SIAM J. COMPUT.
Vol. 41, No. 3, pp. 684–713

HUFFMAN CODING WITH LETTER COSTS: A LINEAR-TIME APPROXIMATION SCHEME*

MORDECAI J. GOLIN†, CLAIRE MATHIEU‡, AND NEAL E. YOUNG§

Abstract. We give a polynomial-time approximation scheme for the generalization of Huffman coding in which codeword letters have nonuniform costs (as in Morse code, where the dash is twice as long as the dot). The algorithm computes a $(1+\epsilon)$-approximate solution in time $O(n + f(\epsilon)\log^3 n)$, where n is the input size.

Key words. Huffman coding with letter costs, polynomial-time approximation scheme

AMS subject classification. 68P30

DOI. 10.1137/100794092

1. Introduction.

The problem of constructing a minimum-cost prefix-free code for a given distribution, known as Huffman coding, is well known and admits a simple greedy algorithm. But there are many well-studied variations of this simple problem for which fast algorithms are not known. This paper considers one such variant—the generalization of Huffman coding in which the encoding letters have nonuniform costs—for which it describes a polynomial-time approximation scheme (PTAS).

Letter costs arise in coding problems where different characters have different transmission times or storage costs [3, 24, 20, 27, 28]. One historical example is the telegraph channel—Morse code. There, the encoding alphabet is $\{\cdot, -\}$, and dashes are twice as long as dots; i.e., $\text{cost}(-) = 2\,\text{cost}(\cdot)$ [10, 11, 22]. A simple data-storage example is the (h,k)-run-length-limited codes used in magnetic and optical storage. There, the codewords are binary and constrained so that each "1" must be preceded by at least h, and at most k, "0's" [17, 13]. (To reduce this problem to Huffman coding with letter costs, use an encoding alphabet with one letter of cost $j+1$ for each string "0^j1," where $h \le j \le k$.)

DEFINITION 1.1 (Huffman coding with letter costs—HULC). *The input is*

- *a probability distribution p on $[n]$,*
- *a* codeword alphabet Σ *of size at most n,*
- *for each letter $\ell \in \Sigma$, a specified nonnegative integer,*[1] $\text{cost}(\ell)$.

The output is a code $\mathcal{X}$ consisting of n codewords, where $\mathcal{X}_i \in \Sigma^*$ is the codeword for probability p_i. The code must be prefix-free. (That is, no codeword is a prefix of any other.) The goal is to minimize the cost of $\mathcal{X}$, which is denoted $\text{cost}(\mathcal{X})$ *and defined to be $\sum_{i=1}^{n} p_i \,\text{cost}(\mathcal{X}_i)$, where, for any string w,* $\text{cost}(w)$ *is the sum of the*

*Received by the editors May 4, 2010; accepted for publication (in revised form) April 10, 2012; published electronically June 26, 2012. A preliminary conference version of this work appeared as "Huffman coding with unequal letter costs" in STOC'02.

http://www.siam.org/journals/sicomp/41-3/79409.html

†Department of Computer Science, Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong (golin@cs.ust.hk). This author's work was partially supported by HK RGC Competitive Research grants HKUST 6137/98E, 6162/00E, and 6082/01E.

‡Department of Computer Science, Brown University, Providence, RI 02912 (claire@cs.brown.edu). This author's work was partially supported by NSF grant CCF-0728816.

§Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA 92521 (neal.young@ucr.edu). This author's work was partially supported by NSF grants CNS-0626912 and CCF-0729071.

[1]The assumption of integer costs is made for technical reasons. In fact the algorithm given here handles arbitrary real letter costs. See section 4.1.

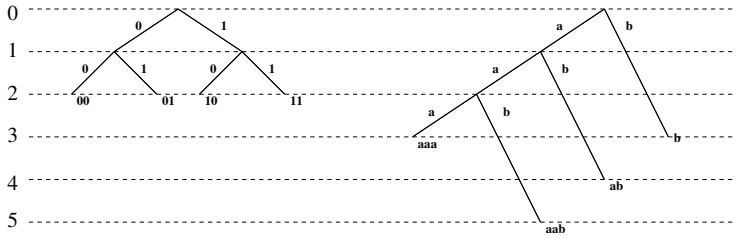

FIG. 1. *Two prefix-free codes, in tree representation. The letter costs are* $(1, 1)$ *and* $(1, 3)$*, respectively. The code on the left is* $\{00, 01, 10, 11\}$*. The code on the right is* $\{aaa, b, ab, aab\}$*. The costs of the two codes are, respectively,* $2(p_1 + p_2 + p_3 + p_4)$ *and* $3(p_1 + p_2) + 4p_3 + 5p_4$*.*

costs of the letters in w. (See Figure 1*.)*

HULC has been extensively studied, since at least 1954. Blachman [3], Marcus [24], and Gilbert [11] give heuristic algorithms. The first algorithm yielding an exact solution is due to Karp, based on integer linear programming [20]. Karp's algorithm does not run in polynomial time. A number of other works use some form of entropy to lower-bound the optimal cost OPT, and give polynomial-time algorithms that compute heuristic solutions of cost at most $\text{OPT} + f(\text{cost})$, where $f(\text{cost})$ is some function of the letter costs [22, 8, 7, 25, 2, 12]. These algorithms are not constant-factor approximation algorithms, even for fixed letter costs, because nontrivial instances can have small OPT. For further references and other uses of HULC, see Abrahams' survey on source coding [1, section 2.7].

However, there is no known polynomial-time algorithm for HULC, nor is it known to be NP-hard. Before now, the problem was not known to have any polynomial-time constant-factor approximation algorithm. Our main result is a PTAS.

THEOREM 1.2 (PTAS for HULC). *Given any* HULC *instance, the tree representation of a prefix-free code of cost at most* $1 + O(\epsilon)$ *times minimum can be computed in time* $O(n) + O_\epsilon(\log^3 n)$*.*

The tree representation is a standard representation of prefix-free codes (see Definition 1.6 and Figure 1). In the $O_\epsilon(\log^3 n)$ term, the subscript ϵ denotes that the hidden constant in the big-O depends on ϵ.

We note without proof that the above PTAS can easily be adapted to show that, given any fixed ϵ, the problem of $(1+\epsilon)$-approximating HULC is in NC (Nick's class—polynomially many parallel processors and polylogarithmic time).

Related problems. When all letter costs are equal, HULC reduces to standard HUFFMAN CODING. The well-known greedy algorithm for HUFFMAN CODING is due to Huffman [16]. The algorithm runs in $O(n)$ time, or $O(n \log n)$ time if p is not sorted.

When the letter costs are fixed integers, Golin and Rote give a dynamic programming algorithm that produces exact solutions in time $O(n^{2+\max_j \text{cost}(\ell_j)})$ [13]. This is improved to $O(n^{\max_j \text{cost}(\ell_j)})$ for alphabets of size 2 by Bradford et al. [4] and for general (but fixed) alphabets by Dumitrescu [9].

When all the *probabilities* are equal (each $p_j = 1/n$), HULC is the VARN CODING problem, which is solvable in polynomial time [28, 23, 6, 26, 14, 5].

Finally, ALPHABETIC CODING is like HUFFMAN CODING but with an additional constraint on the code: the order of the given probabilities matters—their respective codewords must be in increasing alphabetic order. (Here the probabilities are not assumed to be in sorted order.) ALPHABETIC CODING WITH LETTER COSTS

(also called DICHOTOMOUS SEARCH [15] or the LEAKY SHOWER problem [19]) models designing testing procedures where the time required by each test depends upon the outcome [21, (section 6.2.2, Example 33)]. That problem has a polynomial-time algorithm [18].

Basic idea of the PTAS. To give some intuition for the PTAS, consider the following simple idea. Without the prefix-free constraint, HULC would be easy to solve: to find an optimal code $\mathcal{X}$, one could simply enumerate the strings in Σ^* in order of increasing cost, and take $\mathcal{X}_i$ to be the ith string enumerated.

The cost of this optimal non–prefix-free code $\mathcal{X}$ is certainly a lower bound on the minimum cost of any prefix-free code. Now consider modifying $\mathcal{X}$ to make it prefix-free as follows. Prepend to each codeword $\mathcal{X}_i$ its length, encoded in a prefix-free binary encoding. That is, take $\mathcal{X}'_i = \mathrm{enc}(|\mathcal{X}_i|)\,\mathcal{X}_i$, where $\mathrm{enc}(\ell)$ is any natural prefix-free encoding of integer ℓ. (For example, make the standard binary encoding prefix-free by replacing 0 and 1 by 01 and 10, respectively, then append a 00.) The resulting code is prefix-free, because knowing the length of an upcoming codeword is enough to determine where it ends. And, intuitively, the cost of $\mathcal{X}'$ should not exceed the cost of $\mathcal{X}$ by much, because each codeword in $\mathcal{X}$ with ℓ letters has only $O(\log_2 \ell) \le O(\epsilon\ell)$ letters added to it. Thus, the cost of prefix-free code $\mathcal{X}'$ should be at most $1 + O(\epsilon)$ times the cost of $\mathcal{X}$, and thus at most $1 + O(\epsilon)$ times the cost of OPT.

Why does the above idea fail? It fails because $\log_2 \ell$ is not $O(\epsilon\ell)$ when $\ell < O(\epsilon^{-1} \log \epsilon^{-1})$. That is, when a codeword is small, prepending its length can increase its cost by too much. To work around this, we handle the small codewords separately, determining their placement by exhaustive search. This is the basic idea of the PTAS. The rest of the paper gives the technical details.

Terminology and definitions. For technical reasons, we work with a generalization of HULC in which codewords can be restricted to a given universe $\mathcal{U}$, as given next.

DEFINITION 1.3 (HULC with restricted universe). *The input is a* HULC *instance* $(p, \Sigma, \mathrm{cost})$ *and a* codeword universe $\mathcal{U} \subseteq \Sigma^*$. *The universe* $\mathcal{U}$ *is specified by a finite, prefix-free set* $\mathcal{R} \subset \Sigma^*$ *of "roots" such that* $\mathcal{U}$ *consists of the strings with a prefix in* $\mathcal{R}$. *The problem is to find a code of minimum cost among the prefix-free codes whose codewords are in* $\mathcal{U}$.

Formally, $\mathcal{U}$ is defined from the given root set $\mathcal{R} \subset \Sigma^*$ as the set of strings $x \in \Sigma^*$ such that $\mathrm{prefixes}(x) \cap \mathcal{R} \neq \emptyset$, where $\mathrm{prefixes}(x)$ denotes the set of all prefixes of x. The universe is necessarily closed under appending letters (that is, if $x \in \mathcal{U}$ and y has x as a prefix, then $y \in \mathcal{U}$). If $\mathcal{U} = \Sigma^*$ (i.e., $\mathcal{R}$ contains just the empty string), then the problem is HULC as defined at the start of the paper.

In any problem instance, we assume the following without loss of generality:

- There are at most n letters in the alphabet Σ, and they are $\{0, 1, \ldots, |\Sigma| - 1\}$.
- The letter costs are increasing: $\mathrm{cost}(0) \le \mathrm{cost}(1) \le \cdots \le \mathrm{cost}(|\Sigma| - 1)$. (If not, sort them first, adding $O(n \log n)$ or less to the run time.)
- The codeword probabilities are decreasing: $p_1 \ge p_2 \ge \cdots \ge p_n$. (If not, sort them first, adding $O(n \log n)$ to the run time.)

DEFINITION 1.4 (monotone code). *A code* $\mathcal{X}$ *is* monotone *if*

$$\mathrm{cost}(\mathcal{X}_1) \le \mathrm{cost}(\mathcal{X}_2) \le \cdots \le \mathrm{cost}(\mathcal{X}_n).$$

For any code $\mathcal{X}$, reordering its codewords to make it monotone does not increase its cost (since p is decreasing), so we generally focus on monotone codes.

Next we define two more compact representations of codes.

DEFINITION 1.5 (signature representation). *Given a set $\mathcal{X} \subseteq \Sigma^*$, its* signature *is the vector x such that x_i is the number of strings in $\mathcal{X}$ that have cost i. (Recall that letters, and thus codewords, have integer costs.)*

In Figure 1, the first code has signature $(0, 0, 4)$; the second code has signature $(0, 0, 0, 2, 1, 1)$.

Many codes may have the same signature, but any two (monotone) codes with the same signature are essentially equivalent. For example, the signature x of a monotone code $\mathcal{X}$ determines $\text{cost}(\mathcal{X})$: indeed, $\text{cost}(\mathcal{X}_k) = i(k)$, where $i(k)$ is the minimum i such that $x_1 + \cdots + x_i \geq k$.

DEFINITION 1.6 (tree representation). *The* tree representation *of a code $\mathcal{X}$ is a forest with a node $v(s)$ for each string $s \in \text{prefixes}(\mathcal{X}) \cap \mathcal{U}$, and an edge from each (parent) node $v(s)$ to (child) node $v(s')$ if $s' = s\ell$ for some letter $\ell \in \Sigma$. Each root of the forest is labeled with its corresponding string in $\mathcal{R}$.*

For standard Huffman coding (with just two equal-cost letters $\{0, 1\}$ and $\mathcal{U} = \Sigma^*$), the tree representation is a binary tree. Each codeword traces a path from the root, with 0's corresponding to left edges and 1's to right edges. See, for example, $\mathcal{X}_1$ in Figure 1. If $\mathcal{U} \neq \Sigma^*$, the tree representation can be a forest (that is, it can have multiple trees, each with a distinct root in $\mathcal{R}$).

A code is prefix-free if and only if, in its tree representation, all codewords are leaf nodes.

DEFINITION 1.7 (levels). *The* ith level *of a set $\mathcal{X} \subseteq \Sigma^*$ contains the cost-i strings in $\mathcal{X}$. (See the horizontal lines in Figure 1.)*

Additional terminology and notation. Throughout the paper, ϵ is an arbitrary constant strictly between 0 and 1/2. The PTAS returns a near-optimal code—a code of cost $1+O(\epsilon)$ times the minimum cost of any prefix-free code. The terms "nearly," "approximately," etc., generally mean "within a $1 + O(\epsilon)$ factor." The notation $O_\epsilon(f(n))$ denotes $O(f(n))$, where the hidden constant in the big-O can depend on ϵ.

Given a problem instance $\mathcal{I}$, the cost of an optimal solution is denoted $\text{OPT}(\mathcal{I})$, or just OPT if $\mathcal{I}$ is clear from context. As is standard, $[n]$ denotes $\{1, 2, \ldots, n\}$. We let $[i..j]$ denote $\{i, i + 1, \ldots, j\}$.

The rest of the paper proves Theorem 1.2. The value of the second-largest letter cost, i.e., cost(1), is a major consideration in the proof. We first describe a PTAS for the case when $\text{cost}(1) \leq 3/\epsilon$; we then reduce the general case to that one. For efficiency, the PTAS works mainly with code signatures; in the last step, it converts the appropriate signature to a tree representation.

See Figure 2 for a summary of the three remaining sections and the five subsections of section 2.

2. Computing the signature of a near-optimal code when $\text{cost}(1) \leq 3/\epsilon$.

This section gives the core algorithm of the PTAS. Given any instance in which $\text{cost}(1) \leq 3/\epsilon$, the core algorithm computes the signature of a near-optimal prefix-free code for that instance. (Recall that all letter costs are integers.) Formally, in this section we prove the following theorem.

THEOREM 2.1. *Fix any instance $\mathcal{I} = (p, \Sigma, \text{cost}, \mathcal{U})$ of* HULC *with restricted universe such that $\text{cost}(1) \leq 3/\epsilon$. Let P be the cumulative probability distribution for p: $P_\ell = \sum_{k \leq \ell} p_k$ (for $\ell \in [n]$). Let σ be the signature of Σ. Let r be the signature of the roots of $\mathcal{U}$. Assume that P, σ, and r are given as inputs.*

Then the signature and approximate cost of a prefix-free code (for $\mathcal{I}$) with cost at most $(1 + O(\epsilon))\,\text{OPT}(I)$ can be computed in time $O_\epsilon(\log^2 n)$.

Section 2. For instances in which $\text{cost}(1) \leq 3/\epsilon$, the signature x of a near-optimal prefix-free code can be computed in time $O_\epsilon(\log^2 n)$, provided that the following inputs are precomputed: the cumulative probability distribution P (for the distribution p) and the signatures σ and r of, respectively, the alphabet Σ and the roots $\mathcal{R}$ of the universe $\mathcal{U}$. (These inputs p, σ, and r can be precomputed in $O(n)$ time.)

Section 3. From the signature x, the tree can be built in $O(n) + O_\epsilon(\log^2 n)$ time.

Section 4. Any arbitrary instance of HULC reduces to $O_\epsilon(\log n)$ instances with $\text{cost}(1) \leq 3/\epsilon$, which can in turn be solved by the PTAS from sections 2 and 3, giving the full PTAS.

Breakdown of section 2 (finding a near-optimal signature when $\text{cost}(1) \leq 3/\epsilon$). Sections 2.1–2.4 define and analyze certain structural properties related to near-optimal codes. Section 2.5 uses these properties to assemble the PTAS for instances with $\text{cost}(1) \leq 3/\epsilon$.

Section 2.1. In a *τ-relaxed* code, codewords of cost at least a given threshold τ *are* allowed to be prefixes of other codewords. For appropriate (constant) τ, this relaxation (finding a min-cost τ-relaxed code) has a gap of $1 + O(\epsilon)$—a given τ-relaxed code can be efficiently "rounded" into a prefix-free code without increasing the cost by more than a factor of $1 + O(\epsilon)$.

Thus, it suffices to find a near-optimal τ-relaxed code and then round it.

Any τ-relaxed code $\mathcal{X}$ is essentially determined by its set $\mathcal{X}_{<\tau}$ codewords of cost less than τ. This observation alone is enough to give a slow PTAS for instances with $\text{cost}(1) \leq 3/\epsilon$: exhaustively search the possible signatures f of $\mathcal{X}_{<\tau}$ to find the best.

This would give run time $n^{O_\epsilon(1)}$. The remaining subsections improve the time to $O(n) + O_\epsilon(\log^2 n)$.

Section 2.2. Restricting attention to a relatively small subset of τ-relaxed codes, so-called *group-respecting* codes, increases the cost by at most a $1 + O(\epsilon)$ factor. Thus, it suffices to find an optimal group-respecting τ-relaxed code. This observation reduces the search space size to a constant.

Section 2.3. There is a logarithmic-size set $\mathcal{L}$ of levels such that, without loss of generality, we can consider only codes with support in $\mathcal{L}$—that is, codes whose tree representations have (interior or codeword) nodes only in levels in $\mathcal{L}$. Thus, it suffices to find an optimal group-respecting τ-relaxed code with support in $\mathcal{L}$.

Section 2.4. The problem of finding the *signature* of such a code is formally modeled via an integer linear program, ILP. Thanks to section 2.3, ILP has logarithmic size. Further, given the values of just a constant number of key variables of ILP, an optimal (greedy) assignment of the rest of the variables can easily be computed in logarithmic time.

Section 2.5. Putting the above pieces together, the PTAS for instances with $\text{cost}(1) \leq 3/\epsilon$ enumerates the constantly many possible assignments of the key variables in ILP, then chooses the solution giving minimum cost. This gives the signature x of a near-optimal τ-relaxed code, which is converted via the rounding procedure of section 2.1 into the desired signature x' of a near-optimal prefix-free code.

FIG. 2. *Outline of the proof of Theorem* 1.2 *(PTAS for* HULC*).*

Throughout this section, in proving Theorem 2.1, assume $\text{cost}(1) \leq 3/\epsilon$. (The proof holds for any instance in which $\text{cost}(1) = O_\epsilon(1)$; we focus on the case $\text{cost}(1) \leq 3/\epsilon$ only because later we reduce the general case to that case.)

2.1. Allowing codes to be τ-relaxed. In a *τ-relaxed code*, codewords of cost at least τ *can* be prefixes of other codewords, as illustrated in Figure 3.

DEFINITION 2.2 (relaxation τ-RELAX). *Given a* threshold $\tau \geq 0$*, a code $\mathcal{X}$ is* τ-relaxed *if no codeword* of cost less than τ *is the prefix of another codeword. (Prefix-*

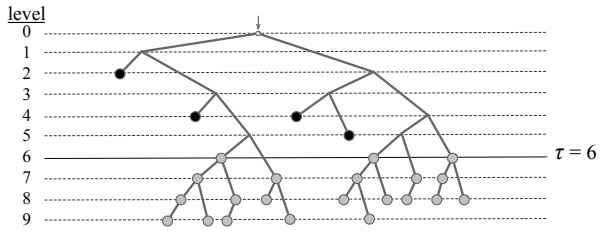

FIG. 3. *Tree representation of a τ-relaxed code $\mathcal{X}$ with four codewords in levels less than $\tau = 6$. The 21 codewords in levels τ and higher can be prefixes of other codewords, so they are taken to be the cheapest 21 strings that have no prefix in $\mathcal{X}$ of cost less than τ.*

free codes are τ-relaxed, but not vice versa.)

τ-RELAX is the problem of finding a minimum-cost τ-relaxed code for a given instance of HULC.

HULC reduces to τ-RELAX. Specifically, if the threshold τ is appropriately chosen, the relaxation changes the optimal cost by at most a $1 + O(\epsilon)$ factor, as shown next.

LEMMA 2.3 (relaxation gap). *Fix threshold $\tau = \lceil \log_2[\mathrm{cost}(1)/\epsilon]\, \mathrm{cost}(1)/\epsilon \rceil$. Given a τ-relaxed code $\mathcal{X}$ for any* HULC *instance, there exists a prefix-free code $\mathcal{X}'$ such that $\mathrm{cost}(\mathcal{X}') = (1 + O(\epsilon))\,\mathrm{cost}(\mathcal{X})$. The code $\mathcal{X}'$ is produced by calling procedure* Round$(\mathcal{X})$.

Proof. The procedure Round is Algorithm 2.1, below. Roughly, for each codeword of cost τ or more in $\mathcal{X}$, Round inserts the cost, i, (encoded in a simple prefix-free binary code, as specified in step 1 of the algorithm) into the codeword, starting at level τ. For technical reasons, instead of the cost i, it actually inserts $i - \hat{\tau}$, where $\hat{\tau}$ is the minimum cost of any codeword in the code of cost at least τ.

Algorithm 2.1. Round: *Construct a prefix-free code from a τ-relaxed code.*

given: *τ-relaxed code $\mathcal{X}$ (for $\tau = \lceil \log_2[\mathrm{cost}(1)/\epsilon]\, \mathrm{cost}(1)/\epsilon \rceil$).*
return: *Prefix-free code $\mathcal{X}'$ of cost $(1 + O(\epsilon))\,\mathrm{cost}(\mathcal{X})$.*

1: Define $\mathrm{enc}(0) = 00$. For integer $i > 0$, define $\mathrm{enc}(i)$ to be the encoding of i obtained from the binary representation of i by replacing each 0 by 01, each 1 by 10, and finally appending 00.
Note that $\{\mathrm{enc}(i) : i = 0, 1, 2, \ldots\}$ is prefix-free.
2: Let $\hat{\tau} = \min\{\mathrm{cost}(\mathcal{X}_k) : \mathrm{cost}(\mathcal{X}_k) \geq \tau\}$.
3: **for** each codeword $\mathcal{X}_k$ of cost τ or more **do**
4: **Round** the codeword: let x be the smallest prefix of $\mathcal{X}_k$ of cost τ or more that is in $\mathcal{U}$; let y be the remaining suffix; replace the codeword $\mathcal{X}_k = xy$ by $\mathcal{X}'_k = x\,\mathrm{enc}(\mathrm{cost}(xy) - \hat{\tau})y$.
5: Return the rounded code $\mathcal{X}'$.

Here is why the code $\mathcal{X}'$ returned by Round is prefix-free. Since $\mathcal{X}$ is τ-relaxed, codewords of cost less than τ are not prefixes of any other codeword. Any codeword of cost $i \geq \tau$, once rounded, cannot be a prefix of any nonrounded codeword because the nonrounded codewords have cost less than τ. It cannot be a prefix of any rounded codeword because in any rounded codeword the string $\mathrm{enc}(i - \hat{\tau})$ (which immediately follows its unique minimal prefix x of cost τ or more in $\mathcal{U}$) uniquely determines the cost of the remaining suffix y. Thus, $\mathcal{X}'$ is prefix-free.

Here is why $\mathcal{X}'$ has cost $(1 + O(\epsilon))\,\mathrm{cost}(\mathcal{X})$. Modifying a codeword of cost $i \geq \tau$

increases its cost by at most $2\,\mathrm{cost}(1)\lceil \log_2 i \rceil$. Since $i \geq \tau$ and τ is chosen[2] so that $\mathrm{cost}(1)\log\tau = O(\epsilon\tau)$, the increase is $O(\epsilon i)$.

Each modified codeword is still in $\mathcal{U}$ because, in any codeword xy that is modified, the unmodified prefix x is in $\mathcal{U}$, so xz is in $\mathcal{U}$ for any string z. □

Remark for intuition—A slow PTAS. Lemma 2.3 alone is enough to give an $n^{O_\epsilon(1)}$-time PTAS for HULC (when $\mathrm{cost}(1) \leq 3/\epsilon$). The intuition is as follows.

A minimum-cost τ-relaxed code $\mathcal{X}$ can be found as follows (much more easily than a minimum-cost *prefix-free* code). Let $\mathcal{X}_{<\tau}$ denote the set containing the codewords in $\mathcal{X}$ of cost less than τ. Given just $\mathcal{X}_{<\tau}$, the optimal way to choose the remaining codewords (those in $\mathcal{X} - \mathcal{X}_{<\tau}$) is *greedily*: those remaining codewords must simply be some $n - |\mathcal{X}_{<\tau}|$ *cheapest available strings among those that have no prefix in* $\mathcal{X}_{<\tau}$. In short, the optimal τ-relaxed code $\mathcal{X}$ is essentially determined by its set $\mathcal{X}_{<\tau}$ of codewords of cost less than τ.

In fact, the code $\mathcal{X}$ is essentially determined by just the *signature* f of this set $\mathcal{X}_{<\tau}$ (the signature f essentially determines $\mathcal{X}_{<\tau}$, which in turn determines $\mathcal{X}$). Each such signature is a distinct function $f : [\tau] \to [0..n]$. There are $(n+1)^\tau$ such functions.

Recall that, as defined in Lemma 2.3, the threshold τ is $O_\epsilon(1)$. (The assumption $\mathrm{cost}(1) \leq 3/\epsilon$ and the choice of $\tau \approx \log[\mathrm{cost}(1)/\epsilon]\,\mathrm{cost}(1)/\epsilon$ imply $\tau = O(\log(1/\epsilon)/\epsilon^2)$.) Thus, the number $(n+1)^\tau$ of such functions is $n^{O_\epsilon(1)}$.

The PTAS is as follows: exhaustively search all such functions f. For each, construct a minimum-cost τ-relaxed code $\mathcal{X}$ such that $\mathcal{X}_{<\tau}$ has signature f. (If any such code $\mathcal{X}$ exists, it can be constructed greedily from just f as described above.) Finally, take $\mathcal{X}^{\min}$ to be the code of minimum cost among the τ-relaxed codes $\mathcal{X}$ obtained in this way, take $\mathcal{X}'$ to be the prefix-free code produced by $\mathsf{Round}(\mathcal{X}^{\min})$, and, finally, return $\mathcal{X}'$.

By Lemma 2.3, the prefix-free code $\mathcal{X}'$ obtained by rounding $\mathcal{X}^{\min}$ has cost $(1 + O(\epsilon))\,\mathrm{cost}(\mathcal{X}^{\min})$. By its construction, $\mathcal{X}^{\min}$ is an optimal τ-relaxed code. Since any prefix-free code is also τ-relaxed, the cost of $\mathcal{X}^{\min}$ is at most the cost of the minimum-cost prefix-free code, OPT. Transitively,

$$\mathrm{cost}(\mathcal{X}') \leq (1 + O(\epsilon))\,\mathrm{cost}(\mathcal{X}^{\min}) \leq (1 + O(\epsilon))^2\,\mathrm{OPT} = (1 + O(\epsilon))\,\mathrm{OPT}\,.$$

That is, the algorithm is a PTAS.

The rest of the paper is about reducing the running time (in sections 2 and 3) and reducing the general case to the case $\mathrm{cost}(1) \leq 3/\epsilon$ (in section 4).

2.2. Restricting to *group-respecting* τ-relaxed codes. By Lemma 2.3, to find a near-optimal prefix-free code, it suffices to find a near-optimal τ-relaxed code $\mathcal{X}$ and then "round" $\mathcal{X}$.

As described in the remark in section 2.1, this fact yields a PTAS, one that works by exhaustively searching the potential signatures f for the set $\mathcal{X}_{<\tau}$ of codewords of cost less than τ. This gives an optimal τ-relaxed code $\mathcal{X}$, which the PTAS then rounds to a near-optimal prefix-free code.

The run time of this PTAS is high because there are $n^{O_\epsilon(1)}$ potential signatures.

To reduce the run time, we next show how to compute a set S of signatures that has *constant* size yet is nonetheless still guaranteed to contain a good signature—that is, the signature f of some set $\mathcal{X}_{<\tau}$ that extends to a near-optimal τ-relaxed code $\mathcal{X}$.

[2]The condition $\mathrm{cost}(1)\log\tau = O(\epsilon\tau)$ is equivalent to $\tau/\log\tau = \Omega(z)$ for $z = \mathrm{cost}(1)/\epsilon$. This holds because the choice of τ implies $\tau \geq z\log z$, which (using $\log z \leq z$ and some algebra) implies $\tau/\log\tau \geq z/2$.

To compute this set S, we restrict our attention to codes that choose the codewords in levels less than τ in a restricted way. In particular, we partition the probabilities $\{p_i\}_i$ into a constant number of groups. We then consider only codes that, within the levels less than τ, *give all probabilities within each group codewords of equal cost.*

The partition G of $p[1..n]$ in question is constructed greedily so that there are $O(\tau/\epsilon) = O_\epsilon(1)$ groups, and, within each group, either there is only one (large) probability or the probabilities sum to $O(\epsilon/\tau)$. Recall that p is decreasing.

DEFINITION 2.4 (grouping). *Given any* HULC *instance* (p, Σ, cost), $\epsilon > 0$, *and* τ *from Lemma* 2.3, *define the* grouping $G = G_{\epsilon,\tau}(p)$ *of* p *to be a partitioning of* p*'s index set* $[n]$ *into some* γ *contiguous groups* $(G_1, G_2, \ldots, G_\gamma)$, *as follows: take* $G_g = (j, j+1, \ldots, h)$, *where* h *is maximal subject to* $p_j + \cdots + p_{h-1} \le \epsilon/\tau$ *(and* j *is just after the previous group ended, i.e.,* $j = 1 + \max G_{g-1}$, *or* $j = 1$ *if* $g = 1$*).*

Given a τ*-relaxed code* $\mathcal{X}$, *say that* $\mathcal{X}$ respects G *if, for each group* G_g, *if any index* k *in* G_g *is assigned a codeword of some cost* i *less than* τ, *then* all *indices in* G_g *are assigned codewords of cost* i. *(Formally, for all* g, *for any* $k, k' \in G_g$, *one has* $\max(\text{cost}(\mathcal{X}_k), \tau) = \max(\text{cost}(\mathcal{X}_{k'}), \tau)$*.)*

The number of groups, γ, is at most τ/ϵ (because each group except the last has total probability at least ϵ/τ). Also, each group $G_g = (j, j+1, \ldots, h)$ either has just one member, or has $p_j + p_{j+1} + p_{h-1} \le \epsilon/\tau$.

Next we argue that there is always a G-respecting τ-relaxed code that is near-optimal. To argue this, we show that any τ-relaxed code (in particular the optimal one) can be modified, by working from level 0 to level $\tau-1$, appending 0's to codewords as necessary to make the code G-respecting, while increasing the cost by at most a $1+\epsilon$ factor. More specifically, since the code is monotone, in any given level $i < \tau$, at most one group G_g is "split" between that level and higher levels, and that group has total probability $O(\epsilon/\tau)$. We "fix" that group (by appending a 0 to its level-i codewords) while increasing the cost of the code by $O(\text{cost}(0)\epsilon/\tau)$. The total cost of fixing all levels in $[0, \tau-1]$ in this way is at most $\tau \times \text{cost}(0)\epsilon/\tau = \text{cost}(0)\epsilon$. This is at most ϵ times the total cost of the code, because any code must cost at least cost(0).

LEMMA 2.5 (grouping gap). *Given a* τ*-relaxed code* $\mathcal{X}$ *for any* HULC *instance, there exists a* τ*-relaxed code* $\mathcal{X}'$ *that is* G*-respecting and such that* $\text{cost}(\mathcal{X}') \le (1+\epsilon)\,\text{cost}(\mathcal{X})$.

Proof. Let $\mathcal{X}$ be any τ-relaxed code. If $\mathcal{X}$ is not monotone, reorder its codewords to make it monotone. For each $i \in [\tau]$, in increasing order, do the following. Since $\mathcal{X}$ is monotone there can be at most one group G_g that is "split" at level i, meaning that some probabilities are assigned codewords of cost i while others are assigned codewords of larger cost. If there is such a group, add a letter 0 to the end of each level-i codeword assigned to that group, and then reorder the codewords above level i to restore monotonicity. This defines $\mathcal{X}'$. (See Figure 4.)

Note that the codewords in $\mathcal{X}'$ are still in $\mathcal{U}$, and that $\mathcal{X}'$ is monotone, G-respecting, and τ-relaxed.

To finish we bound the cost increase. Clearly, reordering codewords to make a code monotone never increases the cost. Then, if a group $G_g = (j, j+1, \ldots, h)$ has its codewords modified for level i, then that group must have at least two members, and $p_j + p_{j+1} + \cdots + p_{h-1}$ must be at most ϵ/τ. Thus, adding a letter 0 to the level-i codewords assigned to G_g increases the cost of the code by at most $\text{cost}(0)\epsilon/\tau$. Since there is at most one such increase for each level $i < \tau$, the total increase in cost is at most $\tau\,\text{cost}(0)\epsilon/\tau = \epsilon\,\text{cost}(0)$. On the other hand, the cost of any code is at least cost(0). Thus, the modified code $\mathcal{X}'$ has cost at most $(1+\epsilon)\,\text{cost}(\mathcal{X})$. ∎

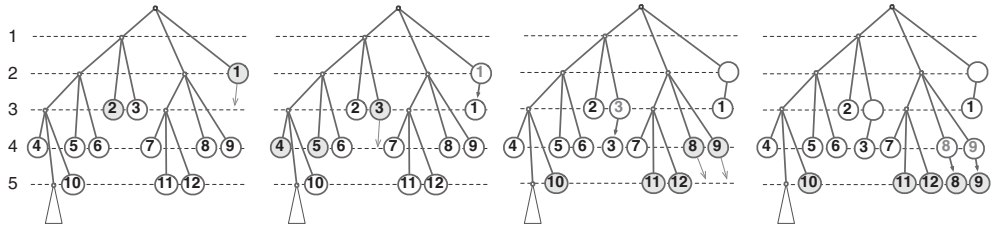

FIG. 4. *Making a code G-respecting. The first four groups are* $G_1 = \{1, 2\}$, $G_2 = \{3, 4, 5\}$, $G_3 = \{6, 7\}$, $G_4 = \{8, 9, 10, 11, 12\}$. *Iterations for levels* $i = 2, 3, 4$ *are shown, from left.*

2.3. Bounding the support of τ-relaxed group-respecting codes. By Lemma 2.5, to find a near-optimal τ-relaxed code, it suffices to find a near-optimal G-respecting τ-relaxed code $\mathcal{X}$.

In this section, we observe that any such code $\mathcal{X}$ (and its prefix-free rounded code $\mathcal{X}'$, per Lemma 2.3) must have *support* in a logarithmic-size set $\mathcal{L}$ of levels. That is, each string in $\text{prefixes}(\mathcal{X}) \cap \mathcal{U}$ (and each node in its tree representation) must have cost in $\mathcal{L}$. Thus, for example, the signature x of such a code has support of logarithmic size.

We use this structural property later in the paper to keep parts of the computation time polylogarithmic. The detailed definition of $\mathcal{L}$ is not important; what is important is that $\mathcal{L}$ can be precomputed easily and has logarithmic size.

DEFINITION 2.6 (limited levels, $\mathcal{L}$). *Given any instance of* τ-RELAX, *let* τ *be as defined in Lemma* 2.3. *Let* $i_{\mathcal{R}}$ *be the minimum cost of any root of* $\mathcal{U}$ *of cost at least* τ. *Let* i_Σ *be the minimum cost of any letter in* Σ *of cost at least* τ. *Let* $\delta = \text{cost}(1)\lceil \log_2 n \rceil$. *Define* $\mathcal{L}$, *the* set of possible levels, *to contain the* $O(\text{poly}(\epsilon^{-1}) \log n)$ *integers in*

$$[0, 2\tau + 3\delta] \ \cup \ [i_{\mathcal{R}}, i_{\mathcal{R}} + 3\delta] \ \cup \ [i_\Sigma, i_\Sigma + \tau + 3\delta]. \tag{2.1}$$

(If $i_{\mathcal{R}}$ *or* i_Σ *is not well defined, take the corresponding interval above to be empty.)*

To verify that $\mathcal{L}$ has logarithmic size, note that, since $\text{cost}(1) \le 3/\epsilon$, it follows that $\tau = O(\text{poly}(\epsilon^{-1}))$ and $\delta = O(\text{poly}(\epsilon^{-1}) \log n)$. Thus, by inspection, $\mathcal{L}$ has size $O(\text{poly}(\epsilon^{-1}) \log n)$.

Next we prove that without loss of generality, in computing and rounding a τ-relaxed code, we can limit our attention to codes having support in $\mathcal{L}$.

The proof is based on local-optimality arguments (and details of the rounding procedure). The rough idea is this. Among the words in levels τ and up that are available to be codewords, let s_τ denote one of minimum cost, as shown in Figure 5. Since codewords in levels τ and above must be taken greedily in any optimal τ-relaxed code, and the n words of the form $s_\tau\{0,1\}^{\lceil \log_2 n \rceil}$ are available to be codewords, it follows that all codewords that lie in level τ or above should have costs in $[\text{cost}(s_\tau), \text{cost}(s_\tau) + \delta]$ (recall $\delta = \text{cost}(1)\lceil \log_2 n \rceil$). To finish the proof, we bound the values that $\text{cost}(s_\tau)$ can take, and we observe that rounding any codeword in level τ or above increases its costs by at most 2δ.

LEMMA 2.7 (limited levels). *Given any instance of* τ-RELAX, *let* $\mathcal{L}$ *be as defined above. Then the following hold:*

(i) *Any minimum-cost* τ*-relaxed* G*-respecting code* $\mathcal{X}$ *has support in* $\mathcal{L}$.

(ii) *Rounding such a code* $\mathcal{X}$ *(per Lemma* 2.3*) gives a prefix-free code* $\mathcal{X}'$ *with support in* $\mathcal{L}$.

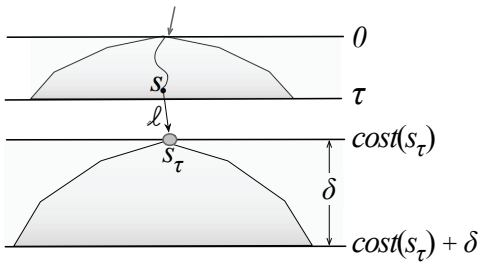

FIG. 5.

Proof. Part (i). Let $\mathcal{X}$ be any minimum-cost G-respecting τ-relaxed code. Assume $\mathcal{X}$ has a codeword of cost at least τ (otherwise all nodes in the tree representation are in $[0, \tau) \subset \mathcal{L}$, and we are done).

Say a string of cost at least τ is *available* if no prefix of the string is a codeword of cost less than τ in $\mathcal{X}$.

Let s_τ be a minimum-cost available codeword. (There is at least one, by the assumption that $\mathcal{X}$ has a codeword of cost at least τ.) Let s be the parent of s_τ, so that $s_\tau = s\ell$ for some $\ell \in \Sigma$, as shown in Figure 5.

The n strings in $S = s_\tau\{0,1\}^{\lceil \log_2 n \rceil}$ are available. Each costs at most $\mathrm{cost}(s_\tau)+\delta$, so, in the tree representation of $\mathcal{X}$, all levels $i > \mathrm{cost}(s_\tau) + \delta$ are empty. (Otherwise $\mathcal{X}$ could be made cheaper by swapping in some string of cost at most $\mathrm{cost}(s_\tau) + \delta$.) Thus, $\mathcal{X}$ has support in $[0, \tau) \cup [\mathrm{cost}(s_\tau), \mathrm{cost}(s_\tau) + \delta]$.

Let $\mathcal{X}'$ be obtained by rounding $\mathcal{X}$ (Lemma 2.3). Any unmodified codeword has cost less than τ. Following the notation of Algorithm 2.1, let $\mathcal{X}'_k = x\,\mathrm{enc}(i - \mathrm{cost}(s_\tau))y$ be any modified codeword, so that $i = \mathrm{cost}(\mathcal{X}_k)$. By the previous paragraph, $i \le \mathrm{cost}(s_\tau)+\delta$, and rounding increases the cost of the codeword by at most $\mathrm{cost}(\mathrm{enc}(\delta)) \le 2\,\mathrm{cost}(1)\lceil \log_2 \delta \rceil \le 2\delta$ (assuming $n \ge 3/\epsilon$) to at most $\mathrm{cost}(s_\tau) + 3\delta$. Also, by the rounding method, the code tree is not modified below level $\mathrm{cost}(s_\tau)$. Thus, $\mathcal{X}'$ and $\mathcal{X}$ have support in $[0, \tau) \cup [\mathrm{cost}(s_\tau), \mathrm{cost}(s_\tau) + 3\delta]$.

To complete the proof, we show that these two intervals are contained within the three intervals $[0, 2\tau + 3\delta] \cup [i_\mathcal{R}, i_\mathcal{R} + 3\delta] \cup [i_\Sigma, i_\Sigma + \tau + 3\delta]$ from the definition of $\mathcal{L}$. By inspection, this will be the case as long as

$$\mathrm{cost}(s_\tau) \in [\tau, 2\tau] \cup \{i_\mathcal{R}\} \cup [i_\Sigma, i_\Sigma + \tau]. \tag{2.2}$$

We use a case analysis to show that (2.2) holds.

If it happens that $s \notin \mathcal{U}$, then s_τ is a root of $\mathcal{U}$, necessarily (by the choice of s_τ) of cost $i_\mathcal{R}$, so (2.2) holds. So assume $s \in \mathcal{U}$. Then $\mathrm{cost}(s) < \tau$. (Otherwise s would be available and have cost less than s_τ, contradicting the choice of s_τ.) If it happens that $\mathrm{cost}(\ell) < \tau$, then $\mathrm{cost}(s_\tau) = \mathrm{cost}(s) + \mathrm{cost}(\ell) < 2\tau$, so $\mathrm{cost}(s_\tau) \in [\tau, 2\tau]$, and (2.2) holds. So assume $\mathrm{cost}(\ell) \ge \tau$. In this case i_Σ is well defined, and $\mathrm{cost}(\ell) \ge i_\Sigma$ (as no letters have cost in $[\tau, i_\Sigma)$, by the definition of i_Σ). In fact it must be that $\mathrm{cost}(\ell) = i_\Sigma$ (otherwise replacing the last letter ℓ in codeword s_τ by the letter of cost i_Σ would give a string that is cheaper than s_τ, contradicting the choice of s_τ). Thus, $i_\Sigma \le \mathrm{cost}(\ell) \le \mathrm{cost}(s_\tau) \le \tau + i_\Sigma$, so $\mathrm{cost}(s_\tau) \in [i_\Sigma, i_\Sigma + \tau]$. ∎

2.4. A mixed integer program to find a min-cost G-respecting τ-relaxed code. In this section we focus on the problem of finding the full signature x of an optimal G-respecting τ-relaxed code $\mathcal{X}$, for a given instance of HULC. We describe

minimize $\sum_{i,k} p_k\, i\, y_{ki}$ s.t.

$$
\begin{aligned}
x_i + w_i &\le r_i + \textstyle\sum_{j<i} \sigma_{i-j} w_j && (i \in [m]) \\
\textstyle\sum_{k\in[n]} y_{ki} &= x_i && (i \in [m]) \\
\textstyle\sum_{i\in[m]} y_{ki} &= 1 && (k \in [n]) \\
w_i, x_i, y_{ki} &\in \mathbb{N}_{\ge 0} && (i \in [m],\ k \in [n])
\end{aligned}
$$

Parameters

- p — probability distribution on $[n]$
- r — signature of $\mathcal{U}$'s root set $\mathcal{R}$
- σ — signature of alphabet Σ
- m — $n \times \max\{\text{cost}(\ell) \mid \ell \in \Sigma\}$

Variables determining code $\mathcal{X}$

- x — signature of codewords ($\mathcal{X}$)
- w — signature of interior nodes
- y — assignment; $y_{ki} = 1$ iff $\text{cost}(\mathcal{X}_k) = i$

FIG. 6. *Karp's integer linear program (*KARP*) for finding a minimum-cost prefix-free code.*

how this problem can by modeled by an integer linear program (ILP) that (thanks to Lemma 2.7) has size $O_\epsilon(\log n)$.

We also identify, within ILP, a particular constant-size vector z of binary variables. (These variables encode the assignment of the groups in G to the levels less than τ.) We show that, given any assignment to just these constantly many binary variables, an optimal assignment of the remaining variables can be computed greedily in $O_\epsilon(\log^2 n)$ time. Thus, by exhaustive search over the $O_\epsilon(1)$ possible assignments to z, one can find an optimal solution to ILP (and hence the signature x of an optimal G-respecting τ-relaxed code) in $O_\epsilon(\log^2 n)$ time.

The integer linear program ILP is a modification of one of Karp's original integer programs [20, sect. IV] for HULC (that is, for finding a minimum-cost *prefix-free* code; in contrast we seek a G-respecting, τ-relaxed code). The variables of ILP are contained in four vectors (w, x, y, z), where x encodes the signature of the codeword set, w encodes the signature of the set of interior nodes, y encodes the assignment of probabilities to levels (y is determined by x, and helps compute the cost), and z encodes the assignment of groups to levels (for levels less than τ). The basic idea (following Karp) is that, since the numbers of various types of nodes available on level i satisfy natural linear recurrences in terms of the numbers at lesser levels, we can model the possible signatures by linear constraints on x and w.

For intuition, we first describe Karp's original integer program for finding a *prefix-free* code (generalized trivially here to allow a universe $\mathcal{U}$ with arbitrary root set $\mathcal{R}$). The inputs to Karp's program are the probability distribution p along with the signatures σ and r of, respectively, the alphabet Σ and the root set $\mathcal{R}$. (Note that $m = n \max\{\text{cost}(\ell) \mid \ell \in \Sigma\}$ is a trivial upper bound on any codeword cost in any optimal code.) Karp's program is in Figure 6.

We call the first constraint in KARP the "capacity" constraint. Note that the vector z is not used in KARP.

THEOREM 2.8 (correctness of KARP, [20, sect. IV]). *In any optimal solution $(w^\star, x^\star, y^\star)$ of* KARP*, the vector $x^\star$ is the signature of a minimum-cost prefix-free code, the cost of which is the cost of $(w^\star, x^\star, y^\star)$.*

Proof sketch. For any prefix-free code $\mathcal{X}$, there is a feasible solution (w, x, y) for KARP of cost $\text{cost}(\mathcal{X})$. To see why, consider the tree representation of $\mathcal{X}$. Let x_i be the number of leaves in level i, let w_i be the number of interior nodes (in $\mathcal{U}$) in level i, and let $y_{ki} = 1$ if $\text{cost}(\mathcal{X}_k) = i$ and $y_{ki} = 0$ otherwise. (So y_{ki} indicates whether probability p_k is assigned to level i.) Taking (w, x, y) as a solution to KARP, the capacity constraint holds because each interior node on level j can have at most σ_{i-j} children in level i. By inspection, the other constraints are also met, and (w, x, y) has

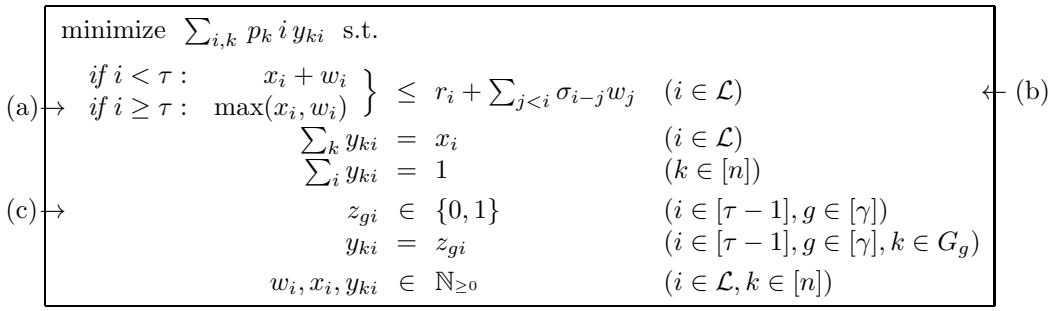

FIG. 7. *An integer program (ILP) for computing an optimal τ-relaxed, G-respecting code. Innovations* (a), (b), *and* (c) *are described in the text.*

cost equal to $\text{cost}(\mathcal{X})$.

Conversely, given any feasible solution (w, x, y), one can greedily construct a code $\mathcal{X}$ with signature x by building its tree representation level by level (in order of increasing $i \in \mathcal{L}$), adding w_i interior nodes and x_i codeword nodes in level i. The capacity constraint ensures that there are enough parents (and roots) to allocate each level's nodes. □

Next we modify KARP to model our problem: finding the signature x of a minimum-cost *G-respecting τ-relaxed* code (instead of a minimum-cost prefix-free code). The modified program, denoted ILP, is shown in Figure 7. The program differs from Karp's in three ways, labeled (a), (b), and (c) in the figure:

(a) *For i above the threshold τ, the left-hand side of the capacity constraint is replaced by* $\max(x_i, w_i)$. This models τ-relaxed codes, in which codeword nodes in level $i \ge \tau$ can also be interior nodes.

(b) *The indices i (and j) range over the set $\mathcal{L}$ of possible levels, instead of $[m]$ (per Definition* 2.6*).* Restricting i and j to levels within $\mathcal{L}$ is without loss of generality by Lemma 2.7.

(c) *There are $\tau\gamma$ new* 0/1 *variables: one variable z_{gi} for each group G_g ($g \le \gamma$) and level $i < \tau$.*

The new z variables enforce the restriction to G-respecting codes. Specifically, they constrain the y variables to force all probabilities within a given group to be assigned to the same level (if any is assigned to a level below τ): z_{gi} will be 1 if and only if group G_g is assigned to level $i < \tau$. (If a group is not assigned to any level below τ, then all its z_{gi}'s will be zero.)

Next we state the formal correctness of ILP: that the feasible solutions to ILP do correspond to the (signatures of the) G-respecting τ-relaxed codes.

LEMMA 2.9 (correctness of ILP). (i) *Given any minimum-cost τ-relaxed G-respecting code $\mathcal{X}$, the integer program* ILP *has a feasible solution (w, x, y, z) of cost* $\text{cost}(\mathcal{X})$*, where x is the signature of $\mathcal{X}$.*

(ii) *Conversely, given any solution (w, x, y, z) of* ILP*, there is a τ-relaxed G-respecting code $\mathcal{X}$ having signature x and with equal (or lesser) cost.*

Proof sketch. (A detailed proof is in the appendix.)

The proof is a simple extension of the proof of Theorem 2.8. In the forward direction, the capacity constraint is met because, in any τ-relaxed code, codeword nodes in levels τ and higher can also be interior nodes. In the backward direction, the code is G-respecting because of the constraint $y_{ki} = z_{gi}$ (for $g \le \gamma$, $k \in G_g$, and $i < \tau$). □

Remark. We remark without proof that the integrality constraints on w, x, and y (in the final line of ILP) can be dropped, giving a mixed integer linear program. (In any optimal basic feasible solution to the latter program, w, x, and y will still take only integer values.)

Note that a particular assignment of the z variables determines the assignment of groups in G within each level in $[0, \tau - 1]$. As previously discussed, this in turn essentially determines the rest of the τ-relaxed code, as codewords in levels τ and above should be chosen greedily. Thus, given any particular assignment of the variables in z, there is a natural optimal assignment of the remaining variables (w, x, y). We call this (w, x, y, z) the *greedy extension* of z. Here is the formal definition.

DEFINITION 2.10 (greedy extension). *Given any z with values in $\{0, 1\}$ such that $\sum_i z_{gi} \leq 1$ for each g, define the* greedy extension *of z for* ILP *to be the tuple $(\hat{w}, \hat{x}, \hat{y}, z)$ of all-integer vectors defined as follows:*

1. *In each level $i < \tau$, in increasing order, define $\hat{x}_i$ and $\hat{w}_i$ as follows. Let $\hat{x}_i$ be the number of probabilities that z assigns to level i; that is, $\hat{x}_i = \sum_{g:z_{gi}=1} |G_g|$. Let $\hat{w}_i$ be the number of interior nodes left available in level i. That is, let $\hat{w}_i$ be maximal subject to the capacity constraint.*

2. *For each level $i \geq \tau$, in increasing order, take interior and codeword nodes greedily: take $\hat{x}_i$ and $\hat{w}_i$ to be maximal subject to the capacity constraint for i and the constraint $\sum_{j \leq i} \hat{x}_j \leq n$.*

3. *Among vectors y such that the tuple $(\hat{w}, \hat{x}, y, z)$ is feasible for* ILP*, let $\hat{y}$ be one giving minimum cost (breaking any ties by assigning probabilities with lesser indices to lesser levels).*

Note: In step 1, if it happens that the capacity constraint is violated even with $\hat{w}_i = 0$, then there is no G-respecting τ-relaxed code for the given z, and the greedy extension of z is not well defined.

In step 2, if it happens that some probabilities are not assigned to any level below τ (i.e., $\sum_{i<\tau} \hat{x}_i < n$) but no nodes are available in higher levels (i.e., for all $i \geq \tau$ the right-hand side of the capacity constraint is 0), then there is no G-respecting τ-relaxed code for the given z, and the greedy extension of z is not well defined.

Since codewords in levels τ and higher should be assigned greedily, the greedy extension is optimal, as shown next.

LEMMA 2.11 (optimality of greedy extension). *Fix any z for which there is any feasible extension (x, w, y, z) for* ILP*. Then the greedy extension $(\hat{w}, \hat{x}, \hat{y}, z)$ of z is well-defined, feasible, and has minimum cost.*

The proof is straightforward; it is given in the appendix.

The next corollary summarizes what is needed from this section.

COROLLARY 2.12 (correctness of ILP). *Fix any instance of* HULC.

(i) *Fix any z that has some feasible extension for* ILP*. Then the greedy extension $(\hat{w}, \hat{x}, \hat{y}, z)$ of z is well defined, feasible, and has minimum cost.*

(ii) *Let $(w^\star, x^\star, y^\star, z^\star)$ be an optimal solution to* ILP*. Then $x^\star$ is the signature of a minimum-cost G-respecting τ-relaxed code.*

Part (i) of the corollary is just Lemma 2.11. Part (ii) follows from Lemma 2.9.

2.5. Proof of Theorem 2.1. We now prove Theorem 2.1, which is restated here for convenience.

THEOREM 2.1. *Fix any instance $\mathcal{I} = (p, \Sigma, \text{cost}, \mathcal{U})$ of* HULC *with restricted universe such that $\text{cost}(1) \leq 3/\epsilon$. Let P be the cumulative probability distribution for p: $P_\ell = \sum_{k \leq \ell} p_k$ (for $\ell \in [n]$). Let σ be the signature of Σ. Let r be the signature of the roots of $\mathcal{U}$. Assume that P, σ, and r are given as inputs.*

0. Let $\tau = \lceil \mathrm{cost}(1) \log_2(\mathrm{cost}(1)/\epsilon)/\epsilon \rceil$.

1. Compute grouping $G = G(p)$ (Definition 2.4) and set of levels $\mathcal{L}$ (Definition 2.6).

2. For each possibly feasible assignment $\hat{z}$ to z in ILP:

2a. compute just $\hat{w}$ and $\hat{x}$ of the greedy extension of $\hat{z}$ (Definition 2.10);

2b. from $\hat{x}$, compute the cost of the greedy extension of $\hat{z}$ (if well defined).

Select $(w^\star, x^\star, z^\star)$ to be the $(\hat{w}, \hat{x}, \hat{z})$ giving minimum cost among those computed.

3. Without explicitly computing the τ-relaxed code $\mathcal{X}$ with signature $x^\star$, compute the signature x' and approximate cost of the prefix-free code $\mathcal{X}' = \mathsf{Round}(\mathcal{X})$.

FIG. 8. *The steps of the PTAS for the case* $\mathrm{cost}(1) \le 3/\epsilon$.

Then the signature and approximate cost of a prefix-free code (for $\mathcal{I}$) with cost at most $(1 + O(\epsilon))\,\mathrm{OPT}(I)$ can be computed in time $O_\epsilon(\log^2 n)$.

Proof. By Lemmas 2.3–2.7 and Corollary 2.12, the steps in Figure 8 give the signature x' and cost.

To finish, we show that each of these steps can be done in $O_\epsilon(\log^2 n)$ time, given P, σ, and r.

Step 1. Compute G (in particular, the first and last index of each group G_g) as follows. By inspection of Definition 2.4, for each group $G_g = (j, \ldots, h)$, the index h can be computed in $O(\log n)$ time from P by binary search. There are at most τ/ϵ groups, so the total time is $O((\tau/\epsilon)\log n) = O_\epsilon(\log n)$.

Compute $\mathcal{L}$ in time $O(|\mathcal{L}|) = O_\epsilon(\log n)$ as follows. Following Definition 2.6, compute $i_\mathcal{R}$ and i_Σ in $O(\tau) = O_\epsilon(1)$ time (assuming r and σ are given as sorted lists or arrays indexed by i), and then enumerate $\mathcal{L}$.

Step 2. There are at most $\tau^{|G|} = O_\epsilon(1)$ possibly feasible assignments to z. (An assignment chooses a level in $[\tau - 1]$, or no such level, for each group index $g \in [\gamma]$; although ILP allows other assignments to z in which $\sum_i z_{gi} > 1$, none of those will have a feasible extension because they force $\sum_i y_{ki} > 1$ for $k \in G_g$.)

For each such assignment $\hat{z}$, to compute just $\hat{w}$ and $\hat{x}$ of the greedy extension (Definition 2.10), observe that all $\hat{x}_i$ with $i < \tau$ can be set in total time $O(|G|) = O(\gamma) = O_\epsilon(1)$ using $\hat{x}_i = \sum_{g:\hat{z}_{gi}=1} |G_g|$. Then, the $\hat{w}_i$ (for $i \in \mathcal{L}$) and the $\hat{x}_i$ (for $i \in \mathcal{L}$, $i \ge \tau$) can each be computed in time $O(|\mathcal{L}|)$ (the time it takes to compute $\sum_{j<i} \sigma_{i-j} w_j$), for a total time of $O(|\mathcal{L}|^2) = O_\epsilon(\log^2 n)$.

Given $\hat{z}$ and $\hat{x}$, the cost of the code can then be computed (without computing y!) as follows. The probability associated with a group G_g is $P[\max G_g] - P[\max G_{g-1}]$. The contribution of levels less than τ to the cost is $\sum_g \sum_{i<\tau} i\,\hat{z}_{gi}(P[\max G_g] - P[\min G_g - 1])$.

The cumulative cost of codewords in levels $i \ge \tau$ can be computed as follows. Consider those groups G_g that are not assigned to the lower levels, in order of increasing g. Break the groups as necessary into smaller pieces, while assigning the pieces monotonically to the levels $i = \tau, \tau + 1, \ldots$, so that each level i is assigned pieces of total size x_i. (At most $|G| + |\mathcal{L}| - \tau$ pieces will be needed to do this.) Once all pieces are assigned levels, compute their cumulative cost as the sum, over the pieces, of the cumulative probability in the piece times the assigned level. In this way, the cost of

the code for a given $\hat{z}$ and $\hat{x}$ can be computed in time $O(|G|\tau + |G| + |\mathcal{L}|) = O_\epsilon(\log n)$.

Since there are $O_\epsilon(1)$ assignments $\hat{z}$ to consider and, for each, $\hat{x}$ can be computed in $O_\epsilon(\log^2 n)$ time, the total time to find the minimum-cost signature x is $O_\epsilon(\log^2 n)$.

Step 3. By inspection of Round in the proof of Lemma 2.3, for each codeword of cost $i \geq \tau$ in $\mathcal{X}$, there is a codeword of cost $i + \text{cost}(\text{enc}(i - \hat{\tau}))$ in $\mathcal{X}'$. Thus, x' can be computed directly from x by taking $x'_i = x_i$ for $i < \tau$, and for the rest, starting with $x'_i = 0$ and then, for each $i \geq \tau$, incrementing $x'_{i'}$ by x_i, where $i' = i + \text{cost}(\text{enc}(i - \hat{\tau}))$.

The cost of $\mathcal{X}'$ is $1 + O(\epsilon)$ times the cost of the τ-relaxed code with signature x, which is, in turn, the cost of the solution (w, x, y, z) to ILP, which is known from the previous step.

This completes the proof of Theorem 2.1. □

The following observations about the proof are useful in the next section. By Lemma 2.7, the code whose signature is produced has support in $\mathcal{L}$. Thus, the tree representation uses only the roots of $\mathcal{U}$ that lie in levels in $\mathcal{L}$. Similarly, by inspection of ILP, its solution requires only those r_i with $i \in \mathcal{L}$. We summarize as follows.

OBSERVATION 2.13. *The computation in Theorem* 2.1 *produces a signature x for a code with support in $\mathcal{L}$. The computation does not require the full signature r of the roots of $\mathcal{U}$, but relies only on the r_i such that $i \in \mathcal{L}$ (the set $\mathcal{L}$ of possible levels from Definition* 2.6*).*

3. Computing the tree representation from the signature. For the case $\text{cost}(1) \leq 3/\epsilon$, Theorem 2.1 proves that the signature (and cost) of a near-optimal prefix-free code can be efficiently computed, but says nothing about computing a more explicit representation of the code. Here we address this by proving Theorem 3.1, which describes how to compute the tree representation in $O(n) + O_\epsilon(\log^2 n)$ time, given the signature x.

Given the signature, it would be easy to compute the tree-representation F using a root-to-leaves greedy algorithm in time $O(|F| + |\mathcal{L}|)$ (where $|F|$ is the number of nodes in F). Roughly, one could just allocate the nodes and edges of F appropriately in order of increasing level $i \in \mathcal{L}$. Unfortunately, F might not have size $O(n)$, because in the worst case it may have many long chains of interior nodes, each with just one child.[3]

One could of course modify F, splicing out nodes with just one child, so as to build a new tree F' whose size is $O(n)$ and whose cost is less than or equal to the cost of F. However, if the algorithm were to explicitly build F from the signature, and then modify F into F' as described, it would still take time at least $O(|F|)$, which could be excessive. To prove the theorem below, we describe how to bypass the intermediate construction of F, instead building F' directly from x, in time $O(|F'|) + O_\epsilon(\log^2 n)$, where $|F'| = O(n)$.

THEOREM 3.1. *Given any instance $\mathcal{I} = (p, \Sigma, \text{cost}, \mathcal{U})$ of* HULC *with restricted universe such that* $\text{cost}(1) \leq 3/\epsilon$*, and given the signature x of some prefix-free code $\mathcal{X}$ with support in $\mathcal{L}$, one can construct the tree representation of a prefix-free code $\mathcal{X}'$ that has cost at most* $\text{cost}(\mathcal{X})$*. The running time is $O(n) + O_\epsilon(\log^2 n)$. The tree representation has $O(n)$ nodes.*

Proof. Starting from the signature x, we first compute various signatures for a tree F whose codeword nodes have signature x. Specifically, we compute both w (the signature of the interior nodes of F) and an "edge signature" e—where e_{ji} is the number of edges from level j to level $i > j$ in F. In fact the signature x does not

[3]Indeed, for some instances, there are signatures that force this to happen.

uniquely determine e or w, so we make some arbitrary choices to fix a particular F with codeword signature x.

Here are the details of how to compute w and e in time $O(|\mathcal{L}|^2)$.

1. To start, initialize vector w so that the capacity constraint for KARP (on the left below) holds with x:

the capacity constraint for KARP	constraints defining edge signature e
$x_i + w_i \;\le\; r_i + \sum_{j<i} \sigma_{i-j} w_j \quad (i \in \mathcal{L})$	$x_i + w_i \;\le\; r_i + \sum_{j<i} e_{ji} \quad (i \in \mathcal{L})$ $w_{ji} \;=\; \lceil e_{ji}/\sigma_{i-j} \rceil \quad (i, j \in \mathcal{L}, j < i)$ $w_j \;=\; \max_{i>j} w_{ji} \quad (j \in \mathcal{L})$

(Achieve this as follows. For each $i \in \mathcal{L}$, in increasing order, choose w_i maximally subject to the ith capacity constraint. This assignment to w will satisfy the capacity constraints (with x) if any assignment to w can.)

2. In the edge-signature constraints on the right above, e_{ji} represents the number of edges from level j to level $i > j$, and w_{ji} represents the number of interior nodes in level j with children in level $i > j$. Initialize the edge signature e and the w_{ji}'s so that these constraints are met. (To do this, take $e_{ji} = \sigma_{i-j} w_j$ and $w_{ji} = w_j$ for all i and j. Since the capacity constraints for KARP are satisfied by x and w, by inspection, the edge-signature constraints for e on the right above will also be satisfied.)

3. Next, *lower* w, e, and possibly r so that all of the edge-signature constraints above are tight. (Achieve this by mimicking a leaves-to-root scan over the tree that deletes "unused" interior nodes and edges, as follows. For each $j \in \mathcal{L}$, in *decreasing* order, for each $i \in \mathcal{L}$ with $i > j$, lower e_{ji} as much as possible subject to the first edge-signature constraint for $i \in \mathcal{L}$, then update w_{ji} and w_i. Finally, if the first edge-signature constraint for some $i \in \mathcal{L}$ is still loose, it must be that $\sum_{j<i} e_{ji} = 0$, so lower r_i to $x_i + w_i$ to make the constraint tight.)

4. In F, if for some edge (a, b), b is a's only child, then call the node a *useless*. (Contracting such edges would give a better code.) Call all other nodes (including codeword nodes) *useful*. For each j, count the number u_j of useless nodes in level j as follows. For definiteness, order the level-j nodes arbitrarily and assume that, for each $i, j \in \mathcal{L}$ with $i > j$, the nodes in level j that have children in level i are the *first* w_{ji} interior nodes in level j, and that all but the last of these w_{ji} nodes has the maximum possible number (σ_{j-i}) of children in level i (so that the last such node has $e_{ji} \bmod \sigma_{j-i}$ children in level i). Then count the useless nodes in level j as follows. Let $i' = \arg\max_i w_{ji}$ and $i'' = \arg\max_{i \neq i'} w_{ji}$ be the two levels having the most and second-most children of nodes in level j. (So $w_i = w_{ji'} \geq w_{ji''}$.) If it happens that $\sigma_{j-i'} = 1$, then the last $w_j - w_{ji''}$ level-j interior nodes have only one child, so $u_j = w_j - w_{ji''}$. Otherwise ($\sigma_{j-i'} \geq 2$), only the last level-j interior node can have just one child (because all others have $\sigma_{j-i'}$ edges to level i'). The number of level-i' children of that last node is $e_{ji'} \bmod \sigma_{j-i'}$. If this quantity is 1 and $w_{ji''} < w_i$ (the node has no children in level i''), then $u_j = 1$, and otherwise $u_j = 0$.

5. Define F' to be the subforest of F induced by useful nodes and their children. Explicitly construct F' as follows. For each level $j \in \mathcal{L}$ in *decreasing* order, do the following. Create the x_j codeword nodes and the $w_j - u_j$ nonuseless interior nodes. Then, following the description of the edges in F from step 4 above, for each $i > j$, add up to e_{ji} edges greedily from each of the first $\min(w_{ji}, w_j - u_j)$ interior nodes (adding at most σ_{i-j} edges from each node) to parentless nodes in level i (giving those nodes parents). If there are not enough parentless nodes in level i to do this, create

new *childless* interior nodes in level i as needed (these new nodes are useless children of nonuseless nodes; in step 6, below, they are the *stubs*). Among all $x_j + w_j - u_j$ new nodes instantiated in level j, designate as many as possible $(\min(x_j + w_j - u_j, r_j))$ as roots, and designate the rest as (temporarily) parentless. Nonroot nodes might be left parentless (these are nodes whose parents were useless in F; in step 6, they are the *orphans*).

6. Next consider the nonroot parentless nodes in F' (call these *orphans*), and the (useless) childless interior nodes in F' (call these *stubs*). The nodes in $F - F'$ are interior nodes with one child whose parents also have one child, so in F the nodes in $F - F'$ form vertex-disjoint paths connecting each orphan d to a unique stub $A(d)$ (the child of d's first nonredundant ancestor in F). Thus, the number of orphans equals the number of stubs. Make a list $a_1, a_2, \ldots, a_k$ of the stubs, and a list $d_1, d_2, \ldots, d_k$ of the orphans, both ordered by increasing level (breaking ties arbitrarily). Finally, modify F' as follows. For each pair of nodes (a_j, d_j), identify a_j and d_j—that is, make d_j the child of a_j's parent in place of a_j. The resulting forest is F''.

Correctness. Let $\mathcal{X}'$ be the monotone code with tree representation F''. By construction, $\mathcal{X}'$ is prefix-free, has codewords in $\mathcal{U}$, and has signature x. To prove that F'' has cost no greater than the cost of F, we observe that each leaf node in F has a corresponding leaf node in F'' and observe that, in the last step of the construction, going from F' to F'', cannot increase the level of any orphan d_j. Indeed, suppose for contradiction that the level of some d_j in F is strictly less than the level of its paired node a_j. Thus, the j stub nodes $A(d_1), A(d_2), \ldots, A(d_j)$ are in levels strictly less than the level of a_j. Each of these j nodes must precede a_j in the ordering $a_1, a_2, \ldots, a_k$ of stub nodes, but only $j - 1$ nodes can do so.

Time. The time for constructing x, w, and e is $O(|\mathcal{L}|^2) = O_\epsilon(\log^2 n)$. By inspection, the forest F'' can be constructed from w, x, and e in time $O(|\mathcal{L}|^2 + |F''|)$. In F'' there are n leaves, and each interior node has at least two children, so $|F''| \le 2n$. ∎

4. Computing the signature of a near-optimal code when cost(1) $\ge 3/\epsilon$. The preceding sections give a complete PTAS for instances of HULC with cost(1) $\le 3/\epsilon$. In this section, the goal is to extend the PTAS to handle arbitrary letter costs. Note that if the letter costs were fixed (not part of the input), then for small (but still constant) ϵ it would be the case that cost(1) $\le 3/\epsilon$, so the PTAS in the preceding sections could be applied as it stands. But since letter costs are part of the input, as we've defined HULC, we cannot assume that cost(1) is constant; we have to handle the case when cost(1) grows asymptotically.

Unfortunately, the PTAS in the preceding sections makes fundamental use of the assumption that cost(1) $= O_\epsilon(1)$. Indeed, that restriction is what ensures that the relaxation gap for τ-relax is $1 + O(\epsilon)$ for some threshold $\tau = O_\epsilon(1)$. In turn, using a threshold τ with value $O_\epsilon(1)$ is central to the polynomial running time. This approach does not seem to extend to handle instances in which the ratio cost(0)/ cost(1) is quite small (e.g., decreasing with n). We need another approach for handling the case when cost(0) is quite small.

4.1. Reducing to coarse letter costs. We start with a simple scaling and rounding step (a standard technique in PTAS's), to bring the letter costs into a restricted form that is easier to work with. Ideally, we would like to make (i) all letter costs integers and (ii) cost(1) $\le 3/\epsilon$, for then the preceding PTAS would apply. We almost achieve these two conditions, failing only in that cost(0) may end up being noninteger. More specifically, we scale and round the costs to make them *coarse*, as follows.

DEFINITION 4.1. *The letter costs are* coarse *if*

- *the second-cheapest letter cost,* $\text{cost}(1)$*, is in the interval* $[1/\epsilon, 3/\epsilon]$*; and*
- *all letter costs are integers, except possibly* $\text{cost}(0)$*, which may instead be the reciprocal of an integer.*

Note well that *throughout this section* $\text{cost}(0)$ *is not necessarily an integer*—it may instead be the reciprocal of an integer, i.e., $\text{cost}(0) = 1/N$ for some integer N. All other letter costs are still integers.

Here are the specific scaling and rounding steps that we use to achieve coarse letter costs.

Subroutine 4.1. *Coarsening the letter costs.*

1: **if** $\text{cost}(1)/\text{cost}(0) \geq 1/\epsilon$, **then**
2: Let N be the maximum integer such that $\frac{\text{cost}(1)}{N\,\text{cost}(0)} \geq 1/\epsilon$.
Initialize $\text{cost}'(\ell) = \frac{\text{cost}(\ell)}{N\,\text{cost}(0)}$ for $\ell \in \Sigma$.
3: **else**
4: Let N be the minimum integer such that $\frac{N\,\text{cost}(1)}{\text{cost}(0)} \geq 1/\epsilon$.
Initialize $\text{cost}'(\ell) = \frac{N\,\text{cost}(\ell)}{\text{cost}(0)}$ for $\ell \in \Sigma$.
5: For each $\ell \in \Sigma$ except $\ell = 0$, round $\text{cost}'(\ell)$ to the integer $\lceil \text{cost}'(\ell) \rceil$.
6: Return cost'.

To conclude section 4.1 we prove that the above procedure does indeed produce coarse letter costs in linear time, and that any instance with arbitrary costs reduces (in an approximation-preserving way) to the same instance but with coarsened costs.

LEMMA 4.2. *Let* $\text{cost}' : \Sigma \to \mathbb{R}_{\geq 0}$ *be the costs output by the coarsening subroutine (given arbitrary letter costs* $\text{cost} : \Sigma \to \mathbb{R}_{\geq 0}$*). Then the following hold:* (i) *The subroutine takes* $O(n)$ *time.* (ii) *The costs* cost' *are coarse.* (iii) *Any code that is near-optimal under* cost' *is also near-optimal under* cost.

Proof. Part (i) is clear by inspection and the assumption that $|\Sigma| \leq n$.

(ii) If the condition in the "if" statement holds (that is, $\text{cost}(1)/\text{cost}(0) \geq 1/\epsilon$), the scaling step makes $\text{cost}'(0)$ $\left(= \frac{\text{cost}(0)}{N\,\text{cost}(0)} = 1/N\right)$ the reciprocal of an integer. Also, the scaling step brings $\text{cost}'(1)$ into the interval $[1/\epsilon, 2/\epsilon)$, because, by the choice of N,

$$\tfrac{1}{2}\,\text{cost}'(1) \leq \tfrac{N}{N+1}\,\text{cost}'(1) = \tfrac{N}{N+1}\,\tfrac{\text{cost}(1)}{N\,\text{cost}(0)} = \tfrac{\text{cost}(1)}{(N+1)\,\text{cost}(0)} < \tfrac{1}{\epsilon} \leq \tfrac{\text{cost}(1)}{N\,\text{cost}(0)} = \text{cost}'(1).$$

Alternatively, if the "else" clause is executed, the scaling step makes $\text{cost}'(0)$ an integer and brings $\text{cost}'(1)$ into the interval $[1/\epsilon, 2/\epsilon)$ because $N \geq 2$ and

$$\tfrac{1}{2}\,\text{cost}'(1) \leq \tfrac{N-1}{N}\,\text{cost}'(1) = \tfrac{N-1}{N}\,\tfrac{N\,\text{cost}(1)}{\text{cost}(0)} = \tfrac{(N-1)\,\text{cost}(1)}{\text{cost}(0)} < \tfrac{1}{\epsilon} \leq \tfrac{N\,\text{cost}(1)}{\text{cost}(0)} = \text{cost}'(1).$$

In either case, the final rounding step (line 5) makes every $\text{cost}'(\ell)$ (for $\ell \geq 1$) an integer. The rounding step also leaves $\text{cost}'(1) \leq 3/\epsilon$, because $\text{cost}'(1) \leq 2/\epsilon$ before rounding and $\lceil 2/\epsilon \rceil \leq 3/\epsilon$ for $\epsilon \leq 1$.

(iii) The scaling steps (lines 1–4) do not change the ratio of any two letter costs. The rounding step changes the relative costs of any two letters by at most a factor of $1 + \epsilon$, because, before rounding, each rounded letter cost, $\text{cost}'(\ell)$, is at least $1/\epsilon$ and so increases by at most a $1 + \epsilon$ factor. Thus, any prefix-free code $\mathcal{X}$ is a near-optimal solution under $\text{cost}'()$ if and only if it is a near-optimal solution under $\text{cost}()$. ∎

4.2. Reducing to coarse letter costs with cost(0) ≥ 1. Appealing to Lemma 4.2, we can now assume without loss of generality that the letter costs are

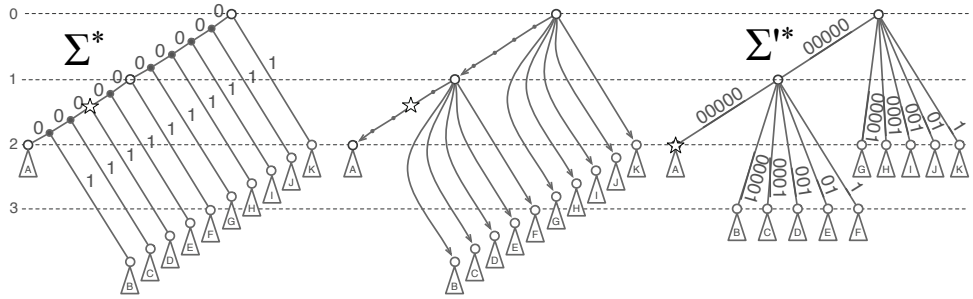

FIG. 9. *Given* $\Sigma = \{0, 1\}$ *with* $\text{cost}(0) = 1/N = 1/5$ *and* $\text{cost}(1) = 1/\epsilon = 2$*, the top of* Σ^* *is on the left, the top of* Σ'^* *is on the right. The "chunk" alphabet* Σ' *has six letters, called "chunks": the cheapest chunk,* $\underline{00000}$*, represents the string* 00000 *in* Σ^**, and costs* 1*. The other five chunks (*$\underline{1}$*,* $\underline{01}$*,* $\underline{001}$*,* $\underline{0001}$*, and* $\underline{00001}$*) represent, respectively, the strings* 1*,* 01*,* 001*,* 0001*, and* 00001*. Each costs* $2 = \text{cost}(1)$*.*

coarse. That is, we assume that $\text{cost}(1) \in [1/\epsilon, 3/\epsilon]$ and that all letter costs are integers except perhaps cost(0), which may instead be the reciprocal of an integer.

If it does happen that cost(0) is an integer, then the condition for the PTAS of the preceding sections is met: all letter costs are integers, and $\text{cost}(1) \leq 3/\epsilon$. So in this case we can apply that PTAS directly to the instance.

So, assume that cost(0) is not an integer. That is, cost(0) equals $1/N$ for some integer $N \geq 2$.

We now confront the core problem of this section: how to deal with an instance in which cost(0) is very small in comparison to cost(1). To handle such an instance, the basic idea is to reduce the problem to the case we've already solved. In particular, we replace the given alphabet by a new alphabet Σ', in which each new letter $\underline{s}$ represents some *string* s over the original Σ. This idea allows us to manipulate the letter costs: by choosing large enough strings s to represent, we can make sure no letter cost in Σ' is too small.

For intuition, consider an example with binary alphabet $\Sigma = \{0, 1\}$. Consider replacing this alphabet with an alphabet Σ' containing the six letters $\underline{00000}$, $\underline{1}$, $\underline{01}$, $\underline{001}$, $\underline{0001}$, and $\underline{00001}$. Call these letters *chunks*. They represent, respectively, the four strings 00000, 1, 01, 001, 0001, and 00001 over Σ. In this way, each string of chunks (i.e., string over Σ') represents a string over Σ in a natural way, For example, the string "$\underline{1}$ $\underline{00000}$ $\underline{01}$" over Σ' represents the string "10000001" over Σ. See Figure 9.

For letter costs, it would be natural to take $\text{cost}(\underline{s})$ equal to the cost of the string over Σ that $\underline{s}$ represents. For the example, if cost(0) is 1/5, it would be natural to take $\text{cost}(\underline{00000}) = 1$, $\text{cost}(\underline{1}) = \text{cost}(1)$, $\text{cost}(\underline{01}) = \frac{1}{5} + \text{cost}(1)$, $\text{cost}(\underline{001}) = \frac{2}{5} + \text{cost}(1)$, etc. But, since our goal is to have all-integer letter costs, we instead round down the costs: $\text{cost}(\underline{00000}) = 1$, $\text{cost}(\underline{1}) = \text{cost}(1)$, $\text{cost}(\underline{01}) = \text{cost}(1)$, $\text{cost}(\underline{001}) = \text{cost}(1)$, etc. Because $\text{cost}(1) \geq 1/\epsilon$, rounding down doesn't alter the "natural" costs by more than a $1 + \epsilon$ factor.

In general, for an arbitrary alphabet Σ, where, say, $\text{cost}(0) = 1/N$, here is how we construct Σ'.

DEFINITION 4.3 (chunk alphabet). *Let* chunk alphabet Σ' *contain the following letters (called* chunks*): one letter denoted* $\underline{0^N}$ *and, for each nonzero letter* $\ell \in \Sigma$*,* N *letters denoted* $\underline{\ell}, \underline{0\ell}, \ldots, \underline{0^{N-1}\ell}$*. (Each underlined string* $\underline{0^i\ell}$ *denotes a single letter in* Σ'*.) Give letter* $\underline{0^N}$ *cost* 1*, and give each letter* $\underline{0^i\ell}$ *cost equal to* $\text{cost}(\ell)$*.*

For any string s' over Σ', let unchunk(s') denote the string over Σ that s' represents. Say a string s over Σ is *chunkable* if $s = \text{unchunk}(s')$ for some s' over Σ'. (These are the strings over Σ that can be cleanly broken into chunks.)

Extending from strings to codes, each code $\mathcal{X}'$ over Σ' represents a code $\mathcal{X}$ over Σ in a natural way, specifically $\mathcal{X}_i = \text{unchunk}(\mathcal{X}'_i)$. Let unchunk($\mathcal{X}'$) denote this code $\mathcal{X}$. Say that a code $\mathcal{X}$ over Σ is *chunkable* if it can be obtained in this way (i.e., all its codewords are chunkable).

Thus, unchunk() gives a bijection between the strings over Σ' and the chunkable strings over Σ. Likewise, it gives a bijection between the codes over Σ' and the chunkable codes over Σ. On consideration, unchunk($\mathcal{X}'$) will be prefix-free if and only if $\mathcal{X}'$ is prefix-free. Thus, this bijection preserves prefix-free-ness and (approximate) cost.

First attempt at PTAS via reduction. The general scheme will be something like the following:

---

(1) Given Σ, construct the chunk alphabet Σ'.
(2) Find a near-optimal prefix-free code $\mathcal{X}'$ over Σ' using PTAS for cost(1) $\leq 3/\epsilon$.
(3) Return the prefix-free code $\mathcal{X} = \text{unchunk}(\mathcal{X}')$ that $\mathcal{X}'$ represents.

The main flaw in this reduction is the following: not all strings over Σ can be broken into chunks from Σ'. In particular, the codewords in the optimal code $\mathcal{X}^\star$ over Σ might not be chunkable. Thus, even if $\mathcal{X}'$ is near-optimal over Σ' a priori, it may happen that unchunk($\mathcal{X}'$) is far from optimal over Σ.

The main technical challenge in this section is to understand this flaw and work around it. To understand the flaw in detail, recall that the codes over Σ' correspond, via the bijection unchunk(), to the chunkable codes over Σ, and this bijection preserves prefix-free-ness and (approximate) cost.

Because of this bijection, the reduction proposed above (after Definition 4.3) will work if and only if the optimal prefix-free code $\mathcal{X}^\star$ over Σ is has approximately the same cost as the optimal *chunkable* prefix-free code over Σ (since the latter code has approximately the same cost as the optimal prefix-free code over Σ'). So, is there always a chunkable prefix-free code whose cost is near that of the optimal prefix-free code $\mathcal{X}^\star$?

Let's consider which strings over Σ are chunkable (that is, can be broken into chunks from Σ'). On consideration,[4] a necessary and sufficient condition for a string s over Σ to be chunkable is that the number of 0's at the end of s should be a multiple of N. Thus, a given code $\mathcal{X}$ over Σ is chunkable if and only if all of its codewords end nicely in that way. Define pad($\mathcal{X}$) to be the code over Σ obtained by padding each codeword in $\mathcal{X}$ with just enough 0's so that the number of 0's at the end of the codeword is a multiple of N.

Then pad($\mathcal{X}^\star$) is a prefix-free, chunkable code over Σ. But how much can padding increase the cost of $\mathcal{X}^\star$? Padding a codeword adds at most $N-1$ 0's to the codeword. This increases each codeword cost by at most $(N-1)\,\text{cost}(0) = (N-1)/N < 1$.

Is this significant? That is, can it increase the cost of the codeword by more than a $1+\epsilon$ factor? In order for this to happen, the codeword must have cost less than $1/\epsilon$. Call any such codeword (of cost less than $1/\epsilon$) a *runt*. Recalling that cost(ℓ) $\geq 1/\epsilon$ for every letter $\ell \in \Sigma - \{0\}$, for a codeword in $\mathcal{X}^\star$ to be a runt it must consist only of

[4] A string with this property can be broken into chunks as follows: first break the string after each occurrence of each nonzero letter, leaving pieces of the form $0^i\ell$ for some i, plus a final piece of the form 0^{iN} for some i; then, within each such piece, break the piece after every Nth 0.

0's. In any prefix-free code, there is either one runt or none, and the only codeword that can be the runt is the cheapest one, $\mathcal{X}_1^\star$.

In sum, the reduction above fails, but just barely, and the reason that it fails is because padding the runt can, in the worst case, increase the cost of the code by too much.

Second attempt. To work around this issue, we handle the runt differently: we use exhaustive search to remove it from the problem, then solve the remaining runt-free problem as described above.

More specifically, we consider all possibilities for the runt in the optimal code: either the optimal code has no runt (in which case the reduction in the first attempt above works), or the optimal code has a runt of the form 0^q for some $q \le n$ such that $\text{cost}(0^q) < 1/\epsilon$. For each possible choice 0^q for $\mathcal{X}_1$, we compute a near-optimal choice for the $n-1$ remaining codewords $\mathcal{X}_2, \mathcal{X}_3, \ldots, \mathcal{X}_n$, given that $\mathcal{X}_1 = 0^q$. We then return the best code found in this way.

How do we find a near-optimal choice for the $n-1$ *remaining* codewords, given a particular choice 0^q for $\mathcal{X}_1$? This problem can be stated precisely as follows:

(4.1) Find a near-optimal prefix-free code of $n-1$ codewords over alphabet Σ, for probabilities $p' = \langle p_2, p_3, \ldots, p_n\rangle/(1-p_1)$, *from the universe $\mathcal{U}_q$ of strings that do not have 0^q as a prefix.*

Since padding any nonrunt codeword to make it chunkable increases its cost by at most a $1+\epsilon$ factor and maintains prefix-free-ness, the problem above reduces in an approximation-preserving way to the following one:

(4.2) Find a near-optimal prefix-free code of $n-1$ codewords over alphabet Σ, for probabilities $p' = \langle p_2, p_3, \ldots, p_n\rangle/(1-p_1)$, from the universe $\hat{\mathcal{U}}_q$ of *chunkable* strings that do not have 0^q as a prefix.

Since the chunkable strings over Σ correspond via the bijection unchunk() to the strings over chunk alphabet Σ', and this bijection preserves prefix-free-ness and approximate cost, the problem above in turn reduces in an approximation-preserving way to the following problem:

(4.3) Find a near-optimal prefix-free set of $n-1$ codewords over *chunk alphabet* Σ', for probabilities $p' = \langle p_2, p_3, \ldots, p_n\rangle/(1-p_1)$, from universe $\mathcal{U}'_q$ of strings s *such that* unchunk(s) *does not have* 0^q *as a prefix.*

Note that the chunk alphabet Σ' in the latter problem (4.3) has integer letter costs, and the second cheapest letter cost is $\text{cost}(\underline{1}) = \text{cost}(1)$, which is in $[1/\epsilon, 3/\epsilon]$. These letter costs are appropriate for the PTAS from the preceding sections. We solve problem (4.3) using that PTAS.

To do so we have to limit the codeword universe $\mathcal{U} = \mathcal{U}'_q$ to those "strings s such that unchunk(s) does not have 0^q as a prefix." The basic idea is to choose an appropriate root set $\mathcal{R}'_q$ for $\mathcal{U}'_q$. For intuition, consider an example with binary alphabet $\Sigma = \{0,1\}$, with $\text{cost}(0) = 1/5$ and $\text{cost}(1) = 2$, shown in Figure 10. The strings over Σ are shown to the left; the strings over the chunked alphabet Σ' are shown to the right. A potential runt 0^7 is marked with $\star$. The strings having 0^7 as a prefix (on the left) and the corresponding strings over Σ' (s such that unchunk(s) has 0^7 as a prefix, on the right) are gray.

The remaining (allowed) strings are those in the subtrees marked $E, F, \ldots, K$ (on both the left and the right). The roots of these subtrees are the roots of $\mathcal{U}'_7$.

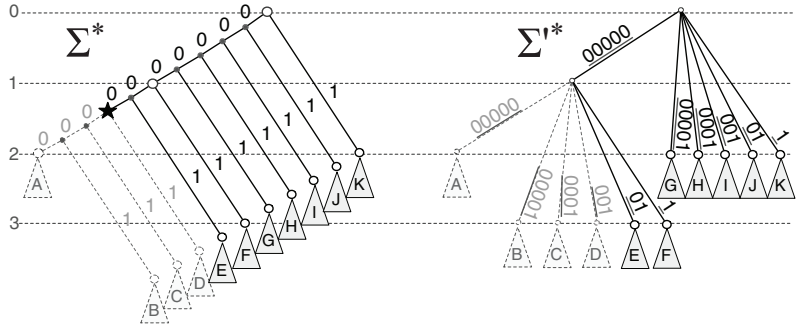

FIG. 10. *An example of chunking. The alphabet is* $0, 1$*, with* $\text{cost}(0) = 1/5$ *and* $\text{cost}(1) = 2$.

In general, given any alphabet Σ where $\text{cost}(0) = 1/N$ for some integer N, and given an arbitrary runt 0^q, we compute the root set $\mathcal{R}'_q$ for the desired universe $\mathcal{U}'_q$ as follows.

Let chunk() denote the functional inverse of unchunk(): if string s is chunkable, then $\text{chunk}(s)$ is the string s' over Σ' such that $\text{unchunk}(s') = s$; likewise, if code $\mathcal{X}$ is chunkable, then $\text{chunk}(\mathcal{X})$ is the code $\mathcal{X}'$ over Σ' such that $\text{unchunk}(\mathcal{X}') = \mathcal{X}$.

The universe $\mathcal{U}'_q$ should contain those strings s' such that $\text{unchunk}(s')$ does not have 0^q as a prefix. The chunkable strings over Σ that do not have 0^q as a prefix are those that start with a prefix of the form $0^i\ell$, where $i < q$ and $\ell \in \Sigma - \{0\}$. Each such string $0^i\ell$ is itself chunkable (as it ends in a letter other than 0). Thus, $\text{unchunk}(s')$ does not have 0^q as a prefix if and only if s' starts with a prefix of the form $\text{chunk}(0^i\ell)$, where $i < q$ and $\ell \in \Sigma - \{0\}$. That is, the universe $\mathcal{U}'_q$ has root set $\mathcal{R}'_q = \{\text{chunk}(0^i\ell) \ : \ i < q,\, \ell \in \Sigma - \{0\}\}$.

Thus, we can reformulate problem (4.3) with an explicit root set as follows:

(4.4)
Find a near-optimal prefix-free set of $n-1$ codewords over chunk alphabet Σ',
for probabilities $p' = \langle p_2, p_3, \ldots, p_n\rangle/(1-p_1)$,
from the universe $\mathcal{U}'_q$ with root set $\mathcal{R}'_q = \{\text{chunk}(0^i\ell) \ : \ i < q,\, \ell \in \Sigma - \{0\}\}$.

We solve this problem using the PTAS from the preceding sections.

Next is a precise summary of the entire reduction.

For efficiency, instead of considering all possible choices 0^q for the root (for all $q < n$ such that $\text{cost}(0^q) < 1/\epsilon$), we further restrict q to be near a power of $1+\epsilon$. This is okay because in any prefix-free code the runt 0^q can be padded with $O(\epsilon q)$ 0's to convert it to this form, without increasing the cost by more than a $1+\epsilon$ factor. (This reduces the number of possibilities for the runt from n to $O_\epsilon(\log n)$.)

DEFINITION 4.4 (reduction).

Forward direction: *Given a* HULC *instance* $\mathcal{I} = (p, \Sigma, \text{cost})$*, the forward direction of the reduction produces a set of instances* $\{\mathcal{I}'_0\} \cup \{\mathcal{I}'_q \mid q \in Q\}$ *over alphabet* Σ'*, where* $Q = \{\lceil \min(n, N/\epsilon)/(1+\epsilon)^j \rceil \mid j \in \mathbb{N}_{\geq 0}\}$ *(one instance for each choice of runt in* OPT*).*

Instance $\mathcal{I}'_0$ *(for the case of no runt in* OPT*) is* $(p, \Sigma', \text{cost}, \mathcal{R}'_0)$ *with chunked alphabet* Σ' *and universe* $(\Sigma')^*$ *(with root set* $\mathcal{R}'_0$ *containing just the empty string).*

For each $q \in Q$*, instance* $\mathcal{I}'_q$ *(for the case of runt* 0^q *in* OPT*) is* $(p', \Sigma', \text{cost}, \mathcal{R}'_q)$*, where* $p' = \langle p_2, p_3, \ldots, p_n\rangle/(1-p_1)$ *and universe* $\mathcal{U}'_q$ *contains the string* s *over* Σ' *such that* $\text{unchunk}(s)$ *doesn't have* 0^q *as a prefix (root set* $\mathcal{R}'_q = \{\text{chunk}(0^i\ell) \ : \ i < q,\, \ell \in \Sigma - \{0\}\}$*).*

Backward direction: *Given any near-optimal prefix-free code $\mathcal{Y}^0$ for $\mathcal{I}'_0$, and near-optimal prefix-free codes $\mathcal{Y}^q$ for each $\mathcal{I}'_q$, the reverse direction of the reduction produces a near-optimal code $\mathcal{X}^{\min}$ for the original instance $\mathcal{I}$ as follows: Let $\mathcal{X}^{\min}$ be a code of near-minimum cost among the codes* $\text{unchunk}(\mathcal{Y}^0)$, *and* $\{0^q\} \cup \text{unchunk}(\mathcal{Y}^q)$ *for* $q \in Q$. *Return* $\mathcal{X}^{\min}$.

By the preceding discussion, the reduction above is correct, as shown next.

LEMMA 4.5 (correctness). *Assuming the codes $\mathcal{Y}^q$ for $q \in \{0\} \cup Q$ are near-optimal prefix-free codes for their respective instances, the code $\mathcal{X}^{\min}$ returned by the reduction above is a near-optimal prefix-free code for $\mathcal{I}$.*

Proof. By construction, all of the codes $\text{unchunk}(\mathcal{Y}^0)$, and $\{0^q\} \cup \text{unchunk}(\mathcal{Y}^q)$ for $q \in Q$, are prefix-free codes over Σ.

To see that at least one of these codes is near-optimal, let $\mathcal{X}^\star$ be an optimal prefix-free code over Σ. In the case that $\mathcal{X}^\star$ has no runt, the code $\text{chunk}(\text{pad}(\mathcal{X}^\star))$ for instance $\mathcal{I}'_0$ has approximately the same cost as $\mathcal{X}^\star$, so the code $\mathcal{Y}^0$ for $\mathcal{I}'_0$ also has approximately the same cost as $\mathcal{X}^\star$, and thus so does $\text{unchunk}(\mathcal{Y}^0)$.

Otherwise code $\mathcal{X}^\star$ has some runt 0^q with $q \le \min(n, N/\epsilon)$. Padding the root to $0^{q'}$ for $q' \in Q$ gives a prefix-free code $\mathcal{X}$ over Σ of approximately the same cost. By construction, for the near-optimal solution $\mathcal{Y}^{q'}$ to instance $\mathcal{I}'_{q'}$, the codewords in $\text{unchunk}(\mathcal{Y}^{q'})$ are a near-optimal choice for the nonrunt codewords for any code over Σ with runt $0^{q'}$. Thus, the cost of the prefix-free code $\{0^{q'}\} \cup \text{unchunk}(\mathcal{Y}^{q'})$ is approximately the same as $\text{cost}(\mathcal{X})$, which is approximately the same as $\text{cost}(\mathcal{X}^\star)$. ∎

4.3. Proof of Theorem 1.2. The full PTAS implements the reduction in Definition 4.4. That is, it uses the PTAS from the preceding section to approximately solve the instances $\{\mathcal{I}'_q\}_q$ produced by the forward direction of the reduction, then computes and returns $\mathcal{X}^{\min}$ following the backward direction of the reduction. By Lemma 4.5, this gives a near-optimal prefix-free code for the given instance. Below is an outline of the steps needed to achieve running time $O(n) + O_\epsilon(\log^3 n)$.

Step 1 (forward direction—computing and solving the instances). For each of the $O_\epsilon(\log n)$ instances $\{\mathcal{I}'_q\}_q$, the PTAS first computes the signature and approximate cost (not the code tree) of the respective solutions $\{\mathcal{Y}'_q\}_q$.

By Theorem 2.1 (section 2.5), for each instance $\mathcal{I}'_q$, the signature and approximate cost of the solution $\mathcal{Y}^q$ can be computed in $O_\epsilon(\log^2 n)$ time given appropriate precomputed inputs. Here is a restatement of that theorem.

THEOREM 2.1. *Fix any instance $\mathcal{I} = (p, \Sigma, \text{cost}, \mathcal{U})$ of* HULC *with restricted universe such that* $\text{cost}(1) \le 3/\epsilon$. *Let P be the cumulative probability distribution for p: $P_\ell = \sum_{k \le \ell} p_k$ (for $\ell \in [n]$). Let σ be the signature of Σ. Let r be the signature of the roots of $\mathcal{U}$. Assume that P, σ, and r are given as inputs.*

Then the signature and approximate cost of a prefix-free code (for $\mathcal{I}$) with cost at most $(1 + O(\epsilon))\,\text{OPT}(I)$ *can be computed in time* $O_\epsilon(\log^2 n)$.

To solve the instances this way, we need to precompute three things for each instance: the cumulative probability distribution, the signature of the chunked alphabet Σ', and the signature of the root set.

Regarding the cumulative probability distributions, in fact there are only two distinct distributions used by the instances: p for $\mathcal{I}'_0$, and p' for the remaining instances. So the necessary cumulative distributions for all instances can be computed in $O(n)$ time.

Regarding the signature σ' of Σ', it can be computed as follows. First, compute the signature σ of $\Sigma - \{0\}$ in $O(n)$ time. Then, according to the definition $\Sigma' =$

$\{\underline{0^N}\} \cup \{\underline{0^i\ell} \ : \ i < N,\ \ell \in \Sigma - \{0\}\}$, take $\sigma'_1 = 1$ and, for j such that $\sigma_j > 0$, take $\sigma'_j = N\sigma_j$. This takes $O(n)$ time since $|\Sigma| \le n$.

Next consider how to compute the root-set signatures. For $\mathcal{I}'_0$, the root set is trivial. For each of the remaining $O_\epsilon(\log n)$ instances $\mathcal{I}'_q$, the PTAS computes the signature of the root set in $O_\epsilon(\log n)$ time using the following lemma.

LEMMA 4.6. *Given the signature σ of $\Sigma - \{0\}$, the signature r^q of the root set of universe $\mathcal{U}'_q$ for $\mathcal{I}'_q$ (restricted to the set $\mathcal{L}$ of possible levels, per Observation 2.13) can be computed in time $O_\epsilon(\log n)$.*

Proof. The root set for instance $\mathcal{I}'_q$ is $\mathcal{R}'_q = \{\text{chunk}(0^j\ell) \ : \ \ell \in \Sigma - \{0\}, 0 \le j < q\}$. The associated multiset of costs is $\{\text{cost}(\text{chunk}(0^j\ell)) \ : \ \ell \in \Sigma - \{0\},\ 0 \le j < q\}$. Expressing the costs explicitly, this is $\{\lfloor j/N \rfloor + \text{cost}(\ell) \ : \ \ell \in \Sigma - \{0\},\ 0 \le j < q\}$.

In this multiset, by calculation, each fixed $\ell \in \Sigma - \{0\}$ contributes N copies of $a + \text{cost}(\ell)$ for each nonnegative integer $a < \lfloor q/N \rfloor$, and $q \bmod N$ copies of $\lfloor q/N \rfloor + \text{cost}(\ell)$. Thus, the multiset can be expressed as

$$\begin{aligned} & N \times \{a + \text{cost}(\ell) \ : \ \ell \in \Sigma - \{0\}, 0 \le a < \lfloor q/N \rfloor\} \\ \bigcup \quad & (q \bmod N) \times \{a + \text{cost}(\ell) \ : \ \ell \in \Sigma - \{0\}, a = \lfloor q/N \rfloor\}. \end{aligned}$$

Introducing variable $i = \text{cost}(\ell) + a$ to eliminate a, and rearranging the inequalities, this is

$$\begin{aligned} & N \times \{i \ : \ \ell \in \Sigma - \{0\}, i - \lfloor q/N \rfloor < \text{cost}(\ell) \le i\} \\ \bigcup \quad & (q \bmod N) \times \{i \ : \ \ell \in \Sigma - \{0\}, \text{cost}(\ell) = i - \lfloor q/N \rfloor\}. \end{aligned}$$

Thus, introducing variable $j = \text{cost}(\ell)$ and recalling that σ_j is the number of cost-j letters in $\Sigma - \{0\}$, a given $i \in \mathcal{L}$ occurs with multiplicity

$$r_i^q \ = \ N \times \sum_{j=i-\lfloor q/N \rfloor + 1}^{i} \sigma_j + (q \bmod N) \times \sigma_{i - \lfloor q/N \rfloor}.$$

Since by assumption 0^q is a runt, $\text{cost}(0^q) < 1/\epsilon$, so $q/N < 1/\epsilon$. Thus, the sum above has at most $1/\epsilon$ terms, and the value of r_i^q for a given i and q can be calculated in $O(1/\epsilon)$ time.

To finish, we observe that the set $\mathcal{L}$ of possible levels for the instance $\mathcal{I}'_q$ can be computed as follows. Per Definition 2.6 (section 2.3), the set is $[0, 2\tau + 3\delta] \cup [i_{\mathcal{R}'_q}, i_{\mathcal{R}'_q} + 3\delta] \ \cup \ [i_{\Sigma'}, i_{\Sigma'} + \tau + 3\delta]$.

The values of τ and δ (resp., $\lceil \log_2 \lceil \text{cost}(\underline{1})/\epsilon \rceil \, \text{cost}(\underline{1})/\epsilon \rceil$ and $\text{cost}(\underline{1}) \lceil \log_2 n \rceil$) are easy to calculate.

By definition, $i_{\Sigma'}$ is the minimum cost of any letter in Σ' of cost at least τ. It can be calculated (just once) in $O(\log n)$ time by binary search over Σ.

By definition, $i_{\mathcal{R}'_q}$ is the minimum cost of any root in $\mathcal{R}'_q$ of cost at least τ. Reinspecting the calculation of r_i^q above, $i_{\mathcal{R}'_q}$ is the minimum value of the form $a + \text{cost}(\ell)$ exceeding $\tau - 1$, for any $\ell \in \Sigma - \{0\}$ and integer $a \in [0, q/N]$. This value can be found in binary search over Σ in $O(\log n)$ time.

Once $\mathcal{L}$ is computed for $\mathcal{I}'_q$, each coordinate of the signature r^q of the root set $\mathcal{R}'_q$ above (restricted to $\mathcal{L}$) can be calculated in $O_\epsilon(1)$ time. Since $|\mathcal{L}| = O_\epsilon(\log n)$, the total time is $O_\epsilon(\log n)$. ∎

In sum, the PTAS precomputes the necessary inputs for all instances $\{\mathcal{I}'_q\}_q$ of the reduction, taking $O_\epsilon(\log n)$ time for each of the $O_\epsilon(\log n)$ instances. It then applies

Theorem 2.1 to solve these instances. Specifically, in $O(n) + O_\epsilon(\log^3 n)$ total time, it computes the signature and approximate cost of a near-optimal prefix-free code $\mathcal{Y}^q$ for every instance $\mathcal{I}'_q$.

Step 2 (backward direction—building the near-optimal code tree). The backward direction of the reduction must return a near-minimum-cost code $\mathcal{X}^{\min}$ among the following candidate codes: unchunk($\mathcal{Y}^0$) and $\{0^q\} \cup$ unchunk($\mathcal{Y}^q$) for $q \in Q$.

At this point, the PTAS has only the signatures and approximate costs of the various codes $\{\mathcal{Y}^q\}_q$. But this is enough information to determine which of the candidate codes above have near-minimum cost. In particular, unchunking a code approximately preserves its cost, so the PTAS knows the approximate costs of each code unchunk($\mathcal{Y}^q$). Then, from the approximate cost of unchunk($\mathcal{Y}^q$), the approximate cost of $\{0^q\} \cup$ unchunk($\mathcal{Y}^q$) is easily calculated. (Recall that each code unchunk($\mathcal{Y}^q$) is for probabilities $p' = \langle p_2, p_3, \ldots, p_n\rangle/(1-p_1)$ and has nonrunt codewords that don't have 0^q as a prefix. By calculation, adding codeword 0^q to the code gives a code for p of cost $p_1 \operatorname{cost}(0^q) + (1 - p_1) \operatorname{cost}(\text{unchunk}(\mathcal{Y}^q))$.) In this way, the PTAS chooses the index q of the best candidate code. The PTAS retains the signature x of the corresponding code $\mathcal{Y}^q$ over Σ'.

One more step remains: to compute the tree representation T of the chosen candidate code $\mathcal{X}^q$ (i.e., unchunk($\mathcal{Y}^0$) if $q = 0$, or $\{0^q\} \cup$ unchunk($\mathcal{Y}^q$) if $q > 0$).

Recall that, by Theorem 3.1, for alphabets with integer letter costs and $\operatorname{cost}(1) \le 3/\epsilon$, given a signature x for a prefix-free code, one can compute a corresponding tree representation F' in $O(n) + O_\epsilon(\log^2 n)$ time. This theorem doesn't solve our problem directly for two reasons: (1) the signature x that we have is for the code $\mathcal{Y}^q$ over chunk alphabet Σ', not for the final code $\mathcal{X}^q$ over Σ; (2) more fundamentally, because $\operatorname{cost}(0) = 1/N < 1$ for Σ, the concept of signature is not particularly useful when working over Σ.

Instead, to compute the tree T for $\mathcal{X}^q$, the PTAS uses Theorem 3.1 to first compute the tree representation F' of the prefix-free $\mathcal{Y}^q$ over Σ'. This takes $O(n) + O_\epsilon(\log^2 n)$ time. The PTAS will then convert this tree representation F' for $\mathcal{Y}^q$ directly into a tree T for $\mathcal{X}^q$, using the following lemma.

LEMMA 4.7. (i) *Given the tree F' for a code $\mathcal{Y}^0$ for $\mathcal{I}'_0$, the tree T for the corresponding code $\mathcal{X}^0 =$* unchunk($\mathcal{Y}^0$) *(or one at least as good) for $\mathcal{I}$ can be constructed in $O(|F'|)$ time.*

(ii) *Given the forest F' for a code $\mathcal{Y}^q$ for $\mathcal{I}'_q$, the tree T for the corresponding code $\mathcal{X}^q = \{0^q\} \cup$* unchunk($\mathcal{Y}^q$) *(or one at least as good) for $\mathcal{I}$ can be constructed in $O(|F'|)$ time.*

Proof. Part (i). In this case ($q = 0$), $\mathcal{Y}^0$ has n codewords, and F' has only a single root node. F' is the tree representation of $\mathcal{Y}^0$ over Σ', and we want to compute the tree representation of $\mathcal{X}^0 =$ unchunk($\mathcal{Y}^0$) over Σ. Note that the unchunk() function simply breaks each chunk $\underline{0^N}$ or $\underline{0^i\ell}$ into its individual letters over Σ.

In the tree representation F' of $\mathcal{Y}^0$ over Σ', each edge (such as $\underline{0001}$) represents a chunk. To "unchunk" the tree, we replace each such edge by a path (such as $0 \to 0 \to 0 \to 1$), adding intermediate nodes as necessary. This can be accomplished by applying a local transformation at each interior node u' of F', as illustrated (from right to left) in Figure 11. Roughly, for each edge labeled $\underline{a_1 a_2 \ldots a_k}$ on the right, there is a corresponding path $a_1 \to a_2 \to \cdots \to a_k$ on the left. However, for efficiency, in fact we do something slightly different. In general, in the tree F', only some of the possible edges might be present. (For example, there is no edge labeled $\underline{001}$ out of u' on the right.) When there are vacancies such as this, we first preprocess the node, replacing edges in T' by cheaper edges if possible. In general, if the node u' has some

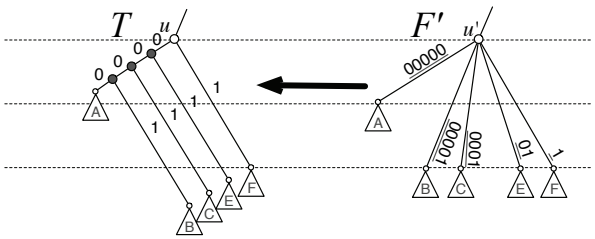

FIG. 11. *The transformation applied to each node of F' to "unchunk" F'.*

d children, we preprocess the node to make sure those d children use the d cheapest possible outgoing edges in Σ'. In this way we avoid constructing overly large trees.

The general construction is as follows. For each interior node u' in F', let v'_0 be the child along the edge labeled $\underline{0^N}$, if any. Let $v'_1, \ldots, v'_d$ be the remaining children. Replace the edges to these latter d children by a subtree $t(d)$ with d leaves, where $t(d)$ is the tree representation for the d cheapest strings in $\{b \mid \underline{b} \in \Sigma', b \neq 0^N\}$. Next, identify each child v'_i for $i \geq 1$ with the ith cheapest leaf in $t(d)$. Then make v'_0 the 0-child of the node at the end of the left spine of $t(d)$. Doing this for all interior nodes gives T.

Part (ii). In this case the forest F' is a collection of trees, each with its own root in the root set $\mathcal{R}'_q$ of $\mathcal{U}'_q$. Let d' be the number of roots. Perform the transformation described in part (i) separately for each tree in the collection. Finally, glue the d' trees together into a single tree T as follows: start with a tree whose d' leaves are the d' cheapest roots in the root set, then, for $j = 1, 2, \ldots, d'$, identify the jth of these leaves with the root of the jth (modified) tree in the collection. Finally, add a leaf $0^{q'}$, where $q' \leq q$ is the minimum such that $0^{q'}$ is not already an interior node in T.

Correctness. By inspection of the construction, each leaf node v' in F' becomes a leaf in T whose cost is at most $1 + \epsilon$ times the cost of the string over Σ that the string of v' originally represented. If $q > 0$, the runt in T has at most q zeros, so it has cost at most $\text{cost}(0^q)$.

Time. Assuming that each interior node u' in F' comes with a list of the edges to its children ordered by increasing cost, the local transformation at each node u' can be done in time proportional to its degree d. Also, gluing together the roots takes time proportional to the number of roots, since in the resulting tree each interior node has degree at least two (recall that the roots unchunk to strings of the form $0^j\ell$ for $j < d$, which hang consecutively off the left spine of T). Thus, the entire transformation can be done in time proportional to the size of F'. □

Since the trees produced via Theorem 3.1 have size $O(n)$, the time the PTAS takes to construct the tree T for the near-optimal code $\mathcal{X}^q$ via Lemma 4.7 is $O(n)$.

This completes the PTAS and the proof of Theorem 1.2. □

5. Remarks.

More precise time bound. The proof of Theorem 1.2 shows that the PTAS runs in $O(n) + O_\epsilon(\log^3 n)$ time. We note without proof that the time is

$$O(n) + \exp\left(O\left(\frac{1}{\epsilon^3}\log^2\frac{1}{\epsilon}\right)\right)\log^3 n.$$

Here is a sketch of the reasoning. By careful inspection of the proof of Theorem 1.2, the time is proportional to $(\tau+1)^\gamma(|\mathcal{L}|^2+\gamma)\,|Q|$. Plugging in $\tau = O(\epsilon^{-2}\log\epsilon^{-1})$

(Lemma 2.3), $\gamma = O(\tau/\epsilon)$ (Definition 2.4), $|\mathcal{L}| = O(\tau + \epsilon^{-1}\log n)$ (Definition 2.6), and $|Q| = O(\epsilon^{-1}\log n)$ (Definition 4.4) gives the claim. (Slightly better bounds can be shown with more careful arguments, including coarsening the letter costs to ceilings of powers of $(1+\epsilon)$ to reduce the number of distinct letter costs.)

Practical considerations. The exhaustive search outlined in section 2.5 is the bottleneck of the computation. In practice, this search can be pruned and restricted to *monotone* group-to-level assignments. Or, it may be faster to use a mixed integer linear program solver to solve the underlying program. In this case, the alternate mixed program in Figure 12 may be easier to solve than ILP, as it integer (in fact 0/1) variables only for the probabilities p_k with $p_k \geq \epsilon/\tau$.

$$
\begin{array}{|l|}
\hline
\text{minimize } \sum_{i,k} p_k\, i\, y_{ki} \text{ s.t.} \\
\begin{array}{rrcll}
\left.\begin{array}{r}\text{if } i<\tau: \\ \text{if } i\geq\tau:\end{array}\right\} & \begin{array}{r} x_i + w_i \\ \max(x_i, w_i)\end{array} & \leq & r_i + \sum_{j<i}\sigma_{i-j}w_j & (i\in\mathcal{L}) \\
& \sum_k y_{ki} & = & x_i & (i\in\mathcal{L}) \\
& \sum_i y_{ki} & = & 1 & (k\in[n]) \\
& w_i, x_i,\ y_{ki} & \geq & 0 & (i\in\mathcal{L}, k\in[n]) \\
& y_{ki} & \in & \{0,1\} & (i\in[\tau-1], k : p_k \geq \epsilon/\tau)
\end{array} \\
\hline
\end{array}
$$

FIG. 12. *A practical alternative to* ILP *with integrality gap* $1+O(\epsilon)$.

Solving this mixed program suffices, because any near-optimal fractional solution (x, w, y) to it can be rounded to a near-optimal integer solution (corresponding to a near-optimal τ-relaxed code), as discussed next.

LEMMA 5.1. *Given any fractional solution (w, x, y) to the mixed program in Figure 12, one can compute in $O(n)+O_\epsilon(\text{polylog}\, n)$ time an integer solution $(\hat{w}, \hat{x}, \hat{y})$ of cost at most $1+O(\epsilon)$ times the cost of (w, x, y).*

Proof sketch. For each $i < \tau$, in increasing order, if x_i and $\sum_k y_{ik}$ have fractional part $f > 0$, do the following. Let $i' = i + \text{cost}(0)$. Decrease x_i by f, increase w_i by f, and increase $x_{i'}$ by f. (This preserves the capacity constraint because increasing $w_{i'}$ by f increases the right-hand side of the capacity constraint for i' by at least f, since $\sigma_{i'-i} = \sigma_{\text{cost}(0)} \geq 1$.) Also, decrease $\sum_k y_{ki}$ by f, and increase $\sum_k y_{ki'}$ by f by (repeatedly, if necessary) decreasing the (nonintegral) $y_{ki} > 0$ with smallest p_k and increasing the corresponding nonintegral $y_{ki'}$.

Since these nonintegral y_{ki}'s have $p_k < \tau/\epsilon$, for each i, the increase in the cost is at most $(\epsilon/\tau) f\, \text{cost}(0)$, which is less than $(\epsilon/\tau)\,\text{cost}(0)$, so the total increase in the cost (for all levels $i < \tau$) is at most $\epsilon\,\text{cost}(0)$.

After this modification, each x_i for $i < \tau$ is an integer. Take $(\hat{w}, \hat{x}, \hat{y})$ to be an optimal, all-integer greedy extension of this assignment to these x_i's. That is, for each $i \in \mathcal{L}$, in increasing order, take $\hat{w}_i$ maximally subject to the capacity constraint, and, if $i \geq \tau$, take $\hat{x}_i$ maximally subject to the capacity constraint. Then take $\hat{y}$ so that the corresponding code is monotone. This greedy extension is optimal by an argument similar to the proof of Lemma 2.11, so it has cost at most the cost of the modified (w, x, y). □

Finding a $(1+\epsilon)$-approximation is in NC. Given that HULC is neither known to be in P (polynomial time), nor known to be NP-hard, it is interesting that the results in this paper extend to show that, given any fixed ϵ, the problem of $(1+\epsilon)$-approximating HULC is in NC (Nick's class—polynomially many parallel processors and polylogarithmic time). (For instances in which $\text{cost}(1) \leq 3/\epsilon$, the cumulative

distribution P and the signatures r and σ necessary for Theorem 2.1 can be computed in NC, and the remaining computation takes time $O_\epsilon(\text{polylog}\, n)$ on one processor. For instances with no restrictions on the cost, one can use the fact that $\mathcal{L} = O_\epsilon(\log n)$ to show that each $O(n)$-time step in the proof of Theorem 1.2 is in NC.)

Open problems. The PTAS in this paper is not a fully polynomial-time approximation scheme (FPTAS). That is, the running time is not polynomial in $1/\epsilon$. Is there an FPTAS? For that matter, is there a polynomial-time exact algorithm? And, of course, is HULC NP-complete?

Appendix.

Proof of Lemma 2.9. *Part* (i). Let F be the forest in the tree representation of $\mathcal{X}$. Each nonempty level i in F is in $\mathcal{L}$, by Observation 2.13.

Let x_i and w_i be, respectively, the number of codewords and interior nodes in level i of F. Let y be the assignment of codewords (or rather codeword costs) to probabilities: that is, $y_{ki} = 1$ if and only if $\text{cost}(\mathcal{X}_k) = i$ (else $y_{ki} = 0$). Let z be the assignment of levels to groups: that is, $z_{gi} = y_{ki}$ for all $i < \tau$, $g \in [\gamma]$, and $k \in G_g$.

First consider the capacity constraint of ILP. Level i of F has at least $x_i + w_i$ nodes, or $\max(x_i, w_i)$ if $i \geq \tau$. Up to r_i of these nodes can be parentless in F because they are roots in $\mathcal{U}$. Each of the rest has a parent in F that is an interior node in F in a level $j < i$. There are at most $\sum_{j<i} \sigma_{j-i} w_j$ nodes with such parents, because each of the w_j interior nodes in a given level j of F can parent at most σ_{i-j} nodes in level i (one for each of the σ_{i-j} letters of cost $i - j$ in Σ). Thus, the capacity constraint is met. By inspection, (w, x, y, z) meets the remaining constraints of ILP, and the cost of (w, x, y, z) is $\text{cost}(\mathcal{X})$. This proves part (i) of the lemma.

Part (ii). Given any set $\mathcal{X} \subset \mathcal{U}$, let $\mathcal{X}_{<\tau}$ denote $\{\mathcal{X}_k \mid \text{cost}(\mathcal{X}_k) < \tau\}$.

Start with $\mathcal{X} \leftarrow \emptyset$. For each $i \in \mathcal{L}$, in increasing order, add to $\mathcal{X}$ any x_i strings from level i of $\mathcal{U}$ that have no prefix in $\mathcal{X}_{<\tau}$.

This construction clearly generates a τ-relaxed code as long as there are enough strings available to assign in each level. There will be, because the construction maintains the following invariant: *for each $j < i$, at least w_j strings in level j of $\mathcal{U}$ are available.* (Recall that a string is available if it has no prefix in $\mathcal{X}_{<\tau}$.) Suppose that this invariant holds before codewords are added from level i. At that point, the number of available strings in level i of $\mathcal{U}$ must be at least the right-hand side of the capacity constraint for i. In the case that $i < \tau$, since the capacity constraint holds, the right-hand side is at least $x_i + w_i$, so placing x_i of the available strings into $\mathcal{X}$ leaves w_i still available, maintaining the invariant. In the case that $i \geq \tau$, the right-hand side is both at least w_i (so the invariant is maintained) and at least x_i (so there are x_i available strings to add to $\mathcal{X}$, without making any string unavailable, since $i \geq \tau$).

Finally, assign to each probability p_k a codeword from $\mathcal{X}$ of cost i' such that $y_{ki'} = 1$. (This is possible because in $\mathcal{X}$ there are $x_i = \sum_k y_{ki}$ codewords of each cost i.) Then, $\text{cost}(\mathcal{X})$ equals the cost of (w, x, y, z). □

Proof of Lemma 2.11. Let (w, x, y, z) be any minimum-cost feasible extension of z. Let $(\hat{w}, \hat{x}, \hat{y}, z)$ be the greedy extension (if it is well defined).

Given z, the constraints of ILP force $\hat{x}_i = x_i$ for $i < \tau$.

By induction on $i \in \mathcal{L}$ (using the maximality of $\hat{w}_i$ and that $\hat{x}_i = x_i$ for $i < \tau$), it follows that $\hat{w}_i \geq w_i$ for all $i \in \mathcal{L}$. Thus, replacing w by $\hat{w}$ in (w, x, y, z) gives a solution $(\hat{w}, x, y, z)$ that is also feasible and optimal.

Now suppose for contradiction that $x_i \neq \hat{x}_i$ for some level i. Fix i' to be the

minimum such level. Note that $i' \geq \tau$ since $\hat{x}_i = x_i$ for $i < \tau$. Since $\hat{x}_i = x_i$ for $i < i'$, and $\hat{x}_{i'}$ is maximal (by definition of the greedy extension), it follows that $x_{i'} < \hat{x}_{i'}$. Thus, the capacity constraint for level i' ($\geq \tau$) is loose for $(\hat{w}, x, y, z)$. Increasing $x_{i'}$ by 1, and decreasing x_j by 1 for some $j > i$ (and adjusting y accordingly) gives a feasible solution that is cheaper than $(\hat{w}, x, y, z)$, contradicting the optimality of $(\hat{w}, x, y, z)$.

Thus, $\hat{x} = x$. Thus, $(\hat{w}, \hat{x}, y, z)$ is feasible. By the choice of $\hat{y}$ in the definition of the greedy extension, the lemma follows. □

Acknowledgments. The authors are very grateful to the two anonymous referees for their patience and helpful comments.

REFERENCES

[1] J. Abrahams, *Code and parse trees for lossless source encoding*, Comm. Inform. Syst., 1 (2001), pp. 113–146.
[2] D. Altenkamp and K. Melhorn, *Codes: Unequal probabilities, unequal letter costs*, J. ACM, 27 (1980), pp. 412–427.
[3] N. M. Blachman, *Minimum cost coding of information*, IRE Trans. Inform. Theory, PGIT-3 (1954), pp. 139–149.
[4] P. Bradford, M. Golin, L. L. Larmore, and W. Rytter, *Optimal prefix-free codes for unequal letter costs and dynamic programming with the Monge property*, J. Algorithms, 42 (2002), pp. 277–303.
[5] S. N. Choi and M. Golin, *Lopsided trees* I: *A combinatorial analysis*, Algorithmica, 31 (2001), pp. 240–290.
[6] N. Cot, *Complexity of the variable-length encoding problem*, in Proceedings of the 6th Southeast Conference on Combinatorics, Graph Theory and Computing, 1975, pp. 211–244.
[7] N. Cot, *Characterization and Design of Optimal Prefix Codes*, Ph.D. thesis, Stanford University, Palo Alto, CA, 1977.
[8] I. Csiszár, *Simple proofs of some theorems on noiseless channels*, Inform. Control, 514 (1969), pp. 285–298.
[9] S. Dumitrescu, *Faster algorithm for designing optimal prefix-free codes with unequal letter costs*, Fund. Inform., 73 (2006), pp. 107–117.
[10] E. N. Gilbert, *How good is Morse code?*, Inform. Control, 14 (1969), pp. 559–565.
[11] E. N. Gilbert, *Coding with digits of unequal costs*, IEEE Trans. Inform. Theory, 41 (1995), pp. 596–600.
[12] M. Golin and J. Li, *More efficient algorithms and analyses for unequal letter cost prefix-free coding*, IEEE Trans. Inform. Theory, 54 (2008), pp. 3412–3424.
[13] M. Golin and G. Rote, *A dynamic programming algorithm for constructing optimal prefix-free codes for unequal letter costs*, IEEE Trans. Inform. Theory, 44 (1998), pp. 1770–1781.
[14] M. J. Golin and N. Young, *Prefix codes: Equiprobable words, unequal letter costs*, SIAM J. Comput., 25 (1996), pp. 1281–1292.
[15] K. Hinderer, *On dichotomous search with direction-dependent costs for a uniformly hidden object*, Optim., 21 (1990), pp. 215–229.
[16] D. A. Huffman, *A method for the construction of minimum redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.
[17] K. A. S. Immink, *Codes for Mass Data Storage Systems*, Shannon Foundations Publishers, Eindhoven, The Netherlands, 1999.
[18] A. Itai, *Optimal alphabetic trees*, SIAM J. Comput., 5 (1976), pp. 9–18.
[19] S. Kapoor and E. M. Reingold, *Optimum lopsided binary trees*, J. ACM, 36 (1989), pp. 573–590.
[20] R. Karp, *Minimum-redundancy coding for the discrete noiseless channel*, IRE Trans. Inform. Theory, IT-7 (1961), pp. 27–39.
[21] D. E. Knuth, *The Art of Computer Programming, Volume* III: *Sorting and Searching*, Addison–Wesley, Reading, MA, 1973.
[22] R. M. Krause, *Channels which transmit letters of unequal duration*, Inform. Control, 5 (1962), pp. 13–24.
[23] A. Lempel, S. Even, and M. Cohen, *An algorithm for optimal prefix parsing of a noiseless and memoryless channel*, IEEE Trans. Inform. Theory, 19 (1973), pp. 208–214.

[24] R. S. Marcus, *Discrete Noiseless Coding*, M.S. thesis, Electrical Engineering Department, MIT, Cambridge, MA, 1957.
[25] K. Mehlhorn, *An efficient algorithm for constructing nearly optimal prefix codes*, IEEE Trans. Inform. Theory, 26 (1980), pp. 513–517.
[26] Y. Perl, M. R. Garey, and S. Even, *Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters*, J. ACM, 22 (1975), pp. 202–214.
[27] L. E. Stanfel, *Tree structures for optimal searching*, J. ACM, 17 (1970), pp. 508–517.
[28] B. Varn, *Optimal variable length codes (arbitrary symbol cost and equal code word probability)*, Inform. Control, 19 (1971), pp. 289–301.