European Journal of Combinatorics 34 (2013) 82–90

Contents lists available at SciVerse ScienceDirect

European Journal of Combinatorics

journal homepage: www.elsevier.com/locate/ejc

Minimax trees in linear time with applications

Paweł Gawrychowski [a], Travis Gagie [b]

[a] Institute of Computer Science, University of Wroclaw, Poland
[b] Department of Computer Science and Engineering, Aalto University, Finland

ARTICLE INFO

Article history:
Available online 16 August 2012

ABSTRACT

A minimax tree is similar to a Huffman tree except that, instead of minimizing the weighted average of the leaves' depths, it minimizes the maximum of any leaf's weight plus its depth. Golumbic (1976) [20] introduced minimax trees and gave a Huffman-like, $\mathcal{O}(n \log n)$-time algorithm for building them. Drmota and Szpankowski (2002) [10] gave another $\mathcal{O}(n \log n)$-time algorithm, which takes linear time when the weights are already sorted by their fractional parts. In this paper we give the first linear-time algorithm for building minimax trees for unsorted real weights.

1. Introduction

In a minimax tree for a multiset $W = \{w_1, \ldots, w_n\}$ of weights, each leaf has a weight w_i, each internal node has weight equal to the maximum of its children's weights plus 1, and the weight of the root is as small as possible. (Notice this is not the same as the definition from game theory.) In other words, if ℓ_i is the depth of the leaf with weight w_i, then $\max_i\{w_i + \ell_i\}$ is minimized. Golumbic [20] showed that if we modify Huffman's algorithm [24] to repeatedly replace the two nodes with smallest weights by a node whose weight is equal to their maximum plus 1, instead of their sum, then it builds a minimax tree instead of a Huffman tree. Like Huffman's algorithm, Golumbic's algorithm takes $\mathcal{O}(n \log n)$ time and can build trees of any degree. Golumbic, Parker [35] and Hoover et al. [21] showed how to use Golumbic's algorithm to restrict circuits' fan-in and fan-out without greatly increasing their sizes or depths. While studying prefix codes with minimum maximum pointwise redundancy, Drmota and Szpankowski [10,11] independently introduced minimax trees as code-trees for generalized Shannon codes [38] and gave another $\mathcal{O}(n \log n)$-time algorithm for building them, which takes linear time when the weights are already sorted by their fractional parts. To see why the two problems are related, consider that, if $P = p_1, \ldots, p_n$ is a probability

E-mail addresses: gawry1@gmail.com (P. Gawrychowski), travis.gagie@gmail.com (T. Gagie).

0195-6698/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ejc.2012.07.016

distribution and each $w_i = \log_2 p_i$, then a minimax tree for W is the code-tree for a prefix code with minimum maximum pointwise redundancy with respect to P. By analyzing their algorithm, Drmota and Szpankowski proved bounds on the redundancy of arithmetic coding, which Baer [3] recently improved by analyzing Golumbic's algorithm. In this paper we show how Drmota and Szpankowski's algorithm can be made to run in linear time on a RAM when each index and weight fits in $\mathcal{O}(1)$ words. Thus, we obtain the first linear-time algorithm for building minimax trees for unsorted real weights.

Between Golumbic's article and Drmota and Szpankowski's, there seems to have been little research on building minimax trees.[1] Several important papers were published, however, on the related problem of building alphabetic minimax trees, in which the leaves' weights must be in a given order from left to right. Hu et al. [23] gave the first $\mathcal{O}(n \log n)$-time algorithm for building alphabetic minimax trees for real weights. Kirkpatrick and Klawe [27] and Coppersmith et al. [7] gave an algorithm (or, more precisely, two algorithms that are equivalent when the trees are to be binary) that builds alphabetic minimax trees for integer weights in linear time, and showed how to use it to restrict circuits' fan-in and fan-out without greatly increasing their sizes or depths and without changing the numbers of edge crossings (and, thus, preserving planarity). Kirkpatrick and Klawe also showed how to combine their algorithm with binary search in order to build alphabetic minimax trees for real weights in $\mathcal{O}(n \log n)$ time. We note that, if their algorithm for integer weights is viewed as an alphabetic analogue of the Kraft Inequality [31] – as it was by Yeung [41] and Nakatsu [33], who independently rediscovered it – then their algorithm for real weights is an alphabetic analogue of Drmota and Szpankowski's. Kirkpatrick and Przytycka [28] gave an $\mathcal{O}(\log n)$-time, $\mathcal{O}(n/\log n)$-processor algorithm for integer weights in the CREW PRAM model. Finally, Evans and Kirkpatrick [12] showed how a generalization of Kirkpatrick and Klawe's algorithm can be used to restructure binary search trees.

We became interested in minimax trees while studying adaptive prefix coding, which we discuss in Section 4. In a previous paper [15] (see also [16,25]) we noted that minimax trees built with Golumbic's algorithm have the same Sibling Property [13,19] as Huffman trees, and turned the Faller–Gallager–Knuth algorithm [30] for adaptive Huffman coding into an algorithm for adaptive Shannon coding. In another previous paper [17] we used a data structure due to Kirkpatrick and Przytycka and a technique for generalized selection due to Klawe and Mumey [29], to make Kirkpatrick and Klawe's algorithm for real weights run in $\mathcal{O}(n \min(\log n, d \log\log n))$ time, where d is the number of distinct values $\lceil w_i \rceil$. In that paper we conjectured that a similar modification could make Drmota and Szpankowski's algorithm run in linear time on unsorted real weights, and in this paper we prove that conjecture.

In Section 2 we consider the preliminary problem of building minimax trees for unsorted integer weights. Notice that, as such weights have no fractional parts, Drmota and Szpankowski's algorithm takes linear time for this problem. However, there are two difficulties when using their algorithm: first, because they considered the weights to be logarithms, they did not address some questions of precision that arise when the weights are large; second, because they were mostly interested in analysis, they were satisfied with computing the depths of minimax trees' leaves in linear time, rather than building the trees themselves. We give two new linear-time algorithms for unsorted integer weights that can handle large weights – i.e., polynomial in n, so that each fits in a constant number of machine words – and that actually build the minimax trees. In Section 3 we present our main result, a linear-time algorithm for building minimax trees for unsorted real weights. Our algorithm is based on Drmota and Szpankowski's but, whereas theirs uses sorting and binary search, ours uses generalized selection, as well as a new data structure to test the Kraft Inequality. Our results generalize to higher degrees and larger code alphabets but, for the sake of simplicity, in this paper we consider only binary

[1] Baer [4] recently pointed out to us that, in this interval, Blumer and McEliece [6] gave essentially the same algorithm that Drmota and Szpankowski later did, but for a different and more general problem; as far as we know, this was the first time anyone noticed the relationship between the two results. We note the algorithm is not optimal for the general problem Blumer and McEliece considered, and it is not clear to us they realized it is optimal for the special case equivalent to building a prefix code with minimum maximum pointwise redundancy.

trees and alphabets; by log we always mean $\log_2$. In Section 4 we discuss how minimax trees are related to problems in adaptive and semi-static prefix coding and group testing.

2. Minimax trees for integer weights

In this section we give two $\mathcal{O}(n)$-time algorithms for building a minimax tree for a multiset of integer weights, both based on the following lemma (which we note applies to any weights, not only integers) and corollary. We write $M(W)$ to denote the weight of the root of a minimax tree for W.

Lemma 1. *If $W = \{w_1, \ldots, w_n\}$ is a multiset of weights and*

$$W' = \left\{ \max\left(w_1, \max_i\{w_i\} - n + 1\right), \ldots, \max\left(w_n, \max_i\{w_i\} - n + 1\right) \right\},$$

then $M(W') = M(W)$. Moreover, any minimax tree for W' becomes a minimax tree for W when we replace the leaves' weights equal to $\max_i\{w_i\} - n + 1$ by the weights in W less than or equal to $\max_i\{w_i\} - n + 1$, in any order.

Proof. Consider a minimax tree T for W. Without loss of generality, we can assume T is strictly binary – i.e., that every internal node has exactly two children – and, therefore, that it has height at most $n - 1$. If $n = 1$, then $W = w_1 = \max_i\{w_i\} - n + 1$. Otherwise, all the leaves have depth at least 1, so $M(W) \geq \max_i\{w_i\} + 1$. Consider any leaf (if one exists) with weight less than $\max_i\{w_i\} - n + 1$ and depth ℓ. Since $\max_i\{w_i\} - n + 1 + \ell \leq \max_i\{w_i\} < M(W)$, increasing that leaf's weight to $\max_i\{w_i\} - n + 1$ and updating its ancestors' weights, does not change the weight $M(W)$ of the root. It follows that $M(W') = M(W)$.

Now consider a minimax tree T' for W'. If we replace the leaves' weights equal to $\max_i\{w_i\} - n + 1$ by the weights in W less than or equal to $\max_i\{w_i\} - n + 1$ and update all the nodes' weights, then the weight $M(W')$ of the root cannot increase nor, by definition, decrease to less than $M(W)$. Since $M(W') = M(W)$, it follows that the re-weighted tree is a minimax tree for W. □

Corollary 2. *When all the weights in W are integers, we can sort W' in $\mathcal{O}(n)$ time.*

Proof. When all the weights in W at least $\max_i\{w_i\} - n + 1$ are integers, all the weights in W' are integers in the interval $[\max_i\{w_i\} - n + 1, \max_i\{w_i\}]$. Since this interval has length $n - 1$, we can sort W' in $\mathcal{O}(n)$ time using either direct addressing, which takes $\mathcal{O}(n)$ extra space, or radix sort, which takes no extra space [14]. □

For our first algorithm, we build and sort W'; build a minimax tree for W' using an implementation of Golumbic's algorithm that takes $\mathcal{O}(n)$ time when the weights are already sorted; and replace the leaves' weights equal to $\max_i\{w_i\} - n + 1$ by the weights in W less than or equal to $\max_i\{w_i\} - n + 1$. We note that Van Leeuwen [39] showed how to implement Huffman's algorithm to take $\mathcal{O}(n)$ time when the weights are already sorted. We could implement Golumbic's algorithm analogously, but we think the implementation below is simpler.

Lemma 3. *Golumbic's algorithm can be implemented to take $\mathcal{O}(n)$ time when the weights are already sorted.*

Proof. We start with the weights stored in a linked list in nondecreasing order, and set a pointer to the head of the list. We then repeat the following procedure until there is only one node left in the list, which is the root of a minimax tree for the given weights: we move the pointer along the list to the last weight less than or equal to the maximum of the first two weights plus 1; remove the first two nodes from the list; make those nodes the children of a new node with weight equal to the maximum of their weights plus one; and insert the new node immediately to the right of the pointer. Notice we remove two nodes for each one we insert, so the total number of nodes is $2n - 1$. Therefore, since the pointer passes over each node once, this implementation takes $\mathcal{O}(n)$ time. □

Building and sorting W' takes $\mathcal{O}(n)$ time, by Corollary 2; building a minimax tree for W' takes $\mathcal{O}(n)$ time, by Lemma 3; replacing the leaves' weights equal to $\max_i\{w_i\} - n + 1$ by the weights in W less than or equal to $\max_i\{w_i\} - n + 1$ takes $\mathcal{O}(n)$ time, because it can be done in any order. By Lemma 1, the resulting tree is a minimax tree for W.

Theorem 4. *Given a multiset W of n integer weights, we can build a minimax tree for W in $\mathcal{O}(n)$ time.*

Our second algorithm differs in its second step: instead of using Golumbic's algorithm to build a minimax tree for W', we use Kirkpatrick and Klawe's $\mathcal{O}(n)$-time algorithm for integer weights to build an alphabetic minimax tree for the sequence V consisting of the weights in W' in non-increasing order. The algorithm's correctness follows from the Kraft Inequality.

Theorem 5 ([31])**.** *If there exists a binary tree whose leaves have depths $\ell_1, \ldots, \ell_n$, then $\sum_i 1/2^{\ell_i} \leq 1$. Conversely, if $\sum_i 1/2^{\ell_i} \leq 1$ and $\ell_1 \leq \cdots \leq \ell_n$, then there exists an ordered binary tree whose leaves, from left to right, have depths $\ell_1, \ldots, \ell_n$.*

By the latter part of Theorem 5 and a standard exchange argument – i.e., if a minimax tree contains two leaves such that the deeper one has a higher weight than the shallower one, then we can swap their weights – there exists a minimax tree for W' in which the leaves' weights are non-increasing from left to right. Therefore, by definition, any alphabetic minimax tree for V is a minimax tree for W'.

3. Minimax trees for real weights

In this section we give the first $\mathcal{O}(n)$-time algorithm for building minimax trees for unsorted real weights. As we noted in the introduction, our algorithm is based on Drmota and Szpankowski's algorithm but avoids sorting, which is the step that determines their algorithm's $\mathcal{O}(n \log n)$ complexity. In addition to yielding an optimal algorithm for an interesting problem with applications in, e.g., circuit design and data compression, we believe the techniques we use in this section may be of independent interest.

To build a prefix code with minimum maximum pointwise redundancy with respect to a given probability distribution $P = p_1, \ldots, p_n$, Drmota and Szpankowski start with a Shannon code for P, in which the codeword for the ith character has length $\lceil \log(1/p_i) \rceil$, for each i; they sort the logarithms by their fractional parts, i.e., $\log(1/p_1) - \lfloor \log(1/p_1) \rfloor, \ldots, \log(1/p_n) - \lfloor \log(1/p_n) \rfloor$; and they find the largest value x such that $\lceil \log(1/p_1) - x \rceil, \ldots, \lceil \log(1/p_n) - x \rceil$ obey the Kraft Inequality. A binary tree with leaves at these depths is the code-tree for a prefix code with minimum maximum pointwise redundancy with respect to P, and a minimax tree for $\{\log p_1, \ldots, \log p_n\}$.

Consider a multiset $W = \{w_1, \ldots, w_n\}$ of weights and let $W' = \{w_1 + c, \ldots, w_n + c\}$ for some value c. By definition, $M(W') = M(W) + c$ and any minimax tree for W' becomes a minimax tree for W when we subtract c from each leaf's weight. If we set $c = -\log(\sum_i 2^{w_i})$, then $\sum_i 2^{w_i + c} = 2^c \sum_i 2^{w_i} = 1$; therefore, $W' = \{\log p_1, \ldots, \log p_n\}$ for some probability distribution $P = p_1, \ldots, p_n$ and we can use Drmota and Szpankowski's algorithm directly to build minimax trees for W' and, thus, for W. Without loss of generality, we henceforth assume the given multiset W of weights is equal to $\{\log p_1, \ldots, \log p_n\}$ for some probability distribution P (so, in particular, each $w_i \leq 0$). We can restate the theorem Drmota and Szpankowski proved to establish the correctness of their algorithm – and which also establishes the correctness of our own – in terms of minimax trees instead of prefix codes, as follows.

Theorem 6 ([10])**.** *If $W = \{w_1, \ldots, w_n\}$ is a multiset of weights (meeting the assumption above), $X = \{x_1, \ldots, x_n\} = \{|w_1| - \lfloor |w_1| \rfloor, \ldots, |w_n| - \lfloor |w_n| \rfloor\}$ and x_i is the largest element in $X \cup \{0\}$ such that*

$$\sum_{x_j \leq x_i} 1/2^{\lfloor |w_j| \rfloor} + \sum_{x_j > x_i} 1/2^{\lceil |w_j| \rceil} \leq 1,$$

then any minimax tree for $\{-\lfloor |w_j| \rfloor : x_j \leq x_i\} \cup \{-\lceil |w_j| \rceil : x_j > x_i\}$ becomes a minimax tree for W when we replace each leaf's weight $-\lfloor |w_j| \rfloor$ or $-\lceil |w_j| \rceil$ by w_j.

If $x_1 \leq \cdots \leq x_n$ and $x_i > 0$ then, by Theorem 6, i is the largest index such that $\{\lfloor |w_j| \rfloor : x_j \leq x_i\} \cup \{\lceil |w_j| \rceil : x_j > x_i\}$ satisfies the Kraft Inequality. To build a minimax tree for W with Drmota and Szpankowski's algorithm, we compute and sort X; use binary search to find i, in each round testing whether the Kraft Inequality holds; build a minimax tree for $\{-\lfloor |w_1| \rfloor, \ldots, -\lfloor |w_i| \rfloor, -\lceil |w_{i+1}| \rceil, \ldots, -\lceil |w_n| \rceil\}$; and replace each leaf's weight $-\lfloor |w_j| \rfloor$ or $-\lceil |w_j| \rceil$ by w_j. Our version differs in three ways: we use generalized selection instead of sorting and binary search; we use a new data structure to test the Kraft Inequality; and we use either of our algorithms from Section 2 to build the minimax tree for $\{-\lfloor |w_1| \rfloor, \ldots, -\lfloor |w_i| \rfloor, -\lceil |w_{i+1}| \rceil, \ldots, -\lceil |w_n| \rceil\}$. In the remainder of this section we first show how to use generalized selection to find i in $\mathcal{O}(n)$ time, excluding the time needed to test the Kraft Inequality; we then show how to perform all the necessary tests in a total of $\mathcal{O}(n)$ time using our new data structure. Since each of our algorithms from Section 2 takes $\mathcal{O}(n)$ time, it follows that we can build a minimax tree for W in $\mathcal{O}(n)$ time.

To find x_i in $\mathcal{O}(n)$ time with general selection, we start with the multiset $X_1 = X \cup \{0\}$ and repeat the following procedure until we reach the empty set: in the rth round, we use the linear-time selection algorithm due to Blum et al. [5] to find the current multiset X_r's median x_m, then test whether

$$\sum_{x_j \leq x_m} 1/2^{\lfloor |w_j| \rfloor} + \sum_{x_j > x_m} 1/2^{\lceil |w_j| \rceil} \leq 1;$$

if so, we remove those elements of X_r that are less than or equal to x_m and recurse on the resulting multiset; if not, we remove those elements of X_r that are greater than or equal to x_m and recurse. The element x_i is the largest median we consider for which the test is positive. Since the size of the multisets decreases by a factor of at least 2 in each round, we use $\mathcal{O}(\log n)$ rounds and we find all the medians in a total of $\mathcal{O}(n)$ time.

By the same arguments we used to prove Lemma 1, we can assume, without loss of generality, that $\lceil |w_j| \rceil \leq n - 1$ for each j. To test the Kraft Inequality, we use a data structure consisting of two n-bit binary fractions, S_1 and S_2, each broken into $(\log n)$-bit blocks and initially set to 0. For $1 \leq k \leq n - 1$, adding $1/2^k$ to either fraction takes $\mathcal{O}(1)$ amortized time, for the same reason that incrementing a binary counter takes $\mathcal{O}(1)$ amortized time (see, e.g., [8, Section 17.3]). Nondestructively testing whether $S_1 + S_2 \leq 1$ takes $\mathcal{O}(n/\log n)$ time, because adding each corresponding pair of blocks takes $\mathcal{O}(1)$ time and, by induction, the number carried from each pair to the next is at most 1; resetting either fraction to 0 takes $\mathcal{O}(1)$ time for each block, i.e., $\mathcal{O}(n/\log n)$ time in total.

Before starting to search for x_i, we set $S_1 = \sum_j 1/2^{\lceil |w_j| \rceil}$ in $\mathcal{O}(n)$ time. Throughout our generalized selection, we maintain the invariant that, at the beginning of the rth round,

$$S_1 = \sum_j 1/2^{\lceil |w_j| \rceil} + \sum_{0 < x_j < \min(X_r)} 1/2^{\lceil |w_j| \rceil}$$

and $S_2 = 0$. In the rth round, we set

$$S_2 = \sum_{\min(X_r) \leq x_j \leq x_m} 1/2^{\lceil |w_j| \rceil}$$

in $\mathcal{O}(|X_r|)$ time. Since

$$\begin{aligned} S_1 + S_2 &= \sum_j 1/2^{\lceil |w_j| \rceil} + \sum_{0 < x_j < \min(X_r)} 1/2^{\lceil |w_j| \rceil} + \sum_{0 < \min(X_r) \leq x_j \leq x_m} 1/2^{\lceil |w_j| \rceil} \\ &= \sum_{x_j \leq x_m} 1/2^{\lfloor |w_j| \rfloor} + \sum_{x_j > x_m} 1/2^{\lceil |w_j| \rceil}, \end{aligned}$$

we can test the Kraft Inequality in $\mathcal{O}(n/\log n)$ time by checking whether $S_1 + S_2 \leq 1$. If the test is positive, then we add S_2 to S_1 in $\mathcal{O}(n/\log n)$ time; if the test is negative, then we do not change S_1. In either case, straightforward calculation shows that, afterwards,

$$S_1 = \sum_j 1/2^{\lceil |w_j| \rceil} + \sum_{0 < x_j < \min(X_{r+1})} 1/2^{\lceil |w_j| \rceil}$$

so the first part of our invariant is maintained. Finally, we reset $S_2 = 0$ in $\mathcal{O}(n/\log n)$ time, so the second part of our invariant is maintained. Since $|X_r| = \mathcal{O}(n/2^r)$, the rth round takes a total of $\mathcal{O}(n/2^r + n/\log n)$ time. Since $\sum_{r\geq 1} n/2^r = n$ and we use $\mathcal{O}(\log n)$ rounds, it follows that our whole generalized selection takes $\mathcal{O}(n)$ time. This completes the proof of our main result.

Theorem 7. *Given a multiset W of n real weights, we can build a minimax tree for W in $\mathcal{O}(n)$ time.*

4. Applications

Suppose a transmitter wants to send a string $s[1 \ldots m]$ over an alphabet of size n to a receiver who already knows the frequencies of the distinct characters in s. Since there are $m!/\prod_i m_i!$ possible arrangements of the characters in s, where m_i is the frequency of the ith distinct character, the transmitter must send at least $\log(m!/\prod_i m_i!)$ bits in the worst case. One way the transmitter can nearly meet this bound is to send the $\lceil \log(m!/\prod_i m_i!)\rceil$-bit binary representation of s's lexicographic rank among all strings with the same composition. A more practical way is to use decrementing arithmetic coding, i.e., encoding each character based on the distribution of characters in the suffix remaining to be encoded. Ignoring the redundancy of arithmetic coding, calculation shows this takes

$$\sum_i \log \frac{m-i+1}{\mathrm{occ}(s[i], s[i \ldots m])} = \log\left(\frac{m!}{\prod_j m_j!}\right)$$

bits, where $\mathrm{occ}(s[i], s[i \ldots m])$ denotes the number of occurrences of $s[i]$ in $s[i \ldots m]$.

Now suppose the receiver does not know the frequencies beforehand. As long as the alphabet is not too large, the transmitter can still nearly meet the same bound by using incrementing arithmetic coding, i.e., encoding each character based on the distribution of characters in the prefix already encoded (with each frequency incremented to avoid null probabilities). Again ignoring the redundancy of arithmetic coding, calculation shows this takes

$$\sum_i \log \frac{i+n-1}{\mathrm{occ}(s[i], s[i \ldots (i-1)]) + 1} < \log\left(\frac{m!}{\prod_j m_j!}\right) + n\log(m+n)$$

bits. For more information about incrementing and decrementing arithmetic coding, we refer the reader to Howard and Vitter's analysis [22].

Finally, suppose the transmitter wants to make a single pass over s, encoding each character as soon as it is read and in such a way that the receiver can decode it as soon as its codeword is read. This problem is called adaptive prefix coding, and it is the reason we became interested in building minimax trees in the first place. If the transmitter encodes each character with a Shannon code based on the distribution of characters in the prefix already encoded (again incrementing each frequency), then it sends at most

$$\sum_i \left\lceil \log \frac{i+n-1}{\mathrm{occ}(s[i], s[i \ldots (i-1)]) + 1} \right\rceil < \log\left(\frac{m!}{\prod_j m_j!}\right) + m + n\log(m+n)$$

bits. It follows from Robbins' extension [37] of Stirling's Formula that, if $H(P)$ is the entropy of the normalized distribution P of characters in s, then

$$mH(P) - \mathcal{O}(n\log(m/n)) \leq \log\left(\frac{m!}{\prod_j m_j!}\right) \leq mH(P) + \mathcal{O}(\log m).$$

Therefore, assuming $n = o(m/\log m)$, adaptive prefix coding has an upper bound of $(H(P)+1)m + o(m)$ bits.

In a recent paper with Nekrich [18] we showed this bound is worst-case optimal. Moreover, we showed that, assuming $n = o(m/\log^{5/2} m)$, it can be achieved while using $\mathcal{O}(1)$ worst-case encoding

and decoding time per character. Nekrich [34] compared implementations of adaptive Shannon coding and adaptive Huffman coding (which has weaker worst-case bounds [32,40]) and found that, in practice, adaptive Shannon coding is faster but produces slightly longer encodings. One reason for this might be that, if the input is generated by a Markov source, say, instead of adversarially then, according to the Asymptotic Equipartition Property (see, e.g., [9]), adaptive Shannon coding and adaptive Huffman coding will eventually behave like semi-static Shannon coding and semi-static Huffman coding, respectively. Of course, it would be nice to improve adaptive Shannon coding's practical performance while retaining its optimal worst-case bounds.

Suppose the transmitter encodes each character with a prefix code with minimum maximum pointwise redundancy with respect to distribution of characters in the prefix already encoded (again incrementing each frequency). For $1 \leq i \leq m$, let $g_i < 1$ be the smallest value such that, in the prefix code the transmitter uses to encode $s[i]$, each character a is assigned a codeword of length at most

$$\log \frac{i+n-1}{\mathrm{occ}(a, s[1 \ldots (i-1)]) + 1} + g_i,$$

and let $g = \sum_i g_i/m$; then the transmitter sends at most $(H(P) + g) + o(m)$ bits. Notice this bound is never worse than adaptive Shannon coding's bound and, although the lower bound we proved with Nekrich implies $g \approx 1$ in the worst case, in practice it might be smaller. We are currently studying how to efficiently implement adaptive prefix coding with minimum maximum pointwise redundancy, aiming to achieve the same bound we proved for adaptive Shannon coding, i.e., $\mathcal{O}(1)$ worst-case time to encode and decode each character. We hope that, although our linear-time construction algorithm for such codes may not help us directly, it will give us a useful insight into their nature.

Fortunately, considering minimax trees for prefix coding has led us to an interesting problem to which our linear-time construction algorithm is directly applicable. Suppose we want to build a good prefix code with which to compress a file but are given only a sample of its characters. Let $P = p_1, \ldots, p_n$ be the normalized distribution of characters in the file, let $Q = q_1, \ldots, q_n$ be the normalized distribution of characters in the sample and suppose our codewords are $C = c_1, \ldots, c_n$. An ideal code for Q assigns the ith character a codeword of length $\log(1/q_i)$ (which may not be an integer), and the average codeword's length using such a code is $H(P) + D(P \parallel Q)$, where $H(P) = \sum_i p_i \log(1/p_i)$ is the entropy of P and $D(P \parallel Q) = \sum_i p_i \log(p_i/q_i)$ is the relative entropy between P and Q. The entropy measures our expected surprise at a character drawn uniformly at random from the file, given P; the relative entropy (also known as the informational divergence or Kullback–Leibler pseudo-distance) measures the increase in our expected surprise when we estimate P by Q, and is often used to quantify how well Q approximates P (see, e.g., [9]).

Consider the best worst-case bound we can achieve, given only Q, on how much the average codeword's length exceeds $H(P) + D(P \parallel Q)$. A result by Katona and Nemetz [26] implies we do not generally achieve a constant bound on the difference when C is a Huffman code for Q. (Given P, of course, the best bound we could achieve on how much the average codeword's length exceeds $H(P)$, would be the redundancy of a Huffman code for P.) For example, if $q_1, \ldots, q_n$ are proportional to $F_n, \ldots, F_1$, where F_i denotes the ith Fibonacci number (i.e., $F_1 = F_2 = 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 3$), then the codewords' lengths are $1, \ldots, n-2, n-1, n-1$ in any Huffman code for Q. If p_n is sufficiently close to 1, then

$$H(P) + D(P \parallel Q) \approx \log(1/q_n) = \log \sum_{i=1}^{n} F_i = n \log \phi + \mathcal{O}(1)$$

but the average codeword's length $\sum_i p_i |c_i| \approx n - 1$, so for large n the difference is about $(1/\log \phi - 1)n \approx 0.44n$, where $\phi \approx 1.62$ is the golden ratio.

As long as $q_i > 0$ whenever $p_i > 0$, the average codeword's length

$$\begin{aligned} \sum_i p_i |c_i| &= \sum_i p_i (\log(1/p_i) + \log(p_i/q_i) + \log q_i + |c_i|) \\ &= H(P) + D(P \parallel Q) + \sum_i p_i (\log q_i + |c_i|) \end{aligned}$$

(if $q_i = 0$ but $p_i > 0$ for some i, then $D(P \parallel Q)$ is infinite). Notice each $|c_i|$ is the length of a branch in the code-tree for C. Therefore, the best bound we can achieve is

$$\begin{aligned}\min_C \max_P \left\{ \sum_i p_i(\log q_i + |c_i|) \right\} &= \min_C \max_i \{\log q_i + |c_i|\} \\ &= M(\log q_1, \ldots, \log q_n),\end{aligned}$$

which is less than 1 by inspection of Drmota and Szpankowski's algorithm (see also [9, Theorem 5.4.3] and [3,11,36]). Moreover, we achieve this bound when the code-tree for C has the same shape as a minimax tree for $\{\log q_1, \ldots, \log q_n\}$.

Now suppose we want to design a good group test (see, e.g., [1,2]) to find the unique target in a set, given only an estimate Q – presumably gained from past experience or experimentation – of the probability distribution P according to which the target is chosen. A group test allows us to choose, repeatedly, a subset of the elements and check whether the target is among them. We can represent a group test as a decision tree in which each leaf is labelled with an element and each internal node is labelled with the concatenation of its children's labels. Because such a decision tree can be viewed as the code-tree for a prefix code, and vice versa, the expected number of checks we make exceeds $H(P) + D(P \parallel Q)$ by as little as possible when the decision tree for our group test has the same shape as a minimax tree for $\{\log q_1, \ldots, \log q_n\}$.

Acknowledgments

Many thanks to Michael Baer, Giovanni Manzini and the anonymous referees for their advice. The second author's work was funded by Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics" while at the University of Eastern Piedmont, Italy; by the Sofja Kovalevskaja Award from the Alexander von Humboldt Foundation and the German Federal Ministry of Education and Research while at Bielefeld University, Germany; and by Millennium Institute for Cell Dynamics and Biotechnology (ICDB) MIDEPLAN Grant ICM P05-001-F while at the University of Chile.

References

[1] R. Ahlswede, I. Wegener, Search Problems, Wiley, 1987.
[2] M. Aigner, Combinatorial Search, Wiley, 1988.
[3] M.B. Baer, Tight bounds on minimum maximum pointwise redundancy, in: Proceedings of the International Symposium on Information Theory, 2008.
[4] M.B. Baer, Personal communication, 2009.
[5] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, Journal of Computer and System Sciences 7 (4) (1973) 448–461.
[6] A. Blumer, R.J. McEliece, The Rényi redundancy of generalized Huffman codes, IEEE Transactions on Information Theory 34 (5) (1988) 1242–1249.
[7] D. Coppersmith, M.M. Klawe, N. Pippenger, Alphabetic minimax trees of degree at most t, SIAM Journal on Computing 15 (1) (1986) 189–192.
[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press, McGraw-Hill, 2001.
[9] T.M. Cover, J.A. Thomas, Elements of Information Theory, second ed., Wiley, 2006.
[10] M. Drmota, W. Szpankowski, Generalized Shannon code minimizes the maximal redundancy, in: Proceedings of the 5th Latin American Symposium on Theoretical Informatics, 2002.
[11] M. Drmota, W. Szpankowski, Precise minimax redundancy and regret, IEEE Transactions on Information Theory 50 (11) (2004) 2686–2707.
[12] W.S. Evans, D.G. Kirkpatrick, Restructuring ordered binary trees, Journal of Algorithms 50 (2) (2004) 168–193.
[13] N. Faller, An adaptive system for data compression, in: Record of the 7th Asilomar Conference on Circuits, Systems and Computers, 1973.
[14] G. Franceschini, S. Muthukrishnan, M. Pătraşcu, Radix sorting with no extra space, in: Proceedings of the 15th European Symposium on Algorithms, 2007.
[15] T. Gagie, Dynamic Shannon coding, in: Proceedings of the 12th European Symposium on Algorithms, 2004.
[16] T. Gagie, Dynamic Shannon coding, Information Processing Letters 102 (2–3) (2007) 113–117.
[17] T. Gagie, A new algorithm for building alphabetic minimax trees, Fundamenta Informaticae 97 (3) (2009) 321–329.
[18] T. Gagie, Y. Nekrich, Worst-case optimal adaptive prefix coding, in: Proceedings of the 11th Symposium on Algorithms and Data Structures, 2009.
[19] R.G. Gallager, Variations on a theme by Huffman, IEEE Transactions on Information Theory 24 (6) (1978) 668–674.

[20] M.C. Golumbic, Combinatorial merging, IEEE Transactions on Computers 25 (11) (1976) 1164–1167.
[21] H.J. Hoover, M.M. Klawe, N. Pippenger, Bounding fan-out in logical networks, Journal of the ACM 31 (1) (1984) 13–18.
[22] P.G. Howard, J.S. Vitter, Analysis of arithmetic coding for data compression, Information Processing and Management 28 (6) (1992) 749–764.
[23] T.C. Hu, D.J. Kleitman, J. Tamaki, Binary trees optimum under various criteria, SIAM Journal on Applied Mathematics 37 (2) (1979) 246–256.
[24] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the IRE 40 (1952) 1089–1101.
[25] M. Karpinski, Y. Nekrich, A fast algorithm for adaptive prefix coding, Algorithmica 55 (1) (2009) 29–41.
[26] G.O.H. Katona, T.O.H. Nemetz, Huffman codes and self-information, IEEE Transactions on Information Theory 22 (3) (1976) 337–340.
[27] D.G. Kirkpatrick, M.M. Klawe, Alphabetic minimax trees, SIAM Journal on Computing 14 (3) (1985) 514–526.
[28] D.G. Kirkpatrick, T.M. Przytycka, An optimal parallel minimax tree algorithm, in: Proceedings of the 2nd Symposium on Parallel and Distributed Processing, 1990.
[29] M.M. Klawe, B. Mumey, Upper and lower bounds on constructing alphabetic binary trees, SIAM Journal on Discrete Mathematics 8 (4) (1995) 638–651.
[30] D.E. Knuth, Dynamic Huffman coding, Journal of Algorithms 6 (2) (1985) 163–180.
[31] L.G. Kraft, A device for quantizing, grouping, and coding amplitude-modulated pulses, M.Sc. Thesis, Massachusetts Institute of Technology, 1949.
[32] R.L. Milidiú, E.S. Laber, A.A. Pessoa, Bounding the compression loss of the FGK algorithm, Journal of Algorithms 32 (2) (1999) 195–211.
[33] N. Nakatsu, Bounds on the redundancy of binary alphabetical codes, IEEE Transactions on Information Theory 37 (4) (1991) 1225–1229.
[34] Y. Nekrich, An efficient implementation of adaptive prefix coding, in: Proceedings of the Data Compression Conference, 2007.
[35] D.S. Parker Jr., Combinatorial merging and Huffman's algorithm, IEEE Transactions on Computers 28 (5) (1979) 365–367.
[36] F. Rezaei, C.D. Charalambous, Robust coding for uncertain sources: a minimax approach, in: Proceedings of the International Symposium on Information Theory, 2005.
[37] H. Robbins, A remark on Stirling's formula, American Mathematical Monthly 62 (1955) 26–29.
[38] C.E. Shannon, A mathematical theory of communication, Bell System Technical Journal 27 (1948) 379–423. 623–645.
[39] J. van Leeuwen, On the construction of Huffman trees, in: Proceedings of the 3rd International Colloquium on Automata, Languages and Programming, 1976.
[40] J.S. Vitter, Design and analysis of dynamic Huffman codes, Journal of the ACM 34 (4) (1987) 825–845.
[41] R.W. Yeung, Alphabetic codes revisited, IEEE Transactions on Information Theory 37 (3) (1991) 564–572.