On Conditional Branches in Optimal Decision Trees

Michael B. Baer
Electronics for Imaging
303 Velocity Way
Foster City, California 94404 USA
Email: Michael.Baer@efi.com

***Abstract*— The decision tree is one of the most fundamental programming abstractions. A commonly used type of decision tree is the alphabetic binary tree, which uses (without loss of generality) "less than" versus "greater than or equal to" tests in order to determine one of n outcome events. The process of finding an optimal alphabetic binary tree for a known probability distribution on outcome events usually has the underlying assumption that the cost (time) per decision is uniform and thus independent of the outcome of the decision. This assumption, however, is incorrect in the case of software to be optimized for a given microprocessor, e.g., in compiling switch statements or in fine-tuning program bottlenecks. The operation of the microprocessor generally means that the cost for the more likely decision outcome can or will be less — often far less — than the less likely decision outcome. Here we formulate a variety of $O(n^3)$-time $O(n^2)$-space dynamic programming algorithms to solve such an optimal binary decision tree problem, optimizing for the behavior of processors with predictive branch capabilities, both static and dynamic. In the static case, we use existing results to arrive at entropy-based performance bounds. Solutions to this formulation are often faster in practice than "optimal" decision trees as formulated in the literature, and, for small problems, are easily worth the extra complexity in finding the better solution. This can be applied in fast implementation of Huffman coding.**

I. Introduction

Consider a problem of assigning grades to tests. These tests might be administered to humans or to objects, but in either case there are grades 1 through n — n being 5 in most academic systems — and the corresponding probabilities of each grade, $p(1)$ through $p(n)$, can be assumed to be known; if unknown, they are assumed to be identical. Each grade is determined by taking the actual score, a, dividing it by the maximum possible score, b, and seeing which of n distinct fixed intervals of the form $[v_{i-1}, v_i)$ the key (ratio) a/b lies in, where $v_0 = -\infty$ and $v_n = +\infty$. This process is repeated for different values of a and b enough times that it is worthwhile to consider the fastest manner in which to determine these scores.

A straightforward manner of assigning scores would be to multiply (or shift) a by a constant k ($\log_2 k$), divide this by b, and use lookup tables on the scaled ratio. However, division is a slow step in most CPUs — and not even a native operation in others — and a lookup table, if large, can take up valuable cache space. The latter problem can be solved by using a numerical comparisons to determine the score, resulting in a *binary decision tree* (also known as an *alphabetic binary tree*). In fact, with this decision tree, we can eliminate division altogether; instead of comparing scaled ratio ka/b with grade cutoff value, v_i, we can equivalently compare ka with bv_i, replacing the slow division of variable integers with a fast multiplication of a variable and a fixed integer. Depending on the application, this can be useful even if $b = 1$. The only matter that remains is determining the structure of the decision tree.

```
if (V >= 34)          A. compare V, 34
                      B. branch to M if V<34
  if (V >= 42)        C. compare V, 42
                      D. branch to K if V<42
    if (V >= 65)      E. compare V, 65
                      F. branch to I if V<65
      P = 1;          G. P = 1
    else              H. go to N
      P = 2;          I. P = 2
  else                J. go to N
    P = 3;            K. P = 3
else                  L. go to N
  P = 4;              M. P = 4
                      N. end
```

Fig. 1. Steps in a simple decision tree

The desired tree has a large variety of applications — including the compilation of switch (case) statements [8], [23] — and an optimized decision tree is known as an *optimal alphabetic binary tree*. Often times these decision trees are hard coded into software for the sake of efficiency, as in the high-speed low-memory ONE-SHIFT Huffman decoding technique introduced in [22] and illustrated using C code in Fig. 2 of the same paper. A shorter but similar decision tree is illustrated in Fig. 1 above by means of C and assembly-like pseudocode. We discuss this sample tree in Section II, where a pictorial representation is given as Fig. 2.

Algorithms used for finding such trees generally find trees with minimum expected path length, or, equivalently, minimum expected number of comparisons [5], [11], [15]. We, however, want a tree that results in minimum average run time. The general assumption in finding an optimal decision tree is that these goals are identical, that is, that each decision (edge) takes the same amount of time (cost) as any other; this is noted in Section 6.2.2 of Knuth's *The Art of Computer Programming* [18, p. 429]. In exercise 33 of Section 6.2.2, however, it is conceded that this is not strictly true; in the first edition, the exercise asks for an algorithm for where there is an inequity in cost between a fixed cost for a left branch and a fixed cost for a right branch [16], and, in the second edition, a reference is given to such an algorithm [13]. Such an approach has been extended to cases where each node has a possibly different, but still fixed, asymmetry [24].

In practice the asymmetry of branches in a microprocessor is different in character from any of the aforementioned formulations. On complex CPUs, such those in the Pentium family, branches are predicted as taken or untaken ahead of execution. If the branch is predicted correctly, operation continues smoothly and the branch itself takes only the equivalent of one or two other instructions, as instructions that would have been delayed by waiting for the branch outcome are instead speculatively executed. However, if the branch is improperly predicted, a penalty for misprediction is incurred, as the results of speculatively executed instructions must be discarded and the processor returned to the state it was at prior to the branch, ready to fetch the correct instruction stream [9]. In the case of the Pentium 4 processor, a mispredicted branch takes the equivalent of scores of instructions [4]. This penalty has only increased with the deeper pipelines of more recent processors.

In this paper, we discuss the construction of alphabetic binary trees that are optimized with respect to the behavior of conditional branches in microprocessors. We introduce a general dynamic programming approach, one applicable to such architecture families as: the Intel Pentium architectures, which use advanced dynamic branch prediction; the ARM architectures, most instances of which use static branch prediction; and Knuth's MMIX architecture, in which branch instructions explicitly "hint" whether or not the corresponding branches are assumed taken or untaken [19, p. 20]. The first two are not only representative of two styles of branching; they are also by far the most popular processor architecture families for 32-bit personal computers and 32-bit embedded applications, respectively. Because the approach introduced here is more general than extant alphabetical and search dynamic programming methods, the algorithms arising out of it are somewhat slower, having $O(n^3)$-time $O(n^2)$-space performance. This generality allows for different costs (run times) for different comparisons due to such behaviors as dynamic branch prediction and the use of conditional instructions other than branches. In the simplest case of static branch prediction, entropy-based performance bounds are obtained based on known results from related unequal edge-cost problems. It should be emphasized that the one-time $O(n^3)$-time $O(n^2)$-space cost of optimization of these (usually small) problems is dwarfed by even the slightest gain in repeated run-time performance. The main contribution is thus a method by which decision trees can be coded on known hardware with minimum expected execution time.

II. No prediction and static prediction

Consider Knuth's pedagogical MMIX architecture [19], which has a simple rule for branching: If a "hint" indicates that a branch should be taken, a taken branch will take fewer cycles; otherwise, an untaken branch will take fewer cycles. Thus, if we know ahead of time which branch is more likely and which less likely, we can hard code the more likely branch to take $1+c$ clock cycles and the less likely branch to take $3+c$ clock cycles, where c represents the time taken by instructions other than the branch itself, e.g., multiplications, additions, comparisons.

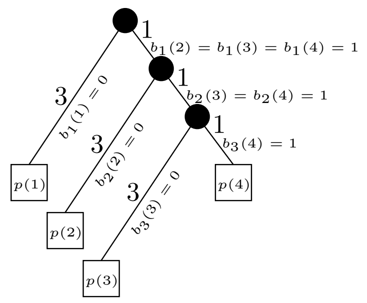

Fig. 2. An optimal branch tree with edge costs for $c = (c_0\ c_1) = (3\ 1)$

It is similarly easy to code the asymmetric bias of the branch for implementations of static branch prediction. In static prediction, opcode or branch direction is used to determine whether or not a branch is taken, the most common rule being that forward conditional branches are presumed taken and backward conditional branches are presumed not taken [9]. Assume, for example, that we want to use a forward branch, which is assumed not to be taken. We thus want the least likely outcome to be that the branch is taken: For example, if it is less likely than not that the item is less than v_i, the branch instruction should correspond to "branch if less than v_i," as in all branches used in Fig. 1.

This *branching problem*, applicable to problems with either no true branch prediction or static branch prediction, considers positive weights c_0 and c_1 such that the cost of a binary path with predictability $b_1 b_2 \cdots b_k$ is

$$\sum_{j=1}^{k} c_{b_j}$$

where $b_j = 0$ for a mispredicted result and $b_j = 1$ for a properly predicted result. Such tree paths are often pictorially illustrated via longer edges on the corresponding tree, so that path height corresponds to path cost, e.g., Fig. 2. This tree corresponds to the C code and pseudocode tree discussed above. Thus the overall expected cost (time) to minimize is

$$T(b) \triangleq \sum_{i=1}^{n} p(i) \sum_{j=1}^{l(i)} c_{b_j(i)}$$

where $p(i)$ is the probability of the ith item, $l(i)$ is the number of comparisons needed, and $b_j(i)$ is 0 if the result of the jth branch for item i is contrary to the prediction and 1 otherwise.

More formally,

Given	$p = (p(1), \ldots, p(n)),\ p(i) > 0$, $\sum_i p(i) = 1$; $c_0, c_1 \in \mathbb{R}_+$ such that $c_0 \geq c_1$
find	B, a full binary tree; b, an assignment of costs to the edges of B such that each nonleaf is connected to one child by an edge with cost c_0 and to the other child by an edge with cost c_1
minimizing	$T(b) \triangleq \sum_{i=1}^{n} p(i) \sum_{j=1}^{l(i)} c_{b_j(i)}$
where	the jth edge along the path from the root to the ith leaf is assigned cost $c_{b_j(i)}$; the number of edges of the path to the ith leaf is $l(i)$.

A sample representation is shown in Fig. 2, labeled with the values of $b_j(i)$. To emphasize the total cost in this pictorial representation, edges are portrayed with height proportional to their cost. The cost (and thus depth) of leaf 3 is, for example,

$$\begin{aligned}\sum_j c_{b_j(3)} &= c_{b_1(3)} + c_{b_2(3)} + c_{b_3(3)} \\ &= c_1 + c_1 + c_0 \\ &= 1 + 1 + 3 \quad = \quad 5.\end{aligned}$$

Table I gives the context for this branching problem among other optimal binary tree problems. These other problems are referred to as in the survey paper [1]. In most problem formulations, edge cost is fixed, and, where it is not fixed, edges generally have costs according to their order, i.e., a left edge has cost c_0 and a right edge has cost c_1. Relaxing this edge-order constraint in the unequal-cost alphabetic problem results in the branching problem we are now considering. Relaxing the alphabetic constraint from either the original alphabetic problem or the branching problem leads to Karp's nonalphabetic problem; since output items in Karp's problem need not be in a given (e.g., alphabetical) order, the tree optimal for the ordered-edge nonalphabetic problem is also optimal for the unordered-edge nonalphabetic problem.

Thus the cost for the optimal tree under Karp's formulation — also called the *lopsided tree problem* — is a lower bound on the cost of the optimal branch tree, whereas the cost for the optimal tree under Itai's (alphabetic) formulation is an upper bound on the cost of the optimal branch tree. This enables the use of bounds in [2] — including the lower bound originally formulated in [20] — for the branching problem. Specifically, if b^{opt} is the optimal branching function, then

$$\frac{H(p)}{d} \leq T(b^{\text{opt}}) \leq \frac{H(p)+1}{d} + \max\{c_0, c_1\}$$

where H is the entropy function $H(p) = -\sum_i p(i)\log_2 p(i)$ and d satisfies $2^{-dc_0} + 2^{-dc_1} = 1$. If $\rho = c_0/c_1$ and x is the sole positive root of $x^\rho + x - 1 = 0$, then $d = -c_1^{-1}\log_2 x$. Thus, for example, when $c = (3\ 1)$,

$$x = \sqrt[3]{\frac{1}{2} + \sqrt{\frac{31}{108}}} - \sqrt[3]{-\frac{1}{2} + \sqrt{\frac{31}{108}}}$$

so $d = \log_2 x^{-1} \approx 0.551$. When $c = (2\ 1)$, $x = 1/\phi$ so $d = \log_2 \phi$, where ϕ is the golden ratio, $\phi = (\sqrt{5}+1)/2$.

The key to constructing an optimizing algorithm is to note that any optimal branching tree must have all its subtrees optimal; otherwise one could substitute an optimal subtree for a suboptimal subtree, resulting in a strict improvement in the result. The branching problem is thus, to use the terminology of [26], *subtree optimal*. Each tree (and subtree) can be defined by its *splitting points*. A splitting point s for the root of the tree means that all items (grades) after s and including s will be in the right subtree while all items before s will be in the left subtree, as per the convention in [6], [15], [18]. Since there are $n-1$ possible splitting points for the root, if we know all potential optimal subtrees for all possible ranges, the splitting point can be found through sequential search of the possible combinations. The optimal tree is thus found through dynamic programming, and this algorithm has $O(n^3)$ time complexity and $O(n^2)$ space complexity, in a similar manner to [6].

The dynamic programming algorithm is relatively straightforward. Each possible optimal subtree for items i through j has an associated cost, $c(i,j)$ and an associated probability $p(i,j)$; at the end, $p(1,n) = 1$ and $c(1,n)$ is the expected cost (run time) of the optimal tree.

The base case and recurrence relation we use are similar to those of [13]. Given unequal branch costs c_0 and c_1 and probability mass function $p(\cdot)$ for 1 through n,

$$\begin{aligned} c(i,i) &= 0 \\ c'(i,j) &= \min_{s \in (i,j]} \{c_0 p(i,s-1) + c_1 p(s,j) + \\ &\qquad c(i,s-1) + c(s,j)\} \\ c''(i,j) &= \min_{s \in (i,j]} \{c_1 p(i,s-1) + c_0 p(s,j) + \\ &\qquad c(i,s-1) + c(s,j)\} \\ c(i,j) &= \min\{c'(i,j), c''(i,j)\} \end{aligned} \qquad (1)$$

where $p(i,j) = \sum_{k=i}^{j} p(i)$ can be calculated on the fly along with $c(i,j)$. The last minimization determines which branch condition to use (e.g., “assume taken” vs. “assume untaken”), while the minimizing value of s is the splitting point for that subtree. The branch condition to use — i.e., the bias of the branch — must be coded explicitly or implicitly in the software derived from the tree.

Knuth [15] and Itai [13] begin with similar algorithms, then reduce complexity by using the property that the splitting point of an optimal tree for their problems must be between the splitting points of the two (possible) optimal subtrees of size $n-1$. The branching problem considered here, however, lacks this property. Consider $p = (0.3\ 0.2\ 0.2\ 0.3)$ and $c = (3\ 1)$, for which optimal trees split either at 2, as in Fig. 2, or at 4, the mirror image of this tree. In contrast, the two largest subtress, as illustrated in the figure and its mirror image, both have optimal splitting points at 3.

The optimal tree of Fig. 2 is identical to the optimal tree returned by Itai's algorithm for order-restricted edges [13].

	restriction on edge order and/or cost		
restriction on output order	**Constant edge cost**	**Fixed edge-cost order**	**Unrestricted edge-cost order**
Alphabetic	Hu-Tucker [5], [6], [11], [18]	Itai [13], [24]	branching problem
Nonalphabetic	Huffman [12], [17], [27]	Karp [3], [7], [14]	

TABLE I
TYPES OF DECISION TREE PROBLEMS

Consider a larger example in which this is not so, the binomial distribution $p = (1\ 6\ 15\ 20\ 15\ 6\ 1)/128$ with $c = (11\ 2)$. If edge order is restricted as in [13], the optimal tree has an expected cost of $967/64 = 15.109375$. If we relax the restriction, as in the problem under consideration here the optimal method has an expected cost of $831/64 = 12.984375$, a 14% improvement.

A practical application of this problem, involving a decision tree, is encountered in implementation of the ONE-SHIFT Huffman decoding technique introduced in [22]. This implementation of optimal prefix coding is fastest for applications with little memory or small caches. Where the ONE-SHIFT technique is the preferred technique, we can apply the methods of this section to optimize the method's decision tree. In the implementation illustrated in [22], the decision tree is used to determine codeword lengths based on 32-bit keys. The suggested "optimal search" strategy involves a hard-coded decision tree in which branches occur if "greater than or equal to" each splitting point; in most static branch schemes, this would result in "less than" taking fewer cycles than "greater than or equal to," but the tree used in [22] was found assuming fixed branch costs [25]. Here we show that we can improve upon this.

Consider the optimal prefix code for random variable X drawn from the Zipf distribution with $n = 2^{16}$, that is,

$$\mathbb{P}[X = i] = \frac{1}{i \sum_{j=1}^{n} j^{-1}}$$

which is approximately equal to the distribution of the n most common words in the English language [28, p. 89]. Using Huffman coding, one can find that this code has codeword lengths, $\ell(X)$, between 4 to 20, with the number of codewords of each size and the probability that the codeword will be a certain size given by Table II.

Consider a decision tree to find codeword lengths with an architecture in which comparisons that result in untaken branches take 3 cycles (for both compare and branch), while comparisons that result in taken branches take 5 cycles. This asymmetry, similar to that of many ARM architectures, is small, but taking advantage of it results in an improved tree. This optimal tree, shown in Fig. 3, takes an average of 15.93 cycles, while the "optimal search" takes an average of 16.44 cycles. This 3.1% improvement, although not as large as the previous examples, is still significant due to the impact of the decision tree on overall algorithm speed.

length (ℓ)	# of codewords		$p(i)$		
4	1	(2^0)	$\mathbb{P}[\ell(X) = 4]$	=	0.08570759
5	2	(2^1)	$\mathbb{P}[\ell(X) = 5]$	=	0.07142299
6	4	(2^2)	$\mathbb{P}[\ell(X) = 6]$	=	0.06509695
7	8	(2^3)	$\mathbb{P}[\ell(X) = 7]$	=	0.06216987
8	16	(2^4)	$\mathbb{P}[\ell(X) = 8]$	=	0.06076807
9	32	(2^5)	$\mathbb{P}[\ell(X) = 9]$	=	0.06008280
10	64	(2^6)	$\mathbb{P}[\ell(X) = 10]$	=	0.05974408
11	128	(2^7)	$\mathbb{P}[\ell(X) = 11]$	=	0.05957570
12	256	(2^8)	$\mathbb{P}[\ell(X) = 12]$	=	0.05949175
13	512	(2^9)	$\mathbb{P}[\ell(X) = 13]$	=	0.05944984
14	1024	(2^{10})	$\mathbb{P}[\ell(X) = 14]$	=	0.05942890
15	2048	(2^{11})	$\mathbb{P}[\ell(X) = 15]$	=	0.05941844
16	4096	(2^{12})	$\mathbb{P}[\ell(X) = 16]$	=	0.05941321
17	8192	(2^{13})	$\mathbb{P}[\ell(X) = 17]$	=	0.05941059
18	16384	(2^{14})	$\mathbb{P}[\ell(X) = 18]$	=	0.05940928
19	32748	(2^{15})	$\mathbb{P}[\ell(X) = 19]$	=	0.05940732
20	2		$\mathbb{P}[\ell(X) = 20]$	=	0.00000262

TABLE II
DISTRIBUTION OF HUFFMAN CODEWORD LENGTHS FOR ZIPF'S LAW

III. MORE ADVANCED MODELS

With dynamic branch prediction [9], which in more advanced forms includes branch correlation, branches are predicted based on the results of prior instances of the same and different branch instructions. This results in complex processor behavior. Often several predictors will be used for the same branch instruction instance; the predictor in a given iteration will be based on the history of that branch instruction instance and/or other branches. In the problem we are concerned with, however, this does not result in as many complications as one might expect; the probability of a given branch outcome conditional on the branches that precede it is identical to the probability of the branch outcome overall. In the case of previous branch outcomes for the same search instance — i.e., those of ancestors in the tree — any given outcome is conditioned on the same events — i.e., the events that lead to the branch being considered. In the case of branches for previous items, if items are independent, so are these branches. In the case of branches outside of the algorithm, these can also be assumed to be either fixed given or independent of the current branch.

Thus, as long as each branch predictor is assigned at most one of the decision tree branches, prediction can be modeled as a random process. This process will result in each predictor converging to a stationary distribution, which can be

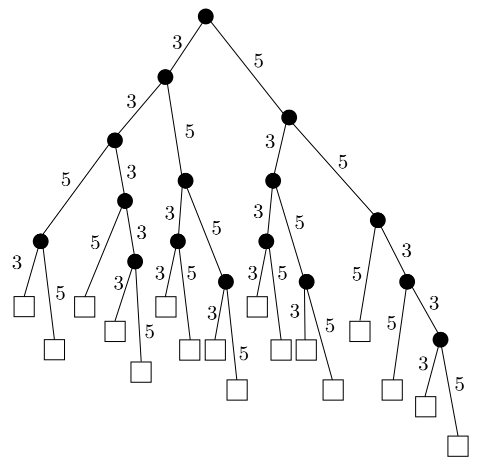

Fig. 3. Optimal branch tree for codeword lengths in optimal prefix coding of Zipf's law

analyzed and optimized for. Simple analysis of the stationary distribution of a branch prediction Markov chain, e.g., [10], can yield the expected time for a given branch direction as a function of the probability of the branch.

Additional performance factors might include an additional asymmetry between taken and untaken branches, the performance of branch target buffers (which are discussed in [21]), and differences among different comparison types. For example, if a $(<, \geq)$ comparison with a certain value has a smaller cost than a comparison with another value — say a comparison with a power of two times a variable is faster due to reduced calculation time — then this can also be taken into account. Similarly, conditional instructions, often preferable to conditional branches, can often be used, but only to eliminate a branch to leaves in the decision tree. Thus branches deciding between only two items might be accounted differently than other types.

With such a variety of coding options, there could be multiple possible costs for any particular decision. A general cost function taking all this into account represents as $C_k(p', p'', i, j, s)$ the cost of choosing the kth of m splitting methods for the step necessary to split a subtree for items $[i, j]$ at splitting point s, with splitting outcome probabilities p' and p''. (The most common value for m is 2, the two choices being to assume a taken branch versus to assume an untaken branch.) The corresponding generalization of (1) is:

$$\begin{aligned} c(i,i) &= 0 \\ c_k(i,j) &= \min_{s \in (i,j]} \{C_k(p(i,s-1), p(s,j), i, j, s) + \\ &\qquad c(i,s-1) + c(s,j)\} \quad \forall k \\ c(i,j) &= \min_{k \in [1,m]} \{c_k(i,j)\}\,. \end{aligned}$$

Once again, this is a simple matter of dynamic programming, and, assuming all C_k are calculable in constant time, this can be done in $O(mn^3)$ time and $O(n^2 + n \log m)$ space, the $\log m$ term accounting for recalculation and storage of the type of cost function (decision method) used for each branch. An even more general version of this could take into account properties of subtrees other than those already mentioned, but we do not consider this here.

References

[1] J. Abrahams, "Code and parse trees for lossless source encoding," *Communications in Information and Systems*, vol. 1, no. 2, pp. 113–146, Apr. 2001.

[2] D. Altenkamp and K. Mehlhorn, "Codes: Unequal probabilities, unequal letter costs," *J. ACM*, vol. 27, no. 3, pp. 412–427, July 1980.

[3] P. G. Bradford, M. J. Golin, L. L. Larmore, and W. Rytter, "Optimal prefix-free codes for unequal letter costs: Dynamic programming with the Monge property," *J. Algorithms*, vol. 42, no. 2, pp. 219–223, Feb. 2002.

[4] A. Fog, "How to optimize for the Pentium® microprocessors," 2004, available from http://www.agner.org/assem/.

[5] A. M. Garsia and M. L. Wachs, "A new algorithm for minimum cost binary trees," *SIAM J. Comput.*, vol. 6, no. 4, pp. 622–642, Dec. 1977.

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

[7] M. J. Golin and G. Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes for unequal letter costs," *IEEE Trans. Inf. Theory*, vol. IT-44, no. 5, pp. 1770–1781, Sept. 1998.

[8] J. L. Hennessy and N. Mendelsohn, "Compilation of the Pascal case statement," *Softw., Pract. Exper.*, vol. 12, no. 9, pp. 879–882, Sept. 1982.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann Publishers, 2003.

[10] P. G. Hoel, S. C. Port, and C. J. Stone, *Introduction to Stochastic Processes*. Boston, MA: Houghton Mifflin Company, 1972.

[11] T. C. Hu and A. C. Tucker, "Optimal computer search trees and variable-length alphabetic codes," *SIAM J. Appl. Math.*, vol. 21, no. 4, pp. 514–532, Dec. 1971.

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sept. 1952.

[13] A. Itai, "Optimal alphabetic trees," *SIAM J. Comput.*, vol. 5, no. 1, pp. 9–18, Mar. 1976.

[14] R. M. Karp, "Minimum-redundancy coding for the discrete noiseless channel," *IRE Trans. Inf. Theory*, vol. 7, no. 1, pp. 27–38, Jan. 1961.

[15] D. E. Knuth, "Optimum binary search trees," *Acta Informatica*, vol. 1, pp. 14–25, 1971.

[16] ——, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 1st ed. Reading, MA: Addison-Wesley, 1973.

[17] ——, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[18] ——, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley, 1998.

[19] ——, *The Art of Computer Programming, Vol. 1, Fascicle 1 : MMIX – A RISC Computer for the New Millennium*. Addison-Wesley, 2005.

[20] R. M. Krause, "Channels which transmit letters of unequal duration," *Inf. Contr.*, vol. 5, no. 1, pp. 13–24, Mar. 1962.

[21] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan. 1984.

[22] A. Moffat and A. Turpin, "On the implementation of minimum redundancy prefix codes," *IEEE Trans. Commun.*, vol. 45, no. 10, pp. 1200–1207, Oct. 1997.

[23] A. Sale, "The implementation of case statements in Pascal," *Softw., Pract. Exper.*, vol. 11, no. 9, pp. 929–942, Sept. 1981.

[24] M. T. Shing, "Optimum ordered bi-weighted binary trees," *Inf. Processing Letters*, vol. 17, pp. 67–70, Aug. 1983.

[25] A. Turpin, Private communication, Nov. 2006.

[26] H. Vaishnav and M. Pedram, "Alphabetic trees—theory and applications in layout-driven logic synthesis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 1, pp. 58–69, Jan. 2001.

[27] J. van Leeuwen, "On the construction of Huffman trees," in *Proc. 3rd Int. Colloquium on Automata, Languages, and Programming*, July 1976, pp. 382–410.

[28] G. K. Zipf, "Relative frequency as a determinant of phonetic change," *Harvard Studies in Classical Philology*, vol. 40, pp. 1–95, 1929.