Karl-Franzens-Universität Graz & Technische Universität Graz

SPEZIALFORSCHUNGSBEREICH F 003

OPTIMIERUNG
und
KONTROLLE

Projektbereich
DISKRETE OPTIMIERUNG

Mordecai J. Golin Günter Rote

A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes with Unequal Letter Costs

Bericht Nr. 84, revidierte Fassung — Jänner 1998 to appear in *IEEE Transactions on Information Theory* **44**, no. 5, (September 1998)

A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes with Unequal Letter Costs

Mordecai J. Golin
Computer Science Dept.
Hong Kong UST
Clear Water Bay
Kowloon, Hong Kong
E-mail: *golin@cs.ust.hk*

Günter Rote
Technische Universität Graz
Institut für Mathematik
Steyrergasse 30, A-8010
Graz, Austria
E-mail: *rote@opt.math.tu-graz.ac.at*

June 26, 1998

Abstract

We consider the problem of constructing prefix-free codes of minimum cost when the encoding alphabet contains letters of unequal length. The complexity of this problem has been unclear for thirty years with the only algorithm known for its solution involving a transformation to integer linear programming. In this paper we introduce a new dynamic programming solution to the problem. It optimally encodes n words in $O(n^{C+2})$ time, if the costs of the letters are integers between 1 and C. While still leaving open the question of whether the general problem is solvable in polynomial time, our algorithm seems to be the first one that runs in polynomial time for fixed letter costs.

Index Terms. Huffman code, optimal code, dynamic programming.

1 Introduction

In this paper we present a new algorithm for constructing optimal-cost prefix-free codes when the letters of the alphabet from which the codewords are constructed have different lengths (costs). This algorithm runs in polynomial time when the costs are fixed positive integers, improving upon the previously best known algorithm which ran in exponential time even for fixed letter costs.

Assume that messages consist of sequences of characters taken from an alphabet of n source symbols and are transmitted over a channel admitting an *encoding alphabet* $\Sigma = \{\alpha_1, \ldots, \alpha_r\}$ containing r characters. The length of letter α_i, symbolizing its cost or transmission time, is $c_i = \mathit{length}(\alpha_i)$. A codeword w is a string of characters in Σ, i.e., $w \in \Sigma^+$. The *length* of $w = \alpha_{i_1}\alpha_{i_1}\ldots\alpha_{i_k}$ is the sum of the lengths of its component letters, $\mathit{length}(w) = \sum_{j=1}^{k} c_{i_j}$. As an example consider the Morse-code alphabet $\{\cdot, -\}$. If $\mathit{length}(\cdot) = c_1 = 1$ and $\mathit{length}(-) = c_2 = 2$ then $\mathit{length}(\cdot - \cdot) = 4$.

Codeword $w = x_1x_2 \dots x_k$ is a *prefix* of codeword $w' = x'_1x'_2 \dots x'_{k'}$ (with $x_j, x'_j \in \Sigma$) if $k \leq k'$ and $x_j = x'_j$ for all $j = 1, \dots, k$. A set of codewords $W = \{w_1, \dots, w_n\}$ is *prefix-free* if no codeword $w \in W$ is a prefix of another codeword $w' \in W$.

A prefix-free *code* assigns a codeword w_i to each of the n source symbols, such that the set $W = \{w_1, \dots, w_n\}$ is prefix-free. For example, the codes $\{0, 10, 110, 1110, 1111\}$ and $\{000, 001, 010, 011, 100, 101, 11\}$ are prefix-free, whereas the code $\{010, 11, 0\}$ is not, because 0 is a prefix of 010. A prefix-free code is uniquely decipherable, which is obviously a very desirable property of any code that is to be used. However, uniquely decipherable codes need not necessarily be prefix-free.

Let $p_1, \dots, p_n$ be the probabilities with which the source symbols occur; these are also the probabilities with which the respective codewords will be used. The number p_i is also called the *weight* or *frequency* of codeword w_i. The cost of the code W is $C(W) = \sum_{i \leq n} \mathit{length}(w_i) \cdot p_i$. This is the expected length of the string needed to transmit one source symbol.

Given integer costs $0 < c_1 \leq c_2 \leq \dots \leq c_r$, and probabilities $p_1 \geq p_2 \geq \dots \geq p_n > 0$, the *Optimal Coding Problem* is to find a code W containing n prefix-free codewords with minimum cost $C(W)$.

As an application, consider the case of *runlength-limited codes*, which is of importance for storage of information on magnetic or optical disks or tapes. The stored information can be viewed as a sequence of bits, but for technical reasons, it is desirable that the length of a contiguous block of zeros between two successive ones (a *run* of zeros) is bounded from above and below [20, 22]. For example, consider the common case of (2, 7)-codes. We can model this in our framework by combining each one-digit with the preceding block of zeros into a single character of our encoding alphabet: $\Sigma = \{\alpha_1, \dots, \alpha_6\} = \{$001, 0001, 00001, 000001, 0000001, 00000001$\}$, with associated lengths 3, 4, 5, 6, 7, 8. Usually, the source symbols are taken as the symbols 0 and 1 with probabilities $p_1 = p_2 = 1/2$. One can now take blocks of b successive input bits and treat them as one source symbol. This gives $n = 2^b$ source symbols with equal probabilities $p_i = 1/2^b$, assuming that different input bits are distributed uniformly and independently.

If $r = 2$ and $c_1 = c_2 = 1$ then the length of codeword w is just the number of characters it contains; if $\alpha_1 = 0$, $\alpha_2 = 1$ then $\mathit{length}(w)$ is the number of bits in w, e.g., $\mathit{length}(01001) = 5$. A minimum-cost binary code of this type is known as a *Huffman-code* and there is a very well known $O(n \log n)$ time greedy algorithm due to Huffman [21] for finding such a code. This greedy algorithm cannot be adapted to solve the general optimal coding problem (in the next section we will see why). In fact, the only known solution for the general problem seems to be the one proposed by Karp when he formulated it in 1961 [14]. He recast the problem as an integer linear program, which he solved by cutting plane methods; this approach does not have a polynomial time bound.

Since then, different authors have studied various aspects of the problem such as finding bounds on the cost of the solution [1, 13, 19] or solving the special case in which all codewords are equally likely to occur ($p_i = 1/n$ for all i) [15, 6, 7, 11, 18, 23, 5], but not much is known about the general problem, not even if it is NP-hard or solvable in polynomial time. The only efficient algorithms known find approximately minimal codes but not the actual minimal ones [9].

In this paper we describe a dynamic programming approach to the general problem. In fact, our algorithm may be viewed as a dynamic programming solution of the integer programming formulation of Karp [14, Theorem 3]. Our approach is to construct a weighted directed acyclic graph with a polynomial number of vertices and arcs and demonstrate that an optimal code corresponds to a shortest path between two specified vertices in the graph. Finding an optimal code therefore reduces to a shortest path calculation.

We first describe an algorithm that is easy to understand and runs in $O\Big((n(r-1))^{C+2}\Big)$ time where $C = c_r$ is the largest letter cost. We then improve the running time to $O(n^{C+2})$. For this second algorithm, some effort is required to prove its correctness.

While our algorithms do not settle the long-standing question of the problem's complexity they appear to be the first solutions that run in polynomial time for fixed letter costs.

The two algorithms are both extremely simple and easy to implement; the bulk of the paper is devoted to deriving them and proving their correctness. The reader interested primarily in code is invited to skip directly to Figures 5 and 6.

The rest of the paper is structured as follows: in the next section we describe the problem in detail, showing how it can be transformed into a problem on trees. We then explain why the standard Huffman encoding algorithm fails in the general case. In section 3 we introduce the procedure of truncating a tree at a given level, which allows us to transform the problem into shortest path construction on graphs. In section 4 we describe the improved algorithm. We conclude in section 5 by describing implementation experiences, some open problems and areas for future research.

Recently, Bradford, Golin, Larmore, and Rytter [2] found a faster algorithm for the binary case ($r = 2$), using techniques for finding minima in monotone matrices. The algorithm needs only $O(n^C)$ time.

A shorter preliminary version of this paper was presented at the 22nd International Colloquium on Automata, Languages, and Programming in Szeged [10].

2 Some Facts Concerning Codes and Trees

Throughout the remainder of the paper it will be assumed that the numbers $c_1 \leq c_2 \leq \cdots \leq c_r$ are fixed positive integers and $1 > p_1 \geq p_2 \geq \cdots \geq p_n > 0$ are fixed positive reals.

Let $W = \{w_1, \ldots, w_n\}$ be a prefix-free set of codewords. We will find it convenient to follow the standard practice and represent W by an ordered, labeled tree T. By an ordered tree we mean one in which the children of a node are specified in a particular order. Each node in T can have up to r children, carrying distinct labels from the set of letters Σ. To build T perform the following for each $w = x_1x_2\ldots x_k \in W$: start from the root and draw the path consisting of an edge labeled x_1 followed by an edge labeled x_2 followed by an edge labeled x_3 an so on until all k characters have been processed. The i-th edge leaving a node corresponds to writing character α_i in the codeword. See Figure 1.

This process will construct a tree corresponding to W; the tree will have n leaves, each of which corresponds to a codeword in W. Note that a codeword w can not correspond to an internal node. This is because if the node corresponding to w appeared on a path leading from the root to a node corresponding to another codeword w' then w would be a prefix of w', contradicting the prefix-free property. Also, note that this correspondence is reversible; every tree with n leaves will correspond to a different prefix-free set of n

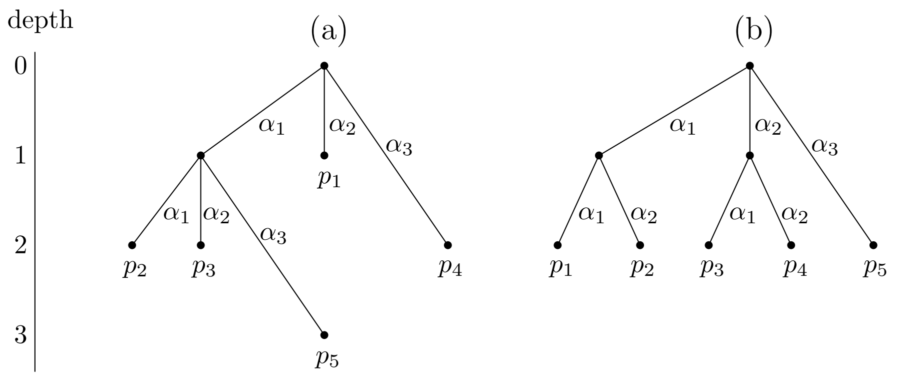

Figure 1: Two codetrees for 5 codewords using the three letter alphabet $\Sigma = \{\alpha_1, \alpha_2, \alpha_3\}$ with respective letter costs $c_1 = c_2 = 1$ and $c_3 = 2$. Figure (a) encodes the set of words $W = \{\alpha_1\alpha_1,\ \alpha_1\alpha_2,\ \alpha_1\alpha_3,\ \alpha_2, \alpha_3\}$, and Figure (b) encodes the set $W = \{\alpha_1\alpha_1,\ \alpha_1\alpha_2,\ \alpha_2\alpha_1,\ \alpha_2\alpha_2, \alpha_3\}$.

codewords.

We draw our trees so that the vertical *length* of an edge labeled α_i is c_i. Such trees are also called *lopsided trees* in the literature. The *depth* of a node $v \in T$, denoted by $depth(v)$, is the sum of the lengths of the edges on the path connecting the root to the node. The root has depth 0. Note that if T represents a code W and a leaf v represents $w \in W$ then our definitions imply that $depth(v) = length(w)$. As usual, the *height* of a tree is the maximum depth of its leaves.

As an example, in Figure 1(a) the codeword $\alpha_1\alpha_3$ is mapped to the leaf v associated with p_5, $depth(v) = 3 = length(\alpha_1\alpha_3)$, and the tree has height 3. When we draw trees in this way, edge labels can be inferred from the edge lengths. (When several characters have the same length, we can assign labels arbitrarily without affecting the quality of the code.) In the sequel, we will therefore omit edge labels in the figures.

Suppose now that T is a tree with n leaves. Label the n leaves as $v = (v_1, \ldots, v_n)$; v_i is the codeword assigned to the i-th input symbol, having probability p_i. Define the cost of the tree under the labeling to be its weighted external path length $cost(T, v) = \sum_{i=1}^{n} depth(v_i) \cdot p_i$.

The following lemma is easy to see:

Lemma 1 *Let T be a fixed tree with n leaves. If v is a labeling of the leaves such that*

$$depth(v_1) \le depth(v_2) \le \cdots \le depth(v_n) \tag{1}$$

then $cost(T, v)$ is minimum over all labelings of the tree.

Proof. We want to find a permutation π which minimizes the inner product of two vectors $(p_i)_{i=1,\ldots,n}$ and $(depth(v_{\pi(i)}))_{i=1,\ldots,n}$, where the entries of the second vector may be arbitrarily permuted. It is well-known that the minimum is achieved by permuting one vector into increasing order and the other one into decreasing order, see [12, p. 261]. □

This lemma implies that in the optimal cost labeling the deepest node is assigned the smallest probability, the next deepest node the second smallest probability and so on, up to the shallowest node which is assigned the highest probability (see Figure 1). Such a code is called a *monotone code*, cf. Karp [14, Section IV].

Since we are interested in minimum cost trees we will restrict our attention to monotone codes. Thus $cost(T)$ can be defined without specifying a labeling of the leaves because the labeling is implied by T.

The Optimal Coding Problem is now seen to be equivalent to the following tree problem: given $(c_1, \ldots, c_r)$ and $p_1 \geq p_2 \geq \cdots \geq p_n$ find a tree T^* with n leaves with minimum cost over all trees with n leaves, i.e.,

$$cost(T^*) = \min\{\, cost(T) \,:\, T \text{ has } n \text{ leaves}\,\}.$$

From here on we will restrict ourselves to discussing the tree version of the problem in place of the original coding formulation.

For example, in Figure 1, we have $(c_1, c_2, c_3) = (1, 1, 2)$. For the probabilities $(p_1, p_2, p_3, p_4, p_5) = (\frac{9}{10}, \frac{1}{40}, \frac{1}{40}, \frac{1}{40}, \frac{1}{40})$, the tree in Figure 1a has minimum cost; for the probabilities $(p_1, p_2, p_3, p_4, p_5) = (\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5})$, the tree in Figure 1b has minimum cost. The corresponding codes are the optimal codes for those probabilities.

Optimal trees have another obvious property:

Lemma 2 *In an optimal tree, every internal node has at least two children.*

Proof. By contracting the edge between a node and its only child we would get a better tree. $\square$

Before continuing, it is instructive to examine why the Huffman encoding algorithm for the case $r = 2$, $c_1 = c_2 = 1$ can not be adapted to work in the general case. Recall that the Huffman encoding algorithm works by constructing the optimal tree from the leaves up. It assumes that it is given a collection of n nodes with associated probabilities $1 > p_1 \geq p_2 \geq \cdots \geq p_n > 0$. It takes the two leaves p_n, p_{n-1} with lowest probability and combines them to form a new node with probability $p = p_n + p_{n-1}$ that it adds to the set while throwing away the two nodes it combined. (For the rest of the algorithm those nodes will always appear as children of p.) It then recurses on the new set of $n - 1$ probabilities, stopping when the set contains only one node. A more complete description of the algorithm can be found in most introductory textbooks on algorithm design, e.g., [21].

Why does this algorithm work? Lemma 1 tells us that p_n and p_{n-1} can always be assigned to two deepest nodes in an optimal tree. In the standard Huffman case of $r = 2$, $c_1 = c_2 = 1$, a node and its sibling have the same depth and, in particular, the deepest node's sibling is also a deepest node. Therefore there is a minimum cost labeling of the optimal-tree in which the leaves assigned weights p_n and p_{n-1} are siblings of each other. Algorithmically, this implies that these two leaves can be combined together as in the Huffman algorithm.

In the general case when $c_1 \neq c_2$, however, the deepest and second deepest node are not necessarily siblings, and thus p_n and p_{n-1} cannot be combined, cf. Figure 2 or Figure 3.

3 A Simple Algorithm for Full Trees

We saw above that building the trees from the bottom up in a straightforward greedy fashion does not work. Instead, we have to consider many possible partial solutions, using a dynamic programming approach. In contrast to the bottom-up approach of the Huffman algorithm we construct the trees from the top down, expanding them level by level. This, however, is not a necessary feature of our algorithm, it is only for ease of exposition.

We construct a graph whose size is polynomial in n and whose arcs encode the structural changes caused by expanding a tree by one level; the cost of an arc will be the cost added to the tree by the corresponding expansion. Trees will then correspond to paths in the graph with the cost of a tree being the cost of the associated path. An optimal-cost tree will correspond to a least cost path between specified vertices in the graph and will be found using a standard single-source shortest path algorithm.

Before proceeding we must address a small technical point. Recall that we had reduced the Optimal-Coding problem to the problem of finding a tree with n leaves and minimal external path length.

A tree is called a *full tree* if all of its internal nodes have the full set of r children. For example, if $r = 2$ any optimal tree must be full, by Lemma 2. Unfortunately, if $r > 2$ this is no longer the case. See Figure 2(a).

For convenience, we will first present an easier version of the algorithm which assumes that the tree we are searching for is full.

Definition 1 *Let T be a tree. By Fill(T) we denote the full tree with the same set of internal nodes as T. In other words, to every internal node which has less than r children, we add the appropriate number of children. These new leaves (the set Fill(T) $-$ T) are called the* missing leaves *of T.*

Since *Fill*(T) may have more than n leaves, we extend the definition of the cost of a tree by padding the sequence

$$(p_1, p_2, \ldots, p_n, 0, 0, 0, \ldots)$$

with sufficiently many zeros. This means that only n leaves are selected with positive probability, and the remaining leaves are ignored in the computation of the cost.

We clearly have $cost(Fill(T)) \leq cost(T)$, because we can obtain an assignment for *Fill*(T) with the same cost as $cost(T)$ by giving probability 0 to the missing leaves. See Figure 2(b). The restriction to full trees is therefore no loss of optimality.

To bound the size of *Fill*(T) we use the fact of Lemma 2 that every internal node of an optimal tree T has at least 2 children. Therefore, T has at most $n - 1$ internal nodes. A full tree with I internal nodes has $1 + Ir - I = 1 + (r - 1)I$ leaves. The full tree that results from augmenting an optimal tree thus has at most $1 + (n - 1)(r - 1) \leq n(r - 1)$ leaves.

We therefore recast the problem of finding the optimal tree for n leaves, and thus the optimal coding problem, as follows: given $(c_1, \ldots, c_r)$ and $p_1 \geq p_2 \geq \cdots \geq p_n$, set $p_i = 0$ for $i > n$ and find the full tree T^* with m leaves $(n \leq m \leq n(r - 1))$, with minimum cost, i.e.,

$$cost(T^*) = \min\{\, cost(T) \,:\, T \text{ has } m \text{ leaves}, \, n \leq m \leq n(r - 1) \,\}.$$

It is this problem that we address now. After finding such a tree and peeling away its 0-probability nodes we will be left with the optimal-coding tree for n leaves.

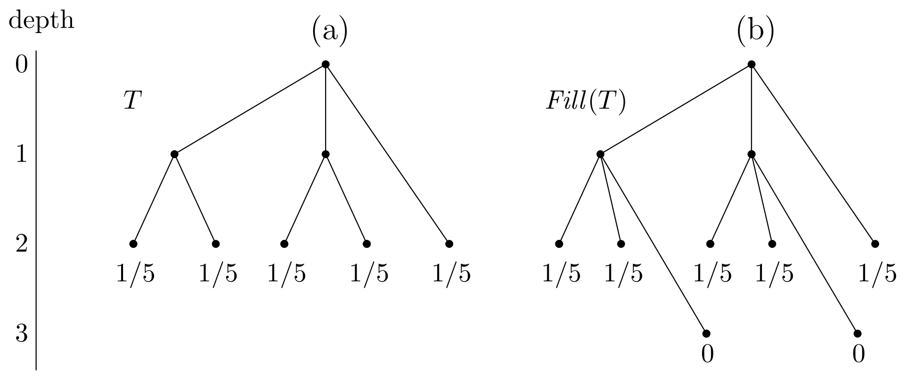

Figure 2: A case in which $r = 3$, $(c_1, c_2, c_3) = (1, 1, 2)$: (a) An optimal tree T which is not full. (b) The augmented tree $\mathit{Fill}(T)$.

We start by examining the structure of trees and how they can change as we expand them level by level. The basic tool we use is the truncated tree:

Definition 2 *Let T be a tree and i a non-negative integer. The* i-th-level truncation *of T is the tree $\mathit{Trunc}_i(T)$ containing the root of T along with all other nodes in T whose parents have depth at most i:*

$$\mathit{Trunc}_i(T) = \mathit{root}(T) \cup \{u \in T \,:\, \mathit{depth}(\mathit{parent}(u)) \leq i\}$$

Figure 3 gives a full tree and Figure 4 shows its truncations to various levels.

We will also need:

Definition 3 *The tree T is an* i-level tree *if all internal nodes $v \in T$ satisfy $\mathit{depth}(v) \leq i$.*

The following are some obvious statements about truncation.

Lemma 3

- *$\mathit{Trunc}_i(T)$ is an i-level tree.*
- *If T is an i-level tree then $\mathit{Trunc}_i(T) = T$.*
- *If T is a full tree then $\mathit{Trunc}_i(T)$ is also a full tree.*
- *$\mathit{Trunc}_i(T)$ has at most as many leaves as T.* $\square$

The idea behind our algorithm will be to build full trees from the top down. Given a tree T with height j we will start from a tree with just the root and successively build the i-level trees $T_i = \mathit{Trunc}_i(T)$, $i = 0, \ldots, j$, where T_0 is the tree containing only the root and its r children and $T_j = T$.

We will find that building T_{i+1} from T_i will not require knowing all of T_i but only (a) the *total* number of leaves with depth at most i in T_i and (b) the number of leaves of T_i on each level $i+1$, $i+2$, …, $i+C$. (There are no leaves beyond depth $i+C$.) To capture this information we introduce the concept of a signature:

Definition 4 *Let T be an i-level tree. The* i-level signature *of T is the $(C+1)$-tuple*

$$\mathit{sig}_i(T) = (m; l_1, l_2, \ldots, l_C)$$

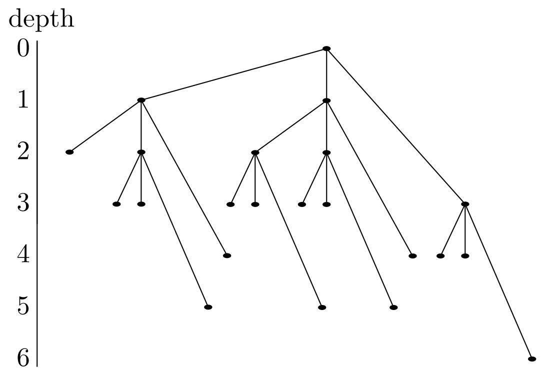

Figure 3: A full tree T with depth 6, having 7 internal nodes and 15 leaves. Each internal node has $r = 3$ children, and $(c_1, c_2, c_3) = (1, 1, 3)$.

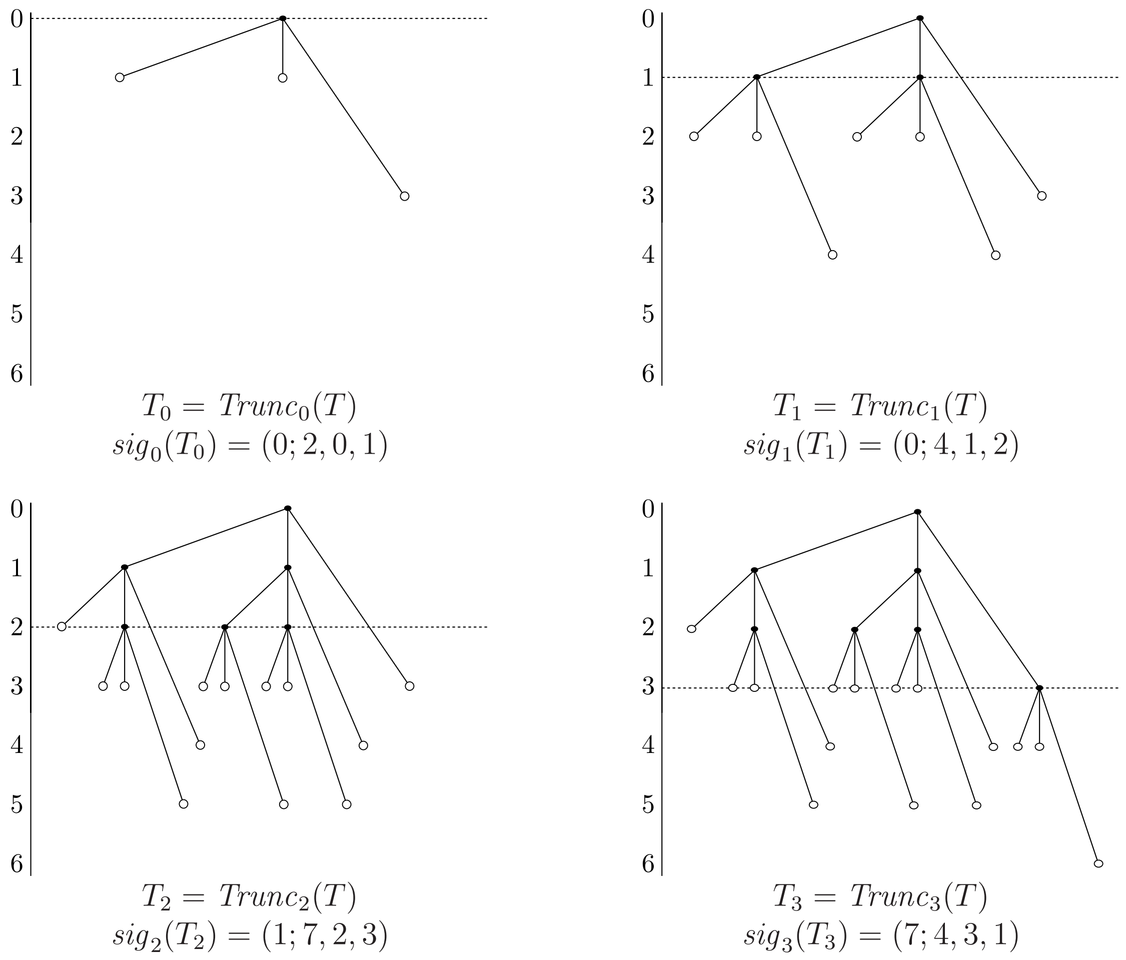
$T_0 = \mathit{Trunc}_0(T)$
$\mathit{sig}_0(T_0) = (0; 2, 0, 1)$
$T_1 = \mathit{Trunc}_1(T)$
$\mathit{sig}_1(T_1) = (0; 4, 1, 2)$
$T_2 = \mathit{Trunc}_2(T)$
$\mathit{sig}_2(T_2) = (1; 7, 2, 3)$
$T_3 = \mathit{Trunc}_3(T)$
$\mathit{sig}_3(T_3) = (7; 4, 3, 1)$

Figure 4: The trees T_0, T_1, T_2, and T_3 for $r = 3$, $(c_1, c_2, c_3) = (1, 1, 3)$. The dotted horizontal line is the *sig* level. Note that $\mathit{Trunc}_3(T) = \mathit{Trunc}_4(T) = \mathit{Trunc}_5(T) = \mathit{Trunc}_6(T) = T$, with $\mathit{sig}_4(T) = (11; 3, 1, 0)$, $\mathit{sig}_5(T) = (14; 1, 0, 0)$, and $\mathit{sig}_6(T) = (15; 0, 0, 0)$.

in which $m = |\{ v \in T : v$ is a leaf, $depth(v) \le i \}|$ is the number of leaves in T with depth at most i and

$$l_k = |\{ u \in T : depth(u) = i + k \}|, \quad k = 1, \ldots, C,$$

is the number of nodes in T of depth $i + k$.

Even though the way in which $sig_i(T_i)$ is computed depends on the truncation level i, the signature itself contains no information identifying the value of i. Also note that the truncation operation cannot increase the total number of leaves in the tree; so, if T is a tree with at most $n(r-1)$ leaves and $sig_i(Trunc_i(T)) = (m; l_1, \ldots, l_C)$, then $m + l_1 + \cdots + l_C \le n(r-1)$.

Suppose T_i is the i-th-level truncation of some code tree T for n symbols with $sig_i(T_i) = (m; l_1, l_2, \ldots, l_C)$. How much information concerning T can be derived from T_i?

First note that the nodes on levels $\le i$ are the same in T and T_i; that is, if $u \in T_i$ is a node with $depth(u) \le i$, then u is a leaf in T_i if and only if the corresponding node is a leaf in T; T therefore has exactly m leaves with depth $\le i$. We can not say anything similar for a node in T_i with depth greater than i; we know it is a leaf in T_i but the corresponding node in T might be an internal node. By Lemma 1, the m largest probabilities in T are assigned to the m highest leaves in T which are the m highest leaves in T_i. All that is known concerning the remaining $n - m$ probabilities is that they will be assigned to nodes in T that have depth greater than i. This leads us to the following definition.

Definition 5 *Let T be an i-level tree with $sig_i(T) = (m; l_1, l_2, \ldots, l_C)$. For $m \le n$, the* i-th level cost *of T is*

$$cost_i(T) = \sum_{t=1}^{m} depth(v_t) \cdot p_t \; + \; i \cdot \sum_{t=m+1}^{n} p_t \tag{2}$$

where $v_1, \ldots, v_m$ are the m highest leaves of T ordered by depth. For $m \ge n$, we define $cost_i(T) = \sum_{t=1}^{n} depth(v_t) \cdot p_t$.

The first term in the sum (2) reflects the cost of the paths to the m leaves which have already been assigned, whereas the second term reflects only part of the cost for reaching the remaining leaves, namely only the part until level i.

Definition 6 *Let $(m; l_1, l_2, \ldots, l_C)$ be a valid signature, i.e., $m, l_1, \ldots, l_C \ge 0$. Set $\mathtt{OPT}[m; l_1, \ldots, l_C]$ to be the minimum cost of a tree with signature $(m; l_1, \ldots, l_C)$. More precisely,*

$$\mathtt{OPT}[m; l_1, \ldots, l_C] = \min \{ \, cost_i(T) : i \ge 0, \; T \textit{ is an } i\textit{-level tree, } \textit{and } sig_i(T) = (m; l_1, \ldots, l_C) \, \}.$$

If T is an i-level tree with $sig_i(T) = (m; l_1, l_2, \ldots, l_C)$ and $m \ge n$ then, because $p_i = 0$ for $i > m$ we find that $cost_i(T) = cost(T)$. Also, T contains at least $m + l_1 + \cdots + l_C$ nodes. Since we may restrict ourselves to trees which have at most $n(r-1)$ nodes the cost of the optimal tree is exactly the minimal cost of $\mathtt{OPT}[m; l_1, l_2, \ldots, l_C]$ where the minimum is taken over all tuples in which $m \ge n$ and $m + l_1 + \cdots + l_C \le n(r-1)$. An optimal tree is one that realizes this cost.

To find this minimum value and its corresponding signature (and the tree with the signature that has that value) we use a dynamic programming approach to fill in the `OPT` table. We will therefore investigate how truncated trees can be expanded level by level: Suppose that T is an i-level tree with $sig_i(T) = (m; l_1, l_2, \ldots, l_C)$ and T' is some $(i+1)$-level tree with $Trunc_i(T') = T$. How can T' differ from T?

By the definition of $Trunc_i$, the two trees must be identical on levels 0 through i in that they contain the same nodes on those levels and a node is a leaf or internal in T if and only if the corresponding node is respectively a leaf or internal in T'. Furthermore the two trees contain exactly the same nodes on level $i+1$ because the parents of those nodes are on level i or higher. The only difference between the trees on level $i+1$ is the status of the l_1 nodes on that level. In T all of these nodes are leaves. In T', some number q of them might be internal with $l_1 - q$ of them being leaves. Since $0 \le q \le l_1$, there are essentially $l_1 + 1$ possible $(i+1)$-level trees T' with $Trunc_i(T') = T$, a different tree corresponding to each possible value of q. Once q is fixed, the number of nodes on levels $i+2$ through $i+1+C$ are also fixed since all such nodes are either nodes in T or children of the q internal nodes at level $i+1$ of T'.

This motivates the following definition:

Definition 7 *Let T be an i-level tree with $sig_i(T) = (m; l_1, l_2, \ldots, l_C)$. Let $0 \le q \le l_1$. The* q-th expansion of T at level $i+1$ *is the tree*

$$T' = Expand_i(T, q)$$

constructed by making q of the leaves at level $i+1$ of T internal nodes with r children.

Note that the definition does not specify *which* q of the l_1 leaves become internal nodes in $Expand_i(T)$. For our purposes, however, this does not matter. Although different choices result in different trees, the number of leaves at each level is fixed, and more importantly, all resulting trees have the same cost. A formal statement of this fact requires a notion of equivalence between trees.

Definition 8 *Two trees T_1 and T_2 are* equivalent *if they have the same number of leaves at every level. We write this as*

$$T_1 \doteq T_2.$$

The following lemma summarizes the obvious properties of this relation.

Lemma 4
- *All trees which may result as the q-th expansion of T at level $i+1$ are equivalent. In other words, $Expand_i(T)$ is unique up to equivalence.*
- *If $T_1 \doteq T_2$, then $Expand_i(T_1, q) \doteq Expand_i(T_2, q)$, provided that the expansions are defined.*
- *If $T_1 \doteq T_2$ are two i-level trees, then $sig_i(T_1) = sig_i(T_2)$.*
- *If $T_1 \doteq T_2$, then $cost_i(T_1) = cost_i(T_2)$, for any i.* □

In order to describe the transformation of signatures effected by expansion, we need the *characteristic vector* $(d_1, d_2, \ldots, d_C)$ associated to an alphabet with length vector $(c_1, c_2, \ldots, c_r)$: for $1 \le j \le C$, d_j is the number of c_i that are equal to j. For example, $(c_1, c_2, c_3, c_4) = (1, 2, 2, 2)$ gives $(d_1, d_2) = (1, 3)$, $(c_1, c_2, c_3) = (1, 1, 3)$ gives $(d_1, d_2, d_3) = (2, 0, 1)$, and $(c_1, c_2, c_3) = (2, 3, 3)$ gives $(d_1, d_2, d_3) = (0, 1, 2)$.

Lemma 5 *Suppose T is an i-level tree with $sig_i(T) = (m; l_1, l_2, \ldots, l_C)$. Let $T' = Expand_i(T, q)$ be its q-th expansion at level $i+1$. Then T' is an $(i+1)$-level tree with*

$$sig_{i+1}(T') = (m + l_1; l_2, \ldots, l_C, 0) + q \cdot (-1; d_1, d_2, \ldots, d_C) \tag{3}$$

where vector addition and multiplication by the scalar q is carried out componentwise (see Example 1 below), and

$$cost_{i+1}(T') = cost_i(T) + \sum_{m<t\leq n} p_t. \tag{4}$$

Proof. Equation (3) follows directly from the discussion preceding the definition: on levels 0 through i, T' has exactly the same leaves as T. On level $i+1$, T' has $l_1 - q$ leaves. The first entry in $sig_{i+1}(T')$ is therefore $m + l_1 - q$. Nodes appearing on level $(i+1)+j$ of T', for $1 \leq j < C$, were either in tree T or are one of the $q \cdot d_j$ children on that level whose parents are on level $(i+1)$. There are exactly $l_{j+1} + qd_j$ of these. Finally, the only nodes appearing on level $i+1+C$ of T' are the qd_C children of the internal nodes on level $i+1$.

The proof of equation (4) follows from a similar analysis. Suppose $m \leq n$; otherwise T' has obviously the same cost as T. The m shallowest leaves in T' are exactly the same as the m shallowest leaves $v_1, \ldots, v_m$ in T, which are the leaves in T at at depth i or less. T' also contains exactly $l_1 - q$ leaves at depth $i+1$ and its remaining leaves are all deeper than $i+1$. Thus, with $m' = \min\{m + l_1 - q, n\}$, we have

$$\begin{aligned} cost_{i+1}(T') &= \left(\sum_{1\leq t\leq m} depth(v_t)p_t + \sum_{m<t\leq m'} (i+1)p_t \right) + (i+1) \sum_{m'<t\leq n} p_t \\ &= \left(\sum_{1\leq i\leq m} depth(v_t)p_t + i \sum_{m<t\leq n} p_t \right) + \sum_{m<t\leq n} p_t \\ &= cost_i(T) + \sum_{m<t\leq n} p_t \end{aligned}$$

□

Example 1. The following table shows the transition from a signature S to S' and S'' by two expansions with $q = 2$ and $q = 3$, respectively. We have $(c_1, c_2, c_3, c_4) = (3, 3, 4, 4)$, and therefore $(d_1, d_2, d_3, d_4) = (0, 0, 2, 2)$.

signature	$(m;\ l_1,\ l_2,\ l_3,\ l_4)$	partial sums
$[\ (-1; d_1, \ldots, d_4) =$	$(-1;\ 0,\ 0,\ 2,\ 2)\ ]$	$(-1, -1, -1,\ 1,\ 3)$
$S =$	$(16; 14,\ 8,\ 2,\ 0)$	$(16,\ 30,\ 38, 40, 40)$
shift level:	$(30;\ 8,\ 2,\ 0,\ 0)$	$(30,\ 38,\ 40, 40, 40)$
$q = 2$: $S' =$	$(28;\ 8,\ 2,\ 4,\ 4)$	$(28,\ 36,\ 38, 42, 46)$
shift level:	$(36;\ 2,\ 4,\ 4,\ 0)$	$(36,\ 38,\ 42, 46, 46)$
$q = 3$: $S'' =$	$(33;\ 2,\ 4, 10,\ 6)$	$(33,\ 35,\ 39, 49, 55)$

The modification of the signature is done in two steps: the first step is the shift, which is due to the change of focus from level sig_i to level sig_{i+1}. The second step is the addition of a suitable multiple of the vector $(-1; d_1, \ldots, d_4)$, which is given in the second line.

The last column gives the sequence of partial sums m, $m + l_1$, $m + l_1 + l_2$, ..., which is important later.

This last lemma tells us that to calculate the extra cost added by a level-i expansion of T and the signature of the new expanded tree it is not necessary to know T or i but only $sig_i(T)$. This motivates us to recast the problem in a graph formulation with vertices corresponding to signatures, arcs to expansions, and the cost of an arc to the cost added by its associated expansion. A path in this graph will correspond to the construction of a tree by successive expansions; so an optimal tree will correspond to a least-cost path of a certain type.

More specifically we define a directed acyclic graph $G = (V, E)$, called the *signature graph* with

$$V = \{ (m; l_1, \ldots, l_C) \; : \; m_1, l_1, \ldots, l_C \geq 0, \; m_1 + l_1 + \cdots + l_C \leq n(r-1) \}$$

and

$$\begin{aligned} E = \{ (S, S') \in V \times V \; : \; & S = (m; l_1, l_2, \ldots, l_C, 0) \text{ and } \exists q, \; 0 \leq q \leq l_1, \text{ such that} \\ & S' = (m + l_1; l_2, \ldots, l_C, 0) + q \cdot (-1; d_1, d_2, \ldots, d_C) \}. \end{aligned}$$

We will often denote an arc $(S, S') \in E$ by $S \xrightarrow{q} S'$, indicating the value of q that defines the arc. We also define a cost function on the arcs of G:

$$c(S, S') = \sum_{m < t \leq n} p_t \quad \text{for } (S, S') \in E, \; S = (m; l_1, l_2, \ldots, l_C).$$

We pause here to point out the motivation behind this function. Suppose T is an i-level tree with $sig_i(T) = S$ and $T' = Expand_i(T, q)$ with $sig_{i+1}(T') = S'$. Then Lemma 5 tells us that

$$cost_{i+1}(T') = cost_i(T) + c(S, S').$$

Note that the change in cost is *independent of* q.

Now let T_0 be the tree containing only the root and its r children. This tree, which we call the *root tree*, is the only 0-level full tree; so $T_0 = Trunc_0(T)$ for every full tree T. Its signature $S_0 = sig_0(T_0) = (0; d_1, d_2, \ldots, d_C)$ is the starting vertex of the graph.

For a directed path

$$P = (S_0 \xrightarrow{q_0} S_1 \xrightarrow{q_1} S_2 \xrightarrow{q_2} \cdots \xrightarrow{q_{j-2}} S_{j-1} \xrightarrow{q_{j-1}} S_j)$$

using j arcs to get from S_0 to S_j, we define the cost of the path in the usual way as the sum of the cost of its arcs, i.e.,

$$c(P) = \sum_{i=1}^{j} c(S_{i-1}, S_i).$$

The following crucial lemma establishes a one-to-one correspondence between paths of length j emanating from S_0 and j-level trees T with $sig_j(T) \in V$.

Lemma 6 (a) *Let T be a j-level tree with $sig_j(T) = (m; l_1, l_2, \ldots, l_C) \in V$. Set $T_i = Trunc_i(T)$, $S_i = sig_i(T_i)$, and let q_i be the number of internal nodes of T at level $i + 1$, for $i = 0, 1, \ldots, j - 1$. Then the path*

$$P(T) = (S_0 \xrightarrow{q_0} S_1 \xrightarrow{q_1} S_2 \xrightarrow{q_2} \cdots \xrightarrow{q_{j-2}} S_{j-1} \xrightarrow{q_{j-1}} S_j)$$

is contained in G, and $c(P(T)) = cost_j(T)$.

(b) *Let*
$$P = (S_0 \xrightarrow{q_0} S_1 \xrightarrow{q_1} S_2 \xrightarrow{q_2} \cdots \xrightarrow{q_{j-2}} S_{j-1} \xrightarrow{q_{j-1}} S_j)$$
be a path in G. Let T_0 be the root tree. Recursively define
$$T_{i+1} = \mathit{Expand}_i(T_i, q_i), \quad i = 0, 1, \ldots, j-1. \tag{5}$$
Then the tree
$$T(P) = T_j \tag{6}$$
is a j-level tree with $\mathit{cost}_j(T(P)) = c(P)$.

Proof. (a) First note that if $\mathit{sig}_j(T) \in V$ then $m + l_1 + \cdots + l_C \leq n(r-1)$. Fix $i < j$. Since the *Trunc*$_i$ operation cannot increase the number of leaves in the tree, $S_i \in V$ as well. The arcs $S_i \xrightarrow{q_i} S_{i+1}$ exist by the definition of the *Trunc* and *Expand* operations. Thus $P(T) \subseteq G$. Straightforward calculation and the fact that $\mathit{cost}_0(T_0) = 0$ yields
$$\begin{aligned} c(P(T)) &= \sum_{0 \leq i < j} c(S_i, S_{i+1}) \\ &= \mathit{cost}_0(T_0) + \sum_{0 \leq i < j} (\mathit{cost}_{i+1}(T_{i+1}) - \mathit{cost}_i(T_i)) = \mathit{cost}_j(T) \end{aligned}$$

(b) The fact that the T_i exist and are i-level trees with $\mathit{sig}_i(T_i) = S_i$ follows from the definition of the *Expand* operation. Equality of costs is obtained by following the above calculation backwards. □

We have just seen (a) that every path P of length j in G from S_0 to S corresponds to a j-level tree $T(P)$ with $\mathit{sig}_j(T(P)) = S$ and $\mathit{cost}_j(T(P)) = c(P)$ and (b) that every j-level tree T corresponds to a j-arc path $P(T)$ from S_0 to $\mathit{sig}_j(T)$ with $\mathit{cost}_j(T) = c(P(T))$. This proves

Lemma 7 *Let $S \in V$ and P a minimum cost path from S_0 to S in G. Then $\mathtt{OPT}[S] = c(P)$. If P contains j arcs and $T(P)$ is as defined in* (5) *and* (6) *then $\mathit{cost}_j(T(P)) = \mathtt{OPT}[S]$.*

The calculation of shortest paths is facilitated by the fact that the graph G is *acyclic* (apart from possible loops at the vertices $(m; 0, \ldots, 0)$). We show this by specifying a specific linear ordering of the vertices which is consistent with the orientation of the arcs (a topological ordering).

Definition 9 *Let $S = (m; l_1, l_2, \ldots, l_C)$, $S' = (m'; l'_1, l'_2, \ldots, l'_C) \in V$. We define the linear order $\prec$ on the signatures so that $S \prec S'$ if and only if the vector*
$$(m + l_1 + \cdots + l_C,\ m + l_1 + \cdots + l_{C-1},\ \ldots,\ m + l_1 + l_2,\ m + l_1,\ m)$$
is lexicographically smaller than
$$(m' + l'_1 + \cdots + l'_C,\ m' + l'_1 + \cdots + l'_{C-1},\ \ldots,\ m' + l'_1 + l'_2,\ m' + l'_1,\ m').$$

```
1. Initialization.
   Set OPT$[m; l_1, \dots, l_C] := \infty$ for all entries with $m + l_1 + \cdots + l_C \le n(r-1)$.
   Set OPT$[0; d_1, d_2, \dots, d_C] := 0$;

2. Loop. Process the array entries OPT$[m; l_1, \dots, l_C]$ with $(l_1, \dots, l_C) \ne (0, \dots, 0)$ in
   lexicographically increasing order of
   $(m + l_1 + \cdots + l_C,\ m + l_1 + \cdots + l_{C-1},\ \dots,\ m + l_1,\ m)$.
   For each entry OPT$[m; l_1, \dots, l_C]$ do the following:
2a.      $new\_cost :=$ OPT$[m; l_1, \dots, l_C] + \sum_{m<t\le n} p_t$;
         for $q := 0$ to $l_1$ do
2b.          Let $(m'; l'_1, \dots, l'_C) := (m + l_1; l_2, \dots, l_C, 0) + q \cdot (-1; d_1, \dots, d_C)$;
2c.          If $m' + l'_1 + \cdots + l'_C \le n(r-1)$ then
                 OPT$[m'; l'_1, \dots, l'_C] := \min\{$OPT$[m'; l'_1, \dots, l'_C],\ new\_cost\}$;
         end for;

3. Termination. $\min\{$ OPT$[m; l_1, \dots, l_C] : m \ge n \}$ is the cost of the optimal code.
```

Figure 5: The simple algorithm

For example, to compare $(4; 1, 2, 4)$ and $(3; 1, 4, 3)$, we form their partial sums sequences $(4,\ 4+1,\ 4+1+2,\ 4+1+2+4) = (4, 6, 7, 11)$ and $(3, 4, 8, 11)$, respectively. We compare these vectors lexicographically, the right-most entry being most significant. Since $11 = 11$ and $7 < 8$, we have $(4; 1, 2, 4) \prec (3; 1, 4, 3)$. In Example 1, the partial sums sequence is indicated in the second column. It is quite easy to see that the signatures become bigger and bigger (in the '$\prec$' order) by expansion. The shift of levels causes a left shift of the sequence, with the right-most entry being duplicated, and then a multiple of the vector $(-1, -1, -1, 1, 3)$, which is lexicographically positive, is added.

Lemma 8 *If $S, S' \in V$, $S \xrightarrow{q} S'$, and $S \ne S'$ then $S \prec S'$.*

Proof. Suppose $S = (m; l_1, l_2, \dots, l_C)$ and $S' = (m'; l'_1, l'_2, \dots, l'_C)$. If $q > 0$, then

$$(m' + l'_1 + \cdots + l'_C) - (m + l_1 + \cdots + l_C) = (-1 + d_1 + \cdots + d_C)q = (r-1)q > 0;$$

so $S \prec S'$.

If $q = 0$ then $m' = m + l_1$ and $l'_j = l_{j+1}$ for $j < C$ and $l'_C = 0$. Thus, $S' \prec S$ unless $S' = S$ in which case $l_1 = l_2 = \cdots = l_C = 0$ and $S = S'$. □

For a directed acyclic graph $G = (V, E)$, a shortest path can be computed in $O(|V| + |E|)$ steps by scanning the vertices in topological order [16, p. 45]. We rewrite the algorithm for our special case and present it in Figure 5. Note that in the algorithm we never explicitly construct the signature graph G but only implicitly use the graph structure to fill in the OPT table properly. In essence, we are performing dynamic programming to fill in the OPT table using an appropriate ordering of the table entries. The algorithm takes $O(|V|) = O\Big((n(r-1))^{C+1}\Big)$ time for Steps 1 and 2a. Each vertex has $l_1 + 1 = O(n(r-1))$

outgoing arcs, and therefore $|E| = O\Big((n(r-1))^{C+2}\Big)$. Each execution of Steps 2b and 2c takes $O(C)$ time. Thus, the total cost of the algorithm is $O\Big(C(n(r-1))^{C+2}\Big)$ time.

Note that the algorithm as presented only calculates the cost of the optimal tree. To actually construct the optimal tree, we have to augment the algorithm by storing a pointer with every array entry, and whenever $\texttt{OPT}[m'; l'_1, \ldots, l'_C]$ is improved we update the pointer to remember where the current optimal value came from. (In fact it is sufficient to store the value q which lead to the current value.) At the end we backtrack from the minimum cost vertex to recover the optimal solution. This is standard dynamic programming practice, and we omit the details.

4 Pruning of Extra Leaves

The algorithm in the previous section restricted its attention to full trees, i.e., trees in which every internal node contains all r of its children. We had to pay for this convenience by constructing trees with as many as $n(r-1)$ leaves, many more than the n leaves actually used in the trees. In this section we improve the algorithm by looking only at trees with at most n leaves, thereby reducing the complexity by a factor of $O((r-1)^{C+2})$. Note that in the binary case of $r = 2$ this makes no difference at all; the results of this section are only of interest when $r > 2$.

We have to relax the requirement of only constructing *full trees*, because optimal trees are not necessarily full, see e.g. Figure 2. This relaxation permits us to transform the construction of an optimal tree into a least-cost path search in a new signature graph $G' = (V', E')$ where

$$V' = \{(m; l_1, \ldots, l_C) \,:\, m, l_1, \ldots, l_C \geq 0,\, m + l_1 + \cdots + l_C \leq n\}.$$

which has size $|V'| = O(n^{C+1})$ and $|E'| = O(n^{C+2})$.

The design of the graph G' and the corresponding algorithm will be complicated by the following technical point: in the previous section we constructed full trees level by level by specifying the number of internal nodes at each level. Since the trees being constructed were full, this specification uniquely determined the tree. If the trees are no longer required to be full then specifying the number of internal nodes on each level no longer uniquely specifies the tree; for each internal node it must also be known which children it has.

As in the previous section, we want to construct the optimal tree level by level. However, if we ever generate more than n leaves, we want to throw away some of them. It appears obvious that it is best to throw away the deepest leaves. Below we will prove that this is true.

Definition 10 *The* reduction *of a tree T (to n leaves) is the tree Reduce(T) obtained by removing all but the n shallowest leaves from T. It may happen that some internal nodes become leaves by this process because they lose all their children. In this case, we remove these additional "unwanted" leaves, and if necessary, we iterate this cleanup process.*

If T does not have more than n leaves, then Reduce$(T) = T$.

In other words, we can think of marking a set of n shallowest leaves in T. The tree *Reduce*(T) is then the unique subtree which has precisely this set of leaves. Similarly as in Definition 7, the set of leaves to be removed is not uniquely specified, but the number of leaves at each level is uniquely determined. In other words, *Reduce*(T) is unique up to equivalence. It is obvious that reduction does not change the cost of a tree:

Lemma 9 *If T is an i-level tree then $cost_i(Reduce(T)) = cost_i(T)$.*

Proof. This follows from the fact that only the n shallowest leaves affect the computation of the cost. $\square$

If T is an i-level tree, then, in going from T to *Reduce*(T), the signature changes as follows:

Let $sig_i(T) = (m; l_1, \dots, l_C)$.
Set $m' := \min\{m, n\}$.
For $j = 1, \dots, C$, successively replace l_j by $l'_j := \min\{l_j, n - (m' + \sum_{k=1}^{j-1} l'_k)\}$.
Then $sig_i(Reduce(T)) = reduce(m; l_1, \dots, l_C) := (m'; l'_1, \dots, l'_C)$.

The modified signature graph $G' = (V', E')$ is defined as follows.

$$V' = \{\, (m; l_1, \dots, l_C) \;:\; m, l_1, \dots, l_C \geq 0,\; m + l_1 + \dots + l_C \leq n \,\}.$$

Let $S = (m; l_1, \dots, l_C) \in V'$. Then there are $l_1 + 1$ arcs $(S, S') \in E'$ leaving S. For each $q = 0, 1, \dots, l_1$ we have an arc $S \xrightarrow{q} S'$ with

$$S' = reduce((m + l_1; l_2, \dots, l_C, 0) + q \cdot (-1; d_1, d_2, \dots, d_C)).$$

The costs are as in the graph G of Section 3. The *starting vertex* is the signature $S_0 = reduce(0; d_1, \dots, d_C)$, which corresponds to the root tree T_0. There is now also a unique *terminal vertex* $\bar{S} = (n; 0, \dots, 0)$.

Example 2. Below we show how Example 1 must be modified for the present section when $n = 40$. After each expansion, we have to insert an additional reduction step.

signature	$(m;\ l_1,\ l_2,\ l_3,\ l_4)$	partial sums
$[\ (-1; d_1, \dots, d_4) =$	$(-1;\ 0,\ 0,\ 2,\ 2)\ ]$	$(-1, -1, -1,\ 1,\ 3)$
$S =$	$(16;\ 14,\ 8,\ 2,\ 0)$	$(16,\ 30,\ 38,\ 40,\ 40)$
shift level:	$(30;\ 8,\ 2,\ 0,\ 0)$	$(30,\ 38,\ 40,\ 40,\ 40)$
$q = 2$:	$(28;\ 8,\ 2,\ 4,\ 4)$	$(28,\ 36,\ 38,\ 42,\ 46)$
reduce: $S' =$	$(28;\ 8,\ 2,\ 2,\ 0)$	$(28,\ 36,\ 38,\ 40,\ 40)$
shift level:	$(36;\ 2,\ 2,\ 0,\ 0)$	$(36,\ 38,\ 40,\ 40,\ 40)$
$q = 3$:	$(33;\ 2,\ 2,\ 6,\ 6)$	$(33,\ 35,\ 37,\ 43,\ 49)$
reduce: $S'' =$	$(33;\ 2,\ 2,\ 3,\ 0)$	$(33,\ 35,\ 37,\ 40,\ 40)$

The reduction step is most easily understood in terms of the rightmost column: We simple reduce all partial sums which are bigger than n to n.

The modification of the main loop in the algorithm is straightforward. Step 2c is replaced by the following two steps.

2c. If $m' + l'_1 + \dots + l'_C > n$, then replace $(m'; l'_1, \dots, l'_C)$ by $reduce(m'; l'_1, \dots, l'_C)$.

2d. Set $\mathtt{OPT}[m'; l'_1, \ldots, l'_C] := \min\{\mathtt{OPT}[m'; l'_1, \ldots, l'_C],\ \mathit{new\_cost}\}$;

In the end, the cost of the optimal tree can be read off the entry $\mathtt{OPT}[n; 0, \ldots, 0]$, which corresponds to the terminal vertex.

However, it turns out that the graph G' is no longer acyclic, see Example 2, where we have $S'' \prec S'$. Therefore it is not obvious that the above modification is enough to compute the shortest path. However, by studying the example carefully we see why such "backward arcs" like (S', S'') need not worry us. In going from S' to S'', 3 leaves at level $i+1$ become internal nodes, causing 9 leaves to be added to the tree. But the following reduction chops off all but 3 of these new leaves. This means that at least one of the 3 new internal nodes has only one child remaining, but, by Lemma 2, such a node cannot occur in an optimal tree.

To prove correctness, we need to show two things. Firstly, every path in the graph from the starting vertex S_0 to $\bar{S} = (n; 0, \ldots, 0)$ corresponds to *some* tree, with appropriate cost. Secondly and more importantly, the optimal tree corresponds to a path from S_0 to $\bar{S}$ which visits the vertices in an order consistent with the lexicographic order $\prec$. These crucial properties are formulated in the following lemma, which is an analog of Lemma 6.

Lemma 10 (a) *Let T be an optimal tree with height denoted by j. Set $T_i = Trunc_i(T)$, $T'_i = Reduce(Fill(T_i))$, $S_i = sig_i(T'_i)$, and let q_i be the number of internal nodes of T at level $i+1$, for $i = 0, 1, \ldots, j-1$. Then the path*

$$P(T) = (S_0 \overset{q_0}{\rightarrow} S_1 \overset{q_1}{\rightarrow} S_2 \overset{q_2}{\rightarrow} \cdots \overset{q_{j-2}}{\rightarrow} S_{j-1} \overset{q_{j-1}}{\rightarrow} S_j)$$

exists in G', with

$$S_0 \prec S_1 \prec \cdots \prec S_j = (n; 0, \ldots, 0),$$

and $c(P(T)) = cost_j(T)$.

(b) *Let*

$$P = (S_0 \overset{q_0}{\rightarrow} S_1 \overset{q_1}{\rightarrow} S_2 \overset{q_2}{\rightarrow} \cdots \overset{q_{j-2}}{\rightarrow} S_{j-1} \overset{q_{j-1}}{\rightarrow} S_j)$$

be a path in G'. Let T_0 be the root tree. Recursively define

$$T_{i+1} = Reduce(Expand_i(T_i, q_i)), \quad i = 0, 1, \ldots, j-1.$$

Then the tree $T(P) = T_j$ is a j-level tree with $cost_j(T(P)) = c(P)$.

Proof. (a) Note first that in T as well as in each T_i, every missing leaf has depth at least j: if a missing leaf of T had depth less than j, we could remove some leaf at level j and add a new leaf in the position of the missing leaf, obtaining a better tree. Since the truncation operator deletes either all children of a node or none of them, it generates no new missing leaves. Therefore the claimed property carries over from T to all trees T_i. The following property follows:

Property 1 *In all trees $Fill(T_i)$, the number of leaves at depth less than j is less than n.*

Optimality of T implies another property.

Property 2 *If $q_i > 0$, then $i + 1 + c_2 \leq j$, and the operation $Expand_i(\cdot, q_i)$ increases the total number of leaves at levels $0, 1, \ldots, j$.*

Proof. If $i+1+c_2 > j$, then a node at level $i+1$ can have at most one child in the tree T. By Lemma 2, a node with one child cannot occur in an optimal tree, and hence there are no internal nodes at level $i+1$.

On the other hand, if $i+1+c_2 \le j$, then at least two children of each node that is expanded lie on levels $0, 1, \ldots, j$, and these children more than compensate for the loss of leaves due to the fact that the leaves at level $i+1$ that are expanded become internal nodes. $\square$

To prove the lemma we must show that $\mathit{Reduce}(\mathit{Expand}_i(T'_i, q_i)) \doteq T'_{i+1}$. We will use the fact that

$$\mathit{Fill}(T_{i+1}) \doteq \mathit{Expand}_i(\mathit{Fill}(T_i), q_i). \tag{7}$$

This is true because both trees are full trees which have the same number of internal nodes at each level.

Let us first deal with the easy case where $\mathit{Fill}(T_i)$ has less than n leaves. Then the reduction operation is void, $T'_i = \mathit{Fill}(T_i)$ and we have

$$\mathit{Fill}(T_{i+1}) \doteq \mathit{Expand}_i(\mathit{Fill}(T_i), q_i) \doteq \mathit{Expand}_i(T'_i, q_i).$$

If we apply the reduction operation to both sides of this equation, we obtain

$$T'_{i+1} = \mathit{Reduce}(\mathit{Fill}(T_{i+1})) \doteq \mathit{Reduce}(\mathit{Expand}_i(T'_i, q_i)),$$

which is what we wanted to show. We also have $S_i \prec S_{i+1}$ because either T'_{i+1} has more leaves in total than T'_i (in case $q_i > 0$), or the reduction operation is void also for $\mathit{Fill}(T_{i+1})$. In the latter case we have $T'_{i+1} = \mathit{Expand}_i(T'_i, q_i)$, and we can apply Lemma 8.

Now let us consider the other case, where $\mathit{Fill}(T_i)$ has at least n leaves. By Property 1, $\mathit{Fill}(T_i)$ has less than n leaves at levels $0, 1, \ldots, j-1$. The same is true for $T'_i = \mathit{Reduce}(\mathit{Fill}(T_i))$ and therefore T'_i has height $k \ge j$. $\mathit{Fill}(T_i)$ and $T'_i = \mathit{Reduce}(\mathit{Fill}(T_i))$ have the same number of leaves at each level between 0 and $k-1$, and both trees have at least n leaves in total at levels $0, \ldots, k$. After expansion, it follows that also the trees $\mathit{Expand}_i(\mathit{Fill}(T_i), q_i)$ and $\mathit{Expand}_i(T'_i, q_i)$ have the same number of leaves at each level between 0 and $k-1$, and moreover, by Property 2, both trees still have at least n leaves in total at levels $0, \ldots, k$. The reduction operation will therefore yield equivalent trees:

$$\mathit{Reduce}(\mathit{Expand}_i(\mathit{Fill}(T_i), q_i)) \doteq \mathit{Reduce}(\mathit{Expand}_i(T'_i, q_i))$$

By applying (7) we obtain

$$T'_{i+1} = \mathit{Reduce}(\mathit{Fill}(T_{i+1})) \doteq \mathit{Reduce}(\mathit{Expand}_i(\mathit{Fill}(T_i), q_i)) \doteq \mathit{Reduce}(\mathit{Expand}_i(T'_i, q_i)),$$

which is what we wanted to show.

To show $S_i \prec S_{i+1}$, let $S_i = \mathit{sig}_i(T'_i) = (m; l_1, l_2, \ldots, l_C)$ and $S_{i+1} = \mathit{sig}_{i+1}(T'_{i+1}) = (m'; l'_1, l'_2, \ldots, l'_C)$. By the above considerations about the numbers of leaves at levels up to k, we have $m + l_1 + \cdots + l_C = m + l_1 + \cdots + l_{C-1} = \cdots = m + l_1 + \cdots + l_{k-i} = n$ and $m + l_1 + \cdots + l_{k-i-1} < n$, but for S_{i+1} we have $m' + l'_1 + \cdots + l'_C = m' + l'_1 + \cdots + l'_{C-1} = \cdots = m' + l'_1 + \cdots + l'_{k-i} = m' + l'_1 + \cdots + l'_{k-(i+1)} = n$. (Note that S_{i+1} is the signature at level $i+1$.) Thus, $S_i \prec S_{i+1}$.

(b) Since the *Expand* and *Reduce* operations are faithfully modeled by the arcs of the graph, it follows that the trees T_i exist and have the given signatures. Equality of costs can be proved in the same way as in Lemma 6, using the fact that reduction does not influence the cost (Lemma 9). $\square$

Thus, as in the previous section, we can actually find the shortest path in $O(|V'| + |E'|) = O(n^{C+2})$ steps, each step taking $O(C)$ time. The code is given in Figure 6. Actually, it is not difficult to be more careful in the implementation and avoid scanning the "backward arcs" of the graph. In the loop of Steps 2b–2d, we let q run only up to $\min\left(l_1, n - (m + \sum_{j=1}^{c_2} l_j)\right)$. We leave it as an exercise for the reader to check that this is correct.

```
1.  Initialization. Set $\texttt{OPT}[m; l_1, \ldots, l_C] := \infty$ for all entries with $m + l_1 + \cdots + l_C \le n$.
    Set $S_0 = \mathit{reduce}(0; d_1, d_2, \ldots, d_C)$. Set $\texttt{OPT}[S_0] := 0$;

2.  Loop. Process the array entries $\texttt{OPT}[m; l_1, \ldots, l_C]$ with
    $(l_1, \ldots, l_C) \ne (0, \ldots, 0)$ in lexicographically increasing order of
    $(m + l_1 + \cdots + l_C,\ m + l_1 + \cdots + l_{C-1},\ \ldots,\ m + l_1,\ m)$.

    For each entry $\texttt{OPT}[m; l_1, \ldots, l_C]$ do the following:

2a.     $\mathit{new\_cost} := \texttt{OPT}[m; l_1, \ldots, l_C] + \sum_{t=m+1}^{n} p_t$;
        for $q := 0$ to $l_1$ do
2b.         Let $(m'; l'_1, \ldots, l'_C) := (m + l_1; l_2, \ldots, l_C, 0) + q \cdot (-1; d_1, \ldots, d_C)$;
2c.         If $m' + l'_1 + \cdots + l'_C > n$ then replace $(m'; l'_1, \ldots, l'_C)$ by $\mathit{reduce}(m'; l'_1, \ldots, l'_C)$.
2d.         Set $\texttt{OPT}[m'; l'_1, \ldots, l'_C] := \min\{\texttt{OPT}[m'; l'_1, \ldots, l'_C],\ \mathit{new\_cost}\}$;
        end for;

3.  Termination. $\texttt{OPT}[n; 0, \ldots, 0]$ is the cost of the optimal code.
```

Figure 6: The improved algorithm.

The bound $|V'| = O(n^{C+1})$ for the number of nonnegative $(C+1)$-tuples $(m; l_1, \ldots, l_C)$ with $m + l_1 + \cdots + l_C \le n$ is only a loose estimate. By looking at the strictly increasing sequence

$$(m,\ m + l_1 + 1,\ m + l_1 + l_2 + 2,\ \ldots,\ m + \cdots + l_C + C) \tag{8}$$

we see that these tuples are in one-to-one correspondence with the family of $(C+1)$-subsets of the set $\{0, \ldots, n + C\}$ and hence their number is $\binom{n+C+1}{C+1}$. (In fact, if one wants to store the table `OPT` as compactly as possible, in an array of $\binom{n+C+1}{C+1}$ entries in the proper lexicographic order $\prec$, it is most convenient to use the vector (8) for indexing.) The number of arcs is now $|E| < (n+1)|V|$, and the processing of each arc involves an overhead of $O(C+1)$ time. This gives a time bound of $O\left((n+1)(C+1)\binom{n+C+1}{C+1}\right)$. We have $(C+1)\binom{n+C+1}{C+1} < 4n^{C+1}$ for all $n \ge 2$, as can be easily proved by induction on C, proving the base cases $C = 1$ and $C = 2$ separately. We have therefore proved the following theorem.

Theorem 1 *The minimum-cost prefix-free code for n words can be computed in $O(n^{C+2})$ time and $O(n^{C+1})$ space, if the costs of the letters are integers between 1 and C.*

5 Conclusions, Implementations, and Open Problems

In this paper we described how to solve the optimal-coding problem in $O(n^{C+2})$ time where the letter lengths $c_1 \le c_2 \le \cdots \le c_r$ are integers, $C = c_r$ is the longest length of an encoding letter and n is the number of symbols to be encoded. This improves upon the previous best known solution due to Karp [14] which solved the problem by transformation to integer linear programming and whose running time could therefore only be bounded by an exponential function of n.

Our algorithm works by creating a weighted graph G' with $O(n^{C+1})$ vertices and $O(n^{C+2})$ arcs and showing that optimal codes (corresponding to minimum cost trees) can be constructed by finding least cost paths in G'.

It is easy to see that the *height* of a tree is exactly the number of arcs in its corresponding path from S_0 to $\bar{S}$. Thus we can also use our formulation to solve the *Length-Limited Optimal Coding Problem.* In this problem, we are given the same data as in the original problem and an integer L, and we want to find a minimum cost code containing no code-word of length more than L. To solve this new problem it is only necessary is to find the least cost path from the source to the sink that uses L or fewer arcs, which can be easily done in $O(Ln^{C+2})$ time.

In a practical implementation of our algorithm many improvements are possible. Recall that our algorithm is equivalent to searching for a shortest path in a directed acyclic graph. The simple shortest path algorithm which we used essentially scans all arcs of the graph. There is a whole range of heuristic graph search algorithms to be considered that might speed up the running time of the algorithm in practice, cf. [17].

One obvious direction for future research is to resolve the complexity of the optimal-coding problem. It is still unknown if the problem is polynomial-time solvable, or if the problem is NP-hard.

Another direction is to relax the restriction that the c_i are integers. Obviously, in any conceivable practical application the given numbers c_i are rationals; therefore they can all be scaled to be integers and our algorithm can be used. However, since the largest integer cost enters into the exponent of the complexity, this approach is in general not feasible. It is challenging to find an algorithm that would solve the problem with, for example, $(c_1, c_2, c_3) = (0.169, 0.3, 0.531)$ in reasonable time, and which could just as easily be applied to incommensurable lengths such as $(c_1, c_2, c_3) = (1, \sqrt{2}, \sqrt{3})$.

It is not known whether the restriction to prefix-free codes in the optimal coding problem, as opposed to the more general class of uniquely decipherable codes, is a severe restriction that excludes codes which would otherwise be optimal, or whether an optimal code in the class of uniquely decipherable codes can always be found among prefix-free codes. See the survey by Bruyère and Latteux [4] for this and related open problems.

We have tested the algorithm for computing optimal codes for the Roman alphabet plus "space", using the probabilities which are given in [3, p. 52] and reproduced in Karp [14]. We ran an experimental implementation of our algorithm in MAPLE on a HP 9000/750 workstation. When the encoding alphabet had 2 letters, $c_1 = 1$, $c_2 = 2$, we found an optimal code with a cost of 5.8599 in 1.5 seconds. For an encoding alphabet with 3 letters, $(c_1, c_2, c_3) = (2, 3, 3)$, we found an optimal cost of 6.7324 in 6 seconds. The only algorithm in the literature for which running times are reported is the algorithm of Karp [14] from 1961. His program took 1 minute for the first example[1] and 5 minutes for the second one on an IBM 704 computer. These running times can hardly be compared. On the one hand, this machine was much slower than today's computers. An IBM 704 in 1955 could carry out about 5,000 floating-point operations per second (0.005 MFLOPS). On the other hand, the MAPLE system is not designed for taking the most efficient advantage of computer hardware. For example, all arithmetic operations are carried out in software, and array indexing is not as efficient as in a conventional programming language. (For the second problem, about 40 % of the total running time was spent initializing the array `OPT`.) We ran an integer programming formulation derived from Karp's on the same workstation as our MAPLE code. The model was formulated in the AMPL modeling language [8], using about 25 lines of code, and was solved with the CPLEX 4.0 software for mixed-integer optimization. (The source for the AMPL model and data is included in Appendix A.) Interestingly, the 3-letter problem was easier to solve than the 2-letter problem. It took 0.19 seconds, 46 branch-and-bound nodes and a total of 207 pivots of the simplex algorithm to solve the 2-letter problem, but only 0.09 seconds, 3 branch-and-bound nodes and 143 simplex iterations to solve the 3-letter problem.

Acknowledgements: The authors would like to thank Dr. Jacob Ecco for introducing them to the Morse Code puzzle which sparked this investigation. The first author will also like to acknowledge that this work was partially supported by HK RGC/CRG grant number HKUST 181/93E and HK RGC/CRG grant number HKUST 652/95E. The second author was supported by the Spezialforschungsbereich F 003, *Optimierung und Kontrolle*, Projektbereich Diskrete Optimierung.

References

[1] Doris Altenkamp and Kurt Mehlhorn, "Codes: unequal probabilities, unequal letter costs," *J. Assoc. Comput. Mach.* **27** (3) (July 1980), 412–427.

[2] Phil Bradford, Mordecai J. Golin, Lawrence L. Larmore, and Wojciech Rytter "Optimal prefix-free codes for unequal letter costs and dynamic programming with the Monge property," to appear in *Algorithms*, Proceedings of the Sixth European Symposium on Algorithms (ESA '98), ed. G. Bilardi and G. F. Italiano Lecture Notes in Computer Science, Springer-Verlag, 1998.

[3] L. Brillouin, *Science and Information Theory*, Academic Press, New York 1956.

[4] Véronique Bruyère and Michel Latteux, "Variable-length maximal codes," in *Automata, Languages and Programming*, Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP 96), ed. F. Meyer auf der Heide and B. Monien. Lecture Notes in Computer Science, Vol. **1099**, Springer-Verlag, 1996, pp. 24–47.

[1]The code we found for the first example was different from Karp's even though, of course, it had the same cost.

[5] Siu-Ngan Choi and M. Golin, "Lopsided trees: analyses, algorithms, and applications," in *Automata, Languages and Programming*, Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP 96), ed. F. Meyer auf der Heide and B. Monien. Lecture Notes in Computer Science, Vol. **1099**, Springer-Verlag, 1996, pp. 538–549.

[6] N. Cot, "A linear-time ordering procedure with applications to variable length encoding," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems*, (1974), pp. 460–463.

[7] N. Cot, "Complexity of the variable-length encoding problem," in *Proceedings of the 6th Southeast Conference on Combinatorics, Graph Theory and Computing*, (1975), pp. 211–224.

[8] Robert Fourer, David M. Gay, and Brian W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Boyd & Fraser, Danvers, Massachusetts, 1993.

[9] E. N. Gilbert, "Coding with digits of unequal cost," *IEEE Transactions on Information Theory* **41** (2), (March 1995), 596–600.

[10] Mordecai Golin and Günter Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes for unequal letter costs," in *Automata, Languages and Programming*, Proceedings of the 22nd International Colloquium on Automata, Languages, and Programming (ICALP 95), Szeged, Hungary, July 1995, ed. F. Gécseg. Lecture Notes in Computer Science, Vol. **944**, Springer-Verlag, 1995, pp. 256–267.

[11] Mordecai J. Golin and Neal Young, "Prefix codes: equiprobable words, unequal letter costs," *SIAM Journal on Computing.* **25** (6) (December 1996), 1281–1292.

[12] G. H. Hardy, J. E. Littlewood, and G. Pólya, *Inequalities*, Cambridge University Press, Cambridge 1967.

[13] Sanjiv Kapoor and Edward Reingold, "Optimum lopsided binary trees," *Journal of the Association for Computing Machinery* **36** (3) (July 1989), 573–590.

[14] R. Karp, "Minimum-redundancy coding for the discrete noiseless channel," *IRE Transactions on Information Theory* **IT-7** (1961), 27–39.

[15] Abraham Lempel, Shimon Even, and Martin Cohen, "An algorithm for optimal prefix parsing of a noiseless and memoryless channel," *IEEE Transactions on Information Theory*, **IT-19**(2) (March 1973), 208–214.

[16] Kurt Mehlhorn *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin, 1984.

[17] Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto 1980.

[18] 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) (April 1975), 202–214,

[19] Serap A. Savari, “Some notes on Varn coding,” *IEEE Transactions on Information Theory*, **40** (1) (Jan. 1994), 181–186.

[20] K. A. Schouhamer Immink, “Runlength-limited sequences,” *Proc. IEEE* **78** (1990), 1744–1759.

[21] Robert Sedgewick, *Algorithms*, 2nd ed., Addison-Wesley, Reading, Mass. (1988).

[22] P. H. Siegel, “Recording codes for digital magnetic storage,” *IEEE Trans. Magn.* **MAG-21** (1955), 1344–1349.

[23] L. E. Stanfel, “Tree structures for optimal searching,” *Journal of the Association for Computing Machinery* **17** (3) (July 1970), 508–517.

A Karp’s integer programming model

AMPL model file:

```
param n;        # number of weights
param maxebene; # maximum permitted level of the tree
set letters;
param C {i in letters} integer >0; # costs of letters
param maxC := max {i in letters} C[i];

param w{i in 1..n} >=0;
check {i in 2..n}:  w[i-1] <= w[i];

param zil; #optional target value for the objective function

var inn{i in 1-maxC .. maxebene} >=0; # inner nodes
var bl {i in 1..maxebene} >=0;        # leaves at each level
var nochnicht {i in 0..maxebene+1} integer >=0;
            # number of leaves still needed below each level

minimize cost:
 ( sum {j in 1..maxebene}   # piece-wise linear function:
       << {i in 1..n-1} i; {i in 1..n} w[i] >> nochnicht[j]
 ) - zil;

subject to
  erhaltung {i in 1..maxebene}:
    inn[i]  <=  ( sum{j in letters} inn[i-C[j]] ) - bl[i];
  forher {i in 1-maxC .. -1}: inn [i] = 0;   # dummy entries
  anfang: inn [0] = 1;
# ende {i in maxebene-maxC+1..maxebene}: inn[i]=0;#redundant

  rest {i in 1..maxebene}:
         nochnicht[i+1] = nochnicht[i] - bl[i];
  rand1: nochnicht[1]=n;
  rand2: nochnicht[maxebene+1]=0;  # boundary conditions
```

AMPL data file:

```
data;
#english letters
#as in Karp 1961

param:
  letters: C :=
        a  1
        b  2  ;
param n := 27;
param
 maxebene := 20;
param zil := 0;

param w :=
 1  10    15  230
 2  10    16  290
 3  10    17  350
 4  20    18  470
 5  30    19  520
 6  80    20  540
 7 105    21  550
 8 110    22  590
 9 120    23  630
10 120    24  654
11 175    25  720
12 210    26 1050
13 225    27 2000
14 225
;
```