this sort of update (operating on elements of different blocks) an update of Type I.

Suppose there are at least three elements in E_r. If

$$w_{t(r)} \;>\; w_{b(r)-1} + w_{b(r)},$$

then we can reduce the average codeword length by executing:

$$\begin{aligned} l_{t(r)} &\leftarrow l_{t(r)} - 1 \\ l_{b(r)-1} &\leftarrow l_{b(r)-1} + 1 \\ l_{b(r)} &\leftarrow l_{b(r)} + 1. \end{aligned}$$

This sort of update (operating on elements of the same block) will be called an update of Type II. Note that after both types of updates, the equality in (1) is satisfied.

Figure 6: *Schematic representation of updates of Types I and II*

Figure 6 shows updates of both types; the current block in each case is indicated by the boldface lines, the codewords which are transferred are indicated by the dotted lines.

The question is now in which order these updates should be executed. If we want to assure a sequence of updates such that at its conclusion no further update of Type I or II is possible for any block, then we cannot simply process the blocks E_r in a single pass: while executing a Type II update in block E_r, codewords are transferred to blocks both above and below E_r. These transfers can in turn cause further updates in both E_{r-1} and E_{r+1}, and so on. Therefore, we could for example start with E_B and proceed bottom-up, passing from E_r to E_{r-1}, except when there was a Type II update in E_r, in which case we return to E_{r+1}. Unfortunately, there are two serious objections to this approach.

First, we have no reasonable bound on the number of steps the algorithm will execute. Theoretically it is possible that a certain codeword will be passed several times back and forth between adjacent blocks. We know that the number of updates is *finite*, since after each of them, $\sum w_i l_i$ is decreased at least by

$$\min\{\, x_{ij} = |w_i - w_j - w_{j+1}| \quad : \quad 1 \le i < j < n \text{ and } x_{ij} > 0\}.$$

But the number of updates can be $\Omega(n^2)$, as for Garey's algorithm (nevertheless, the use of the new algorithm can still be justified in certain cases, since the space complexity is reduced from quadratic to linear).

Secondly, even if all the possible updates were executed, this does not guarantee that the optimum is achieved. As example, take the weights to be the first few Fibonacci numbers, say $(w_1, \ldots, w_{14}) = (377, 244, \ldots, 3, 2, 1, 1)$. The corresponding Huffman code has $l_i = i$ for $1 \le i \le 13$ and $l_{14} = 13$. If we impose a bound $B = 6$, the lengths vector is changed by the first part of the algorithm to $(1, 2, 5, 5, 5, 5, 6, \ldots, 6)$, with $\sum w_i l_i = 2777$. The smoothing action of §3.1 then changes the vector to $(1, 3, 3, 4, 5, 5, 6, \ldots, 6)$ and reduces $\sum w_i l_i$ to 2633. But now, no block of codewords satisfies the conditions necessary for either type of update, and on the other side, the minimum of $\sum w_i l_i$ is 2599, which is obtained by the lengths vector $(2, 2, 3, 3, 4, 4, 6, \ldots, 6)$.

The first objection motivated us to design a *single pass* algorithm for the updates, even at the price of missing some of them; the second objection justified this approach, since anyhow, even a more sophisticated scanning procedure does not assure optimality. We now describe informally the procedure for updates of Type I and II (the formal algorithm appears in the Appendix).

Process the blocks E_r sequentially, starting with $r = B$ and decreasing r after each iteration. For a given block E_r, try first to execute an update of Type I; if it succeeded, repeat, until no Type I update is possible any more. Now try Type II updates (except for $r = B$, since codewords of length $B + 1$ are not allowed) and execute as many as possible. After each update, the limits of the affected blocks (E_{r-2}, E_{r-1} and E_r for Type I, E_{r-1}, E_r and E_{r+1} for Type II) are accordingly redefined. This terminates the current iteration and we pass to E_{r-1}.

Note that if a codeword is lengthened by a Type II update, it is not handled any more.

In order to bound the time complexity of this procedure, let us consider the codeword x, corresponding to w_j for some fixed $1 \leq j \leq n$. The codeword x changes possibly several times its length during the execution of the updates.

Lemma. *Immediately after the i-th time the codeword x was shortened in an update of Type I, the number of codewords above x in the same block is at least $(3/2)^{i-1}$.*

Proof: By induction on i. For $i = 1$, suppose x was transferred from E_r to E_{r-1}; there is another element y, which was transferred from E_{r-2} to the top of E_{r-1}. Even if E_{r-1} was initially empty and if x was the top element of E_r, there is, after the update, $(3/2)^0 = 1$ element above x in E_{r-1}.

Suppose the lemma is true for i, and that after the i-th time x is shortened in a Type I update, x belongs to E_r and has R elements above it in this block. Assume the next Type I shortening transfers x from E_h to E_{h-1}, for some $h \leq r$. It follows that the transfers of x from E_s to E_{s-1}, for $h < s \leq r$, were all during updates of Type II. However, Type II updates process the elements of a block E_s sequentially from the top downwards, so that if x is transferred to E_{s-1}, so were the elements above x in E_s. Therefore the number of elements above x in the same block, immediately after having transferred x, cannot decrease, and there are $R' \geq R$ elements above x in E_h. Since x passes from E_h to E_{h-1} during a Type I update, this is true also for the R' elements above x, because there is no Type I update after a Type II update in the same block. For every pair of elements which are transferred from E_h to E_{h-1}, there is a third element which is transferred from E_{h-2} to E_{h-1}, and which will also be above x in E_{h-1}. Thus the total number of elements above x in E_{h-1} is at least

$$\left.\begin{array}{ll} \frac{3}{2}R' + 1 & \text{if } R' \text{ is even} \\ \frac{3}{2}(R'-1) + 2 & \text{if } R' \text{ is odd} \end{array}\right\} \geq \frac{3}{2}R' \geq \frac{3}{2}\left(\frac{3}{2}\right)^{i-1} = \left(\frac{3}{2}\right)^{i}.$$

∎

Theorem 2. *The number of steps of the update algorithm is $O(n \log n)$.*

Proof: We count the number of updates, since there is a constant amount of work for each of them. If a codeword x is lengthened in a Type II update, it is transferred, together with an adjacent codeword, into a block which has already been handled by the algorithm. Thus the total number of Type II updates cannot exceed $n/2$. From the lemma we know that the number of times the i-th codeword can be shortened in a Type I update is at most $\lfloor \log_{3/2} i \rfloor + 1$, thus an upper bound on the total number of Type I updates is $n + \sum_{i=1}^{n} \lfloor \log_{3/2} i \rfloor = O(n \log n)$. ∎

Theorem 2 provides an upper bound for a theoretical worst case distribution; actually the total number of updates never exceeded $0.4\,n$ in all our experiments.

One could ask why we have chosen a bottom-up scan, from the longest codewords to the shorter ones. Alternatively, a top-down scan would start with the block of shortest codewords, and proceed to the longer ones. The definition of a Type I update must then be changed to act on the bottom element of E_r and the two top elements of E_{r+2}, instead of the two top elements of E_r and the bottom element of E_{r-2}. When only a small part of the tree is rearranged, there are no possible updates in the upper blocks, since these are identical to the blocks in the Huffman code, for which the codewords have optimal lengths. Thus a top-down scan yields updates only in the last few blocks, if at all. On the other hand, a bottom-up scan starts precisely at the blocks which are different from the corresponding blocks in the Huffman code, and by updates of Type I, the changes can propagate also to blocks above those of the reorganized subtree.

Another question may be why for a given block, the updates of Type I precede those of Type II. In a Type II update, two elements of the current block are transferred to a block with higher index, to which the algorithm will not return any more. Thus even if this transfer possibly allows another update, it will not be done. On the other hand, all the elements transferred in an update of Type I belong to blocks which are going to be handled later. Therefore, when coming to reduce the number of updates we are possibly missing, we try to minimize the number of Type II updates. Clearly, every Type I update executed from E_r decreases the chance that there will be afterward a Type II update in E_r.

We have implemented both the bottom-up and the top-down approaches. As expected, the former gave generally better results, even though there were some rare exceptions. We have also experimented with non-sequential scans, returning to previously visited blocks if they were changed by updates in the current block. For this variant, the difference between top-down and bottom-up was even smaller, and in any case, the total number of updates was smaller than $0.6\,n$.

4. Examples and Experimental Results

The new method significantly improves the time and space complexities of the optimal algorithms, but its usefulness in many practical applications depends on the actual loss in compression efficiency. There is however a problem in choosing an adequate model for a "typical" probability distribution. Lacking any other information, one often assumes a uniform probability distribution, but the corresponding Huffman code is of fixed length, so that the problem addressed in this work is not relevant. Another well-known distribution is Zipf's law, defined by the weights $w_i = H_n/i$, for $1 \le i \le n$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number. This law is believed to govern the distribution of the most common words in a large natural language text. The corresponding Huffman tree is not very skew, because there are no great differences between the smallest weights. For example, the depth K of the Huffman tree for Zipf's law with $n = 100$ and $n = 200$ is respectively 9 and 10. Imposing as bound $B = K - 1$, the relative increase of the average codeword

length when using the new algorithm instead of the optimal one is of 0.06% for $n = 100$ and of 0.03% for $n = 200$. For $B = K - 2$ (which is the minimal possible depth on these examples), the corresponding results are 0.74% and 1.42%.

Even though these figures can be considered as close to optimal, we feel that Zipf's law is not a representative example of a distribution on which one wishes to apply an algorithm for bounding the depth of a tree. Indeed, when there are many nodes on the lowest level as in this case, there can be a significant difference between the B-restricted and the non-restricted optimal trees, even when B is close to K. A more typical application would be a case where B is chosen by some technical constraint which is independent of K, and for which there is only a small number of nodes on levels $> B$; e.g., if one wishes to store the list of codewords in a table, it may be desirable to fit every codeword in one or two bytes. At the other end of the spectrum, the bound $B = \lceil \log_2 n \rceil$ will seldom be requested. The main reason for using variable length codes, in spite of their complicated processing, is their reduced storage requirements. Generally, this advantage is almost lost with such a bound, so one will rarely prefer this alternative to the simple and almost as efficient fixed length code.

We have therefore decided to test the compression efficiency of the new method empirically on various "real-life" weight distributions, similarly to Knuth [17], who checked his dynamic Huffman coding algorithm on, e.g., a file of Grimm's *Fairy Tales*. For any given set of n weights, the Huffman tree was built, with depth K. Using then Garey's algorithm, the optimal B-restricted trees were constructed for all possible values of B, $\lceil \log_2 n \rceil \leq B \leq K - 1$. Finally the optimal trees were compared first with the trees obtained by the ROT Algorithm of §2, then with the improved trees based on the refinements of §3.

The first class of sets of weights consists of probability distributions of the characters of the alphabet for various natural languages. The distribution of the 26 letters of English is in Heaps [10]; the distribution of the 29 letters of Finnish is from Pesonen [19]; the distribution for French (including blank) is from Brunet [2]; for German, the distribution of 30 letters (including blank and *Umlaute*) is given in Bauer & Goos [1]; for Hebrew (30 letters including two kinds of apostrophes and blank), we have computed the distribution using the database of the Responsa Retrieval Project (see for example Fraenkel [4]) of about 40 million Hebrew and Aramaic words; the distribution for Italian (26 letters) can be found in Gaines [6], and for Russian (32 letters) in Herdan [11]. The results for this first set are summarized in Table 1.

	Statistics	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
English	4.1852	**opt**	0.01	0.30	**opt**	**opt**					
	10	**opt**	0.01	**opt**	**opt**	**opt**					
Finnish	4.0449	**opt**	0.96	1.69	0.70	0.45	0.18	0.12	0.01	**opt**	0.00
	15	**opt**	**opt**	**opt**	**opt**	**opt**	**opt**	0.02	**opt**	**opt**	**opt**
French	3.9708	1.61	1.14	**opt**	0.12	0.04	0.03				
	11	**opt**	**opt**	**opt**	**opt**	0.03	**opt**				
German	4.1479	**opt**	0.44	0.71	0.12	0.15	0.01	**opt**			
	12	**opt**	0.14	0.11	**opt**	0.07	0.01	**opt**			
Hebrew	4.2851	**opt**	**opt**	0.01	**opt**						
	9	**opt**	**opt**	**opt**	**opt**						
Italian	4.0000	**opt**	1.06	0.48	0.21	**opt**	**opt**				
	11	**opt**	0.03	0.00	0.17	**opt**	**opt**				
Russian	4.4480		0.42	0.11	0.04						
	9		0.09	**opt**	**opt**						

Table 1: *Experimental results — Distribution of letters in natural languages*

The first column contains statistical information for each language: the average length of a codeword for the unrestricted Huffman tree and below the natural depth of this tree. The following columns correspond each to another value of the bound B, which appears in the header line. For each language and each column, two values are listed: the upper one is the relative increase (in percent) of the average codeword length when the ROT Algorithm of §2 is used instead of Garey's optimal algorithm, i.e., using the notation of Theorem 1: $(L_R/L_O - 1) \times 100$. The number listed below is the corresponding result when the improved algorithm of §3 is used. When the optimum is reached, this is indicated by the letters **opt**, hence if an entry contains 0.00, this means that the algorithm is sub-optimal, but that the deviation from the minimum is smaller than 0.005%. These explanations apply also to the two following tables.

The second class of weight distributions included larger sets: the distribution of bigrams in English and Hebrew. For English, the probabilities given in [10] are of low precision (10^{-4}), therefore the number of character-pairs which have "non-zero" probability is only 378; for Hebrew, we have computed the distribution with high precision (10^{-10}) and got 743 pairs with non-zero probability. Table 2 summarizes the experiments on the bigrams.

	Statistics	*9*	*10*	*11*	*12*	*13*	*14*	*15*
English	7.6085 13	2.82 0.00	1.25 0.01	0.05 0.03	0.03 0.00			
Hebrew	8.0370 23		6.54 0.25	1.44 0.03	0.23 0.04	0.06 0.01	0.09 0.00	0.01 0.00
		16	*17*	*18*	*19*	*20*	*21*	*22*
Hebrew	**con't**	0.01 0.00	0.01 0.00	0.00 0.00	0.00 0.00	0.00 0.00	0.00 0.00	0.00 0.00

Table 2: *Experimental results — Distribution of bigrams*

In order to test distributions of another kind, we have computed the frequency of appearance of different symbols in 5762 source lines of PLI programs. The number of different characters was 59, but more than 2/3 of this file consisted of blanks. Therefore we have also evaluated the distribution of a similar file, for which leading and trailing blanks in each record were omitted. The first line of Table 3, headed PLI+, corresponds to the distribution with leading and trailing blanks, the second line, headed PLI−, corresponds to the distribution after having omitted these blanks.

In the following method for the compression of sparse bit-vectors, Huffman coding is applied to items of a completely different nature: first the given vectors are partitioned into bytes (8-bit blocks), then statistics are collected on the frequency of appearance of the elements of a set S, consisting of the 255 possible non-zero bytes to which t elements $\{a_0, \ldots, a_{t-1}\}$ have been adjoined; the latter represent the first t basis elements of a *numeration system*, e.g., $\{1, 2, 4, 8, \ldots\}$ for the standard binary numeration system. The idea is to consider the length k of each *run* of 0-bytes, and to "decompose" k uniquely as a linear combination of the basis elements a_i. Finally, Huffman codes are assigned to the elements of S; to the different non-zero bytes correspond different codewords, and every run of 0-bytes is encoded by the codewords of the corresponding basis elements. The set of basis elements is a parameter; various choices are suggested in [5], where this method is described in more detail.

	Statistics	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
PLI+	2.6757 16	14.1 **opt**	**opt** **opt**	0.60 **opt**	0.38 0.08	0.04 0.04	0.06 0.02	0.01 0.01	0.00 0.00	**opt** **opt**	0.00 0.00
PLI−	4.8518 15	2.34 **opt**	0.59 0.59	0.80 **opt**	0.11 **opt**	0.07 0.00	0.02 0.00	0.01 0.00	**opt** **opt**	**opt** **opt**	
POW2	4.6992 14				51.6 13.8	8.13 4.40	1.12 0.91	0.06 0.02	**opt** **opt**		
FIB2	5.0513 14				43.0 9.70	25.1 3.87	1.09 0.77	0.03 0.03	**opt** **opt**		

Table 3: *Experimental results — PLI and compression of bit-vectors*

For the present application, we have chosen the standard binary numeration system (third line of Table 3, headed POW2) and the binary Fibonacci numeration system (see [16], Exercise 1.2.8–34), the basis elements of which are Fibonacci numbers (fourth line of Table 3, headed FIB2). The statistics were collected from 56588 bit-vectors of 42272 bits each, which were constructed at the Responsa Project: each vector serves as an "occurrence map" for a different word, the bit-position referring to the number of the document, where the value at position i is 1 if and only if the given word appears in the i-th document.

The experiments show that actually the optimal value is often reached, and in the great majority of the cases, the deviation from the optimum is smaller than 1%. The rare exceptions are usually when the bound B has its minimal possible value, but, as was pointed out earlier, in this case one will rather use a fixed length code. Therefore the method presented herein is an attractive alternative in situations where n is large (so that the optimal method is not only very time-consuming, but often even not feasible, because of the quadratic space complexity), and B is close to the natural depth of the Huffman tree — a case for which the optimal algorithm takes its longest time. For example, due to the space requirements, we could run Garey's algorithm on the Hebrew bigrams only by night and batch; choosing $B = 17$, the job took about 18 minutes of CPU on our IBM 3081. On the other hand, using our sub-optimal method we got in a few seconds, on-line, a result which exceeded the optimal one only by 0.00007%.

APPENDIX

We bring here the formal description of the Algorithm ROT for bounding the depth of a binary tree. The algorithm will be presented in an Algol-like language that uses "**fi**" and "**od**" to close "**if**" and "**do**". For the second refinement, a bottom-up single scan is executed. However, the statements necessary to execute a non-sequential scan (returning to a block already handled if it was touched by the last update), are added as comments into boxes with heading: *For non-sequential scan*.

The data structures involved are:

1. an extended binary tree; each node q has the following fields: $left(q)$, $right(q)$ and $N(q)$, storing resp. a pointer to the left son, a pointer to the right son and the number of leaves in the subtree rooted at q.
2. a stack ST; as in [16], the statements $ST \Leftarrow p$ and $p \Leftarrow ST$ are used resp. for "**push** p into the stack ST" and "**pop** the top element from ST and put it into p".
3. two auxiliary vectors $t(r)$ and $b(r)$, $0 \leq r \leq n$, containing the index of the top, resp. bottom, element of E_r, the block of codewords of length r. If $E_r = \emptyset$, then $t(r) = b(r) = 0$.

For a given set of integers l_i such that $\sum 2^{-l_i} = 1$, the following procedure BUILD(q, $level$) constructs a "canonical" tree rooted at q, which is on level $level$, and having its leaves on levels l_i. The procedure uses the global variable i, the index to the next element in the sequence l_i.
The function *new* allocates a new node, the function *free* returns unneded nodes to the pool of available space, and Λ stands for the null-pointer.

```
procedure  BUILD(q, level)
    if  level = l_i  then
        left(q) ← right(q) ← Λ
        N(q) ← 1
        i ← i + 1
    else
        L ← new
        left(q) ← L
        call  BUILD(L, level + 1)
        R ← new
        right(q) ← R
        call  BUILD(R, level + 1)
        N(q) ← N(L) + N(R)
    fi
end BUILD
```

A L G O R I T H M R O T

```
begin
    head ← new
    i ← 1
    call  BUILD( head , 0 )
```

Push the nodes on the path from the root to (but not including) the rightmost leaf into the stack ST.

```
    p ← head
    level ← 0
    repeat
        level ← level + 1
        ST ⇐ p
        p ← right(p)
    until  right(p) = Λ
```

p now points to the rightmost leaf; $level$ is the depth of the tree, and the top element in ST is the father of p.

if $level > B$ **then**
repeat
$level \leftarrow level - 1$
$q \Leftarrow ST$
until $N(q) \leq 2 * *(B - level)$

Here, q points to the root of the subtree which will be reorganized; first update the lengths, then the procedure BUILD can be invoked.

$i \leftarrow n - N(q) + 1$
for $j \leftarrow i$ **to** $i - N(q) + 2 * *(B - level) - 1$ **do**
$l_j \leftarrow B - 1$ **od**
for $j \leftarrow i - N(q) + 2 * *(B - level)$ **to** n **do**
$l_j \leftarrow B$ **od**
call BUILD$(q, level)$

First refinement: Smoothing the transition point
i is the index of the leftmost leaf of $T^*(q)$, the reorganized subtree; *changed* is a Boolean variable indicating if the current tree will be changed.

changed $\leftarrow$ **false**
if $(i > 1)$ **and** $(l_{i-1} \leq l_i - 2)$ **and**
$\big(w_{i-1} < (l_i - l_{i-1} - 1) \times w_i + w_{i+1} + (l_{i+1} - l_i) \times w_{i+2}\big)$ **then**
if $l_{i+1} \neq l_i$ **then** $l_{i+2} \leftarrow l_{i+2} - 1$ **fi**
$l_{i-1} \leftarrow l_{i-1} + 1$
$l_i \leftarrow l_{i-1}$
$l_{i+1} \leftarrow l_{i+1} - 1$
changed $\leftarrow$ **true**
fi

Second refinement: Transfer to adjacent blocks
First, the vectors $t(r)$ and $b(r)$ are initialized.

for $r \leftarrow 0$ **to** n **do**
$t(r) \leftarrow b(r) \leftarrow 0$ **od**
$j \leftarrow 0$
for $r \leftarrow 1$ **to** n **do**
if $l_r \neq j$ **then**
$b(j) \leftarrow r - 1$
$j \leftarrow l_r$
$t(j) \leftarrow r$
fi
od
$b(l_n) \leftarrow n$

Execute now a bootom-up scan over the blocks, trying first updates of Type I; the Boolean variable *succeding* serves to control the loop of updates.

for $r \leftarrow B$ **to** 2 **step** -1 **do**
  succeding $\leftarrow$ **true**
  while *succeding* **do**
    succeding $\leftarrow$ **false**
    if $(b(r-2) > 0)$ **and** $(b(r) > t(r))$ **and**
      $(w_{b(r-2)} < w_{t(r)} + w_{t(r)+1})$ **then**
        succeding $\leftarrow$ **true**
        changed $\leftarrow$ **true**

        Update of lengths

        $l_{b(r-2)} \leftarrow l_{b(r-2)} + 1$
        $l_{t(r)} \leftarrow l_{t(r)} - 1$
        $l_{t(r)+1} \leftarrow l_{t(r)+1} - 1$

        Update of vectors $t(r)$ and $b(r)$

        $t(r-1) \leftarrow b(r-2)$
        $b(r-1) \leftarrow t(r) + 1$
        $b(r-2) \leftarrow b(r-2) - 1$
        $t(r) \leftarrow t(r) + 2$
        if $t(r-2) > b(r-2)$ **then**
          $t(r-2) \leftarrow b(r-2) \leftarrow 0$ **fi**
        if $t(r) > b(r)$ **then** $t(r) \leftarrow b(r) \leftarrow 0$ **fi**
    fi
  od ⟦ end of loop for Type I updates ⟧

Now try updates of Type II, but not for $r = B$.
For non-sequential scan:
*changed*2 is a Boolean variable, indicating if there will be any change due to Type II updates.
  *changed*2 $\leftarrow$ **false**

succeding $\leftarrow$ **true**
while *succeding* **do**
  succeding $\leftarrow$ **false**
  if $(r < B)$ **and** $(b(r) - 1 > t(r))$ **and**
    $(w_{t(r)} > w_{b(r)-1} + w_{b(r)})$ **then**
      succeding $\leftarrow$ **true**

changed ← **true**

Update of lengths *For non-sequential scan:* *changed*2 ← **true**

$l_{t(r)} \leftarrow l_{t(r)} - 1$
$l_{b(r)-1} \leftarrow l_{b(r)-1} + 1$
$l_{b(r)} \leftarrow l_{b(r)} + 1$

Update of vectors $t(r)$ and $b(r)$

$b(r-1) \leftarrow t(r)$
$t(r) \leftarrow t(r) + 1$
$b(r) \leftarrow b(r) - 2$
if $t(r-1) = 0$ **then** ⟦ block E_{r-1} was empty ⟧
$t(r-1) \leftarrow b(r-1)$ **fi**
if $t(r) > b(r)$ **then** ⟦ block E_r became now empty ⟧
$t(r) \leftarrow b(r) \leftarrow 0$ **fi**

For non-sequential scan: $t(r+1) \leftarrow b(r) - 1$ **if** $b(r+1) = 0$ **then** ⟦ block E_{r+1} was empty ⟧ $b(r+1) \leftarrow t(r+1) + 1$ **fi**

fi
od ⟦ end of loop for Type II updates ⟧

For non-sequential scan: **if** *changed*2 **then** $r \leftarrow r - 2$ **fi**

od ⟦ end of second refinement ⟧

The vector of lengths being updated, the procedure BUILD will be invoked again if there is any change, after having freed the space occupied by the Huffman tree.

if *changed* **then**
free space of the tree
head ← *new*
$i \leftarrow 1$
call BUILD(*head* , 0)
fi
fi ⟦ end of bounding the depth ⟧
end ⟦ of Algorithm ROT ⟧

REFERENCES

[1] **Bauer F.L., Goos G.,** *Informatik, Eine einführende Übersicht, Erster Teil,* Springer Verlag, Berlin (1973).

[2] **Brunet E.**, *Le Vocabulaire de Jean Giraudoux — Structure et Evolution*, Editions Slatkine, Genève (1978).

[3] **Choueka Y., Klein S.T., Perl Y.**, Efficient variants of Huffman codes in high level languages, *Proc. 8-th ACM-SIGIR Conf.,* Montreal (1985) 122–130.

[4] **Fraenkel A.S.,** All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, expanded summary, *Jurimetrics J.* **16** (1976) 149–156.

[5] **Fraenkel A.S., Klein S.T.,** Novel compression of sparse bit-strings — preliminary report, *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 169–183.

[6] **Gaines H.F.,** *Cryptanalysis, A Study of Ciphers and their solution*, Dover Publ. Inc., New York (1956).

[7] **Garey M.R.**, Optimal binary search trees with restricted maximal depth, *SIAM J. of Comp.* **3** (1974) 101–110.

[8] **Gilbert E.N.,** Codes based on inaccurate source probabilities, *IEEE Trans. on Inf. Th.* **IT–17** (1971) 304–314.

[9] **Gilbert E.N., Moore E.F.**, Variable-length binary encodings, *The Bell System Technical Journal* **38** (1959) 933–968.

[10] **Heaps P.,** *Information Retrieval, Computational and Theoretical Aspects*, Academic Press (1978).

[11] **Herdan G.,** *The Advanced Theory of Language as Choice and Chance*, Springer-Verlag, New York (1966).

[12] **Hu T.C., Tan K.C.,** Path length of binary search trees, *SIAM J. of Appl. Math.* **22** (1972) 225–234.

[13] **Huffman D.,** A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[14] **Katona G.O.H., Nemetz T.O.H.,** Huffman codes and self-information, *IEEE Trans. on Inf. Th.* **IT–22** (1976) 337–340.

[15] **Knuth D.E.,** Optimum binary search trees, *Acta Informatica* **1** (1971) 14–25.

[16] **Knuth D.E.,** *The Art of Computer Programming, Vol I, Fundamental algorithms*, Addison-Wesley, Reading, Mass. (1973).

[17] **Knuth D.E.,** Dynamic Huffman coding, *J. of Algorithms* **6** (1985) 163–180.

[18] **McMillan B.,** Two inequalities implied by unique decipherability, *IRE Trans. on Inf. Th.* **IT–2** (1956) 115–116.

[19] **Pesonen J.,** Word inflexions and their letter and syllable structure in Finnish newspaper text, Research Rep. 6/1971, Dept. of Special Education, University of Jyräskylä, Finland (in Finnish, with English summary).

[20] **Schwartz E.S., Kallik B.,** Generating a canonical prefix encoding, *Comm. of the ACM* **7** (1964) 166–169.

[21] **Van Leeuwen J.,** On the construction of Huffman trees, *Proc. of the* 3^{rd} *ICALP Conf.*, Edinburgh University Press (1976) 382–410.

[22] **Van Voorhis D.C.**, Constructing codes with bounded codeword lengths, *IEEE Trans. on Inf. Th.* **IT–20** (1974) 288–290.