Optimal Prefix-Free Codes for Unequal Letter Costs: Dynamic Programming with the Monge Property

Proposed Running Head: Prefix Codes for Unequal Letter Costs

Phil Bradford [1] **Mordecai J. Golin** [2] **Lawrence L. Larmore** [3]
Wojciech Rytter [4]

[1]Max-Planck-Institut für Informatik, 66123 Saarbruecken, Germany

[2]`Corresponding Author:` Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong. email:`golin@cs.ust.hk`. Tel: (852) 2358-6993. Fax: (852) 2358-1477. This research partially supported by HK RGC CERG grants 652/95E, 6082/97E,6137/98E, and HKUST DAG 98/99.EG23

[3]Department of Computer Science, University of Nevada, Las Vegas, NV 89154-4019. Email: `larmore@cs.unlv.edu`. Research supported by NSF grant CCR-9503441.

[4]Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097 Warszawa, Poland, and Department of Computer Science, University of Liverpool.

Abstract

In this paper we discuss the problem of finding optimal prefix-free codes for unequal letter costs, a variation of the classical Huffman coding problem. Our problem consists of finding a minimal cost prefix-free code in which the encoding alphabet consists of unequal cost (length) letters, with lengths α and β. The most efficient algorithm known previously requires $O(n^{2+\max(\alpha,\beta)})$ time to construct such a minimal-cost set of n codewords, provided α and β are integers. In this paper we provide an $O(n^{\max(\alpha,\beta)})$ time algorithm. Our improvement comes from the use of a more sophisticated modeling of the problem, combined with the observation that the problem possesses a "Monge property" and that the SMAWK algorithm on monotone matrices can therefore be applied.

Keywords: Dynamic Programming, Huffman Codes, Lopsided Trees, Monge Matrix, Monotone Matrix, Prefix-Free Codes.

1 Introduction

Finding optimal prefix-free codes for unequal letter costs (and the associated problem of constructing optimal lopsided trees) is an old classical problem. It consists of finding a minimal cost prefix-free code in which the encoding alphabet consists of unequal cost (length) letters, of lengths α and β, $\alpha \leq \beta$. The code is represented by a *lopsided tree* in the same way that a Huffman tree represents a solution for the Huffman coding problem. Despite this similarity, the case of unequal letter costs seems much harder to solve than the classical Huffman problem; no polynomial time algorithm is known for general letter costs, despite a rich literature on the problem. (See *e.g.*, [1] for a survey.) However, there are known polynomial time algorithms when α and β are integer constants [8].

The problem of finding the minimum cost tree in this case was first studied in 1961 by Karp [10] who solved the problem of finding optimal prefix-free codes for (possibly non-binary) encoding alphabets with unequal integral cost letters by reduction to integer linear programming. This yielded an algorithm with time complexity exponential in n, where n is the number of leaves in the code tree. Since then there has been much work on variations of the problem, such as bounding the cost of the optimal tree (Altenkamp and Mehlhorn [2], Kapoor and Reingold [9], and Savari [18]), restriction to the special case when all of the weights are equal (Cot [6], Perl Gary and Even [17], and Choi and Golin [5]), and approximating the optimal solution (Gilbert [7]). However, it is still not known whether the basic problem is polynomial-time solvable, or is $\mathcal{NP}$-hard.

The only published technique other than Karp's for solving the general problem is due to Golin and Rote [8]. For r-letter encoding alphabets in which the respective costs of the letters are integers satisfying $c_1 \leq c_2 \leq \cdots \leq c_r$, they describe an $O\left(n^{c_r+2}\right)$-time dynamic programming algorithm that constructs the tree in a top-down fashion; in our binary encoding case this becomes $O(n^{\beta+2})$-time and is currently the most efficient known algorithm for small β. In this paper, we introduce a different dynamic programming approach, obtaining a bottom-up algorithm, and saving a quadratic factor in time complexity. A straightforward algorithmic realization of this approach would also run in $O(n^{\beta+2})$-time, but we describe two techniques which decrease the time complexity, each by a factor of $\Theta(n)$. The first technique transforms the search space into a larger, but more tractable, one. The second uses monotone-matrix concepts, *i.e.*, the *Monge property* [16] and the SMAWK algorithm [3].

Before proceeding we point out that there are two well understood special cases of the general arbitrary-costs arbitrary-weights problem: the Varn problem, see [20], in which all the leaf weights are the same, and the Huffman problem, in which all the letter-costs are the same. In the Varn case there is an $O(n)$ top down node-spltting algorithm; in the Huffman case

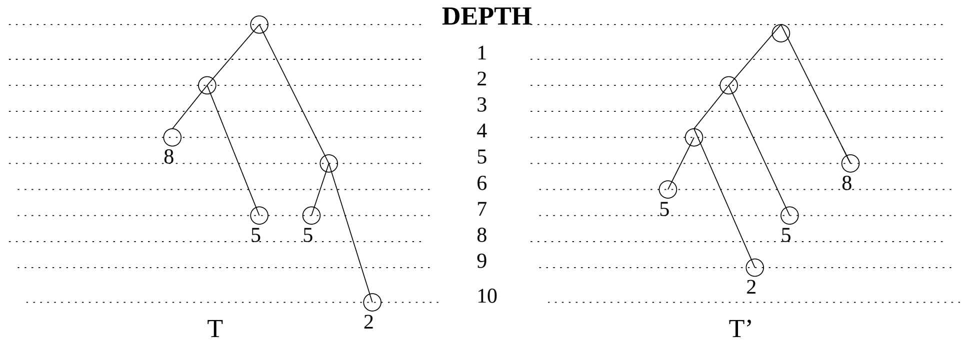

Figure 1: Two lopsided $2,5$ trees. The trees are labeled with $P = \{2, 5, 5, 8\}$.

there is an $O(n)$ bottom-up merging algorithm. Both of these algorithms take advantage of the special combinatorial structure their input restrictions imply to utilize (different) greedy techniques to build their optimal trees. In the general, arbitrary-weights, arbitrary-letter-costs case that this paper addresses, both both of these greedy algorithms can easily be shown to fail.

We also point out that unlike in Golin and Rote [8] our method is limited to *binary* code trees. The technique that we employ does not seem easily generalizable to general r-ary trees since the new representations of binary trees that we use and the special combinatorial properties they permit us to exploit do not seem to extend to the r-ary case.

Our approach requires a better understanding of the combinatorics of lopsided trees, which, in turn, requires introducing some definitions. Let α, β be positive integers, $\alpha \le \beta$. A *binary lopsided* α, β *tree* (or just a *lopsided tree*, if α and β are understood) is a binary tree in which every non-leaf node u of the tree has two children, where the length of the edge connecting u to its left child is α, and the length of the edge connecting u to its right child is β. Figure 1 shows two 2,5 lopsided trees.

Let T be a lopsided tree and $v \in T$ some node. Then

$$\begin{aligned} depth(T, v) &= \text{sum of the lengths of the edges connecting } root(T) \text{ to } v \\ depth(T) &= \max\{depth(T, v) \,:\, v \in T\} \end{aligned}$$

For example, tree T in Figure 1 has depth 10 while tree T' has depth 9.

Let $P = (p_1, p_2, \ldots, p_n)$ be a sequence of nonnegative weights. Let T be a lopsided tree with n leaves labeled $v_1, v_2, \ldots, v_n$. The *weighted external path length* of T with respect to P is

$$cost(T, P) = \sum_i p_i \cdot depth(T, v_i).$$

Given P, our problem is to construct a lopsided tree T that minimizes $cost(T, P)$. Returning to Figure 1 we find that, for $P = \{2, 5, 5, 8\}$ tree T has

$$cost(T, P) = 2 \cdot 10 + 5 \cdot 7 + 5 \cdot 7 + 8 \cdot 4 = 122$$

while tree T' has

$$cost(T', P) = 2 \cdot 9 + 5 \cdot 7 + 5 \cdot 6 + 8 \cdot 5 = 123.$$

With a little more work it is not hard to see that tree T is a minimal cost lopsided 2, 5 tree for P. As was pointed out quite early [10] (see [8] for a more recent detailed description) this problem is equivalent to finding a minimal cost prefix-free code in which the encoding alphabet consists of two unequal cost (length) letters, of lengths α and β. If $\alpha = \beta$ the problem reduces directly to the standard Huffman coding problem.

Note that, given any particular tree T, the cost actually depends upon the enumeration of the leaves of T, the cost being minimized when leaves of greater depth always have smaller or equal weight. We therefore will assume that the leaves of T are enumerated in nonincreasing order of their depth, *i.e.*, $depth(T, v_1) \geq depth(T, v_2) \geq \cdots \geq depth(T, v_n)$, and that $p_1 \leq p_2 \leq \cdots \leq p_n$. This assumption will be used implicitly throughout the paper.

In the next section we will introduce some sequences that are related to trees and introduce some properties that permit us to restate our problem as a problem about sequences rather than trees. In Sections 4 and 5, we prove most of those properties. In Section 6, we discuss how to use the Monge property to reduce the running time of the algorithm. In Section 7 we prove a key lemma stated in Section 5. Section 8 concludes.

2 Three types of sequences related to lopsided trees

Let n and P be fixed. Throughout the paper, we describe a tree only by how many leaves it has at each level. This description is justified by the fact that $cost(T, P) = cost(T', P)$, if T and T' have the same number of leaves at every level. In what follows we say that node v is on *level* i of T if v is i levels from the bottom of T, i.e., if $i = depth(T) - depth(T, v)$.

We consider three classes of sequences:

1. for a given tree T, the **numbers-of-leaves sequence** $\Delta(T)$= $(\delta_0(T), \delta_1(T), \ldots, \delta_{d-1}(T))$, where $\delta_i(T)$ is the number of leaves which are below or at a given level i.

2. for a given tree T, its **characteristic sequence**, denoted by $seq(T)$, which is the sequence $B_T = (b_0, b_1, \ldots, b_{d-1})$ in which b_i is the number of right children at or below level i, for all $0 \leq i < d = depth(T)$. See Figure 2.

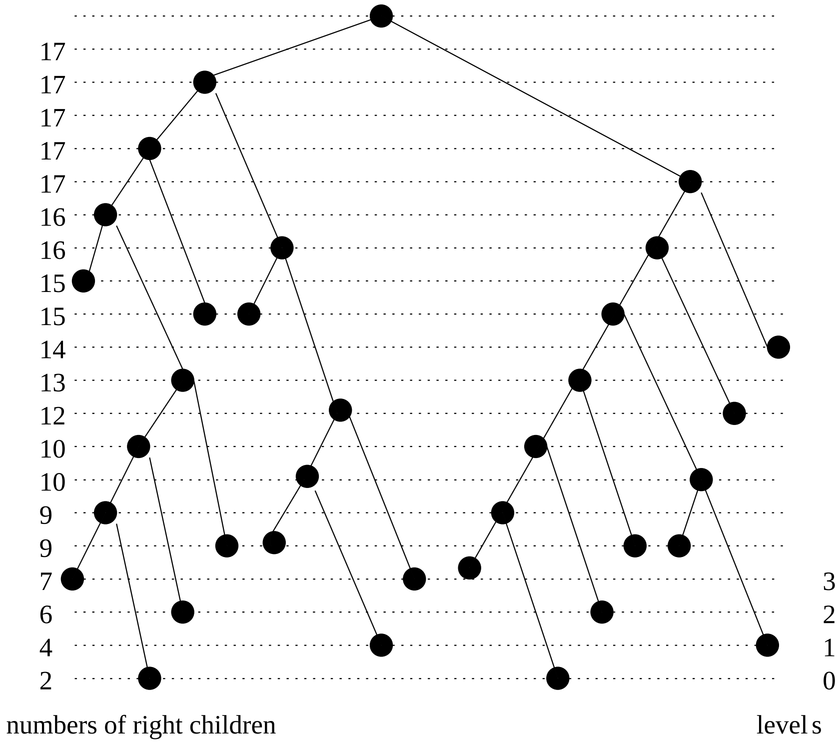

Figure 2: A 2-5 tree T with $depth(T) = 20$ and its characteristic sequence

$$seq(T) \;=\; (2, 4, 6, 7, 9, 9, 9, 10, 10, 12, 13, 14, 15, 16, 16, 17, 17, 17, 17, 17).$$

3. monotonic sequences $B = b_0,\, b_1,\, \ldots,\, b_{d-1}$ of nonnegative integers which end in the β-tuple $(n-1, n-1, \ldots, n-1)$. A sequence is **monotonic** if $b_0 \le b_1 \le \cdots \le b_{d-1}$. Denote the set of such monotonic sequences ending in β-tuple $(n-1, n-1, \ldots, n-1)$ by $\mathcal{M}_n$. If T is a tree, we shall see below that $seq(T) \in \mathcal{M}_n$, but if $B \in \mathcal{M}_n$, there may be no tree for which B is a characteristic sequence. We say a sequence $B \in \mathcal{M}_n$ is *legal* if $B = seq(T)$ for some tree T.

We now provide some intuition as to how these definitions arise. $\Delta(T)$ is introduced because $\delta_i(T) - \delta_{i-1}(T)$ is the number of leaves on level i, and, as mentioned above, these values can be used to calculate $cost(T, P)$. In the next section we will see that $\Delta(T)$ can be reconstructed from $seq(T)$.

Monotonic sequences are a generalization of characteristic sequences. For any tree T, $seq(T)$ is monotonic by definition. If T is a tree with n leaves then T must have $n-1$ internal nodes and thus $n-1$ right children. The top β levels of T (not counting the root) cannot contain any right children.

Thus, $seq(T)$ terminates in a β tuple $(n-1, n-1, \ldots, n-1)$ and $\mathcal{M}_n$ contains the set of all legal sequences.

In section 4 we will introduce a quantity, $cost(B, P)$, defined for monotonic sequences.

This cost will have two important properties. The first property is that this new cost function is effectively a generalization of the cost function on trees defined above; the second property is that minimum cost is always achieved on a sequence which is the characteristic sequence of some tree. Formally, the first property is

P1. **Consistency of the cost function:**

$$cost(seq(T), P) = cost(T, P)$$

Thus, the problem of finding a minimum cost lopsided tree is totally equivalent to that of finding a minimum cost *legal* sequence in $\mathcal{M}_n$. The reason for introducing all of the notation is the next important property, which will be proven in Lemma 3.

P2. **Key-property:**

For each $B \in \mathcal{M}_n$, not necessarily a legal one, and weight set P, $|P| = n$, a lopsided tree $BuildTree(B)$ can be constructed such that $cost(BuildTree(B), P) \leq cost(B, P)$. Furthermore $BuildTree(B)$ can be constructed in $O(n^2)$ time.

Intuitively (we go into greater detail in Section 5) $BuildTree(B)$ is defined so that if $B = seq(T)$ for some min-cost tree T, then $T = BuildTree(B)$. Property (P1) then implies that $cost(BuildTree(B), P) = cost(T, P) = cost(B, P)$. Defining $BuildTree(B)$ so that it has this property is not hard. The difficult part is proving that *all* $B \in \mathcal{M}_n$, even if they are legal and do not correspond to a minimum-cost tree or, worse, even if they are not legal, satisfy $cost(BuildTree(B), P) \leq cost(B, P)$. As we shall see in the next section, this inequality is important because it immediately leads to an algorithmic approach to solving our problem.

3 General structure of the algorithm

If B is a min-cost monotonic sequence and $T = BuildTree(B)$, then by (P1) and (P2) we have

$$cost(seq(T), P) = cost(T, P) \leq cost(B, P).$$

The minimality of B then implies that $cost(seq(T), P) = cost(B, P)$ and thus $seq(T)$ is also a min-cost sequence in $\mathcal{M}_n$. Legal sequences are a subset of $\mathcal{M}_n$ so this immediately implies that $seq(T)$, by definition a legal sequence, is a *min-cost* legal sequence and, from (P1), that T is a min-cost lopsided tree.

Our algorithm will therefore be to find a min-cost monotonic sequence $B \in \mathcal{M}_n$ and then

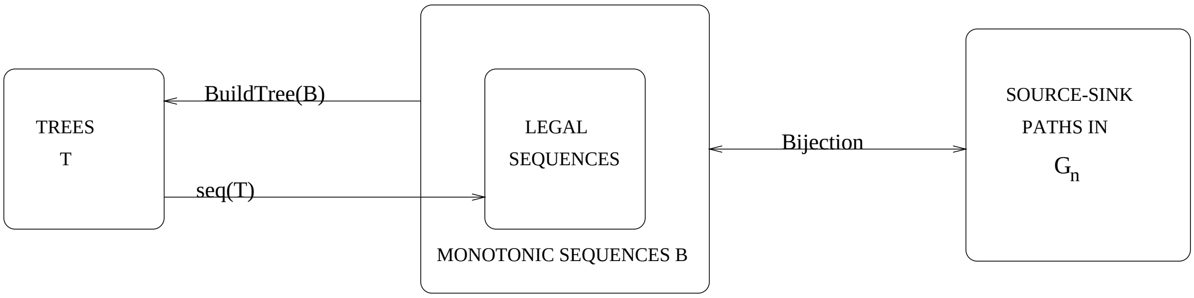

Figure 3: Relations between lopsided trees, monotonic sequences and graphs.

Algorithm *Optimal_Tree_Construction*

1. compute a minimum-cost source-sink path π in the special graph G_n defined in Section 6.
2. construct a monotonic sequence $B \in \mathcal{M}_n$ corresponding to π;
3. **return** *BuildTree*(B)

Figure 4: Top level view of the optimal tree construction algorithm.

build the min-cost tree $T = \mathit{BuildTree}(B)$. The nontrivial aspect of our algorithm, and the fact which will save us our first $\Theta(n)$ factor in our running time, is that the above properties permit us to search for an optimum among *all* sequences in $\mathcal{M}_n$, not just the legal ones. Essentially they permit us to search in a larger, but more tractable, search space.

In Section 6 we show how to actually find min-cost monotonic sequences. We construct a particular edge-weighted graph G_n, with designated source and sink, such that there is a one-one correspondence between the monotonic sequences in $\mathcal{M}_n$ and the source-sink paths in G_n. This correspondence will have the further property that $\mathit{cost}(B, P)$ will be exactly the weight of the path corresponding to B. Finding a min-cost sequence is therefore reduced to finding a min-cost source-sink path in G_n. We will also see that this optimization problem satisfies a *Monge* property that will enable it to be solved a factor of $\Theta(n)$ faster than it would normally require.

The relationship between lopsided trees, sequences and paths is sketched in Figure 3. The general structure of the algorithm is given in Figure 4.

4 Defining the cost in terms of sequences

The main goal of this section is to define a $cost(B, P)$ for all $B \in \mathcal{M}_n$ that has the property that the cost of a tree T and its associated characteristic sequence $seq(T)$ will be the same. We start by defining values:

Definition 1

$$S_i = \begin{cases} \sum_{j \le i} p_j & \text{if } 1 \le i \le n \\ \infty & \text{otherwise.} \end{cases}$$

With this definition it is straightforward to write the cost of T as a function of $\Delta(T) = (\delta_0(T), \delta_1(T), \ldots, \delta_{d-1}(T))$ where $\delta_i(T)$ is the number of leaves which are below or at a given level i.

Lemma 1

$$cost(T, P) = \sum_{0 \le k < depth(T)} S_{\delta_k(T)}.$$

Proof:

$$\begin{aligned} Cost(T,P) &= \sum_{i \le n} p_i \cdot depth(v_i) \\ &= \sum_{0 \le j \le depth(T)} j \sum_{\{v_i :\ depth(v_i) = j\}} p_i \\ &= \sum_{1 \le j \le depth(T)} \sum_{\{v_i : depth(v_i) \ge j\}} p_i \\ &= \sum_{1 \le j \le depth(T)} \sum_{\{v_i : height(v_i) \le n-j\}} p_i \\ &= \sum_{0 \le k < depth(T)} S_{\delta_k(T)} \end{aligned}$$

□

We now define a cost on monotonic sequences B and then, in the next lemma, see that this cost is identical to the tree cost on T if the sequence is the legal sequence $B = seq(T)$.

Definition 2 *Let* $B = b_0, b_1, \ldots, b_{d-1}$ *be a monotonic sequence.* $(\forall k,\ 0 \le k < d)$, *set*

$$N_k(B) = b_k + b_{k-(\beta-\alpha)} - b_{k-\beta}$$

where $b_j = 0$ *for all* $j < 0$. *Now define*

$$cost(B, P) = \sum_{0 \le k < d} S_{N_k(B)}$$

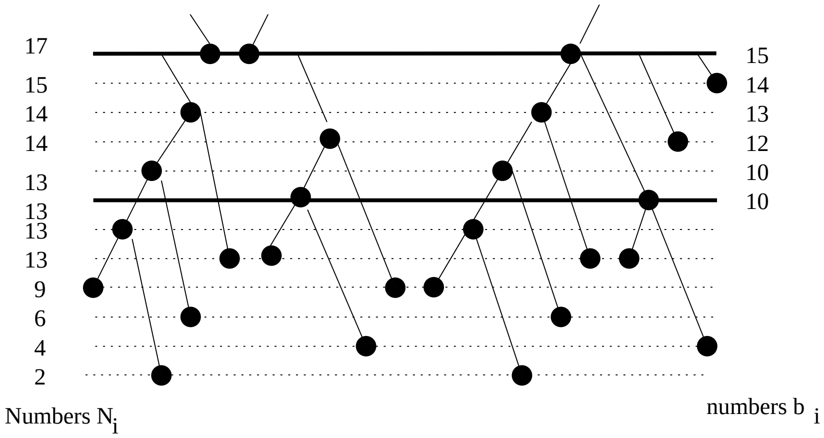

Figure 5: The bottom forest F_{11} of the tree T from Figure 2.

If B is the sequence for some tree T then
$N_k(B) = \delta_k(T)$, the number of leaves on or below level k.

Lemma 2 *Let T be a lopsided tree and $B = seq(T) = b_0, b_1, \ldots, b_{d-1}$. Then*

- **(A)** $(\forall i,\ 0 \le i < d = depth(T))$,

$$\delta_i(T) = N_i(seq(T)) = b_i + b_{i-(\beta-\alpha)} - b_{i-\beta}$$

- **(B)** $cost(T, P) = cost(B, P)$

Proof: Let $\mathcal{F}_k = forest_k(T)$ be the forest consisting of all nodes at level k and below (See Figure 5). From our perspective its most useful property will be the fact that a node $u \in \mathcal{F}_k$ is a leaf in $\mathcal{F}_k$ if and only if the same node is a leaf in T.

Note that $\delta_k(T)$, previously defined as the the number of leaves on or below level k in T, is therefore also the exact number of leaves in $forest_k(T)$.

To prove (A), note that $\mathcal{F}_i$ is a forest, hence

$$\begin{aligned} N_i(T) &= \{u \in F_i \,:\, u \text{ is a leaf in } \mathcal{F}_i\} &(1)\\ &= \textit{Number of internal nodes in } \mathcal{F}_i + \textit{Number of trees in } \mathcal{F}_i &(2) \end{aligned}$$

The first summand in the last line is easily calculated. A node at height k is internal in $\mathcal{F}_i$ if and only if it is the parent of some right child at level $k - \beta$. Thus

$$\textit{Number of internal nodes in } \mathcal{F}_i = b_{i-\beta}. \tag{3}$$

The second summand is only slightly more complicated to calculate. The number of trees in $\mathcal{F}_i$ is exactly the same as the number of tree-roots in $\mathcal{F}_i$.

Now note that a node in $\mathcal{F}_i$ is a tree-root in $\mathcal{F}_i$ if and only if its parent is not in $\mathcal{F}_i$. Thus a right child at height k in $\mathcal{F}_i$ is a tree-root if and only if $i - \beta < k \leq i$ and there are exactly $b_i - b_{i-\beta}$ such nodes.

Similarly a left child at height k is a tree-root if and only if $i - \alpha < k \leq i$.

This may occur if and only if the left child's right sibling is at height k, where $i - \beta < k \leq i - (\beta - \alpha)$. The number of such nodes is therefore

$$b_{i-(\beta-\alpha)} - b_{i-\beta}.$$

We have therefore just seen that

$$\textit{Number of trees in } \mathcal{F}_i = (b_i - b_{i-\beta}) + (b_{i-(\beta-\alpha)} - b_{i-\beta}). \tag{4}$$

Combining (3) and (4) completes the proof of (A). (B) follows from Lemma 1 and (A). □

5 Description of the function *BuildTree*

Each characteristic sequence describes the unique "shape" of a lopsided tree. Although intuitive, the reconstruction of a tree from its characteristic sequence can be rather technical. The main goal of this section is to describe a procedure that reconstructs *min-cost* trees from their sequences and what happens when we try to reconstruct a sequence corresponding to a non min-cost tree or even a sequence that corresponds to no tree at all.

Our construction is guided by the requirement that it be reversible for min-cost trees, i.e., if $B = \textit{seq}(T)$ for some min-cost tree T, then $T = \textit{BuildTree}(B)$. If $B = \textit{seq}(T)$ for some non min-cost tree it will be possible that $T \neq \textit{BuildTree}(B)$; if B is not legal then $T = \textit{BuildTree}(B)$ will still exist but of course $\textit{seq}(T) \neq B$.

So, now, assume that $B = \textit{seq}(T)$ is a legal sequence for some min-cost tree T. The weight p_1 is associated with a leaf at level 0, and the left sibling of this leaf is associated with some other weight p_k. To define $\textit{BuildTree}(B)$ so that it works backwards to construct T it must determine how k can be identified.

Observe that we may assume that this left sibling is a lowest leaf in the tree which is a left-child, *i.e.*, a lowest left node in T. Such a node appears on level $\beta - \alpha$ (see tree T in Figure 6). The number of leaves below this level is $b_{\beta-\alpha-1}$. Thus, since we list items consecutively with respect to increasing levels, a lowest left-child leaf has index $k = \textit{FirstLeft}(B)$, where

$$\textit{FirstLeft}(B) = b_{\beta-\alpha-1} + 1$$

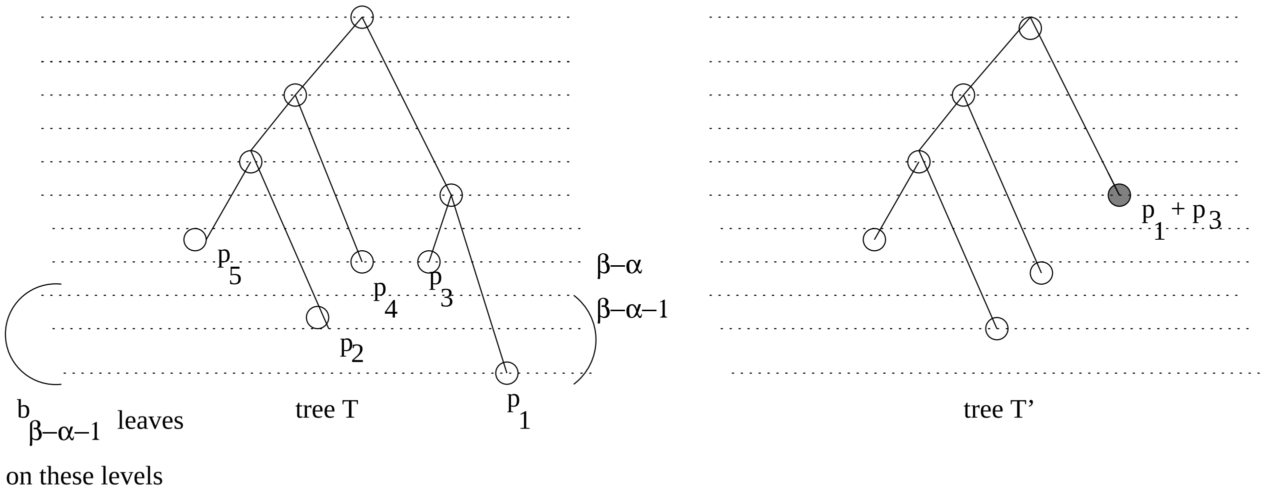

Figure 6: The correspondence between trees T, T' and their sequences: $T' = merge(T, 1, 3)$ and $seq(T) = B = (1, 2, 2, 3, 3, 4, 4, 4, 4, 4)$; $seq(T') = dec(B) = B' = (0, 1, 1, 2, 2, 3, 3, 3, 3, 3)$; $FirstLeft(B) = b_{\beta-\alpha-1} + 1 = b_{5-2-1} + 1 = 3$ and $cost(T) = cost(T') + 2p_4 + 5p_1$.

We state, without proof, the intuitive fact[1] that, if T is an optimal tree in which p_1, p_k label sibling leaves, then the tree T' that results by (i) removing those leaves and (ii) labeling their parent (now a leaf) with $p_1 + p_k$ will also be an optimal tree for the leaf set $P' = P \cup \{p_1 + p_k\} - \{p_1, p_k\}$. (See tree T' in Figure 6.) Calculation shows that

$$cost(T, P) = cost(T', P') + \beta \cdot p_1 + \alpha \cdot p_k. \tag{5}$$

If the leaves with weights p_1, p_k are siblings in a tree T then denote by $T' = merge(T, 1, k)$ the tree in which those leaves are removed and their parent is replaced by a leaf. (see Figure 6.) For the sequence $B = (b_0, b_1, \ldots b_d)$ denote

$$dec(B) = B' = (b_0 - 1, b_1 - 1, b_2 - 1, \ldots b_d - 1).$$

Note that (after any leading zeros are deleted) this sequence is the characteristic sequence of $T' = merge(T, 1, k)$.

Assume Γ is a sorted sequence of positive integers, x is a positive integer, and $insert(\Gamma, x)$ is the sorted sequence obtained by inserting x into Γ in the correct position. Now denote by $delete(P, p_1, p_k)$ the sequence P with elements p_1 and p_k deleted, and define

$$P' = package\_merge(P, 1, k) = insert(delete(P, p_1, p_k), p_1 + p_k).$$

[1] This fact is not needed for later proofs; it is only given to help provide some intuition as to why the algorithm is defined the way it is.

function *BuildTree(B): given weights P*

1. If $n = 2$ note that $P = \{p_1, p_2\}$ for some $p_1 \le p_2$.
   Return one root with two children;
   left child has weight p_2, right child has weight p_1.

   If $n > 2$ then

2. $k = FirstLeft(B)$

3. $P' = package\_merge(P, 1, k)$

4. $B' = dec(B)$

5. Delete leading zeros from B'

6. $T' = BuildTree(B')$ using weights P' (recursive step)

7. Let $u_1, u_2, \ldots, u_{n-1}$ be the leaves of T' enumerated so that
   $depth(T, u_1) \ge depth(T, u_2) \ge \cdots \ge depth(T, u_{n-1})$
   Let $p'_1 \le p'_2 \le \cdots \le p'_{n-1}$ be the weights in P'.
   Let j be an index such that $p'_j = p_1 + p_k$.

8. Replace $u_j \in T'$ by an internal node with two children.
   Call the resulting new tree, T

9. Return T.

Figure 7: Procedure *BuildTree(B)*.

For example if $P = \{2, 3, 4, 5, 10\}$ then

$$\begin{aligned} P' = delete(P, 2, 4) &= \{3, 5, 10\}, \\ insert(P', 6) &= \{3, 5, 6, 10\}, \\ package\_merge(P, 1, 3) &= \{3, 5, 6, 10\}. \end{aligned}$$

The reason for introducing this notation is that P' will be the weights that $T' = merge(T, 1, k)$ will be labelled with.

The observations above lead us to the algorithm $BuildTree(B)$ in Figure 7 which, for $B \in \mathcal{M}_n$ and P with $|P| = n$, builds a lopsided tree with n leaves.

As an example of how the algorithm works suppose that $\alpha, \beta = 2, 5$, $B = (1, 2, 2, 3, 3, 4, 4, 4, 4, 4)$

and $P = \{1,1,1,1,1\}$. Set $B_5 = B$ and $P_5 = P$. We will run *BuildTree*(B_5) for $P_5 = \{1,1,1,1,1\}$. For $i = 4,3,2$, let B_i be $dec(B_{i+1})$ with leading zeros deleted, i.e., the two smaller sequences on which *BuildTree* is recursively called, and let P_i be the P with which B_i is called. The table in Figure 8 collects the values generated by the algorithm. Note that $k = FirstLeft(B) = b_{(5-2)-1} + 1 = b_2 + 1$. The p_k column contains the value of p_k in the current P_i. Figure 9 shows the trees $T_i = BuildTree(B_i)$ (for P_i) that are generated. Note that T_5, the tree that is the final result, satisfies $seq(T_5) = B_5$, i.e.,

$$Buildtree(B_5) = B_5.$$

It is not difficult to show that T_5 is a min-cost tree for P_5. This is a special case of a general rule; if $B = seq(T)$ where T is a min-cost tree for P then *BuildTree(B)* will construct a tree whose shape, i.e., the number of nodes per level, is exactly the same as that of T. The proof of this fact is a straightforward induction on n using the definition of $FirstLeft(B)$ and the fact that if T is minimal for P then T' is minimal for P'. We do not include it here because it is not needed for the algorithm.

We also note that the algorithm is well defined for *all* $B \in \mathcal{M}_n$ and $|P| = n$: the proof is by induction. It is obviously well defined for $B \in \mathcal{M}_2$ and $|P| = 2$. If $n > 2$ then $k = FirstLeft(B) = b_{\beta-\alpha-1} + 1 \leq n$ so p_k exists and $P' = package\_merge(P, 1, k)$ is well defined so steps 1-5 are well defined. Since $B' = dec(B) \in \mathcal{M}_{n-1}$ and $|P'| = n - 1$ this means that when the algorithm recursively calls *BuildTree*(B') using P' it receives a well defined result and step 6 is well defined as well. Finally, from the definition of $P' = package\_merge(P, 1, k)$ we know that there exists some j with $p'_j = p_1 + p_k$. Thus step 8 is also well defined.

To bound the running time note that the recursion only goes to a depth of $n-1$ and each step requires at most $O(n)$ time so the entire procedure only needs $O(n^2)$ time.

As an example of the algorithm run on a legal sequence corresponding to a *non-minimal* tree we refer back to tree T' in Figure 1 which was not min-cost for $P = \{2,5,5,8\}$. Set $B_4 = seq(T')$. We will run *BuildTree*(B_4) for $P_4 = P$. For $i = 3,2$, let B_i be $dec(B_{i+1})$ with leading zeros deleted, i.e., the two smaller sequences on which *BuildTree* is recursively called, and let P_i be the P with which B_i is called. The table in Figure 10 collects the values generated by the algorithm. Figure 11 shows the trees $T_i = BuildTree(B_i)$ (for P_i) that are generated. Note that *BuildTree*(B_4) generates tree T_4 in the diagram which is not T', i.e., $BuildTree(seq(T')) \neq T'$. Referring back to Figure 1 we see that T_4 actually is the other tree, T, in that figure.

If $B \in \mathcal{M}_n$ is not legal then *BuildTree*(B) will still build some tree but it will not have any meaning for us. As an example suppose that $\alpha, \beta = 2, 5$, $B = (2,2,2,2,2,2)$ and $P = (1,1,1)$ with $n = 3$. Working through the two possible cases of trees with three leaves we see that B

i	B_i	$k = FirstLeft(B_i) = b_2 + 1$	P_i	p_k	$p_1 + p_k$
5	$(1, 2, 2, 3, 3, 4, 4, 4, 4, 4)$	3	$\{1, 1, 1, 1, 1\}$	1	2
4	$(1, 1, 2, 2, 3, 3, 3, 3, 3)$	3	$\{1, 1, 1, 2\}$	1	2
3	$(1, 1, 2, 2, 2, 2, 2)$	3	$\{1, 2, 2\}$	2	3
2	$(1, 1, 1, 1, 1)$	$--$	$\{2, 3\}$	$--$	$--$

Figure 8: The values generated by $Buildtree(B_5)$ on P_5 and its recursive calls

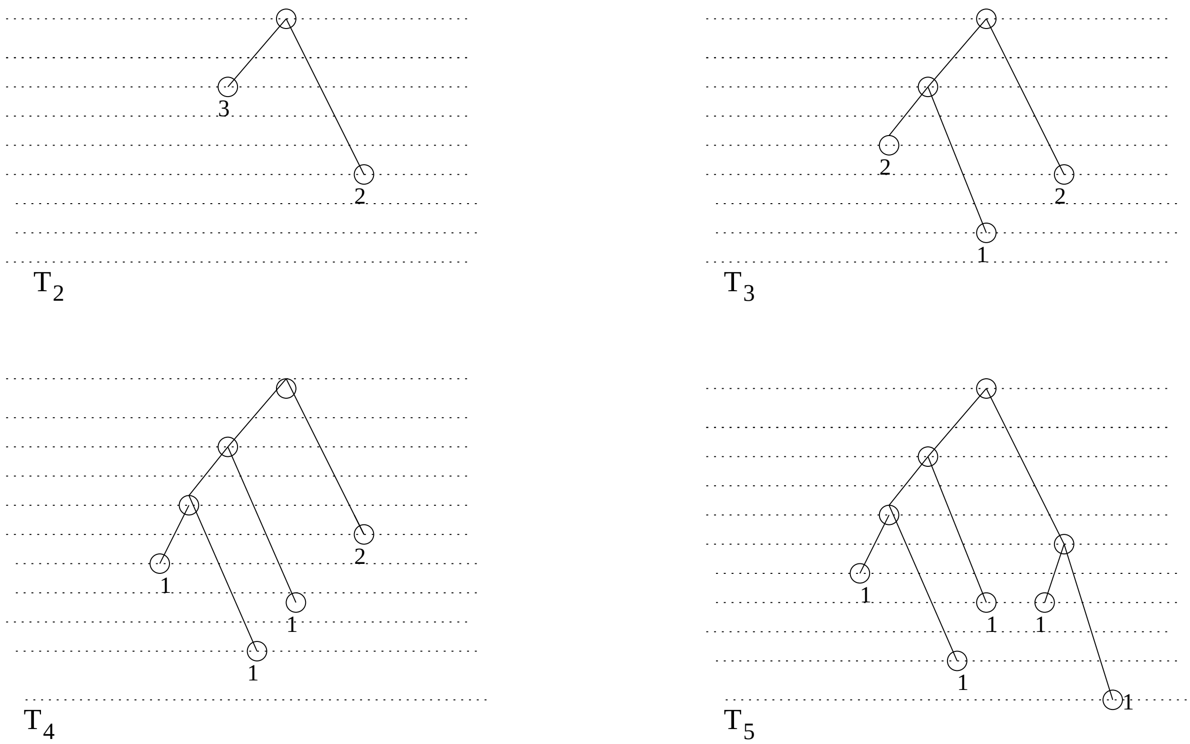

Figure 9: Trees generated by $BuildTree(B_5)$ and its recursive calls. Tree T_i is labelled by P_i.

i	B_i	$k = FirstLeft(B_i) = b_2 + 1$	P_i	p_k	$p_1 + p_k$
4	$(1,1,2,2,3,3,3,3,3)$	3	$\{2,5,5,8\}$	5	7
3	$(1,1,2,2,2,2,2)$	3	$\{5,7,8\}$	8	13
2	$(1,1,1,1,1)$	$--$	$\{7,13\}$	$--$	$--$

Figure 10: The values generated by $Buildtree(B_4)$ on P_4 and its recursive calls

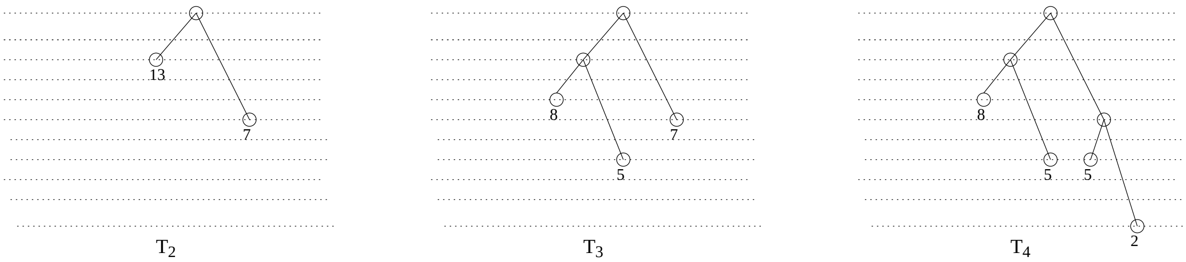

Figure 11: Trees generated by $BuildTree(B_4)$ and its recursive calls. Tree T_i is labelled by P_i.

does not correspond to either of them so B is *not* legal. Referring to Figure 12 we see that $Buildtree(2,2,2,2,2,2)$ does construct a perfectly reasonable tree.

The most important property of the operation *BuildTree* is stated by the following lemma (whose somewhat technical proof is postponed to Section 7).

Lemma 3 *For all* $B \in \mathcal{M}_n$,

$$cost(BuildTree(B), P) \leq cost(B, P).$$

As mentioned previously if $B = seq(T)$ for some min-cost tree T, then $BuildTree(B)$ will have the same shape as T so, from Lemma 2,

$$cost(BuildTree(B), P) = cost(T, P) = cost(B, P)$$

and the inequality in the lemma reduces to an equality.

The inequality in the lemma can be strict, though. For example, referring back to the construction in Figures 10 and 11 we have

$$cost(BuildTree(B_4), P_4) = cost(T_4, P_4) = 122 < 123 = cost(B_4, P_4).$$

For an example starting with a non-legal B, suppose again that $\alpha, \beta = 2, 5$, $B = (2,2,2,2,2,2)$ and $P = (1,1,1)$ with $n = 3$. Then $N_3(B) = b_3 + b_0 = 2 + 2 = 4 > n$ so $S_{N_3(B)} = \infty$ and $cost(B, P) = \infty$. On the other hand $BuildTree(B)$ is a well defined tree (see Figure 12) with

i	B_i	$k = \mathit{FirstLeft}(B_i) = b_2 + 1$	P_i	p_k	$p_1 + p_k$
3	$(2,2,2,2,2,2)$	3	$\{1,1,1\}$	1	2
2	$(1,1,1,1,1,1)$	$--$	$\{1,2\}$	$--$	$--$

Figure 12: The values and trees generated by $\mathit{Buildtree}(2,2,2,2,2,2)$ on $(1,1,1)$. Note that $(2,2,2,2,2,2)$ is not a legal sequence but $\mathit{Buildtree}$ is still well-defined on it.

$\mathit{cost}(\mathit{BuildTree}(B), P) < \infty$ (actually, $\mathit{cost}(\mathit{BuildTree}(B), P) = 16$). Thus, trivially

$$\mathit{cost}(\mathit{BuildTree}(B), P) < \mathit{cost}(B, P).$$

A direct corollary of Lemma 3 is the correctness theorem.

Theorem 1 (correctness theorem)
If $B \in \mathcal{M}_n$ is of minimal cost then $\mathit{BuildTree}(B)$ is an optimal lopsided tree.
The cost of the optimal tree equals the cost of the optimal monotonic sequence.

Proof: Let B be a min-cost sequence, $T = \mathit{seq}(B)$ and T' be a min-cost tree. By Lemma 2, Lemma 3, and the fact that $\mathit{cost}(T', P) \le \mathit{cost}(T, P)$ we have

$$\mathit{cost}(\mathit{seq}(T'), P) \le \mathit{cost}(T', P) \le \mathit{cost}(T, P) \le \mathit{cost}(B, P).$$

From the minimality of B we have that $\mathit{cost}(\mathit{seq}(T'), P) \ge \mathit{cost}(B, P)$ so $\mathit{cost}(T, P) = \mathit{cost}(B, P) = \mathit{cost}(T', P)$ and T is an optimal lopsided tree with the cost of T equaling the cost of B. $\square$

6 The shortest path computation and the Monge property

In this section we first show how to find a minimum cost monotonic sequence by performing a shortest path calculation in a special weighted graph G_n. We then show that the structure of this graph is special enough that the problem we are trying to solve has a Monge property, enabling us to use the SMAWK algorithm to get a better running time.

We will use the weighted directed graph $G_n = (V_n, E_n)$ where

$$V_n = \{(i_0, i_1, \dots, i_{\beta-1}) \,:\, 0 \le i_0 \le i_i \le \cdots \le i_{\beta-1} \le n-1\},$$

of all non-increasing β-tuples of nonnegative integers in the range $[0 \dots n-1]$. Two vertices $u, v \in V_n$ will be connected by an edge in E_n if and only they "overlap" in a $(\beta - 1)$-tuple:

Definition 3 *Let u, v, $u \neq v$ be any two vertices in V_n such that $u = (i_0, i_1, i_2, \dots, i_{\beta-1})$, $v = (i_1, i_2, \dots, i_{\beta-1}, i_\beta)$ where*

$$0 \le i_0 \le i_1 \le i_2 \le \dots i_{\beta-1} \le i_\beta \le n-1.$$

Then $(u, v) \in E_n$. E_n contains no other edges.

Furthermore, for u, v as above with $(u, v) \in E_n$ we define Weight and EdgeCost by

$$Weight(u, v) = EdgeCost(i_0, i_1, \dots, i_\beta) = S_{i_\beta + i_\alpha - i_0}$$

Note that the structure of G_n is only dependent upon n and not upon P; P is only used to define the S_i and thus the edge weights.

A β-tuple $i_0, i_1, \dots, i_{\beta-1}$ is *lexicographically smaller* than another β-tuple $j_0, j_1, \dots, j_{\beta-1}$ if $\exists k < \beta - 1$ such that (a) $(\forall t,\ t < k)$, $i_t = j_t$ and (b) $i_k < j_k$. Observe that if (u, v) is an edge in E_n, then the fact that $(\forall j,\ 0 < j \le \beta)$, $i_{j-1} \le i_j$ in $(i_0, i_1, i_2, \dots, i_{\beta-1}, i_\beta)$ guarantees that u is lexicographically smaller (as a β-tuple) than v. In other words the lexicographic ordering on the nodes is a topological ordering of the nodes of V_n; the existence of such a topological ordering implies that G_n is acyclic. Note that the β-tuple of zeros, $(0, \dots 0)$, is a source. We refer to this node as the *initial node* (or the *source*) of the graph. Note also that the β-tuple $(n-1, \dots, n-1)$ is a sink, which we call the *final node* (or the *sink*) of the graph.

As we shall now see there is a cost-preserving one-to-one correspondence between source-sink paths in G_n and monotonic sequences in $\mathcal{M}_n$.

First suppose $B = b_0, b_1, \dots, b_{d-1}$ is any monotonic sequence terminating in the β-tuple $(n-1, n-1, \dots, n-1)$. Define $u_{-1} = (0, 0, \dots, 0)$ and $\forall k,\ 0 \le k \le d-1$ set

$$u_k = (b_{k-\beta}, b_{k+1-\beta}, \dots, b_k)$$

where $b_i = 0$ when $i < 0$. Then u_{-1} is the initial node and u_{d-1} the final node, thus

$$u_{-1}\, u_0\, u_1\, u_2\, \dots\, u_{d-1}$$

is a path from the initial to the final node. This will be the path corresponding to B.

Now note that for $k \geq 0$

$$Weight(u_{k-1}, u_k) = S_{b_k + b_{k-(\beta-\alpha)} - b_{k-\beta}}.$$

Thus, the cost of the path is exactly $cost(B, P) = \sum_{0 \leq k < d} S_{b_k + b_{k-(\beta-\alpha)} - b_{k-\beta}}$. Note that if B_1 and B_2 are two different monotonic sequence starting with $(0, 0, \ldots, 0)$ and terminating in $(n-1, n-1, \ldots, n-1)$ then the paths associated with them are different. Thus, this mapping from monotonic sequences to paths is one-one.

Next suppose that

$$u_{-1}\, u_0\, u_1\, u_2\, \ldots\, u_{d-1}$$

is some path connecting the initial and final nodes. For $0 \leq k \leq d-1$ let b_k be the β^{th} element of the β-tuple u_k, *i.e.*, if $u_k = (i_0, i_1, \ldots, i_{\beta-1})$ then $b_k = i_{\beta-1}$. Then $B = b_0,\, b_1,\, \ldots,\, b_{d-1}$ is the sequence associated with the path. It is not difficult to see that B is monotonic and terminates in the final node, *i.e.*, the β-tuple $(n-1, n-1, \ldots, n-1)$ and that the path corresponding to β is just the original path we started from.

Combining the above constructions we obtain a cost preserving bijection between monotonic sequences in $\mathcal{M}_n$ and paths in G_n connecting the initial and final nodes. As an aside, note that given a path (sequence), its corresponding sequence (path) can be read off quite simply in $O(d) = O(n)$ time.

The path/sequence correspondence together with Lemma 2 implies that given a tree T and $B = seq(T)$, the cost of the path corresponding to B equals $cost(T)$.

Example.
The tree T_5 in Figure 9 has $B = seq(T) = (1, 2, 2, 3, 3, 4, 4, 4, 4, 4)$ and its corresponding path in the graph G_5 is:

$$(0,0,0,0,0) \xrightarrow{S_1} (0,0,0,0,1) \xrightarrow{S_2} (0,0,0,1,2)$$

$$\xrightarrow{S_2} (0,0,1,2,2) \xrightarrow{S_4} (0,1,2,2,3) \xrightarrow{S_5} (1,2,2,3,3) \xrightarrow{S_5} (2,2,3,3,4)$$

$$\xrightarrow{S_5} (2,3,3,4,4) \xrightarrow{S_5} (3,3,4,4,4) \xrightarrow{S_5} (3,4,4,4,4) \xrightarrow{S_5} (4,4,4,4,4)$$

where the notation $(i_0, i_1, i_2, i_3, i_4) \xrightarrow{S_{i_5+i_2-i_0}} (i_1, i_2, i_3, i_4, i_5)$ denotes an edge from $(i_0, i_1, i_2, i_3, i_4)$ to $(i_1, i_2, i_3, i_4, i_5)$ with cost $S_{i_5+i_2-i_0}$.

The cost of this path, and also of the tree T_5 is

$$S_1 + 2 \cdot S_2 + S_4 + 6 \cdot S_5.$$

The above observations can be restated as

Observation 1 *Assume T is a tree and $B = seq(T)$. Then $cost(T) = cost(B)$ equals the cost of the path in G_n corresponding to B.*

Combining this with the correctness theorem (Theorem 1) gives

Observation 2 *The cost of a shortest path from the initial node to the final node is the same as the cost of a minimum cost tree. Furthermore, given a minimum cost path, a minimum-cost tree can be reconstructed from it in $O(n^2)$ time.*

Note that G_n is acyclic and has $O(n^{\beta+1})$ edges. The standard dynamic-programming shortest path algorithm would therefore find a shortest path from the source to the sink, and hence a minimum cost tree, in $O(n^{\beta+1})$ time. This improves upon the best known algorithm for finding min-cost trees, which runs in $O(n^{\beta+2})$ time [8]. We now discuss how to improve by another factor of $\Theta(n)$ to find such a path, and thus a min-cost tree, in $O(n^{\beta})$ time.

Our algorithm cannot construct the entire graph since it is too large. Instead we use the fact that the graph has a Monge property.

A 2-dimensional $k \times r$ matrix A is defined to be a *Monge matrix* [16] if for all $1 \leq i < k$, $1 \leq j < r$,

$$A(i,j) + A(i+1, j+1) \leq A(i, j+1) + A(i+1, j) \tag{6}$$

To use this definition we need to define appropriate matrices. For any vertex u in the graph G_n, define $cost(u)$ to be the least weight (cost) of any path in G_n from the initial node to u.

Now let $\delta = (i_1, i_2, \ldots, i_{\beta-1})$ be any monotonic $(\beta-1)$-tuple of integers such that $0 \leq i_1$ and $i_{\beta-1} \leq n$. For $0 \leq i \leq i_1$ and $i_{\beta-1} \leq j \leq n-1$, define

$$EdgeCost_\delta(i,j) = EdgeCost(i, i_1, \ldots, i_{\beta-1}, j) = S_{j+i_\alpha - i}.$$

Now define the matrix $A_\delta(i,j)$ for $0 \leq i \leq i_1$ and $i_{\beta-1} \leq j \leq n-1$ by

$$A_\delta(i,j) = cost(i, i_1, \ldots, i_{\beta-1}) + EdgeCost_\delta(i,j).$$

The intuition behind these definition is that $EdgeCost_\delta(i,j)$ is the cost of the edge between vertices (i, δ) and (δ, j) in G_n and $A_\delta(i,j)$ is the cost of the path from the initial node of G_n to (δ, j) that first traverses the shortest path from the initial node to (i, δ), and then takes the edge connecting (i, δ) to (δ, j).

Theorem 2 (Monge-property theorem)
Let $\delta = (i_1, i_2, \ldots, i_{\beta-1})$ be any fixed monotonic $(\beta-1)$-tuple of integers such that $0 \leq i_1$ and $i_{\beta-1} \leq n$. Then the matrix A_δ is a two-dimensional Monge matrix.

Proof: Let $\delta = (i_1, i_2, \ldots, i_{\beta-1})$. We prove Equation (6), where $A = A_\delta$. If the right hand side of inequality 6 is infinite, we are done. Otherwise, by the definitions of A_δ and the S_k we have

$$\begin{aligned}
&A_\delta(i, j+1) + A_\delta(i+1, j) - A_\delta(i, j) - A_\delta(i+1, j+1) \\
&\quad = \mathit{EdgeCost}_\delta(i, j+1) + \mathit{EdgeCost}_\delta(i+1, j) - \mathit{EdgeCost}_\delta(i, j) - \mathit{EdgeCost}_\delta(i+1, j+1) \\
&\quad = S_{j+1+i_\alpha-i} + S_{j+i_\alpha-(i+1)} - S_{j+i_\alpha-i} - S_{j+1+i_\alpha-(i+1)} \\
&\quad = p_{j+1+i_\alpha-i} - p_{j+i_\alpha-i} \\
&\quad \geq 0
\end{aligned}$$

which completes the proof.

□

A 2×2 matrix A is defined to be *monotone* if either $A_{11} \leq A_{12}$ or $A_{21} \geq A_{22}$. An $n \times m$ matrix A is defined to be *totally monotone* if every 2×2 submatrix of A is monotone. It is known, see e.g., [4], that a two-dimensional Monge matrix is always totally monotone.

The SMAWK algorithm [3] takes as its input a function which computes the entries of an $n \times m$ totally monotone matrix A and gives as output a non-decreasing function f, where $1 \leq f(i) \leq m$ for $1 \leq i \leq n$, such that $A_{i,f(i)}$ is the minimum value of row i of A. The time complexity of the SMAWK algorithm is $O(n+m)$, provided that each computation of an A_{ij} takes constant time. Note that since every Monge matrix is totally monotone all of the matrices A_δ are totally monotone. This fact permits us to use the SMAWK algorithm to prove:

Theorem 3 (Shortest-path theorem)
For $\beta > 1$ a shortest path from a source node to the sink node in G can be constructed in $O(n^\beta)$ time.

Proof: Our approach is to calculate $cost(u)$ for *all* monotonic β-tuples u. In particular, this will calculate the cost of the shortest path to the final node, which is the cost of the optimal tree.

For fixed monotone $(\beta-1)$-tuple $\delta = (i_1, i_2, \ldots, i_{\beta-1})$, note that (i, δ) and (δ, j) are β-tuples, and thus vertices of G_n for any $i \leq i_1$. Furthermore for any $i_{\beta-1} \leq j < n$.

$$(\forall j,\ j \geq i_{\beta-1}),\ \ cost(\delta, j) = \min\{A_\delta(i, j)\ :\ i \leq i_1\}$$

Also note that $A_\delta(i, j)$ can be calculated in constant time provided the values of $cost(i, \delta)$ are known. This means that, given a fixed δ, if the values of $cost(i, \delta)$ are already known for all i, then the values of $cost(\delta, j)$ for all j can be calculated in total time $O(n)$ by the SMAWK algorithm. We call this $O(n)$ time step, *processing* δ.

Our algorithm to calculate $cost(i_0, i_1, \ldots, i_{\beta-1})$ for all β-tuples is simply to process all of the $(\beta - 1)$ tuples in lexicographic order. Processing in this order ensures that at the time we process δ the values of $cost(i, \delta)$ are already known for all i.

Using the SMAWK algorithm, each of the $O(n^{\beta-1})$ $(\beta - 1)$-tuples can be processed in linear time, so the entire algorithm uses $O(n^\beta)$ time, as stated.[2]

Note that in this proof we actually only show how to calculate the cost of the shortest path. Transforming this calculation into construction of the actual path uses standard dynamic programming backtracking techniques. We leave the details to the reader. □

Theorem 4 (main result)
A minimum cost lopsided tree can be constructed in $O(n^\beta)$ time.

Proof:
If $\beta = 1$, use the basic Huffman encoding algorithm which runs in $O(n)$ time if the list of weights is already sorted. Otherwise, apply the algorithm *Optimal_Tree_Construction* from the end of section 3.

This tells us to first find a minimum-cost source-sink path π which Theorem 3 tells us can be computed in $O(n^\beta)$ time. It then tells us to construct $B \in \mathcal{M}_n$ corresponding to π. This can be done in $O(n)$ time; the B so constructed is a minimum-cost one.

Finally it tells us to apply the algorithm $BuildTree(B)$ from Section 5. This takes $O(n^2)$ time and Theorem 1 ensures us that this tree will be a minimum-cost one. □

7 Proof of Lemma 3

The main goal of this section is to prove Lemma 3, i.e., to show that

$$cost(B, P) \leq cost(BuildTree(B), P)$$

for any monotonic sequence $B \in \mathcal{M}_n$. The proof is based upon three technical lemmas about sequences of integers.

If $\Gamma = x_1, x_2, \ldots, x_n$ is any sorted sequence of positive integers in nondecreasing order, let $Pref\_Sum_t(\Gamma) = \sum_{i=1}^{t} x_i$ denote the sum of the first t entries of Γ. The following two lemmas are straightforward:

[2] The constant implicit in the $O()$ is actually quite small since all of the work in the algorithm is done by the SMAWK algorithm and the SMAWK algorithm has a very small constant in its running time. See, e.g., [13], for a detailed discussion.

Lemma 4 (insertion-sort lemma)
If $t \leq length(\Gamma)$ and Γ is a sorted sequence then

1. $Pref\_Sum_t(insert(\Gamma, x)) \leq Pref\_Sum_t(\Gamma)$,

2. $Pref\_Sum_t(insert(\Gamma, x)) \leq Pref\_Sum_{t-1}(\Gamma) + x$.

Proof: Immediate □

Lemma 5
Recall from Section 5 that

$$package\_merge(P, 1, k) = insert(delete(P, p_1, p_k), p_1 + p_k).$$

If $j \geq k$ and $P' = package\_merge(P, 1, k)$ then

1. $Pref\_Sum_{j-2}(P') \leq Pref\_Sum_j(P) - p_1 - p_k$,

2. $Pref\_Sum_{j-1}(P') \leq Pref\_Sum_j(P)$.

Proof: Let $\Gamma = delete(P, p_1, p_k)$. Observe that for $j \geq k$ we have

$$Pref\_Sum_{j-2}(\Gamma) = Pref\_Sum_j(P) - p_1 - p_2 \tag{7}$$

To prove **(1)** apply point (1) of Lemma 4 and Equation (7) to the sequence Γ, where $P' = insert(\Gamma, x)$ with $x = p_1 + p_k$ and $t = j - 2$.

To prove **(2)** apply point (2) of Lemma 4 with $x = p_1 + p_2$. From equation (7) we have

$$Pref\_Sum_{j-1}(P') = Pref\_Sum_{j-1}(insert(\Gamma, x)) \leq Pref\_Sum_{j-2}(\Gamma) + x = Pref\_Sum_j(P).$$

This completes the proof. □

Lemma 6 (key-lemma)
Let $k = FirstLeft(B) = b_{\beta-\alpha-1} + 1$, $P' = package\_merge(P, 1, k)$ and $B' = dec(B)$. Then

$$cost(B', P') \leq cost(B, P) - \beta \cdot p_1 - \alpha \cdot p_k.$$

Proof:
Recall that $cost(B, P) = \sum_{0 \leq k < d} S_{N_k(B)}$ where $N_k(B) = b_k + b_{k-(\beta-\alpha)} - b_{k-\beta}$ and

$$S_i = \begin{cases} \sum_{j \leq i} p_j & \text{if } 1 \leq i \leq n \\ \infty & \text{otherwise.} \end{cases}$$

Observe that

$$N_i(B') = \begin{cases} N_i(B) - 1 & \text{if } i < \beta - \alpha \\ N_i(B) - 2 & \text{if } \beta - \alpha \le i < \beta \\ N_i(B) - 1 & \text{if } \beta \le i < d \end{cases}$$

In what follows we assume that $\forall i,\ N_i(B) \le n$ since otherwise $S_{N_i(B)} = \infty$, $cost(B,P) = \infty$ and the lemma is trivially true. Note that $N_i(B) \le n$ will also imply that $N_i(B') \le n - 1$.

Now denote the i^{th} term of the cost of B as

$$\mathit{term}(i,B) = S_{N_i(B)} = \mathit{Pref\_Sum}_{N_i(B)}(P)$$

and the i^{th} term of the cost of B' as

$$\mathit{term}(i,B') = S_{N_i(B')} = \mathit{Pref\_Sum}_{N_i(B')}(P')$$

We now proceed with a case by case analysis.

CASE 1: $i < \beta - \alpha$.

In this case $\mathit{term}(i,B) - \mathit{term}(i,B') = p_1$. Summing over all i yields

$$\sum_{0 \le i < \beta - \alpha} \mathit{term}(i,B') = \sum_{0 \le i < \beta - \alpha} \mathit{term}(i,B) - (\beta - \alpha)p_1. \tag{8}$$

CASE 2: $\beta - \alpha \le i < \beta$

In this case $\mathit{term}(i,B) = \mathit{Pref\_Sum}_j(P)$ and $\mathit{term}(i,B') = \mathit{Pref\_Sum}_{j-2}(P')$ for some $j \ge k = \mathit{FirstLeft}(B)$, and, by Lemma 5, the difference between these values is at least $p_1 + p_k$. Hence

$$\sum_{\beta - \alpha \le i < \beta} \mathit{term}(i,B') \le \sum_{\beta - \alpha \le i < \beta} \mathit{term}(i,B) - \alpha(p_1 + p_k). \tag{9}$$

CASE 3: $\beta \le i$

In this case $\mathit{term}(i,B) = \mathit{Pref\_Sum}_j(P)$ and $\mathit{term}(i,B') = \mathit{Pref\_Sum}_{j-1}(P')$ for some $j \ge k = \mathit{FirstLeft}(B)$. By Lemma 5, $\mathit{term}(i,B') \le \mathit{term}(i,B)$. Hence

$$\sum_{\beta \le i} \mathit{term}(i,B') \le \sum_{\beta \le i} \mathit{term}(i,B). \tag{10}$$

Combining (8), (9) and (10) we obtain the result. □

We can now prove Lemma 3, i.e.

$$\forall B \in \mathcal{M}_n,\ \ cost(\mathit{BuildTree}(B),P) \le cost(B,P). \tag{11}$$

The proof will be by induction on n. If $n = 2$ then $B = (1, 1, \ldots, 1) \in \mathcal{M}_2$ is a d-tuple with $d \geq \beta$ and $P = \{p_1, p_2\}$ for some $p_1 \leq p_2$. By definition, $S_1 = p_1$ and $S_2 = p_1 + p_2$.

Working through the calculations we find that

$$N_k(B) = \begin{cases} b_k = 1 & \text{if } 0 \leq k < \beta - \alpha \\ b_k + b_{k-(\beta-\alpha)} = 2 & \text{if } \beta - \alpha \leq k < \beta \\ b_k + b_{k-(\beta-\alpha)} - \beta_{k-\beta} = 1 & \text{if } \beta \leq k < d \end{cases}$$

so

$$\begin{aligned} cost(B, P) &= \sum_{0 \leq k < d} S_{N_k(B)} \\ &= (\beta - \alpha)p_1 + \alpha(p_1 + p_2) + (d - \beta)p_1 \\ &= \alpha p_2 + dp_1. \end{aligned}$$

Recall that for $n = 2$, $T = BuildTree(B)$ is a root with two children. Therefore

$$cost(BuildTree(B), P) = \alpha p_2 + \beta p_1.$$

Thus

$$cost(BuildTree(B, P)) = \alpha p_2 + \beta p_1 \leq \alpha p_2 + dp_1 = cost(B, P)$$

and (11) holds for $n = 2$.

So now suppose that (11) holds for $n - 1$; we will prove that it also holds for n.

Let $B \in \mathcal{M}_n$, $|P| = n$. Set $T = BuildTree(B)$, $k = FirstLeft(B)$, $\overline{B} = dec(B)$ and $P' = package\_merge(P, 1, k)$. Let B' be $\overline{B}$ with all leading zeros (if any exist) deleted and set $T' = BuildTree(B')$ (for P').

From the induction hypothesis we know that

$$cost(BuildTree(B'), P') \leq cost(B', P') \tag{12}$$

and from Lemma 6 we have that

$$cost(\overline{B}, P') \leq cost(B, P) - \beta \cdot p_1 - \alpha \cdot p_k.$$

Leading zeros contribute nothing to the cost of a monotonic sequence, though, so $cost(\overline{B}, P) = cost(B', P)$ implying

$$cost(B', P') \leq cost(B, P) - \beta \cdot p_1 - \alpha \cdot p_k. \tag{13}$$

Let $u_1, u_2, \ldots, u_{n-1}$ be the leaves of T' enumerated so that

$$depth(T, u_1) \geq depth(T, u_2) \geq \cdots \geq depth(T, u_{n-1}).$$

Let $p'_1 \le p'_2 \le \cdots \le p'_{n-1}$ be the weights in P'. By definition $cost(T', P') = \sum_i p'_i \cdot depth(T, u_i)$.

Let j be an index such that $p'_j = p_1 + p_k$. Recall that *BuildTree*(B) constructs T by starting with T', taking leaf u_j and replacing it with an internal node with two children, both of which are leaves. Let v_L be the left child of u_j and v_R be the right one. Then the leaves of T are

$$\{u_1, u_2, \ldots, u_{n-1}, v_L, v_R\} - \{u_j\}.$$

Label these leaves with the weights in P as follows: for $i \ne j$ label u_i with p_i; label v_L with p_k and v_R with p_1. Then the external path length of T *associated with this labelling* is

$$\begin{aligned}
\sum_{i \ne j} p'_i \cdot depth(T, u_i) \quad &+ \quad p_1 \cdot depth(T, v_R) + p_k \cdot depth(T, v_L) \\
&= \quad \sum_{i \ne j} p'_i \cdot depth(T, u_i) + p_1 \cdot (depth(T', u_j) + \beta) + p_k \cdot (depth(T', u_j) + \alpha) \\
&= \quad \sum_{i} p'_i \cdot depth(T, u_i) + p_1\beta + p_k\alpha \\
&= \quad cost(T', P') + p_1\beta + p_1 k\alpha.
\end{aligned}$$

The last thing to notice is that, as discussed at the end of Section 1, $cost(T, P)$ is the minimum external path length of T under *all possible permutations of the assignments of the weights in P to the leaves of T*. Thus $cost(T, P)$ is upperbounded by the external path length of T associated with the given labelling and

$$cost(T, P) \le p_1\beta + p_1\alpha + cost(T', P'). \tag{14}$$

Combining (12), (13) and (14) gives

$$\begin{aligned}
cost(T, P) \quad &\le \quad p_1\beta + p_1\alpha + cost(T', P') \\
&\le \quad p_1\beta + p_1\alpha + cost(B', P') \\
&\le \quad cost(B, P)
\end{aligned}$$

and we have shown that (11) is valid for n and thus completed the proof of Lemma 3 .

8 Final remarks

In this paper we revisited the problem of finding optimal prefix-free codes for unequal integral letter costs α, β with $\alpha \le \beta$. The best previous known algorithm ran in $O(n^{\beta+2})$ time; the algorithm presented here runs in $O(n^{\beta})$. The reduction in running time was achieved in two ways. The first was by noting that it is possible to transform the problem into one of searching for optimal monotonic sequences (a slightly easier task) and then reconstructing optimal trees, and thus codes, from an optimal monotonic sequence. The second was by showing that the monotonic sequence problem possesses a Monge property, permitting the use of the SMAWK algorithm.

Our construction does not work for the case of constant weights and outdegree $k > 2$ (i.e. the case of general k-ary code). The reason for this is that our representation of binary trees as monotonic sequences does not seem to to extend well to the k-ary tree case. (There are some possible extensions but none that we have tried have good combinatorial properties that lead to efficient algorithms).

The big open question still remaining for this problem is exhibiting whether or not it is NP-hard, when the weights of the edges are not constant and the outdegree k is a part of the input.

Another interesting set of questions follows from the observation that our algorithm, like that of [8], solves the min-cost tree problem by finding the min-cost path in some graph. In both algorithms the graph *structure* only depends upon the costs α, β, and n. It does *not* depend upon the weights of the leaves; they only enter into the algorithm as defining the *weights* of the edges. The question is whether we can we use this graph structure to develop new algorithms for various special min-cost tree problems.

Suppose, for example, that we know the min-cost tree for a set of weights $P = (p_1, p_2, \ldots, p_n)$ and then change p_n to p'_n. This will change a restricted number of edge weights in the graph. To find the min-cost tree for $P' = (p_1, p_2, \ldots, p_{n-1}, p'_n)$ we would need to find a min-cost path for the same graph structure but with the new edge-weights. Is it possible to use the fact that we know both (i) the old min-cost path and and (ii) how the edge-weights change, to develop an efficient algorithm for constructing the new min-cost path and thus the new min-cost tree? As another example, let us return to the Varn problem, i.e., all of the $p_1 = 1$. In this case the edge weights have a very simple structure. Would it be possible to use this structure to rederive the greedy Varn algorithm directly from our graph representation? In essence, the question we are raising here is whether the "min-cost path in a graph" formulation of prefix-free coding could lead to a better understanding of the structure of such codes in special cases and better algorithms for those cases.

We conclude by pointing out, without proof, that the algorithm *Optimal_Tree_Construction* can be straightforwardly extended to the problem of finding an optimal height-limited lopsided tree. A height-limited tree is one without nodes of depth greater than L, L a given parameter. The optimal *height-limited tree problem* is to find a min-cost tree with n leaves for given weights P with tree height limited by L. This is equivalent to finding optimal (L) length-limited Huffman Codes. In [11] it was shown that these two problems can be solved in $O(nL)$

time.

The optimal *height-limited lopsided tree problem* is similar. It is again to find a min-cost tree with n leaves for given weights P with tree height limited by L. The only difference here is that the edges have unequal integral lengths α, β with $\alpha \leq \beta$.

We can prove the following result:

Theorem 5 (height limited trees)
We can construct a minimum cost lopsided tree, with height limited by L, in $O(n^\beta \cdot L)$ time.

The idea is to show that a minimum cost lopsided tree, with height limited by L, will correspond to a sequence $B \in \mathcal{M}_n$ which is minimum-cost among all sequences with length $\leq L$. Such a sequence can in turn be found by finding the least expensive source-sink path in G_n that has link length, i.e., number of edges, $\leq L$. Using the Monge property such a path and thus a min-cost height-L limited lopsided tree, can be found in $O(n^\beta \cdot L)$ time. Because no new ideas are needed we only state the result and do not provide further details.

Acknowledgments: The authors would like to thank the anonymous referees for their comments and suggestions.

References

[1] Julia Abrahams, "Code and Parse Trees for Lossless Source Encoding," *Sequences'97,* (1997).

[2] Doris Altenkamp and Kurt Mehlhorn, "Codes: Unequal Probabilities, Unequal Letter Costs," *J. Assoc. Comput. Machinery,* **27** (3) (July 1980), 412–427.

[3] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica,* **2** (1987), 195–208.

[4] R.E. Burkard, B. Klinz and R. Rudolf, "Perspectives of Monge properties in optimization," *Discrete Applied Mathematics,* **70**(2) (24 Sept. 1996), 95-161.

[5] Siu-Ngan Choi and M. Golin, "Lopsided trees: Algorithms, Analyses and Applications," *Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming* (*ICALP 96*), (1996), 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), 460–463.

[7] E. N. Gilbert, "Coding with Digits of Unequal Costs," *IEEE Trans. Inform. Theory,* **41** (1995), 596–600.

[8] M. Golin and G. Rote, "A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes for Unequal Letter Costs," *IEEE Transactions on Information Theory,* **44**(5) (September 1998), 1770-1781.

[9] Sanjiv Kapoor and Edward Reingold, "Optimum Lopsided Binary Trees," *Journal of the Association for Computing Machinery* **36** (3) (July 1989), 573–590.

[10] R. M. Karp, "Minimum-Redundancy Coding for the Discrete Noiseless Channel," *IRE Transactions on Information Theory,* **7** (1961), 27-39.

[11] L. L. Larmore and D. S. Hirschberg, "A Fast Algorithm for Optimal Length-Limited Huffman Codes," *Journal of the Association for Computing Machinery,* **37**(3) (July 1990) 464-473.

[12] L. L. Larmore, T. Przytycka, W. Rytter, "Parallel computation of optimal alphabetic trees," *SPAA93.*

[13] L. L. Larmore and B. Schieber, "On-Line Dynamic Programming with Applications to the Prediction of RNA Secondary Structure," *Journal of Algorithms,* **12**(3) (Sept 1991), 490-515.

[14] 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.

[15] K. Mehlhorn, "An Efficient Algorithm for Constructing Optimal Prefix Codes," *IEEE Transactions on Information Theory,* **IT-26** (1980) 513-517.

[16] G. Monge, *Déblai et remblai*, Mémoires de l' Académie des Sciences, Paris, (1781) 666-704.

[17] 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.

[18] Serap A. Savari, "Some Notes on Varn Coding," *IEEE Transactions on Information Theory*, **40** (1) (Jan. 1994), 181–186.

[19] Robert Sedgewick, *Algorithms,* Addison-Wesley, Reading, Mass.. (1984).

[20] D.F. Varn "Optimal variable length codes (arbitrary symbol cost and equal code word probability)" Information and Control **19** (1971) 289-301.