Prefix Codes: Equiprobable Words, Unequal Letter Costs

Mordecai J. Golin * *Neal Young* †

March 25, 1994

Abstract

We consider the following variant of Huffman coding in which the costs of the letters, rather than the probabilities of the words, are non-uniform: Given an alphabet of *unequal-length* letters, find a minimum-average-length prefix-free set of n codewords over the alphabet. 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 previously best known $O(nr \min\{\log n, r\})$ one due to Perl, Garey, and Even [5].

Keywords: Algorithms, Huffman Codes, Prefix Codes, Trees.

1 Introduction

The well-known Huffman coding problem [2] is the following: given a sequence of probabilities $\langle p_1, p_2, ..., p_n \rangle$, construct a binary prefix code $\langle w_1, w_2, ..., w_n \rangle$ minimizing the expected length $\sum_i p_i \text{length}(w_i)$. (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 codewords to be strings over an arbitrary alphabet of $r \geq 2$ letters. Further, the letters are allowed to have arbitrary non-negative lengths $\langle c_1 \leq c_2 \leq \cdots \leq c_r \rangle$. 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. This generalization of Huffman coding to a variable-length alphabet has been considered by many authors, including Altenkamp and Melhorn [1], and Karp [3]. Apparently no polynomial-time algorithm for it is known, nor is it known to be NP-hard.

In this paper we consider the special case of the general problem in which the codewords are sent with equal probability, i.e., each p_i equals $1/n$. This is a variant of Huffman coding in which the lengths of the letters, rather than the codeword probabilities, are non-uniform. This problem is equivalent to one of finding a tree of a particular type that has minimal external path length among all trees of that

*Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong. Partially supported by HK RGC Competitive Research Grant HKUST 181/93E. Email: golin@cs.ust.hk

†UMIACS, University of Maryland, College Park, MD 20742. Partially supported by NSF grants CCR-8906949 and CCR-9111348. Email: young@umiacs.umd.edu.

Automata, Languages and Programming (ICALP), 605-617 (1994) doi:10.1007/3-540-58201-0_102

type with n leaves. These two equivalent problems were previously considered by Perl, Garey, and Even [5], who gave an $O(rn \min\{r, \log n\})$-time algorithm. In what follows we describe a simpler, $O(n \log^2 r)$-time algorithm based on new insights into the structure of optimal codes.

In section 2 we define *shallow* trees and their properties and prove that there is a small set of shallow trees that, among themselves, must contain a tree with minimal external path length. In Section 3 we use the properties of shallow trees to develop an algorithm that constructs all of them quickly. The shallow tree with minimal cost will be the one that describes an optimal encoding.

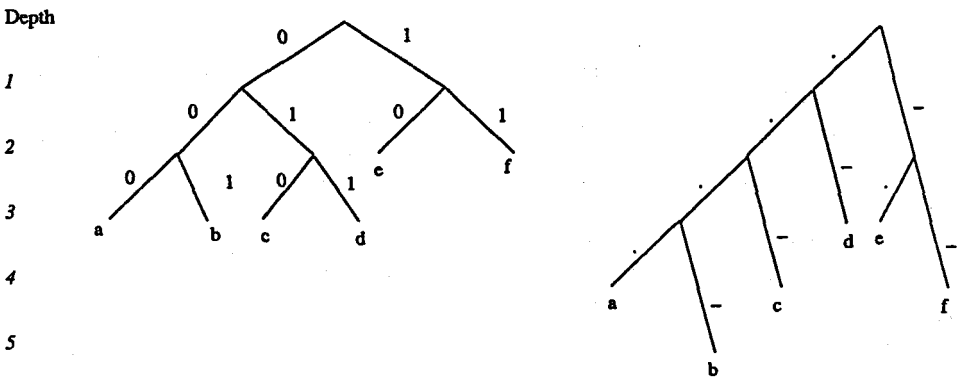


Figure 1: Two Huffman trees for the 6 symbols a,b,c,d,e,f, which all occur with probability 1/6. The tree on the left is the optimal tree that uses the alphabet $\{0,1\}$, $length(0) = length(1) = 1$ while the tree on the right is for the alphabet $\{.,-\}$ with $length(.) = 1$ and $length(-) = 2$. The corresponding sets of codewords are

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

and

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

2 Shallow Trees

Fix an instance of the problem, given by the lengths $\langle c_1 \leq c_2 \leq \cdots \leq c_r \rangle$ of the letters and the number n of (equiprobable and prefix-free) codewords required.

We assume the standard tree representation of prefix codes. The finite words over the alphabet of r letters correspond to the nodes of the infinite, rooted, ordered r-ary tree. If an edge in the tree goes from a node to its ith child, the edge has length c_i and is labeled with the ith letter in the alphabet. The labels along the path from the root to a 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.

In the remainder of the text, the term "tree" refers to any subtree T containing the root. In any such tree, n of the leaves will be identified as *terminals*; their

corresponding words form a prefix code. The term "node" refers to any node of the infinite tree, while the term "non-terminal" refers to any node in the subtree T that is not a terminal. The notation $\text{child}_i(u)$ denotes the ith child of node u.

The *cost* $c(T)$ of such a tree is the sum of the depths of the terminals. This is also called the *external weighted path length* of the tree. The goal is to find an optimal (minimum-cost) tree. A *proper* tree is a tree in which every non-terminal has degree at least two. It is easy to see that some optimal tree is proper so we may restrict ourselves to finding an optimal proper tree.

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

Intuitively, this suggests that the optimal, proper trees have the following form for some m. The non-terminals are some m shallowest (i.e., least-depth) nodes of the infinite tree, while the terminals are some n shallowest children of these nodes in the infinite tree. (In general, when we refer to the children of a *set* of nodes we exclude the nodes in the set itself.) Note that the "m shallowest" nodes are not necessarily unique. Our algorithm constructs a sequence of such trees, one for each possible number of non-terminals, and returns the best one. Note too that, in the definition of a shallow tree, a node may be non-terminal but still have no children. It is for this reason that we talk of *terminal* and *non-terminal* nodes in place of the more common *internal* nodes and *leaves*.

Formally, a tree T is *shallow* provided that (i) for any non-terminal u of T and any node w that is not a non-terminal of T, $\text{depth}(u) \leq \text{depth}(w)$ and (ii) for any terminal u of T and any node w of the infinite tree that is not in T but is a child of a non-terminal of T, $\text{depth}(u) \leq \text{depth}(w)$.

Shallow trees have the nice property that they are optimal among all trees that share the same number of non-terminals.

Lemma 1 *Any shallow tree T satisfies $c(T) \leq c(T')$ for every proper tree T' with the same number of non-terminals.*

Proof. Fix a shallow tree T. If there are no proper trees T' with the same number of non-terminals, the lemma is trivially true. Otherwise, among such trees, consider those that minimize $c(T')$. Among these let T^* be one that maximizes the number of shared non-terminals with T (where T^* and T are considered as finite subsets of the infinite tree).

Suppose for contradiction that the set of non-terminals of T differs from that of T^*. Among all non-terminals of T that are not non-terminals of T^* let u be one whose parent is a non-terminal of T^*. Let w be any non-terminal of T^* that is not a non-terminal of T. Since T is shallow, $\text{depth}(u) \leq \text{depth}(w)$.

In T^*, node u, if present, is a terminal. Node w, on the other hand, has at least two terminal descendants, because T^* is proper. In T^*, consider swapping u and w's subtrees. (More specifically, make u a non-terminal. If u was a terminal in T^*, make w a terminal, otherwise delete w. For each previous descendant x of w, delete x and add the corresponding descendant y of u (as a terminal if x was a terminal).) The swap doesn't increase $c(T^*)$ yet increases the number of non-terminals shared with T. By the choice of T^*, this is a contradiction.

Thus, T and T^* have the same set of non-terminals. Since T is shallow, clearly $c(T) \leq c(T^*)$. □

As an aside, a similar argument proves something like the converse: if a proper tree is optimal among all trees with the same number of non-terminals, then it is shallow.

Lemma 1 implies that it suffices to consider shallow, proper trees:

Lemma 2 *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 *non-terminals. Then one of the* T_m *is proper and optimal.*

Proof. Let m be the minimum number of non-terminals of any optimal tree. Since the optimal tree has degree bounded by r, $m \geq m_{\min}$. By Lemma 1, T_m is optimal. Further, T_m must be proper; otherwise, it would be easy to construct an optimal tree with fewer non-terminals. □

It is this lemma which is at the core of our algorithm for finding an optimal tree; the algorithm generates such a sequence of shallow trees and returns the one which has minimal 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 identification of a minimal cost shallow tree in $O(n \log^2 r)$ time.

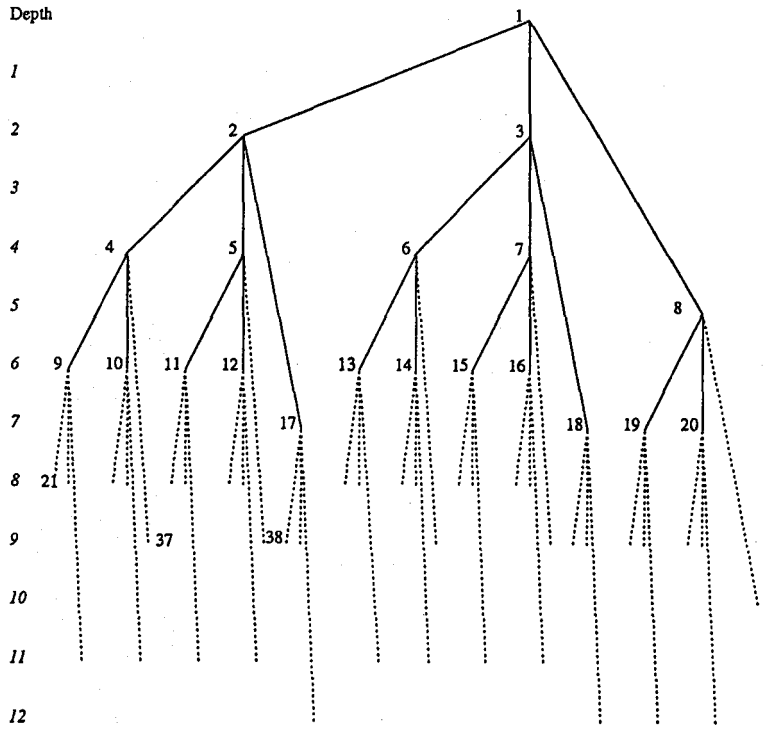


Figure 2: The top of a labelled infinite tree with $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$.

2.1 Defining the Trees

To determine a unique sequence of trees, order 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 and both are ith children for some i, then $u < w$ iff parent(u) < parent(w). For the sake of notation, identify each node with its rank in this ordering, 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 labelling for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. These values of r and c_j will be the ones we assume in all later examples as well.

For each $m \geq m_{\min}$, let T_m denote the "shallowest" tree with m non-terminals with respect to the ordering of the nodes. That is, the non-terminal nodes of T_m are the nodes $\{1, ..., m\}$; the terminals are the minimum n nodes among the children of $\{1, ..., m\}$ in the infinite tree. 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 labelling of Figure 2.

By Lemma 2, to find an optimal tree it suffices to consider the set of trees $\{T_m : T_m \text{ is proper}\}$.

2.2 Relation of Successive Trees

Next we turn our attention to the relation of T_{m+1} to T_m.

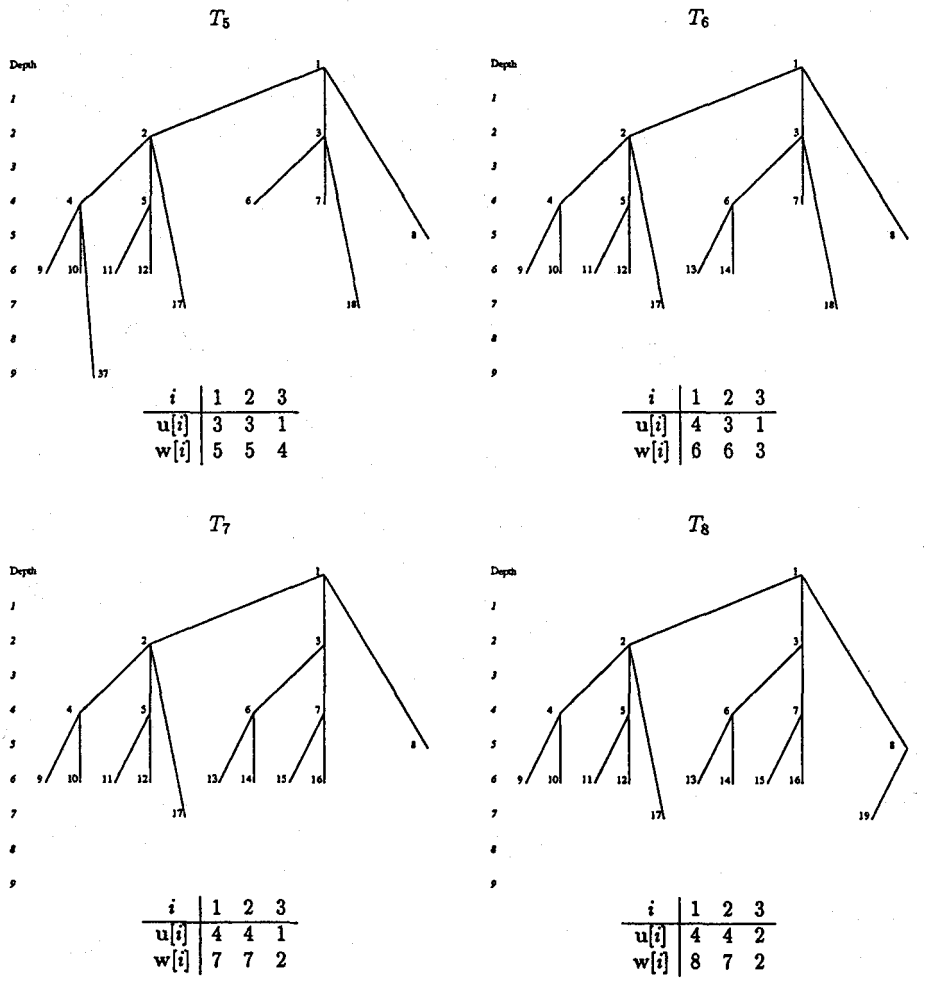
Lemma 3 *For $m \geq m_{\min}$, the new non-terminal (node $m+1$) in T_{m+1} is the minimum terminal of T_m.*

Proof. The parent of $m+1$ is in $\{1, ..., m\}$, so $m+1$ is one of the children of $\{1, ..., m\}$ in the infinite tree. Among these children, $m+1$ is necessarily the minimum. The result follows from the definition of T_m. □

Lemma 4 *For $m \geq m_{\min}$, provided the new non-terminal (node $m+1$) has degree at least one in T_{m+1}, 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 degree d in T_{m+1}. Let the set of children of nodes $\{1, ..., m\}$ in the infinite tree be C. 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 $C - \{m+1\}$. These $n-d$ nodes, together with node $m+1$ (the minimum node in C), are the the $n-d+1$ minimum nodes in 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 3 and 4 is that they will allow an efficient construction of T_{m+1}. Moreover, they also imply that if T_m is not proper, neither is any subsequent tree.



i	1	2	3
u[i]	3	3	1
w[i]	5	5	4

i	1	2	3
u[i]	4	3	1
w[i]	6	6	3

i	1	2	3
u[i]	4	4	1
w[i]	7	7	2

i	1	2	3
u[i]	4	4	2
w[i]	8	7	2

Figure 3: The trees T_5, T_6, T_7, and T_8 for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. 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 $c(T_8) = 62$.

Lemma 5 *One of the trees* $\langle T_{m_{\min}}, T_{m_{\min}+1}, \ldots, T_{m_{\max}} \rangle$ *is optimal and proper, where* $m_{\max} = \min\{m : T_{m+1}$ *is improper*$\} - 1$.

Proof. By lemma 4, if T_m is improper, then so is T_{m+1} — either node $m+1$ has degree zero in T_{m+1} or the non-terminal in T_m that had degree less than two also has degree less than two in T_{m+1}. Hence, for $m \geq m_{\max}$, tree T_m is improper. Thus lemma 2 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 referring back to Figure 3 shows that T_8 is improper. The lemma then implies that one of T_5, T_6, or T_7 must have minimal external path length. Straight calculation shows that T_6 with $c(T_6) = 59$ is the optimal one.

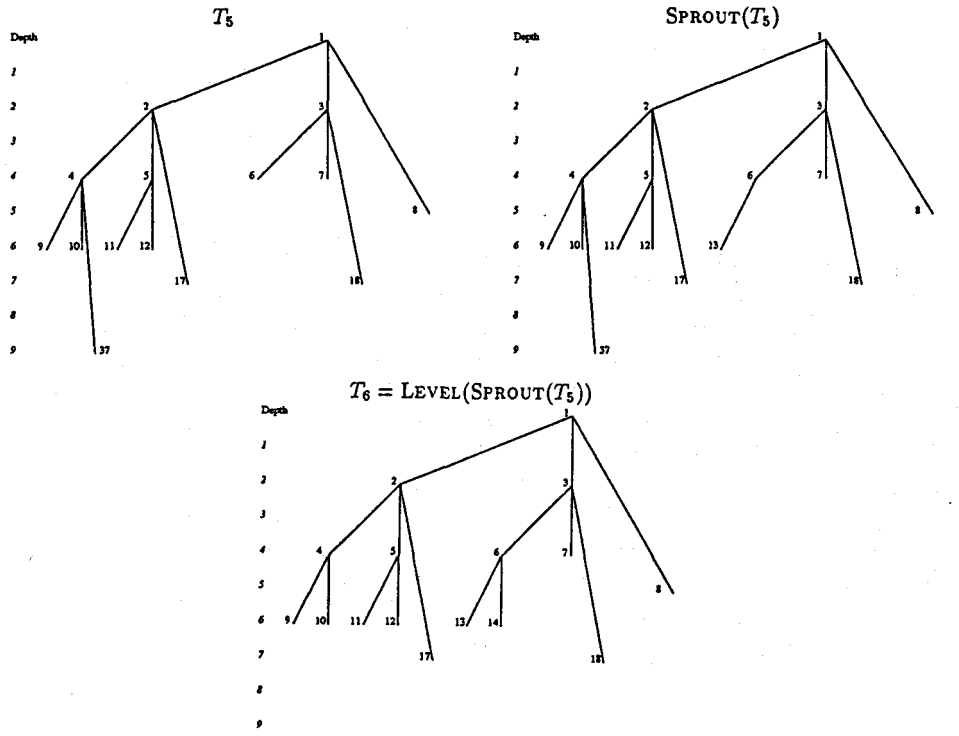


Figure 4: SPROUTing and LEVELing T_5 yields T_6.

As an aside, note that a proper tree can have at most $n-1$ non-terminals corresponding to every non-terminal having exactly two children. This implies that $m_{\max} \leq n-1$, a fact which will later be needed in the proof of Lemma 9.

3 Computing the Trees

Two basic operations are used to compute the trees.

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

To LEVEL a tree is to add c children of the maximum non-terminal to the tree as terminals and to remove the c largest terminals in the tree. The c children are the 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 $T_{m_{\min}}$ then repeatedly SPROUTs and LEVELs to obtain successive trees until the tree so obtained is not proper. Lemmas 3 and 4 imply that, as long as node $m+1$ has degree at least one 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 6 *Let $m = m_{\max}$. If node $m+1$ has degree one in T_{m+1} then* SPROUT*ing and* LEVEL*ing T_m yields tree T_{m+1}. If node $m+1$ has degree zero 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 non-terminal $m+1$ has degree one. Hence, the algorithm always correctly identifies $T_{m_{\max}}$ and terminates correctly, having cosidered 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 non-terminal. 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 are replaced in computing each tree and because $m_{\max} \leq n-1$, only $O(n)$ trees are computed. This approach yields an $O(rn \log n)$-time algorithm.

By a more careful use of the structure of the trees, we improve upon this analysis 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 non-terminals in each priority queue to at most r. This yields an $O(n \log^2 r)$-time algorithm.

Both reductions will be seen to follow from the observation that T_m must have the following simple structure.

Lemma 7 *In any T_m, if u and w are non-terminals with $u < w$, and the ith child of w is in the tree, then so is the ith child of u. If the ith child of w is a non-terminal, then so is the ith child of u.*

Proof. Straightforward from the definition of T_m and the condition on breaking ties in ordering the nodes. □

Corollary 8 *Node m has minimal degree among all non-terminals 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 degree of non-terminal m in T_m. Although this degree might be r for many m, the sum of these degrees is $O(n \log r)$:

Lemma 9 *Let d_m be the degree of non-terminal m in tree T_m. Then $\sum_m d_m$ is $O(n \log r)$.*

Proof. By Lemma 7, within T_m, node m is the lowest-degree non-terminal. The sum of the m non-terminals' degrees is $(m+n-1)$. Thus, d_m is at most the average $(m+n-1)/m = 1 + (n-1)(1/m)$.

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

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

3.2 Limiting the Relevant Terminals

To reduce the number of terminals that must be considered in finding the minimum and maximum terminals, consider, for each i, the terminals which are ith children.

Lemma 10 *In any T_m, for any i, the set of non-terminals whose ith children are terminals is of the form $\{u_i, u_i+1, ..., w_i\}$ for some u_i and w_i. The minimum among terminals that are ith children is $child_i(u_i)$ (the ith child of u_i). The maximum among these terminals is $child_i(w_i)$.*

Proof. A straightforward consequence of Lemma 7. □

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 $\{\text{child}_i(u_i),\ i = 1, \ldots, r\}$. The minimum terminal in T can be found by storing only these r particular children in a priority queue in place of storing 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 maximum terminal can be found in $O(\log r)$ time by storing $\{\text{child}_i(w_i),\ i = 1, \ldots, r\}$, in a priority queue.

3.3 The Algorithm in Detail

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

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

and returns one which has smallest external path length. $T_{m_{\max}}$ is the last proper tree in the sequence. Lemma 5 guarantees that this sequence contains an optimal tree so the tree that the algorithm returns is 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 will therefore run 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 degree of node m in tree T_m. As presented, the algorithm computes only the cost of an optimal tree. It is easily modified to compute the actual tree. Note that to check that the current tree T_m is proper, by Observation 6 and Corollary 8, it suffices to check that non-terminal m has degree at least two.

```
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
```

Figure 5: Algorithm to Find An Optimal Variable-Length Prefix Code

Recall that the nodes of the infinite tree are labelled 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 $u < v$ iff parent(u) < parent(v). Depending upon $c_1, c_2, \ldots, c_r$, there may be many such labellings. The algorithm to be presented breaks ties using the specific rule that if u and v have the same depth, $u = \text{child}_i(u')$, $v = \text{child}_j(v')$, and $u' < v'$ then $u < v$. If $u' = v'$ then $u < v$ iff $i < j$ (this can only occur if $c_i = c_j$). Figure 2 illustrates this labelling for $r = 3$, $c_1 = 2$, $c_2 = 2$, and $c_3 = 5$. Fixing the labelling also fixes the shallow trees. Figure 3 illustrates the shallow trees with 10 non-terminals for these r and c values.

A tree T_m can be fully represented by the following data structures:

N — The number of terminals.

m — The number of non-terminals. Also the rank of the maximum non-terminal.

C — The sum of the depths of the terminals.

mDeg — The degree of non-terminal **m**.

D[u] — The depth of each non-terminal u.

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

w[i] — The rank of the maximum non-terminal (if any) whose ith child is a terminal ($1 \le i \le r$).

low-queue — A priority queue for finding the minimum terminal.
Contains $\{\text{child}_i(\mathbf{u}[i]) : \text{appropriate } i\}$.

high-queue — A priority queue for finding the maximum terminal.
Contains $\{\text{child}_i(\mathbf{w}[i]) : \text{appropriate } i\}$.

For an example refer back to figure 3. Tree T_6 has

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

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

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

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

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

Generally, the algorithm knows the ranks of the non-terminals in the current tree, but not the terminals. The relevant terminals are referenced via their parents, generally one of the non-terminals **m**, **u**[i], or **w**[i]. The order of any two terminals, $\text{child}_i(u)$ and $\text{child}_j(w)$, is determined by considering their respective depths $\mathbf{D}[u] + c_i$ and $\mathbf{D}[w] + c_j$. If one terminal has lesser depth, that terminal has the smaller label. Otherwise, ties are broken in the manner described at the beginning of this

subsection. If $u < w$ then $\text{child}_i(u) < \text{child}_j(w)$. Otherwise $u = w$ and $\text{child}_i(u) < \text{child}_j(w)$ for $i < j$. This is how the r terminals are ordered within each of the priority queues. Inserting and deleting an item from each of these queues can be done in $O(\log r)$ time. Since the minimum terminal in **low**-queue is the minimum terminal in T_m this gives a method of finding the minimum terminal in T_m in $O(\log r)$ time. Similarly, using **high**-queue permits finding the largest terminal in T_m in $O(\log r)$ time.

The reason for using this indirect method of comparing terminals in place of explicitly calculating and comparing the terminals' labels is that the terminals might have rank higher than $\Theta(n \log^2 r)$. Explicitly calculating the labels could therefore cost more than all the rest of the algorithm, effectively destroying the $O(n \log^2 r)$ running time of the algorithm.

In the definitions of the priority queues, "appropriate" values of i are those for which some non-terminal has an ith child in the current tree. Note that Lemmas 4 and 7 imply that if, for some i and T_m, no non-terminal has an ith child in T_m, then no non-terminal has an ith child in T_{m+1}. Subsequently, ith children of non-terminals will always be "inappropriate" and need not be considered.

Corresponding to each variable $\mathbf{u}[i]$ (resp. $\mathbf{w}[i]$) is a terminal in **low**-queue (resp. **high**-queue). When such a variable is changed, the priority queues are updated by the following routine:

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

Line 2 replaces the old value of $\text{child}_i(\mathbf{u}[i])$ in **low**-queue ($\text{child}_i(\mathbf{w}[i])$ in **high**-queue) by its new value. 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.

The routines SPROUT and LEVEL are shown in Figure 6.

```
SPROUT(T)
   —Make the minimum terminal a non-terminal —
1    m ← m + 1;
2    Let child_i(u[i]) be the minimum terminal in low-queue.
3    D[m] ← D[u[i]] + c_i; u[i] ← u[i] + 1; UPDATE-Qs(T, i)
4    C ← C − D[m]; mDeg ← 0;
   —Add smallest child as a terminal —
5    ADD-TERMINAL(T)

LEVEL(T)
1  WHILE (mDeg < r and child_{mDeg+1}(m) is less than
           the max. terminal child_i(w[i]) in high-queue) DO
2          ADD-TERMINAL(T)
           —Delete the maximum terminal —
3            C ← C − (D[w[i]] + c_i)
4            w[i] ← w[i] − 1; UPDATE-Qs(T, i)

ADD-TERMINAL(T)
1  mDeg ← mDeg + 1; C ← C + D[m] + c_mDeg;
2  w[mDeg] ← m; UPDATE-Qs(T, mDeg)
```

Figure 6: The Operations SPROUT and LEVEL.

Construction of the First Trees. Tree $T_{m_{\min}}$ has a simple structure. Its non-terminals 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 $1 \leq m < m_{\min}$, define T_m to be the tree with non-terminals $\{1, ..., m\}$ and *all* of the $(r-1)m+1$ children of $\{1, ..., m\}$ as terminals. The proof of Lemma 3 generalizes easily to these trees; node $m+1$ is the minimum terminal of T_m.

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}, $m < m_{\min} - 1$ 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 non-terminal, and add all of its children to T_m as terminals.

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 degree of the new non-terminal m in T_m. By Lemma 9 this part of the algorithm runs in $O((\sum_m d_m) \log r) = O(n \log^2 r)$ time.

Acknowledgements: The authors would like to thank Dr. Jacob Ecco for introducing us to the Morse Code puzzle which sparked this investigation.

References

[1] Doris Altenkamp and Kurt Melhorn. Codes: Unequal probabilies, unequal letter costs. *Journal of the Association for Computing Machinery*, 27(3):412–427, July 1980.

[2] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE 40*, volume 10, pages 1098–1101], September 1952.

[3] Richard Karp. Minimum-redundancy coding for the discrete noiseless channel. *IRE Transactions on Information Theory*, January 1961.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching* Addison-Wesley, Reading, Mass., 1973.

[5] Y. Perl, M. R. Garey, and S. Even. Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters. *Journal of the Association for Computing Machinery*, 22(2):202–214, April 1975.