Alphabetic Coding with Exponential Costs[☆]

Michael B. Baer

VMware, Inc., 71 Stevenson St., 13th Floor, San Francisco, CA 94105-0901, USA

Abstract

This note considers an alphabetic binary tree formulation in a family of nonlinear problems. An application of this family occurs when a random outcome needs to be determined via alphabetically ordered search within a stochastic time window. Rather than finding a decision tree minimizing $\sum_{i=1}^{n} w(i)l(i)$, this variant involves minimizing $\log_a \sum_{i=1}^{n} w(i)a^{l(i)}$ for a given $a \in (0,1)$. Herein a dynamic programming algorithm finds the optimal solution in $O(n^3)$ time and $O(n^2)$ space; methods traditionally used to improve the speed of optimizations in related problems, such as the Hu-Tucker procedure, fail for this problem. This note thus also introduces two algorithms which can find a suboptimal solution in linear time (for one) or $O(n \log n)$ time (for the other), with associated redundancy bounds guaranteeing their coding efficiency.

Keywords: Approximation algorithms, dynamic programming, information retrieval, Rényi entropy, tree searching

1. Introduction

Applications such as searching[9] and coding theory[7] make extensive use of binary trees. We denote the length (number of edges) of a path from the root to node $i \in \{1, 2, \ldots, n\}$ of the tree as $l(i)$, and the weight (usually probability) of the leaf as $w(i)$. Given a set of weights, Huffman's algorithm[7] finds a tree minimizing cost function

$$\sum_{i=1}^{n} w(i)l(i) \tag{1}$$

and Hu and Tucker's algorithm [6] finds an optimal *alphabetic* tree:

Definition 1. An **alphabetic** tree is a tree with leaves in numerical order given inorder tree traversal $(1, 2, \ldots, n$ from left to right, ignoring internal nodes, which are unlabeled).

Multiple papers independently considered the problem of minimizing *cost*

$$L_a(w, \boldsymbol{l}) \triangleq \log_a \sum_{i=1}^{n} w(i)a^{l(i)} \qquad a > 0,\ a \neq 1 \tag{2}$$

for unconstrained (Huffman-like) minimization[6, p. 254] [11, p. 485] [8, p. 231], the solution of which is very similar to that of Huffman's algorithm. All three papers primarily concern $a > 1$; the first of these — by Hu, Kleitman, and Tamaki [6] — extends the Hu-Tucker algorithm to solve the alphabetically constrained version of this problem, whereas Humblet [8] noted that the Huffman-like solution also solves the unconstrained (2) for $a < 1$, in which $\log_a x$ is monotonically decreasing and the objective's summation term is thus maximized.

A recent paper showed that the $a < 1$ problem describes certain situations of single-shot decision-making[2]. Given a window of time corresponding to a memoryless random variable, if we wish to find the leaf of the binary tree through constant-time edge traversal, this is found in time with probability $a^{L_a(w,\boldsymbol{l})}$ — the *tree weight*, which we thus wish to maximize — for some known $a < 1$. However, solving the alphabetic version of this problem remained unaddressed.

Here we present an $O(n^3)$ algorithm for minimizing (2) that is somewhat similar to Gilbert and Moore's method [5] for (1). One might posit that similar modifications of more efficient methods like Hu-Tucker could speed up the algorithm. However, we show that this is not true for the aforementioned Hu-Kleitman-Tamaki modification — which only succeeds for $a > 1$ — and for a similarly modified Knuth tree search algorithm[9]. Finally we present two methods, related to those for the linear problem, which find suboptimal solutions in $O(n)$ and $O(n \log n)$ time, leading to simple coding efficiency bounds for both these solutions and the optimal ones.

2. Optimal Alphabetic Trees

Because the alphabetic tree problem imposes leaf order, each decision of which child to take, represented by a 0 (for left) or 1 (for right), is equivalent to a question of the form, "Is the output greater than or equal to s?"

[☆]Material in this paper was presented at the 2006 International Symposium on Information Theory, Seattle, Washington, USA.[1]

Email address: mbaer@vmware.com (Michael B. Baer)

Figure 1: **Procedure for Finding an Optimal Code**

1. $W_{j,j} \leftarrow w(j)\ \forall\ j \in [1,n],\ W_{j,k} \leftarrow 0\ \forall\ 1 \le j < k \le n$ {initialize}
2. **for** $z \leftarrow 1$ to $n-1$ {right index minus left index}
3. **for** $j \leftarrow 1$ to $n-z$ {left (node) index}
4. **for** $s \leftarrow j+1$ to $j+z$ {find max split}
5. **if** $W_{j,j+z} < aW_{j,s-1} + aW_{s,j+z}$
6. $W_{j,j+z}, s_{j,j+z} \leftarrow (aW_{j,s-1} + aW_{s,j+z}, s)$

where s is one of the possible symbols, a symbol we call the *splitting point*:

Definition 2. The **splitting point** of an internal node (or the corresponding subtree that has it as its root) is the smallest index among the leaves of the right subtree of that internal node. Each **codeword** $c(i)$ is the sequence of bits corresponding to the sequence of decisions (path) to arrive at leaf i. The overall set of codewords — alphabetic code C — fully describes the tree, as does length vector $\boldsymbol{l}$, the sequence of lengths $\{l(i)\}$.

Fig. 1 is our pseudocode adaptation of the dynamic programming approach of Gilbert and Moore[5] to this problem (2), and is equivalent to

$$W_{j,k} \leftarrow a \max_{s\in\{j+1,j+2,\ldots,k\}} (W_{j,s-1} + W_{s,k}) \quad \text{starting with } W_{j,j} \leftarrow w(j) \tag{3}$$

for $1 \le j \le k \le n$, operating in $O(n^3)$ time (as can be seen from inner loop 4) and $O(n^2)$ space (as can be seen from initialization step 1). This inductively maximizes tree weight $W_{j,k}$ for items j through k for each value of $k-j$ from 0 to $n-1$, thereby minimizing the corresponding optimum tree cost, $L_a(w,\boldsymbol{l}) = \log_a W_{1,n}$. As long as left and right subtrees of a given (sub)tree are optimal — e.g., for the main tree, $W_{1,s-1} = \sum_{i=1}^{s-1} w(i)a^{l(i)-1}$ and $W_{s,n} = \sum_{i=s}^{n} w(i)a^{l(i)-1}$, so that tree weight $W_{1,n}$ is $a(W_{1,s-1} + W_{s,n})$ — a substitution argument (e.g., [9]) means that, for the best s, the tree will be optimal. Backtracking (not shown in Fig. 1) finds the implied tree in a top-down fashion, each splitting point s for subtree with range $[j,k]$ denoting two child subtrees of range $[j,s-1]$ and $[s,k]$.

Knuth [9] reduced the algorithmic complexity of Gilbert and Moore's method for (1) by using the fact that the splitting point of an optimal tree of size n must be between the splitting points of the two optimal subtrees of size $n-1$. With (2), this no longer holds. Consider $a = 0.6$ with input weights $w = (8,1,9,6)$. The splitting point of $(8,1,9)$ is $s = 3$ ($w(s) = w(3) = 9$, yielding subtrees with $(8,1)$ and (9)), and the splitting point of $(1,9,6)$ is $s = 4$ ($w(s) = 6$). However, the optimal splitting point of $(8,1,9,6)$ is $s = 2$ ($w(s) = 1$).

Similarly, for (2) with $a > 1$, the Hu-Tucker-Kleitman method finds an optimal alphabetic solution. The algorithm begins with the input weights arranged as leaves in numerical order $(1,2,\ldots,n$ in a line). It then combines the two items i and j that, of all pairs of items without a leaf separating them, have a minimum weight sum, putting it in the place of either node, both of which are now (ordered) children. In the original Hu-Tucker algorithm (equivalent to a approaching 1), this item is given weight $w(i) + w(j)$, whereas the Hu-Kleitman-Tamaki modification uses weight $aw(i) + aw(j)$. Both algorithms then find the minimum weighted pair among those pairs of distinct items (uncombined input leaves and combined items) without any *uncombined* leaf between them, placing the resulting node in the place of either original node. Continuing on, we obtain a tree that is not necessarily alphabetical, but which has the same lengths as an alphabetic tree which can be easily reconstructed, (optimally) solving the problem (for $a > 1$).

However, consider again $a = 0.6$, this time for weights $(8,1,9,6,2)$. The Hu-Kleitman-Tamaki method first combines 6 and 2, then 8 and 1, then the first combined node with 9, and finally the remaining two nodes, resulting in a tree with lengths $\boldsymbol{l}' = (2,2,2,3,3)$ and $L_a(w,\boldsymbol{l}') \approx -4.121$. However, a tree with lengths $\boldsymbol{l}'' = (1,3,3,3,3)$, having $L_a(w,\boldsymbol{l}'') \approx -4.232$, shows that the Hu-Kleitman-Tamaki solution is nonoptimal.

Result 1. Knuth's method for speeding up dynamic programming fails for $a < 1$, as does the Hu-Kleitman-Tamaki method (which was optimal for $a > 1$).

3. Suboptimal Algorithms and Bounds

In this section, we add the assertion $\sum_{i=1}^{n} w(i) = 1$ to our problem, which can be considered an optimization of (2) with constraints:

1. The binary tree Kraft inequality, $\sum_{i=1}^{n} 2^{-l(i)} \le 1$;
2. The constraint that $l(i)$ is a natural number;
3. The alphabetic constraint.

All three constraints are necessary to have an alphabetic code. The first and second of these are necessary and sufficient for the lengths to correspond to a binary tree. Relaxing the second and third allows for a numerical solution which can bound the performance of the optimal solution. The numerical solution, $l^\dagger$, shown by Campbell [3, 4] is

$$l^\dagger \triangleq -\frac{1}{1+\log_2 a} \cdot \log_2 w(i) + \log_2 \left(\sum_{j=1}^{n} w(j)^{\frac{1}{1+\log_2 a}} \right).$$

Taking $l^s(i) \triangleq \lceil l^\dagger(i) \rceil$, similarly to a Shannon code, we have a code that violates only the alphabetic constraint.

In order to obtain a near-optimal solution, the algorithm in Fig. 2 has a linear-time variant patterned after

Figure 2: **Procedure for Finding a Near-Optimal Code**

1. Start with an optimal or near-optimal nonalphabetic code with length vector $\boldsymbol{l}^{\text{non}}$, either the Huffman-like $\boldsymbol{l}^{\text{h}}$ or the Shannon-like $\boldsymbol{l}^{\text{s}}$.
2. Find the set of all *minimal points*: i such that $1 < i < n$, $l^{\text{non}}(i) < l^{\text{non}}(i-1)$, and $l^{\text{non}}(i) < l^{\text{non}}(i+1)$; or $i \in [j, j+k]$ minimizing $w(i)$ for $l^{\text{non}}(j-1) > l^{\text{non}}(j) = l^{\text{non}}(j+1) = \cdots = l^{\text{non}}(j+k) < l^{\text{non}}(j+k+1)$.
3. Assign a preliminary alphabetic code with lengths $l^{\text{pre}}(i) = l^{\text{non}}(i) + 1$ for all minimal points and $l^{\text{pre}}(i) = l^{\text{non}}(i)$ for all other items. The first codeword is $l^{\text{pre}}(1)$ zeros. Each additional codeword $c(i)$ follows,
   (a) if $l^{\text{pre}}(i) \le l^{\text{pre}}(i-1)$, by truncating $c(i-1)$ to $l^{\text{pre}}(i)$ bits and adding 1 to the integer that the binary codeword represents, **or**,
   (b) if $l^{\text{pre}}(i) > l^{\text{pre}}(i-1)$, by adding 1 to the integer/codeword $c(i-1)$ and appending $l^{\text{pre}}(i) - l^{\text{pre}}(i-1)$ zeros.
4. Traverse the resulting code tree, removing redundant nodes by replacing any only child with its grandchild or grandchildren. This process ends with an alphabetic code satisfying $\sum_{i=1}^{n} 2^{-l(i)} = 1$.

---

that in [12] — relying on $\boldsymbol{l}^{\text{s}}$ — and an $O(n \log n)$-time variant patterned after [10] — instead using $\boldsymbol{l}^{\text{h}}$, those lengths obtained from the optimal code tree for the problem lacking the alphabetic constraint.

Every step after the first takes linear time with linear space, thus the overall complexity of the algorithms. Step 3 is the method by which Nakatsu showed that any nonalphabetic code can be made into an alphabetic code with similar lengths[10]. (The use of weights as a tie breaker and the nonlinearity of the problem do not change the validity of the algorithm.) Step 4 is the method by which Yeung showed that any alphabetic code can be made into another alphabetic code with $\sum_{i=1}^{n} 2^{-l(i)} = 1$ without lengthening any codewords[12]. Thus this is a hybrid and extension of these two approaches.

For $w = (8/26, 1/26, 9/26, 6/26, 2/26)$ with $a = 0.6$, applying the Shannon-like version of this algorithm, we find that $\boldsymbol{l}^{\text{s}} = (2, 13, 1, 4, 10)$, preliminary codeword lengths are $\boldsymbol{l}^{\text{pre}}_{\text{s}} = (2, 13, 2, 4, 10)$, and the preliminary code is

$$C = (00, 01\mathit{00000000000}, 10, 11\mathit{00}, 11\mathit{01000000}).$$

The italicized bits are redundant, and therefore so are the corresponding nodes in the code tree. They are thus removed in Step 4, yielding the code tree with lengths $(2, 2, 2, 3, 3)$. Tree weight is $a^{L_a(w,\boldsymbol{l})} \approx 0.316$ (or $L_a(w, \boldsymbol{l}) \approx 0.851$), close to the optimal weight of about 0.334 (or $L_a(w, \boldsymbol{l}^*) \approx 0.843$). Using the Huffman-like suboptimal algorithm yields $\boldsymbol{l}^{\text{h}} = (2, 4, 1, 3, 4)$, a preliminary tree with lengths $\boldsymbol{l}^{\text{pre}}_{\text{h}} = (2, 4, 2, 3, 4)$, and an output tree with lengths $(2, 2, 2, 3, 3)$, identical to the above. The same w with $a = 0.7$ yields an optimal tree in the Huffman-like version.

These approaches lead to coding bounds for $a \in (0.5, 1)$. If $a \le 0.5$, an optimal nonalphabetic code is a fixed code known as the unary code (shown, e.g., in [2]), so bounds for such cases follow via similar means; we omit these in the interest of space. Similarly, although the nonalphabetic terms here are useful in combination with other known bounds (e.g., [1]) to further improve these alphabetic bounds, we omit the more complicated improved bounds.

Theorem 1. *Given $a \in (0.5, 1)$ and weight function w, let*

- $L^{\bar{\text{a}}}_a(w)$ *be (2) optimized over alphabetic codes (as in prior section),*
- $L^{\tilde{\text{h}}}_a(w)$ *result from the suboptimal $\boldsymbol{l}^{\text{h}}$-based $O(n \log n)$-time algorithm,*
- $L^{\tilde{\text{s}}}_a(w)$ *result from the suboptimal $\boldsymbol{l}^{\text{s}}$-based $O(n)$-time algorithm,*
- $L^{\text{s}}_a(w)$ *be $L_a(w, \boldsymbol{l}^{\text{s}})$, $L^{\text{h}}_a(w)$ be $L_a(w, \boldsymbol{l}^{\text{h}})$, and $L^{\text{non}}_a(w)$ be $L_a(w, \boldsymbol{l}^{\text{non}})$ (using those $\boldsymbol{l}$ values from Fig. 2).*

Then

$$\begin{aligned} H_\alpha(w) &\le L^{\text{h}}_a(w) \le L^{\bar{\text{a}}}_a(w) \le L^{\tilde{\text{h}}}_a(w) \\ &< 1 + L^{\text{h}}_a(w) < 2 + H_\alpha(w) \end{aligned} \tag{4}$$

$$\begin{aligned} H_\alpha(w) &\le L^{\text{h}}_a(w) \le L^{\bar{\text{a}}}_a(w) \le L^{\tilde{\text{s}}}_a(w) \\ &< 1 + L^{\text{s}}_a(w) < 2 + H_\alpha(w) \end{aligned} \tag{5}$$

where $H_\alpha(w)$ is the Rényi entropy for $\alpha = (1 + \log_2 a)^{-1}$

$$\begin{aligned} H_\alpha(w) &= \frac{1}{1-\alpha} \log_2 \sum_{i=1}^{n} w(i)^\alpha \\ &= (\log_a 2a) \left(\log_2 \sum_{i=1}^{n} w(i)^{\frac{1}{1+\log_2 a}} \right) \end{aligned}$$

Proof. This is a corollary of Campbell's Shannon-like bounds for $a > 0.5$ [3, 4] — $H_\alpha(w) \le L^{\text{h}}_a(w) \le L^{\text{s}}_a(w) < 1 + H_\alpha(w)$ — along with the facts that (a) the two suboptimal algorithm lengths corresponding to items 1 and n are no greater than those in $\boldsymbol{l}^{\text{non}}$ and (b) no other length exceeds the corresponding length in $\boldsymbol{l}^{\text{non}}$ by 1 or more. This results in $L^{\tilde{\text{h}}}_a(w) < 1 + L^{\text{h}}_a(w)$ and $L^{\tilde{\text{s}}}_a(w) < 1 + L^{\text{s}}_a(w)$ due to (2), and, since no alphabetic tree is better than the optimal alphabetic tree and no alphabetic tree is better than the optimal Huffman-like tree, we arrive at (4) and (5).

The lower limit to $L^{\bar{\text{a}}}_a(w)$ is satisfied by $(0.5, 0.5)$, while the upper limit is approached by $(\epsilon, 1 - 2\epsilon, \epsilon)$, for which $H_\alpha(w) \to 0$ and $L^{\bar{\text{a}}}_a(w) \to 2$.

Both these algorithms and the bounds due to analogous inequalities apply to $a > 1$ and to the traditional alphabetic problem ($a \to 1$, where H_1 is Shannon entropy [4]). For the traditional problem, the $\boldsymbol{l}^{\mathrm{h}}$-based suboptimal version of the above algorithm is a strict improvement on Yeung's [12] due to Step 4.

Acknowledgments

The author would like to thank T. C. Hu and J. David Morgenthaler for discussions and encouragement on this topic.

References

[1] M. B. Baer. Rényi to Rényi — source coding under siege. In *Proc., 2006 IEEE Int. Symp. on Information Theory*, pages 1258–1262, July 9–14, 2006.

[2] M. B. Baer. Optimal prefix codes for infinite alphabets with nonlinear costs. *IEEE Trans. Inf. Theory*, IT-54(3):1273–1286, Mar. 2008.

[3] L. L. Campbell. A coding problem and Rényi's entropy. *Inf. Contr.*, 8(4):423–429, Aug. 1965.

[4] L. L. Campbell. Definition of entropy by means of a coding problem. *Z. Wahrscheinlichkeitstheorie und verwandte Gebiete*, 6:113–118, 1966.

[5] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell Syst. Tech. J.*, 38:933–967, July 1959.

[6] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM J. Appl. Math.*, 37(2):246–256, Apr. 1979.

[7] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40(9):1098–1101, Sept. 1952.

[8] P. A. Humblet. Generalization of Huffman coding to minimize the probability of buffer overflow. *IEEE Trans. Inf. Theory*, IT-27(2):230–232, Mar. 1981.

[9] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[10] N. Nakatsu. Bounds on the redundancy of binary alphabetical codes. *IEEE Trans. Inf. Theory*, IT-37(4):1225–1229, July 1991.

[11] D. S. Parker, Jr. Conditions for optimality of the Huffman algorithm. *SIAM J. Comput.*, 9(3):470–489, Aug. 1980.

[12] R. W. Yeung. Alphabetic codes revisited. *IEEE Trans. Inf. Theory*, IT-37(3):564–572, May 1991.