SIAM J. COMPUT.
Vol. 25, No. 6, pp. 1281–1292, December 1996
© 1996 Society for Industrial and Applied Mathematics
008

PREFIX CODES: EQUIPROBABLE WORDS, UNEQUAL LETTER COSTS*

MORDECAI J. GOLIN† AND NEAL YOUNG‡

Abstract. We consider the following variant of Huffman coding in which the costs of the letters, rather than the probabilities of the words, are nonuniform: "Given an alphabet of r letters *of nonuniform length*, find a minimum-average-length prefix-free set of n codewords over the alphabet"; equivalently, "Find an optimal r-ary search tree with n leaves, where each leaf is accessed with equal probability but the cost to descend from a parent to its ith child depends on i." We show new structural properties of such codes, leading to an $O(n \log^2 r)$-time algorithm for finding them. This new algorithm is simpler and faster than the best previously known $O(nr \min\{\log n, r\})$-time algorithm, due to Perl, Garey, and Even [*J. Assoc. Comput. Mach.*, 22 (1975), pp. 202–214].

Key words. algorithms, Huffman codes, prefix codes, trees

AMS subject classification. 68Q25

1. Introduction.

The well-known Huffman coding problem [3] is the following: given a sequence of access probabilities $\langle p_1, p_2, \ldots, p_n \rangle$, construct a binary prefix code $\langle w_1, w_2, \ldots, w_n \rangle$ minimizing the expected length $\sum_i p_i \cdot \text{length}(w_i)$. A *binary prefix code* is a set of binary strings, none of which is a prefix of another.

A natural generalization of the problem is to allow the words of the code to be strings over an arbitrary alphabet of $r \geq 2$ letters and to allow each letter to have an arbitrary nonnegative length. The length of a codeword is then the sum of the lengths of its letters. For instance, the "dots and dashes" of Morse code are a variable-length alphabet with length corresponding to transmission time. (See Figure 1.) This generalization of Huffman coding to a variable-length alphabet has been considered by many authors, including Altenkamp and Mehlhorn [1] and Karp [5]. Apparently, no polynomial-time algorithm for it is known, nor is it known to be NP-hard.

A prefix code in which the codewords $\langle w_1, w_2, \ldots, w_n \rangle$ are in alphabetical order is called *alphabetic* [1]. In this case, the underlying tree represents an r-ary *search tree.* The length of the ith letter corresponds to the time required to descend from a node into its ith subtree. This time is often a function of i in search-tree algorithms, for instance, when the subtree to descend into is chosen by sequential search. An optimal alphabetic code thus corresponds to a minimum-expected-cost search tree.

In this paper, we consider the special case in which the codewords occur with equal probability, i.e., each p_i equals $1/n$. With this restriction, the alphabetic and nonalphabetic problems are equivalent. The problem may be viewed as a variant of Huffman coding in which the lengths of the letters, rather than the codeword probabilities, are nonuniform. Alternatively, it may viewed as the problem of finding an optimal r-ary search tree, where the search queries are uniformly distributed but the time to descend from a parent to its ith child depends on i. For the complexity results stated in this paper, the algorithms return a tree representing an optimal code.

* Received by the editors May 25, 1994; accepted for publication (in revised form) March 1, 1995.

† Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong (golin@cs.ust.hk). The research of this author was partially supported by HK RGC Competitive Research grant HKUST 181/93E.

‡ UMIACS, University of Maryland, College Park, MD 20742. Current address: Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510 (neal.young@dartmouth.edu). The research of this author was partially supported by NSF grants CCR-8906949 and CCR-9111348.

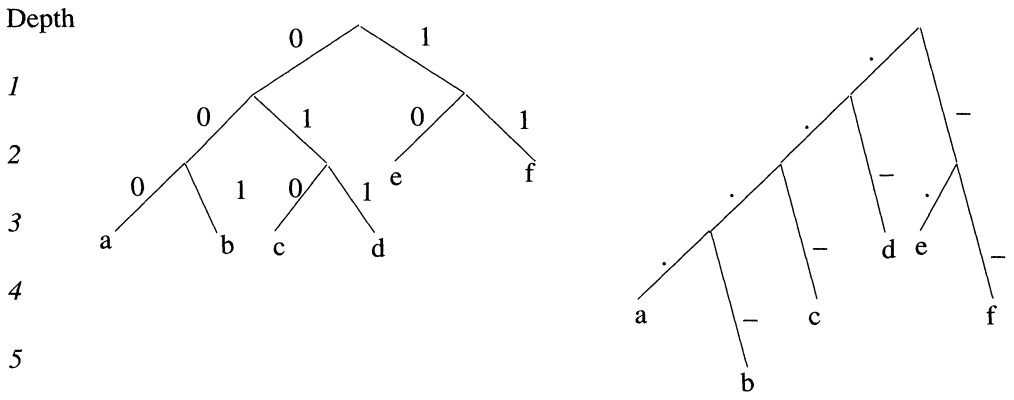


FIG. 1. *Two trees for the six symbols a, b, c, d, e, and f, each occurring with probability* 1/6. *The tree on the left is the optimal tree that uses the alphabet* $\{0,1\}$, *with length*(0) = *length*$(1) = 1$, *while the tree on the right is for the alphabet* $\{.,\_\}$ *with length*$(.) = 1$ *and length*$(\_) = 2$. *The corresponding sets of codewords are*

$$a = 000, \quad b = 001, \quad c = 011, \quad d = 011, \quad e = 10, \quad f = 11$$

and

$$a = ...\,, \quad b = ...\_\,, \quad c = ..\_\,, \quad d = .\_\,, \quad e = \_.\,, \quad f = \_\_\,.$$

In 1989, Kapoor and Reingold [4] described a simple $O(n)$-time algorithm for the binary case $r = 2$. In 1975, Perl, Garey, and Even [7] gave an $O(rn \min\{r, \log n\})$-time algorithm (though due to a typographical error, their abstract incorrectly claims an $O(rn)$-time algorithm). In the same year, Cot [2] described an $O(r^2 n)$-time algorithm. In 1971, Varn [8] gave an algorithm without analyzing its complexity. It appears Varn's algorithm requires $\Omega(rn)$ time.

In this paper, we describe an $O(n \log^2 r)$-time algorithm based on new insights into the structure of optimal trees. In §2, we define *shallow* and *proper* trees and prove that some proper shallow tree is optimal. In §3, we develop the algorithm, which efficiently constructs all proper shallow trees and returns one representing an optimal prefix code.

2. Shallow trees. Fix an instance of the problem, given by the respective lengths $\langle c_1 \le c_2 \le \cdots \le c_r \rangle$ of the r letters in the alphabet and the number n of (equiprobable and prefix-free) codewords required. We assume the standard tree representation of prefix codes, as described in the following definition.

DEFINITION 2.1. *The* infinite r-ary tree *is the infinite, rooted, r-ary tree. Each tree edge has a length and a label—an edge going from a node to its ith child has length* c_i *and is labeled with the ith letter in the alphabet.*

A node *is a node of the infinite r-ary tree.* The finite words over the alphabet of r letters correspond to the nodes. The labels along the path from the root to any node spell the corresponding word and the length of the path is the length of this word. A prefix code corresponds to a set of nodes none of which is a descendant of another. (See Figure 1.)

DEFINITION 2.2. *A* tree *is any subtree T of the infinite r-ary tree containing the root. In any tree, n of the leaves will be identified as* terminals; *their corresponding words form a prefix code. The remaining nodes in the tree are referred to as* nonterminals.

Given a node u, the notation $child_i(u)$ denotes u's ith child; $depth(u)$ denotes the depth (the length of the corresponding codeword); $parent(u)$ denotes the parent.

The cost $c(T)$ *of such a tree is the sum of the depths of the terminals—also called the* external weighted path length *of the tree.*

A proper *tree is a tree in which every nonterminal has at least two children.*

The goal is to find an optimal tree with n terminals. It is easy to see that some optimal tree is proper; thus we restrict our attention to proper trees.

Our basic tool for understanding the structure of optimal trees is a swapping argument. For example, in any proper optimal tree, no nonterminal is deeper than any terminal. Otherwise, the terminal and the subtree rooted at the nonterminal could be swapped, decreasing the average depth of the terminals.

We use a swapping argument to prove that an optimal proper tree has the following form for some m. The nonterminals are the m shallowest (i.e., least-depth) nodes of the infinite tree, while the terminals are the n shallowest available children of these nodes in the infinite tree. We call such a tree *shallow*; here is the precise definition.

DEFINITION 2.3. *A tree T is* shallow *provided that*

(i) *for any nonterminal $u \in T$ and any node w (not necessarily in T) that is not a nonterminal, $depth(u) \leq depth(w)$ and*

(ii) *for any terminal $u \in T$ and any node w that is not in T but is a child of a nonterminal, $depth(u) \leq depth(w)$.*

Note that a nonterminal of an (improper) shallow tree might have no children in the tree. This is why we refer to "terminal" and "nonterminal" nodes in place of the more common "internal nodes" and "leaves."

As a simple example, consider the basic binary tree; $r = 2$, $c_1 = c_2 = 1$. A proper binary tree T will be shallow if and only if there is some depth l such that (a) every node u in the infinite tree with $depth(u) < l$ is a nonterminal in T and (b) all terminals of T are on levels l and $l + 1$. Conditions (a) and (b) are necessary and sufficient conditions for T to have minimum external path length among all binary trees with the same number of leaves; see, e.g., [6, §5.3.1]. So, a binary tree has minimum external path length for its number of leaves if and only if it is shallow. For example, the binary tree on the left-hand side of Figure 1 has minimum external path length among all trees with six leaves because it fulfills conditions (a) and (b) with $l = 2$. As we will see later, though, for most values of r and c_i, shallowness alone does not imply optimality. However, if a shallow tree has the right number of nonterminals, then it is optimal.

LEMMA 2.4. *Let m^* be the minimum number of nonterminals in any optimal tree. Then any shallow tree with m^* nonterminals is optimal and proper.*

Proof. Fix a shallow tree T with m^* nonterminals. We will show the existence of an optimal tree with the same nonterminals as T. Since T is shallow, by property (ii), this will imply that T is optimal. By the choice of m^*, T is also proper (otherwise there would be an optimal proper tree with fewer nonterminals).

It remains to show the existence of an optimal tree with the same nonterminals as T. Let T^* be an optimal (and therefore proper) tree with m^* nonterminals. Let N and N^* be the sets of nonterminals of T and T^*, respectively. If $N = N^*$, we are done. Otherwise, let u be a minimum-depth node in $N - N^*$, so that u's parent is in N^*. Let u^* be a node in $N^* - N$. Note that, since T is shallow, $depth(u^*) \geq depth(u)$ but that, in T^*, u^* is a nonterminal (with at least two terminal descendants) while u is either a terminal or not present.

In T^*, swap the subtrees rooted at u and u^*. Specifically, make u a nonterminal

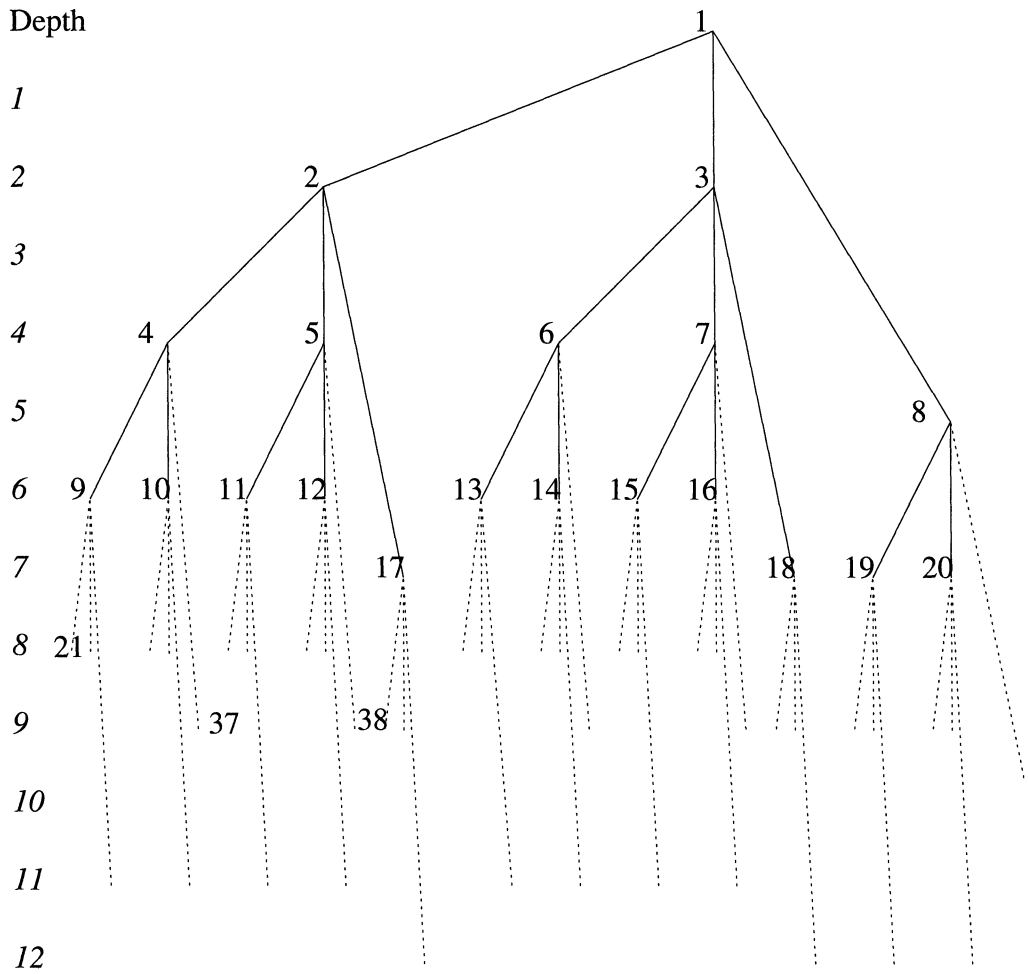


FIG. 2. *The top of a labeled infinite tree with* $r = 3$, $c_1 = 2$, $c_2 = 2$, *and* $c_3 = 5$.

and, for each descendant v^* of u^*, delete it and add the corresponding descendant v of u. If v^* was a terminal, make v a terminal; otherwise, make v a nonterminal. If u was a terminal, make u^* a terminal; otherwise, delete u^*. Call the resulting tree T'.

From $depth(u^*) \geq depth(u)$, it follows that $c(T') \leq c(T^*)$. Thus T' is also optimal. Note that T' shares one more nonterminal with T than does T^*. Thus repeated swapping produces an optimal tree with the same nonterminals as T. □

Note that $m^* \geq (n-1)/(r-1)$ since each node has degree at most r.

COROLLARY 2.5. *Let* $m_{\min} = \lceil (n-1)/(r-1) \rceil$. *Let* $\langle T_{m_{\min}}, T_{m_{\min}+1}, T_{m_{\min}+2}, \ldots \rangle$ *be any sequence of shallow trees such that for each* m, T_m *has* m *nonterminals. Then one of the* T_m *is proper and optimal.*

The algorithm generates a sequence of shallow trees as above and returns the one which has minimum cost. The lemma guarantees that this tree will be optimal. The rest of the paper is devoted to examining the properties of shallow trees which enable the enumeration of the proper shallow trees in $O(n \log^2 r)$ time.

2.1. Defining the trees.

Ordering the nodes. Label the nodes of the infinite tree as $1, 2, 3, \ldots$ in order of increasing depth. Break ties arbitrarily, except that if two nodes u and w are of equal depth, both are ith children of their respective parents, and $\text{parent}(u) < \text{parent}(w)$, then let $u < w$ (this is needed for Lemma 3.2). For the sake of notation, identify each node with its label so that 1 is the root, 2 is a minimum-depth child of the root, etc. Figure 2 illustrates the top section of such a labeling for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. These values of r and c_j are the ones we use in all later examples.

DEFINITION 2.6. *For each $m \geq m_{\min}$, define T_m to be the tree whose nonterminals are $\{1, \ldots, m\}$ and whose terminals are the minimum n nodes among the children of $\{1, \ldots, m\}$ in $\{m+1, m+2, \ldots\}$.*

Thus T_m is the "shallowest" tree with m nonterminals with respect to the ordering of the nodes. Since the ordering of the nodes respects depth, each T_m is shallow. Figure 3 presents T_5, T_6, T_7, and T_8 for $n = 10$ using the labeling of Figure 2.

2.2. Relation of successive trees. Next, we turn our attention to the relation of T_{m+1} to T_m.

LEMMA 2.7. *For $m \geq m_{\min}$, the new nonterminal (node $m+1$) in T_{m+1} is the minimum terminal of T_m.*

Proof. The parent of $m+1$ is in $\{1, \ldots, m\}$, so $m+1$ is the minimum child of $\{1, \ldots, m\}$ in $\{m+1, m+2, \ldots\}$. The result follows from the definition of T_m. □

LEMMA 2.8. *For $m \geq m_{\min}$, provided the new nonterminal (node $m+1$) in T_{m+1} has at least one child, each terminal of T_{m+1} is either a child of $m+1$ or a terminal of T_m.*

Proof. Let node $m+1$ have d children in T_{m+1}. Let $\mathcal{C}$ denote the set of children of nodes $\{1, \ldots, m\}$ in $\{m+1, m+2, \ldots\}$. The terminals of tree T_{m+1} consist of the minimum d children of node $m+1$ together with the minimum $n-d$ nodes in $\mathcal{C} - \{m+1\}$. These $n-d$ nodes together with node $m+1$ (the minimum node in $\mathcal{C}$) are the $n-d+1$ minimum nodes in $\mathcal{C}$. If $d \geq 1$, then by the definition of T_m, each such node is a terminal in T_m. □

The main significance of Lemmas 2.7 and 2.8 is that they will allow an efficient construction of T_{m+1}. Moreover, they imply that if T_m is not proper, then neither is any subsequent tree.

LEMMA 2.9. *One of the trees $\langle T_{m_{\min}}, T_{m_{\min}+1}, \ldots, T_{m_{\max}} \rangle$ is optimal and proper, where $m_{\max} = \min\{m : T_{m+1} \text{ is improper}\}$.*

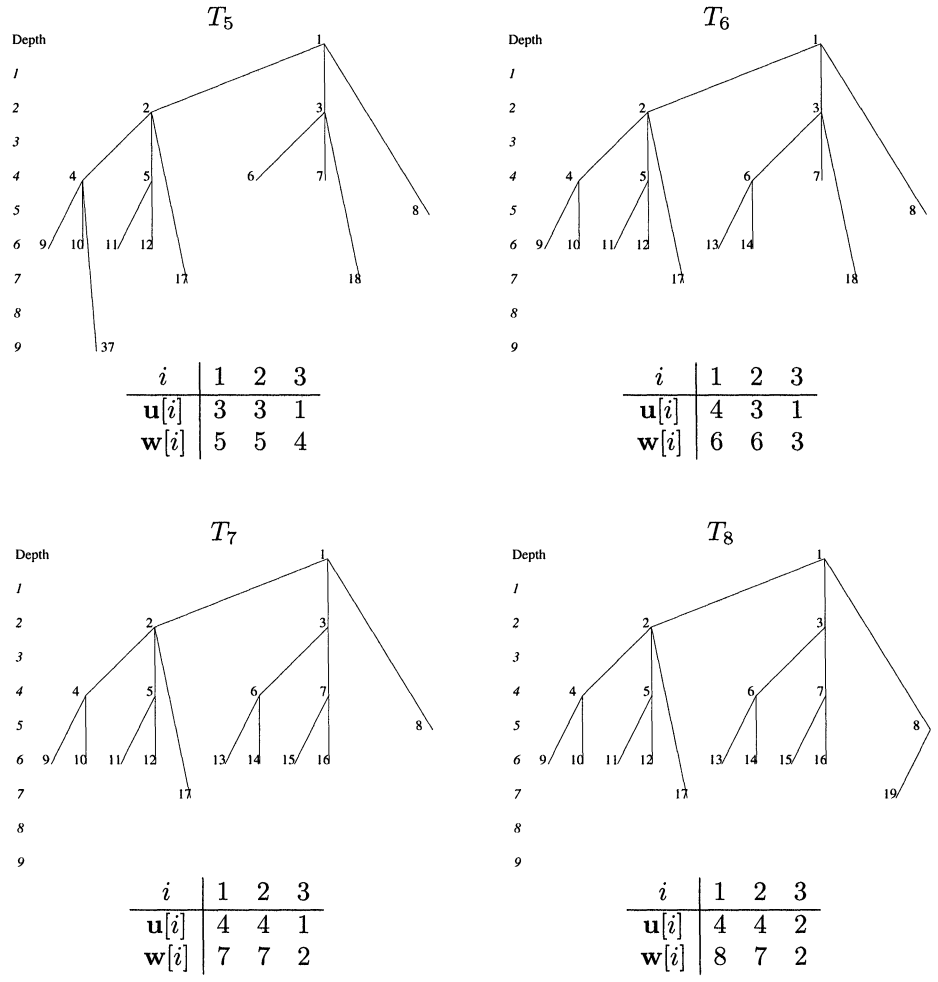
Proof. By Lemma 2.8, if T_m is improper, then so is T_{m+1}—either node $m+1$ has no children in T_{m+1} or the nonterminal in T_m that had less than two children also has less than two children in T_{m+1}. Hence, for each $m > m_{\max}$, tree T_m is improper. Thus Corollary 2.5 implies that one of the trees $\langle T_{m_{\min}}, T_{m_{\min}+1}, \ldots, T_{m_{\max}} \rangle$ is proper and optimal. □

For $n = 10$, $m_{\min} = \lceil \frac{10-1}{3-1} \rceil = 5$ and (as shown in Figure 3) T_8 is improper. The lemma then implies that one of T_5, T_6, or T_7 must have minimum external path length. Calculation shows that T_6 with $c(T_6) = 59$ is the optimal one.

3. Computing the trees. The algorithm uses the following two operations to compute the trees.

To SPROUT a tree is to make its minimum terminal a nonterminal and to add the minimum child of this nonterminal as a terminal.

To LEVEL a tree is to add c children of the maximum nonterminal to the tree as terminals and to remove the c largest terminals in the tree. The c children are the



i	1	2	3
$\mathbf{u}[i]$	3	3	1
$\mathbf{w}[i]$	5	5	4

i	1	2	3
$\mathbf{u}[i]$	4	3	1
$\mathbf{w}[i]$	6	6	3

i	1	2	3
$\mathbf{u}[i]$	4	4	1
$\mathbf{w}[i]$	7	7	2

i	1	2	3
$\mathbf{u}[i]$	4	4	2
$\mathbf{w}[i]$	8	7	2

FIG. 3. *The trees T_5, T_6, T_7, and T_8 for $r = 3$, $c_1 = 2$, $c_2 = 2$, $c_3 = 5$, and $n = 10$. The node numbering is that of the previous figure. Calculating the external path lengths, we find that $c(T_5) = 60$, $c(T_6) = 59$, $c(T_7) = 60$, and $c(T_8) = 62$.*

minimum c children not yet in the tree, where c is maximum such that all children added are less than all terminals deleted.

The algorithm computes the initial tree $T_{m_{\min}}$ and then repeatedly SPROUTs and LEVELs to obtain successive trees until the tree so obtained is not proper. Lemmas 2.7 and 2.8 imply that, as long as node $m+1$ has at least one child in T_{m+1} (it will if T_{m+1} is proper), SPROUTing and LEVELing T_m yields T_{m+1}. Figure 4 illustrates this operation.

OBSERVATION 3.1. *Let $m = m_{\max}$. If node $m+1$ has at least one child in T_{m+1} then* SPROUT*ing and* LEVEL*ing T_m yields tree T_{m+1}. If node $m+1$ has no children in T_{m+1}, then the maximum terminal in T_m is less than the minimum child of node $m+1$ and* SPROUT*ing and* LEVEL*ing T_m yields a tree in which nonterminal $m+1$ has one child. Hence the algorithm always correctly identifies $T_{m_{\max}}$ and terminates*

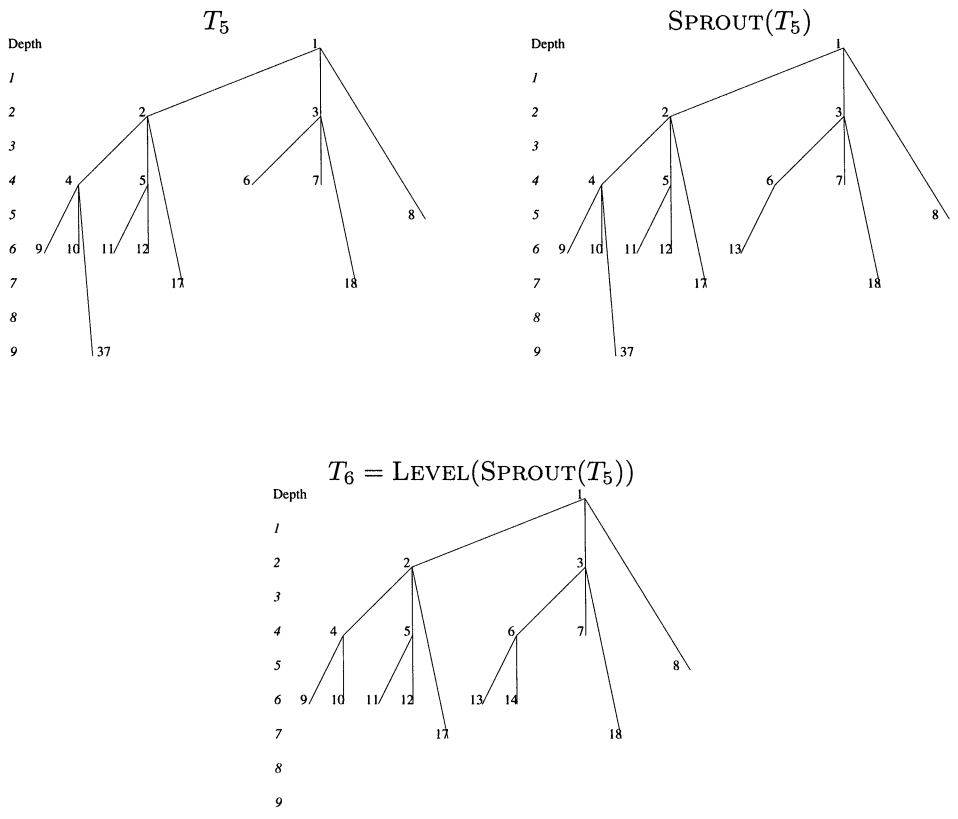


FIG. 4. SPROUT*ing and* LEVEL*ing* T_5 *yields* T_6.

correctly, having considered all relevant trees.

To SPROUT requires identification and conversion of the minimum terminal of the current tree, whereas to LEVEL requires identification and replacement of (no more than r) maximum terminals by children of the new nonterminal. One could identify the maximum and minimum terminals in $O(\log n)$ time by storing all terminals in two standard priority queues (one to detect the minimum, the other to detect the maximum). At most r terminals would be replaced in computing each tree and, because $m_{\max} \leq n - 1$, only $O(n)$ trees would be computed. This approach yields an $O(rn \log n)$-time algorithm.

By a more careful use of the structure of the trees, we improve this in two ways. First, we give an amortized analysis showing that in total, only $O(n \log r)$, rather than $O(rn)$, terminals are replaced. Second, we show how to reduce the number of nonterminals in each priority queue to at most r. This yields an $O(n \log^2 r)$-time algorithm.

Both improvements follow from the tie-breaking condition on the ordering of the nodes, which guarantees that T_m must have the following structure.

LEMMA 3.2. *In any* T_m*, if* u *and* w *are nonterminals with* $u < w$ *and the* i*th child of* w *is in the tree, then so is the* i*th child of* u*. If the* i*th child of* w *is a nonterminal, then so is the* i*th child of* u*.*

Proof. The proof is straightforward from the definition of T_m and the condition on breaking ties in ordering the nodes (in §2.1). □

COROLLARY 3.3. *Node m has a minimum number of children among all nonterminals in T_m.*

3.1. Only $O(n \log r)$ replacements total. The number of terminals replaced while obtaining T_m from T_{m-1} is at most the number of children of nonterminal m in T_m. Although this might be r for many m, the sum of the numbers of children is $O(n \log r)$.

LEMMA 3.4. *Let d_m be the number of children of nonterminal m in tree T_m. Then $\sum_m d_m$ is $O(n \log r)$.*

Proof. By Corollary 3.3, within T_m, node m has the fewest children. The total number of children of the m nonterminals is $m+n-1$. Thus d_m is at most the average $(m+n-1)/m = 1 + (n-1)(1/m)$.

$$\sum_{m=m_{\min}}^{m_{\max}} d_m \le (m_{\max} - m_{\min} + 1) + (n-1) \sum_{m=m_{\min}}^{m_{\max}} 1/m$$
$$= O(m_{\max} - m_{\min} + n \log(m_{\max}/m_{\min})).$$

The result follows from $m_{\min} = \lceil \frac{n-1}{r-1} \rceil$ and $m_{\max} \le n-1$. □

3.2. Limiting the relevant terminals. To reduce the number of terminals that must be considered in finding the minimum and maximum terminals, we partition the terminals into r groups. The ith group consists of the terminals that are ith children ($i = 1, \ldots, r$).

LEMMA 3.5. *In any T_m, for any i, the set of nonterminals whose ith children are terminals is of the form $\{u_i, u_i + 1, \ldots, w_i\}$ for some u_i and w_i. The minimum among terminals that are ith children is $\text{child}_i(u_i)$ (the ith child of u_i). The maximum among these terminals is $\text{child}_i(w_i)$.*

Proof. This is a straightforward consequence of Lemma 3.2. □

Figure 3 presents u_i and w_i for the trees T_5, T_6, T_7, and T_8 when $n = 10$.

This lemma implies that the minimum terminal in T_m is the minimum among $\{\text{child}_i(u_i) : i = 1, \ldots, r\}$. Our algorithm finds the minimum terminal in T by maintaining these r particular children (rather than all n terminals) in a priority queue. This reduces the cost of finding the minimum from $O(\log n)$ to $O(\log r)$. Similarly, the algorithm finds the maximum terminal in $O(\log r)$ time by maintaining $\{\text{child}_i(w_i) : i = 1, \ldots, r\}$ in an additional priority queue.

OBSERVATION 3.6.[1] *As an aside, one can prove using Lemma 3.5 that, for any m such that $m_{\min} < m < m_{\max}$, $c(T_{m+1}) - c(T_m) \ge c(T_m) - c(T_{m-1})$. That is, the sequence of tree costs is unimodal. To prove this, consider building T_{m+1} from T_m.* SPROUT*ing increases the cost by c_1;* LEVEL*ing decreases the cost with each swap. For each swap in building T_{m+1} from T_m, one can show there was a corresponding swap in building T_m from T_{m-1} and that the decrease in cost (from T_m to T_{m+1}) due to the former is bounded by the decrease in cost (from T_{m-1} to T_m) due to the latter. Thus, in practice, the algorithm could be modified to stop and return T_{m-1} when $c(T_m) \ge c(T_{m-1})$.*

[1] This observation is due to R. Fleischer.

3.3. The algorithm in detail. The full algorithm has two distinct phases. The first phase constructs the base tree $T_{m_{\min}}$. The second phase starts with $T_{m_{\min}}$ and, by SPROUTing and LEVELing, iteratively constructs the sequence of shallow trees

$$\langle T_{m_{\min}}, T_{m_{\min}+1}, T_{m_{\min}+2}, \ldots, T_{m_{\max}} \rangle$$

and returns one which has smallest external path length. $T_{m_{\max}}$ is the last proper tree in the sequence, i.e., $T_{m_{\max}+1}$ is improper. Lemma 2.9 guarantees that the algorithm returns an optimal tree. We now describe how to implement the first part of the algorithm in $O(n \log r)$ time and the second in $O(n \log^2 r)$ time; the full algorithm therefore runs in $O(n \log^2 r)$ time.

The skeleton of the final algorithm is shown in Figure 5. Procedure CREATE-$T_{m_{\min}}$ creates tree $T_{m_{\min}}$, the variable **C** contains the external path length of current tree T_m, and **mDeg** contains the number of children of node m in tree T_m. As presented, the algorithm computes only the cost of an optimal tree. It can easily be modified to compute the actual tree. Note that to check that the current tree T_m is proper, by Observation 3.1 and Corollary 3.3, it suffices to check that nonterminal m has at least two children.

```
COMPUTE-TREES(⟨c_1, c_2, ..., c_r⟩, n)
 1. CREATE-T_{m_min}
 2. WHILE (mDeg ≥ 2) DO
                —Compute T_{m+1} from T_m—
 3.             SPROUT(T)
 4.             LEVEL(T)
 5.             C_min ← min{C, C_min}
 6. RETURN C_min
```

FIG. 5. *Algorithm to find an optimal variable-length prefix code.*

The routines SPROUT and LEVEL are shown in Figure 6.

Recall that the nodes of the infinite tree are labeled in order of increasing depth with ties broken arbitrarily except for the requirement that if u and v are both of equal depth and both are ith children of their respective parents, then $u < v$ if and only if parent$(u) <$ parent(v). Depending upon $c_1, c_2, \ldots, c_r$, there may be many such labelings. The algorithm we present breaks ties lexicographically—suppose u and v have the same depth and let $u = \text{child}_i(u')$ and $v = \text{child}_j(v')$; then $u < v$ if and only if $u' < v'$ (or $u' = v'$ and $i < j$). Figure 2 illustrates this labeling for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. The sequence of shallow trees is fully determined by this labelling. Figure 3 illustrates the shallow trees with 10 nonterminals for these r and c values.

The algorithm represents the current tree T_m with the following data structures:

N is the number of terminals.

m is the number of nonterminals. It is also the rank of the maximum nonterminal.

C is the sum of the depths of the terminals.

mDeg is the number of children of nonterminal **m**.

D$[u]$ is the depth of each nonterminal u.

u$[i]$ is the rank of the minimum nonterminal (if any) whose ith child is a terminal $(1 \le i \le r)$.

w$[i]$ is the rank of the maximum nonterminal (if any) whose ith child is a terminal $(1 \le i \le r)$. If no nonterminal has a terminal ith child, then **u**$[i] >$ **w**$[i]$.

SPROUT(T)
—*Make the minimum terminal a nonterminal*—
1. $\mathbf{m} \leftarrow \mathbf{m} + 1$;
2. Let child$_i(\mathbf{u}[i])$ be the minimum terminal in **low-queue**.
3. $\mathbf{D}[\mathbf{m}] \leftarrow \mathbf{D}[\mathbf{u}[i]] + c_i$; $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + 1$; UPDATE-QS($T, i$)
4. $\mathbf{C} \leftarrow \mathbf{C} - \mathbf{D}[\mathbf{m}]$; $\mathbf{mDeg} \leftarrow 0$;
—*Add smallest child as a terminal*—
5. ADD-TERMINAL(T)

LEVEL(T)
1. WHILE ($\mathbf{mDeg} < r$ and child$_{\mathbf{mDeg}+1}(\mathbf{m})$ is less than
the max. terminal child$_i(\mathbf{w}[i])$ in **high-queue**) DO
2. ADD-TERMINAL(T)
—*Delete the maximum terminal*—
3. $\mathbf{C} \leftarrow \mathbf{C} - (\mathbf{D}[\mathbf{w}[i]] + c_i)$
4. $\mathbf{w}[i] \leftarrow \mathbf{w}[i] - 1$; UPDATE-QS($T, i$)

ADD-TERMINAL(T)
1. $\mathbf{mDeg} \leftarrow \mathbf{mDeg} + 1$; $\mathbf{C} \leftarrow \mathbf{C} + \mathbf{D}[\mathbf{m}] + c_{\mathbf{mDeg}}$;
2. $\mathbf{w}[\mathbf{mDeg}] \leftarrow \mathbf{m}$; UPDATE-QS($T, \mathbf{mDeg}$)

FIG. 6. *Operations* SPROUT *and* LEVEL.

low-queue is a priority queue for finding the minimum terminal. It contains $\{\text{child}_i(\mathbf{u}[i]) : \mathbf{u}[i] \leq \mathbf{w}[i]\}$.

high-queue is a priority queue for finding the maximum terminal. It contains $\{\text{child}_i(\mathbf{w}[i]) : \mathbf{u}[i] \leq \mathbf{w}[i]\}$.

For an example, refer back to Figure 3. Tree T_6 has

$$N = 10, \qquad C = 59, \qquad \mathbf{mDeg} = 2,$$

$$D[1] = 0, \qquad D[2] = 2, \qquad D[3] = 3, \qquad D[4] = 4, \qquad D[5] = 4, \qquad D[6] = 4,$$

$$\mathbf{u}[1] = 4, \qquad \mathbf{u}[2] = 3, \qquad \mathbf{u}[3] = 1, \qquad \mathbf{w}[1] = 6, \qquad \mathbf{w}[2] = 6, \qquad \mathbf{w}[3] = 3,$$

$$\mathbf{low\text{-}queue} = \{\text{child}_1(4), \text{child}_2(3), \text{child}_3(1)\},$$

$$\mathbf{high\text{-}queue} = \{\text{child}_1(6), \text{child}_2(6), \text{child}_3(3)\}.$$

The priority queues are maintained as follows. In general, a terminal in T_m can have rank (label) arbitrarily larger than m. The algorithm explicitly maintains the ranks and depths of the m nonterminals in the current tree; the algorithm compares the ranks of terminals in the priority queues via the ranks and depths of their (nonterminal) parents. When $\mathbf{u}[i]$ or $\mathbf{w}[i]$ changes to reflect a new current tree, the queues are updated by the following routine:

UPDATE-QS(T, i)
1. IF ($\mathbf{u}[i] \leq \mathbf{w}[i]$) THEN
2. Update child$_i(\mathbf{u}[i])$ in **low-queue** and child$_i(\mathbf{w}[i])$ in **high-queue**
to maintain the queues' invariants.
3. ELSE Delete both nodes from their respective queues.

Line 2 replaces the old child$_i(\mathbf{u}[i])$ in **low-queue** (child$_i(\mathbf{w}[i])$ in **high-queue**)

by the new one when $\mathbf{u}[i]$ ($\mathbf{w}[i]$) changes. Line 3 will only be executed if $\text{child}_i(\mathbf{u}[i]) > \text{child}_i(\mathbf{w}[i])$, which will only happen if the tree no longer contains any ith child as a terminal. Note that Lemmas 2.8 and 3.2 imply that if, for some i and T_m, no nonterminal has an ith child in T_m, then no nonterminal has an ith child in T_{m+1}.

Construction of the first tree. Tree $T_{m_{\min}}$ has a simple structure. Its nonterminals are the nodes $\langle 1, 2, \ldots, m_{\min}\rangle$. Its terminals are the n shallowest children of nodes $\langle 1, 2, \ldots, m_{\min}\rangle$.

To construct $T_{m_{\min}}$, we assume that $n > r$; otherwise, $T_{m_{\min}}$ is simply the root and its first n children. For $1 \le m < m_{\min}$, define T_m to be the tree with nonterminals $\{1, \ldots, m\}$ and *all* of the $(r-1)m+1$ children of $\{1, \ldots, m\}$ as terminals. The proof of Lemma 2.7 generalizes easily to these trees; node $m+1$ is the minimum terminal of T_m.

CREATE-$T_{m_{\min}}(T)$

—*Create* T_1—

1. $m_{\min} = \lceil \frac{n-1}{r-1} \rceil$; $\mathbf{D}[1] \leftarrow 0$; $\mathbf{C} = \sum_{i=1}^{\min\{r,n\}} c_i$;
2. CREATE **low-queue**; CREATE **high-queue**;
3. FOR $i = 1$ to $\min\{r, n\}$ DO
4. $\quad \mathbf{u}[i] \leftarrow \mathbf{w}[i] \leftarrow 1$; UPDATE-QS$(T, i)$;

—*Create* $\langle T_2, T_3, \ldots, T_{m_{\min}-1}\rangle$—

5. FOR $\mathbf{m} = 2$ to $(m_{\min} - 1)$ DO
6. $\quad$ Let $\text{child}_i(\mathbf{u}[i])$ be the minimum terminal in **low-queue**.
7. $\quad \mathbf{D}[\mathbf{m}] \leftarrow \mathbf{D}[\mathbf{u}[i]] + c_i$; $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + 1$; UPDATE-QS$(T, i)$;
8. $\quad$ FOR $j = 1$ to r DO
9. $\quad\quad \mathbf{w}[j] \leftarrow \mathbf{m}$; UPDATE-QS$(T, j)$;
10. $\quad \mathbf{C} \leftarrow \mathbf{C} - \mathbf{D}[\mathbf{m}] + \sum_{j=1}^{r}(\mathbf{D}[\mathbf{m}] + c_j)$;

—*Create* $T_{m_{\min}}$—

11. $\mathbf{m} = m_{\min}$; $\Delta = n - (r-1)(m_{\min} - 1)$;
12. Let $\text{child}_i(\mathbf{u}[i])$ be the minimum terminal in **low-queue**.
13. $\mathbf{D}[\mathbf{m}] \leftarrow \mathbf{D}[\mathbf{u}[i]] + c_i$; $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + 1$; UPDATE-QS$(T, i)$
14. FOR $j = 1$ to Δ DO
15. $\quad \mathbf{w}[j] \leftarrow \mathbf{m}$; UPDATE-QS$(T, j)$;
16. $\mathbf{C} \leftarrow \mathbf{C} - \mathbf{D}[\mathbf{m}] + \sum_{j=1}^{\Delta}(\mathbf{D}[\mathbf{m}] + c_j)$;
17. $\mathbf{mDeg} = \Delta$;
18. LEVEL(T);

FIG. 7. *Operation* CREATE-$T_{m_{\min}}$.

The tree T_1 is easy to construct. It is the tree with 1 root and r children. Inductively construct the tree T_m from the tree T_{m-1}, $m < m_{\min}$, as follows: find the minimum terminal in T_m by taking the minimum terminal out of **low-queue**. Label this node m, make it a nonterminal, and add all of its children to T_m as terminals. The details are shown in Figure 7.

Finally, construct $T_{m_{\min}}$ from $T_{m_{\min}-1}$ by making the lowest terminal of $T_{m_{\min}-1}$ into node $m_{\min}$. Add the $n - (r-1)(m_{\min} - 1)$ minimum children of node $m_{\min}$ as terminals, bringing the total number of terminals in the current tree to n. Level the resulting tree.

Since only $O(n/r)$ trees are constructed while computing $T_{m_{\min}}$ and each tree

can be constructed from the previous tree in $O(r \log r)$ time, the time required to compute $T_{m_{\min}}$ is $O(n \log r)$. (If desired, the time for each tree T_m with $m < m_{\min}$ can be reduced to $O(\log r)$ because maximum terminals are not replaced in constructing such a tree.)

Construction of the remaining trees. The algorithm constructs the sequence of trees

$$\langle T_{m_{\min}}, T_{m_{\min}+1}, T_{m_{\min}+2}, \ldots, T_{m_{\max}} \rangle,$$

as described previously. Tree T_m is found by SPROUTing and then LEVELing its predecessor T_{m-1}. The cost is $O(d_m \log r)$ time, where d_m is the number of children of the new nonterminal m in T_m. By Lemma 3.4, this part of the algorithm runs in $O\left(\left(\sum_m d_m\right) \log r\right) = O(n \log^2 r)$ time.

Acknowledgments. The authors would like to thank Dr. Jacob Ecco for introducing us to the Morse code puzzle, which sparked this investigation. They would also like to thank Siu Ngan Choi and Rudolf Fleischer (who made Observation 3.6—the unimodality of the tree costs) for their careful reading of an earlier manuscript and subsequent comments.

REFERENCES

[1] D. ALTENKAMP AND K. MELHORN, *Codes: Unequal probabilies, unequal letter costs*, J. Assoc. Comput. Mach., 27 (1980), pp. 412–427.

[2] N. COT, *Complexity of the variable-length encoding problem*, in Proc. 6th Southeast Conference on Combinatorics, Graph Theory, and Computing, Congressus Numerantium XIV, Utilitas Mathematica Publishing, Winnepeg, MB, Canada, 1975, pp. 211–244.

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

[4] S. KAPOOR AND E. M. REINGOLD, *Optimum lopsided binary trees*, J. Assoc. Comput. Mach., 36 (1989), pp. 573–590.

[5] R. KARP, *Minimum-redundancy coding for the discrete noiseless channel*, IRE Trans. Inform. Theory, IT-7 (1961), pp. 27–39.

[6] D. E. KNUTH, *The Art of Computer Programming, Volume* III: *Sorting and Searching*, Addison–Wesley, Reading, MA, 1973.

[7] Y. PERL, M. R. GAREY, AND S. EVEN, *Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters*, J. Assoc. Comput. Mach., 22 (1975), pp. 202–214.

[8] B. VARN, *Optimal variable length codes (arbitrary symbol cost and equal code word probability)*, Inform. Control, 19 (1971), pp. 289–301.