Huffman Coding with Unequal Letter Costs

Mordecai J. Golin * Claire Kenyon † Neal E. Young ‡

November 12, 2001

Abstract

In the standard Huffman coding problem, one is given a set of words and for each word a positive frequency. The goal is to encode each word w as a codeword $c(w)$ over a given alphabet. The encoding must be prefix free (no codeword is a prefix of any other) and should minimize the weighted average codeword size $\sum_w \mathrm{freq}(w)\,|c(w)|$. The problem has a well-known polynomial-time algorithm [15].

Here we consider the generalization in which the letters of the encoding alphabet may have non-uniform lengths. The goal is to minimize the weighted average codeword length $\sum_w \mathrm{freq}(w)\,\mathrm{cost}(c(w))$, where $\mathrm{cost}(s)$ is the sum of the (possibly non-uniform) lengths of the letters in s. Despite much previous work, the problem is not known to be NP-hard, nor was it previously known to have a polynomial-time approximation algorithm. Here we describe a polynomial-time approximation scheme (PTAS) for the problem.

1 Introduction

Given a set of W of n words with associated frequencies $p_1 \geq p_2 \geq \cdots \geq p_n > 0$ and an *encoding alphabet* Σ, the *prefix coding* problem, sometimes known as the *Huffman encoding* problem is to find a prefix-free code over Σ of minimum cost. This problem is very well studied and has a well-known $O(n \log n)$-time greedy algorithm due to Huffman [15]. Here we consider the generalization of the problem in which the letters used for encoding can have different costs. That is, letting $r = |\Sigma|$, the r letters have associated costs $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_r$ and the cost of a codeword is defined to be the sum of the costs of its letters (rather than the length of the codeword).

This generalization is motivated by coding problems in which different characters have different transmission times or storage costs [5, 22, 19, 28, 29]. One example is the telegraph channel [10, 11] in which $\Sigma = \{\cdot, -\}$ and $\ell_2 = 2\ell_1$, i.e., in which dots are twice as long as dashes. Another is the (a, b) run-length-limited codes used in magnetic and optical storage [16, 12], in which the codewords are binary and constrained so that each $\mathbf{1}$ must be preceded by at least a, and at most b, $\mathbf{0}$'s. (This example can be modeled by the problem studied here by using an encoding alphabet of $r = b - a + 1$ characters $\{0^k 1 : k = a, a+1, \ldots, b\}$ with associated costs $\{\ell_i = a + i - 1\}$.)

The literature contains many algorithms for the generalized problem. The special case when all the *probabilities* are equal (but not the letter lengths), known as the Varn coding problem, is solvable in polynomial-time [29, 1, 7, 25, 13, 6]. For the generalized problem, Blachman [5], Marcus [22], and (much later) Gilbert [11] give heuristic constructions. Karp gave the first algorithm

*Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong. Partially supported by HK RGC Competitive Research Grant HKUST 6137/98E. Email: golin@cs.ust.hk

†Laboratoire de Recherche en Informatique (LRI), Université Paris-Sud, France. Email: kenyon@lri.fr

‡Akamai Technologies, Cambridge, MA, USA. Email: neal@acm.org.

yielding an exact solution (assuming the letter costs are integers); Karp's algorithm transforms the problem into an integer program and does not run in polynomial time [19]. Karp's result was followed by many [21, 9, 8, 23, 3] presenting solutions of cost at most OPT + $f(\ell_1, \ell_2, \ldots, \ell_r)$ where OPT is the cost of the optimal code and $f(\ell_1, \ell_2, \ldots, \ell_r)$ is some fixed function of the edge costs (with the different algorithms having different $f(\cdot)$). Golin and Rote [12] gave a dynamic programming algorithm that produces exact solutions in $O(n^{\ell_r+2})$ time for the special case when the ℓ_i are restricted to be integers. Bradford et. al. [24] improved this to $O(n^{\ell_r})$ when $r = 2$.

For further references on Huffman coding with equal letter costs, see Abrahams' recent survey on source coding [2, Section 2.7], which contains a section on the problem.

Despite the extensive literature, there is no known polynomial-time algorithm for the generalized problem, nor is the problem known to be NP-hard. Before this work, the problem was not known to have any polynomial-time approximation algorithm. Our main result here is a polynomial-time approximation scheme (PTAS) for it:

Theorem 1 *Given an instance* $((p_i), (\ell_j))$ *of the Huffman coding problem with unequal letter costs, and given a positive* ϵ*, there exists an algorithm which constructs a prefix code of cost at most* $(1+\varepsilon)$OPT*; this algorithm runs in time* $nd\log(n)\exp(O(\ln(1/\epsilon)^2/\epsilon^3))$*, where* d *is the number of distinct letter costs.*

The algorithm is based on a new relaxation of Huffman coding with unequal letter costs called the *k-prefix code problem.* The relaxation allows codewords of cost more than k to be prefixes of other codewords. The algorithm uses straightforward grouping and enumeration techniques to find a near-minimum-cost k-prefix code (where k is a constant depending only on ϵ), and then converts this k-prefix code into a true prefix code using a "rounding scheme" that increases the cost by at most a $1 + O(\epsilon)$ factor.

The techniques introduced in this paper can also be used to construct PTAS's for the *regular-language prefix-coding* problem, a different generalization of Huffman coding that asks for a minimum-cost prefix code under the additional restriction that all codewords belong to a given regular language $\mathcal{L}$. For example, the binary codes, constrained so all codewords must end in a **1**, are used for group testing and the construction of self-synchronizing codes [4, 26]. Binary codes whose codewords contain at most a specified number of **1**'s are used for energy minimization of transmissions in mobile environments [27]. Algorithms (other than exhaustive search) for the regular-language prefix-coding problem generalize [12] and run in time $n^{\Theta(S(\mathcal{L}))}$ where $S(\mathcal{L})$ is the number of states in the smallest deterministic finite automaton that accepts $\mathcal{L}$. In this extended abstract we do not discuss how the techniques here extend to that problem.

Alphabetic coding is like the generalized problem considered here, but with an additional constraint on the code: the codewords must be chosen in increasing alphabetic order (with respect to the words to be encoded). This problem arises in designing testing procedures in which the time required by a test depends upon the outcome of the test [20, 6.2.2, ex. 33] and has also been studied under the names *Dichotomous search* [14] or the *leaky shower* problem [18]. Alphabetic coding has a polynomial-time algorithm [17].

2 Notations and definitions

A problem instance is specified by a set W of n words with associated frequencies $p_1 \geq p_2 \geq \cdots \geq p_n > 0$, an alphabet Σ of $r \geq 2$ letters with associated costs $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_r$, and an $\epsilon > 0$. The *Huffman coding problem with unequal letter costs* is the problem of finding a prefix code of minimum cost.

Definitions 1 *A* **code** *(usually denoted c) is an injective map from W to Σ^*, whose image $c(W)$ is called the set of* **codewords of** *c. A set $S \subset \Sigma^*$ is* **prefix-free** *if no element of S is a prefix of any other element of S; a* **prefix code** *is one whose set of codewords is prefix-free. The* **cost** *of a code c is $\sum_{i=1}^{n} p_i \operatorname{cost}(c(w_i))$, where $\operatorname{cost}(x)$ is the sum of the costs of the letters of x.*

Unless otherwise stated, all the codes considered here are **ordered**, i.e. the map c assigns codewords of smaller costs to words of larger probability: $\operatorname{cost}(c(w_1)) \leq \operatorname{cost}(c(w_2)) \leq \cdots \leq \operatorname{cost}(c(w_n))$. Clearly, any optimal code must be ordered.

Definition 2 *A* **k-prefix code** *is a code in which no codeword of cost less than k is a prefix of any other codeword.*

Note that a k-prefix code may be non-uniquely decipherable, and that the problem of designing a k-prefix code is used here solely as an intermediate tool for solving the original problem.

We generally use w to denote a word to be encoded, x to denote a potential codeword (a string over Σ^*), $\operatorname{cost}(x)$ to denote the sum of the costs of the letters in x's, and $|x|$ (the **size** of x) to denote the number of letters. To distinguish the given words W from the potential codewords Σ^*, we call the former **words** and the latter latter **strings**. We use d to denote the number of distinct letter costs.

In the remainder of the paper we assume the following without loss of generality: (i) either ϵ is an integer multiple of ℓ_1 or vice versa (this can be guaranteed by decreasing ϵ by at most a factor of 2), (ii) $1 \leq \ell_2 \leq 1+\epsilon$ (this can be guaranteed by scaling all the letter costs), and (iii) for $i > 1$, ℓ_i is a multiple of ϵ (this can be guaranteed by rounding up the letter costs, while increasing OPT by at most a $1+\epsilon$ factor).

In the main sections (particularly Lemmas 7 and 8) we also assume that $\ell_1 > \epsilon/n$. The special case $\ell_1 \leq \epsilon/n$ is easy and is dealt with in Section 7.

Before we explain the main algorithm (Algorithm 3), we first explain two subroutines used in that algorithm: Algorithm 1 for constructing a so-called *leveled* k-prefix code meeting certain constraints, and Algorithm 2 for converting such a code into a true prefix code.

3 How to find an optimal k-prefix code

Here we assume $\ell_1 \geq \epsilon/n$ and that ϵ is an integer multiple of ℓ_1 or vice versa. We explain Algorithm 1, which finds an optimal k-prefix-free code meeting some given constraints. The first constraint is that the code should be *leveled*:

Definitions 3 *For $i \in \{1, \ldots, (k-1)/\epsilon\}$, define the ith* **level of** *Σ^* to be the set of strings x such that $\ell_2 + (i-1)\epsilon \leq \operatorname{cost}(x) < \ell_2 + i\epsilon$. Define level 0 to be the set of strings that cost less than ℓ_2.*

A code is **maximal within level** *i if every codeword in level i is of cost $\ell_2 + i\epsilon - \min\{\ell_1, \epsilon\}$.*

A code is **leveled** *if it is maximal within all levels.*

Note that level 0 can only contain words of a^*, where a is the letter of cost ℓ_1. Note also that by assumption either (i) ℓ_1 is an integer multiple of ϵ, in which case each level $i > 0$ consists of the strings of cost $1 + (i-1)\epsilon$ and every code is trivially maximal, of (ii) ℓ_1 evenly divides ϵ, in which case every string in level i has cost $1 + i\epsilon + j\ell_1$ for some integer $j \leq n$.

The second constraint that the constructed codeword must meet is specified by a tuple $f = (f(0), \cdots, f((k-1)/\epsilon))$ of integers, with the following meaning.

Algorithm 1 – builds a leveled k-prefix-free set of codewords given the level 0 codeword and the number of codewords per level.

INPUT: Letter costs, directed graph D, constraints $(f(0), f(1), \ldots, f((k-1)/\epsilon))$.

OUTPUT: leveled k-prefix code with $f(i)$ codewords in each level $i \geq 1$ and a codeword in level zero of size $f(0)$ (if $f(0) > 0$). If no such code exists, returns "inconsistent".

1: $S \leftarrow \emptyset$
2: For any node ℓ of D, define $v_S(\ell)$ to be the number of strings of cost ℓ having no prefix in S. (The algorithm will compute some of these values as it proceeds.)
3: If $f(0) > 0$, then $S \leftarrow S \cup \{a^{f(0)}\}$, where a is the smallest letter. For each node ℓ on level 0, initialize $v_\ell = v_S(\ell)$ accordingly: $v_\ell = 1$ for $\ell < f(0)$ or if $f(0) = 0$; $v_\ell = 0$ otherwise.
4: **for** $i = 1, 2, \ldots, (k-1)/\epsilon$ **do**
5: Consider the nodes of D in level i, by order of increasing costs. For each node ℓ, initialize v_ℓ using the recurrence $v_\ell = \sum_{i=1}^{r} v_{\ell-\ell_i}$.
6: Let $\ell = \ell_2 + i\epsilon - \min\{\ell_1, \epsilon\}$. If $v_\ell < f(i)$, then return "inconsistent". Otherwise, choose $f(i)$ codewords of cost ℓ and add them to S. Decrement v_ℓ by $f(i)$.
7: Complete S by adding $n - |S|$ strings of minimum cost among the strings of cost $\geq k$ that don't have a prefix of cost $< k$ in S. If $|S| < n$ and there aren't any such strings, return "inconsistent". (Find the $n - |S|$ strings by extending D as needed beyond cost k.)
8: Return the set of codewords S.

1. If $f(0) = 0$, then there must be no codeword in level 0; if $f(0) > 0$, then the set of codewords must contain the codeword $a^{f(0)}$ from level 0.

2. For every level $i > 1$, the set of codewords must contain exactly $f(i)$ codewords in level i.

Algorithm 1 uses a directed graph D which is computed by Algorithm 3 and given to Algorithm 1 as input. The nodes of D are all possible codeword costs in $[0, k]$, with an arc from ℓ to ℓ' if and only if $\ell' - \ell \in \{\ell_1, \ell_2, \cdots, \ell_r\}$.

Here is the analysis of Algorithm 1.

Lemma 1 *Given the constraint tuple $f = (f(0), f(1), \ldots, f((k-1)/\epsilon))$, if there exists a leveled k-prefix code consistent with the constraint, then Algorithm 1 constructs one of minimum cost.*

Given the digraph D, the algorithm can be implemented to run in time $O(ndk/\epsilon)$.

Proof. It is fairly easy to verify Algorithm 1 correctly computes each v_ℓ and produces a leveled k-prefix code consistent with the constraint. (Note that each v_ℓ is determined by the constraints of being leveled and being consistent with f.)

Among the codes meeting the constraints of being leveled and consistent with f, the chosen code has minimum cost because the cost of codewords on levels $0, 1, \ldots, (k-1)/\epsilon$ is determined by those constraints, and, given the codewords in those levels, the chosen code takes a set of codewords of cost $\geq k$ of minimum possible cost.

The running time follows by careful inspection. ◇

4 How to convert a k-prefix code into a prefix code

Algorithm 2 converts a k-prefix code into a prefix code. The next lemma captures what we need to know about Algorithm 2 in order to use it in the main algorithm (Algorithm 3):

Algorithm 2 — converts a k-prefix code into a fully prefix code

INPUT: letter costs; k-prefix code c
OUTPUT: fully prefix code c'

1: Let a denote the letter of cost ℓ_1 and b denote the letter of cost ℓ_2.
2: For positive integer i, define $\mathrm{enc}(i) = b'_1 b'_1 b'_2 b'_2 b'_j b'_j ab$ where $b'_1 b'_2 \ldots b'_j$ is obtained from the binary representation $b_1 b_2 \ldots b_j$ of i by replacing each "0" with "a" and each "1" with b. Define $\mathrm{enc}(0) = ab$.
3: **for** each codeword of cost $\geq k$ **do**
4: Let α be the smallest prefix of cost $\geq k$. Let β be the remaining suffix. Replace the codeword by the new codeword $\alpha\,\mathrm{enc}(i)\beta\, b$, where i is the number of b's in β.
5: Return the modified code.

Lemma 2 *Given any k-prefix-free code c of cost α, Algorithm 2 constructs a prefix-free code c' of cost at most $\alpha[1 + \ell_2(5 + 2\log_2 k)/k]$.*

Proof. First we analyze the cost. Let $c(w) = \alpha\beta$ and $c'(w) = \alpha\,\mathrm{enc}(i)\beta\, b$, respectively, be an original and modified codeword in Algorithm 2. From $i \leq \mathrm{cost}(\beta)$ it follows that $\mathrm{cost}(c'(w)) \leq \mathrm{cost}(c(w)) + \ell_2[5 + 2\log_2 \mathrm{cost}(c(w))]$. Since $\mathrm{cost}(c(w)) \geq k$ if the codeword is modified, each modified codeword costs at most $1 + \ell_2(5 + 2\log_2 k)/k$ times as much as the original.

Next we show that c' is prefix free. Suppose $c'(v)$ is a prefix of $c'(w)$ for some $v, w \in W$. Since the original code was k-prefix free, it must be that

$$\begin{aligned} c'(v) &= \alpha\,\mathrm{enc}(i)\beta b \\ \text{and} \quad c'(w) &= \gamma\,\mathrm{enc}(j)\delta b \end{aligned}$$

where α and γ each have cost $\geq k$ but have no proper prefix of cost $\geq k$, and where i and j are the number of b's in β and δ, respectively (as in Algorithm 2). Since $c'(v)$ is a prefix of $c'(w)$, α is a prefix of $c'(w)$, which means $\alpha = \gamma$. Thus, $\mathrm{enc}(i)$ is a prefix of $\mathrm{enc}(j)\delta b$. Since every letter in $\mathrm{enc}()$ is doubled except the last two, it must be that $i = j$. Thus, βb is a prefix of δb. But (since $i = j$) β has the same number of b's as δ, so it must be that $\beta = \delta$. Finally, we can conclude that $\alpha\beta = \gamma\delta$. Since these were the original codewords assigned to v and w, it must be that $v = w$.

$\diamond$

5 The main algorithm

The main algorithm is Algorithm 3. It consists of:

1. Some preprocessing (Steps 1, 2, 3, and 4).

2. Using Algorithm 1 a constant number of times as a subroutine, for $O_\epsilon(1)$ different choices of $(f(0), \ldots, f((k-1)/\epsilon))$ (Step 6).

3. Choosing the best k-prefix code among those output by Algorithm 2, and transforming it into a fully prefix-free code (Step 9).

We now analyze the cost of the code produced by this algorithm.

Lemma 3 *To find a $1+O(\epsilon)$-optimal solution to the original problem, it suffices to find a $1+O(\epsilon)$-optimal solution to the problem as modified by Step 1.*

Algorithm 3 — finds a $(1+O(\epsilon))$-optimal prefix-free code

INPUT: Word frequencies $p_1 \geq p_2 \geq \cdots \geq p_n > 0$; letter costs $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_r$; $\varepsilon > 0$.

OUTPUT: prefix-free code of cost at most $(1+O(\epsilon))$OPT

1: Divide each ℓ_i by ℓ_2 so $\ell_2 = 1$. Lower ϵ by at most a factor of 2 so that either it is an integer multiple of ℓ_1 or it evenly divides ℓ_1. For each $i > 1$, round ℓ_i up to the next multiple of ϵ.

2: Choose $k = \Theta(\log(1/\epsilon)/\epsilon)$ so that $k-1$ is a multiple of ϵ.

3: Construct a directed graph D whose nodes are all possible costs of codewords cost in $[0, k]$, with an arc from ℓ to ℓ' iff $\ell' - \ell \in \{\ell_1, \ell_2, \cdots, \ell_r\}$. Do this by enumerating the vertices and edges of D using breadth-first search from the root (node 0).

4: Greedily partition W into groups.

- The first group is $G_1 = \{w_1\}$.
- Take i maximum such that $p_i > (1-p_1)\epsilon^2/(2k)$. Then $G_2 = \{w_2\}$, $G_3 = \{w_3\}$, ..., and $G_i = \{w_i\}$.
- While W is not empty, greedily take for the next group $\{w_j, w_{j+1}, \ldots, w_\ell\}$, where $p_j + \cdots + p_\ell \leq (1-p_1)\epsilon^2/k < p_j + \cdots + p_{\ell+1}$.

5: By exhaustive search, guess whether there is a codeword of cost < 1 and what its size is (call it $f(0)$), and, for each $i = 1, 2, \ldots, (k-1)/\epsilon$, guess which groups will be assigned to level i (i.e., with costs in $[1+\varepsilon(i-1), 1+\varepsilon i)$) and let $f(i)$ denote the total number of words in those groups.

6: **for** each guess $(f(0), \ldots, f((k-1)/\epsilon))$ **do**

7: Use Algorithm 1 to construct an optimal k-prefix-free code consistent with the guess (if one exists).

8: From all the codes constructed, choose the code c with smallest cost.

9: Use Algorithm 2 to convert c into a fully prefix-free code c'.

10: Return c'.

Proof. Standard scaling and rounding arguments. ◇

Lemma 4 *Step 4 partitions W into at most $O(k/\epsilon^2)$ groups.*

Proof. Take any two consecutive groups other than G_1. The cumulative probability of the words in the two groups is at least $(1-p_1)\epsilon^2/(2k)$. Thus there can be at most $1 + 2k/\epsilon^2$ such groups. ◇

Lemma 5 *$OPT \geq 1 - p_1$.*

Proof. At most one codeword can belong to a^*. All the other codewords contain at least one letter which costs at least 1, and their cumulative frequency is at least $1 - p_{\max} = 1 - p_1$. ◇

We now focus on Step 8. The analysis of Algorithm 1 implies that Step 5 of the algorithm finds a code that is optimal among leveled k-prefix codes. The next lemma implies that this code has cost at most $1 + O(\epsilon)$ times the minimum cost of any k-prefix code.

Lemma 6 *For any k-prefix code c, there exists a leveled k-prefix code c' that maps group elements within each group to the same level, and such that* $\text{cost}(c') \leq \text{cost}(c)(1+O(\varepsilon))$.

Proof. Let c be an optimal k-prefix code. We modify c level by level so that for each i, c is maximal within level i and so that level i contains all or none of the elements of each group, as follows. Since level 0 contains at most one codeword, and the first group contains exactly one element, this is already true for level 0. Assume that we have already modified c to guarantee those properties for levels up to $i-1$, and consider level i.

Leveling phase for level i**.** First modify c so that it is maximal within level i by taking every codeword z in level i and padding it with enough a's (i.e., replace z by za^j) so that its cost is $\ell_2 + i\epsilon - \min\{\ell_1, \epsilon\}$, i.e. so that adding one more letter a would move the codeword out of level i. (Here we use the assumption that every letter cost is a multiple of ϵ, with the possible exception of ℓ_1 which then evenly divides ϵ.)

Grouping phase for level i**.** Next modify the code so that level i contains all or none of the elements of each group. Let n_j be the cardinality of the jth group. Levels 0 through $i-1$ contain a total of $n_1 + \cdots + n_p$ codewords, for some integer p. Let x be the number of codewords in level i, and let q be such that $n_{p+1} + \cdots + n_q \le x < n_{p+1} + \cdots + n_{q+1}$. We then modify c by taking, in level i, the $x - (n_{p+1} + \cdots + n_q)$ codewords of smallest cost, and padding them with an a to move them out of level i. Then c is still maximal within level i, and now contains codewords for all elements of the groups $p+1, \ldots, q$, and no element of any other group.

When this construction has been done for every i, we obtain a leveled code c' that maps elements within each group to the same level.

What is the cost of code c' ? We examine the increase in cost level by level. Let $t = n_1 + \cdots + n_p$ be the total number of codewords contained within levels 0 through $i-1$, just before dealing with level i.

In the leveling phase for level i, codewords in levels $\le i-1$ are not changed, while codewords within level i stay in level i so their cost never changes by more than ϵ. Suppose that after the leveling phase level i contains δ codewords. Then $\text{cost}(c)$ has increased by at most $\epsilon\,(p_{t+1} + \cdots + p_{t+\delta})$.

Let $\delta' = t + n_{p+1} + \cdots + n_q$. In the grouping phase for level i, the codewords $t+1, \ldots, t+\delta'$ are kept on level i while the codewords $t+\delta'+1, \ldots t+\delta$ are padded with an a. The cost added by the padding is then at most $\ell_1\,(p_{t+\delta'+1} + \cdots + p_{t+\delta}) < (1+\epsilon)(1-p_1)\epsilon^2/k$ since $\ell_1 \le 1+\epsilon$, the grouping phase does not move groups of one word, and groups of more than one word have total probability $\le (1-p_1)\epsilon^2/k$.

Let P_m be the sum of all of the probabilities of words in the mth group. The total cost increase due to the leveling and grouping at level i is at most

$$\epsilon \sum_{m=p+1}^{q} P_m + (2\epsilon+1) \sum_{j=t+\delta'+1}^{t+\delta} p_j \le \epsilon \sum_{m=p+1}^{q} P_m + \frac{(1+2\epsilon)(1-p_1)\epsilon^2}{k}.$$

We now sum over all $(k-1)/\epsilon$ levels. The second term sums to $(1+2\epsilon)(1-p_1)\epsilon = O(\epsilon OPT)$ by Lemma 5. The first term sums to either $\epsilon(1-p_1)$ or to ϵ, depending on whether c maps the first word to level 0 or to a level ≥ 1. In either case this is $O(\epsilon \text{cost}(c)) = O(\epsilon OPT)$.

◇

Together with Lemma 1, Lemma 6 implies that at the end of Step 8, Algorithm 3 finds a code that has cost at most $(1+O(\epsilon))OPT$. Finally, Lemma 2 implies that for k chosen as in Step 2, our main algorithm finds a prefix-free code that has cost at most $(1+O(\epsilon))OPT$.

6 Analysis of running time

We first need to analyze the directed graph D which was built in Step 3 of Algorithm 3 and used by Algorithm 1.

Lemma 7 *Graph D has at most nk/ϵ nodes and dnk/ϵ edges, where d is the number of distinct letter costs among $\{\ell_1, \ldots, \ell_r\}$.*

Proof. From Step 1 of Algorithm 3 all letter costs other than ℓ_1 are multiples of ϵ, and ℓ_1 is either a multiple of ϵ or evenly divides ϵ. Moreover, we've assumed $\ell_1 > \epsilon/n$. Thus each of the k/ϵ levels has at most n distinct costs. Thus D has at most nk/ϵ nodes; each node has outdegree at most d, the number of distinct letter costs. ◇

In particular, in Step 3 of Algorithm 3 graph D can be constructed in time $O(ndk/\epsilon)$, moreover, using an array with a bucket for each possible codeword cost, we can access the nodes of D in constant time.

Lemma 8 *Algorithm 1 can be implemented to run in time $O(ndk/\epsilon)$.*

Proof. Since $\ell_1 > \epsilon/n$, there are $O(n/\epsilon)$ nodes on level 0, so Step 3 of Algorithm 1 takes time $O(n/\epsilon)$.

The loop of line 4 is iterated $k/\epsilon = O_\epsilon(1)$ times. The total time for line 5, is at most $O(1)$ per edge, therefore at most $d \times nk/\epsilon$.

Total time for Step 4 is $O(n)$ ($O(1)$ per codeword). (The exact cost of each iteration depends on details of implementation of the codeword set S. We use an implicit representation of S by a tree, and we highlight an edge e of D and give it a codeword weight $w(e)$ if it is used to build $w(e)$ codewords of S. This data structure can be used to obtain the desired time bounds $O_\epsilon(d \sum_i f(i)) = O_\epsilon(dn)$.)

Finally, Step 7 is implemented by doing a breadth-first search to extend D beyond cost k, starting from all the nodes of D which have $v_\ell \geq 1$. Since the outdegree of any node is bounded by d, and we stop as soon as we have identified at most n shortest eligible strings of cost $\geq k$, this has complexity $O(nd)$.

Overall, the algorithm thus has running time $O(ndk/\epsilon)$. ◇

Remark: Constructing an explicit representation of S could conceivable have complexity $\Theta(n^2)$, since just writing down the codewords takes time n^2 if, say, $S = \{b, ab, a^2b, \ldots, a^n b\}$, as might be the case when $\ell_1 \ll 1$.

Lemma 9 *Algorithm 2 can be implemented in time $O(n)$.*

Proof. (sketch) First represent S with a tree as in the usual Huffman encoding problem, compacting the representation as you go so that every internal node has at least two children, and so the total number of nodes is $O(n)$. Then calculate the number of b's recursively in time $O(n)$. Finally, modify the tree to insert the extra letters. This can be implemented in time $O(n)$. ◇

We now go back to Algorithm 3. It is easy to see that everything outside Step 6 takes time $O(nd)$. From Lemma 4, there are only $O(k/\epsilon^2)$ groups, and $O(k/\epsilon)$ levels, thus $\exp(O(\ln(k/\epsilon)k/\epsilon^2))$ maps from groups to levels, hence $\exp(O(\ln(k/\epsilon)k/\epsilon^2))$ choices for $(f(1), \ldots, f(k/\epsilon))$.

How about $f(0)$: how many possibilities must we try? As written, the algorithm tries every possibility in level 0, which could mean as many as $\Theta(n)$, however it is easy to modify the algorithm so that we only try $O(\log n)$ possibilities for $f(0)$: indeed, rounding costs up to the nearest power of $(1+\epsilon)$, it is enough to try $f(0) = 0, 1, \ldots, 1/\epsilon, (1/\epsilon)(1+\epsilon), (1/\epsilon)(1+\epsilon)^2, \ldots, 1$.

Since each iteration takes time $O(ndk/\epsilon)$, we obtain a complexity of $nd\log(n)\exp(O(\ln(k/\epsilon)^2k/\epsilon^2))$. Expanding k, this is $nd\log(n)\exp(O(\ln(1/\epsilon)^2/\epsilon^3))$.

7 Dealing with the case $\ell_1 \leq \epsilon/n$

Recall that a is the shortest letter and b the second shortest letter, of cost $\ell_2 = 1$. First notice that if b is a codeword, then replacing b by ba^n only increases the cost by a factor of $1+\epsilon$. Next, notice that one can always add to the code the strings bb, bab, ba^2b, ..., $ba^{n-1}b$ (removing any codewords which have those as a prefix). It is now straightforward to design an algorithm. Let $\Sigma' \subset \Sigma$ denote the letters of $\Sigma \setminus \{a, b\}$ whose cost are less than 2.

For each guess $f(0) \in \{1, (1+\epsilon), (1+\epsilon)^2, \ldots, n\}$, build a code in a greedy manner by taking the following strings as codewords one by one in increasing order of cost until you have n codewords:

- $a^{f(0)}$
- $ba^n, aba^n, a^2ba^n, \ldots, a^{f(0)-1}ba^n$
- for each $\sigma \in \Sigma'$ by order of increasing cost of σ, take $\sigma, a\sigma, a^2\sigma, \ldots, a^{f(0)-1}\sigma$
- $bb, bab, ba^2n, \ldots, ba^{n-1}b$.

The observations at the beginning of this section imply that the code built by the greedy construction is within $1+\varepsilon$ of the minimum cost code that contains $a^{f(0)}$. Outputting the best code among the codes thus constructed therefore outputs a code within $(1+\epsilon)OPT$.

The greedy construction can be implemented in $O(n+d)$ time; constructing all n such codes corresponding to the different choices of $a(0)$ can therefore be done in $O(\log(n)(n+d)/\epsilon)$ time.

References

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

[2] Julia Abrahams. Code and parse trees for lossless source encoding. *Communications in Information and Systems*, 1(2):113–146, April 2001.

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

[4] Toby Berger and Raymond W. Yeung. Optimum 1-ended binary prefix codes. *IEEE Transactions on Information Theory*, 36(6):1434–1441, November 1990.

[5] N. M. Blachman. Minimum cost coding of information. *IRE Transactions on Information Theory*, PGIT-3:139–149, 1954.

[6] Siu-Ngan Choi and M. Golin. Lopsided trees: Algorithms, analyses and applications. In *Proc. of the 23rd International Colloquium on Automata Languages and Programming (ICALP '96)*, pages 538–549, 1996.

[7] N. Cot. Complexity of the variable-length encoding problem. In *Proc. 6th Southeast Conference on Combinatorics, Graph Theory and Computing*, pages 211–244, 1975.

[8] Norbert Cott. *Characterization and Design of Optimal Prefix Codes*. PhD Thesis, Stanford University, Department of Computer Science, June 1957.

[9] I. Csisz'ar. Simple proofs of some theorems on noiseless channels. *Inform. Contr.*, 514:285–298, 1969.

[10] E. N. Gilbert. How good is morse code. *Inform Control*, 14:585–565, 1969.

[11] E. N. Gilbert. Coding with digits of unequal costs. *IEEE Trans. Inform. Theory*, 41:596–600, 1995.

[12] 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):1770–1781, 1998.

[13] M. Golin and N. Young. Prefix codes: Equiprobable words, unequal letter costs. *SIAM Journal on Computing*, 25(6):1281–1292, December 1996.

[14] K. Hinderer. On dichotomous search with direction-dependent costs for a uniformly hidden objec. *Optimization*, 21(2):215–229, 1990.

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

[16] K. A. S. Immink. *Codes for Mass Data Storage Systems*. Shannon Foundations Publishers, 1999.

[17] I. Itai. Optimal alphabetic trees. *SIAM J. Computing*, 5:9–18, 1976.

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

[19] Richard Karp. Minimum-redundancy coding for the discrete noiseless channel. *IRE Transactions on Information Theory*, IT-7:27–39, January 1961.

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

[21] R. M. Krause. Channels which transmit letters of unequal duration. *Inform. Contr.*, 5:13–24, 1962.

[22] R.S. Marcus. *Discrete Noiseless Coding*. M.S. Thesis, MIT E.E. Dept, 1957.

[23] K. Mehlhorn. An efficient algorithm for constructing nearly optimal prefix codes. *IEEE Trans. Inform. Theory*, 26:513–517, September 1980.

[24] L. L. Larmore P. Bradford, M. Golin and W. Rytter. Optimal prefix-free codes for unequal letter costs and dynamic programming with the monge property. In *Proc. Sixth European Symposium on Algorithms (ESA'98) (LNCS 1461)*, pages 43–54, 1998.

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

[26] A. De Santis R. M. Capocelli and G. Persiano. Binary prefix codes ending in a 1. *IEEE Transactions on Information Theory*, 40(4):1296–1302, July 1994.

[27] E. Korach S. Dolev and D. Yukelson. The sound of silence: Guessing games for saving energy in mobile environment. In *Proc. of the Eighteenth Annual Joint Conference of IEEE Computer and Communications Societies (IEEE INFOCOM'99)*, pages 768–775, 1995.

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

[29] B. Varn. Optimal variable length codes (arbitrary symbol cost and equal code word probability). *Information Control*, 19:289–301, 1971.