To appear in IEEE Transactions on Information Theory, October 1978.

December 1977 ESL-P-787

# Variations on a Theme by Huffman

by

Robert G. Gallager*

Massachusetts Institute of Technology
Department of Electrical Engineering
and Computer Science
and
Electronic Systems Laboratory

## Abstract

In honor of the twenty-fifth anniversary of Huffman Coding, four new results about Huffman codes are presented. The first result shows that a binary prefix condition code is a Huffman code iff the intermediate and terminal nodes in the code tree can be listed by non-increasing probability so that each node in the list is adjacent to its sibling. The second result upper bounds the redundancy (expected length minus entropy) of a binary Huffman code by $P_1 + \log_2[2(\log_2 e)/e] = P_1 + .086$ where $P_1$ is the probability of the most likely source letter. The third result shows that one can always leave a code word of length 2 unused and still have a redundancy of at most one. The fourth result is a simple algorithm for adapting a Huffman code to slowly varying estimates of the source probabilities. In essence, one maintains a running count of uses of each node in the code tree and lists the nodes in order of these counts. Whenever the occurrence of a message increases a node count above the count of the next node in the list, the nodes, with their attached subtrees, are interchanged.

---

*This work was supported in part by NSF under Grant NSF-ENG76-24447 and in part by Codex Corp., Newton, Mass. 02195.

## 1) Introduction

Since the appearance 25 years ago of Huffman's [1] classical paper on minimum redundancy variable length source coding, Huffman coding has remained one of the most familiar topics in information theory, but has not seen widespread application. One of the difficulties for application arises in the situation of a single source that produces non-binary letters at a fixed rate and whose output is to be transmitted over a synchronous binary communication channel. If one uses a variable length code, then the rate at which binary digits are delivered to the channel will fluctuate, and the expected rate will depend on the relative frequencies of the source letters. This means, first, that buffering is required between the source and channel, and, second, that the system will fail if the source statistics become such that the expected rate of binary digits exceed the channel capacity. For these reasons, and also for processing simplicity, there has been increasing standardization on fixed length source codes, for example ASCII (American Standard Code for Information Interchange).

The above arguments against variable length source codes are now much less compelling than before for several reasons. The first is the growth of statistical multiplexors, concentrators, and data networks. Since each of these allocate communication resources to sources on the basis of need, buffering exists as a central part of such systems. Because of the large number of sources, mistaken assumptions about some of the source statistics lead to inefficiency but not failure when using variable length codes. In addition such systems require large amounts of protocol, or control information, and the use of fixed length codes for this control, as, for

example, in packet headers, turns out to be very inefficient. Finally the fact that processing costs and storage costs are dropping very much faster than communication costs has fundamentally changed the trade-off between communication efficiency and processing complexity. For all these reasons, one can expect to see much greater use of variable length codes in the future.

The author has recently been studying possible uses of source coding in data networks, and, rather surprisingly, the four quite elementary results described in the abstract turned up. In order to simplify the reader's task as much as possible, we first state and prove each result for the case of binary code words, and then extend the result in the appendix to arbitrary code alphabets. We start with the sibling property, since that forms the basis for the other results. Section 3 on redundancy and Section 4 on adaptive Huffman coding are independent and can be read in either order.

## 2) The Sibling Property

Let A be a discrete source with K letters, $2 \leq K < \infty$, and let $P_k$ denote the probability of letter $a_k$, $1 \leq k \leq K$. It is customary to assume $P_k > 0$, but we allow at most one $P_k$ to be 0 in order to simplify examples in which $P_k$ is allowed to approach zero arbitrarily closely. Let $x_k = (x_k(1), x_k(2), \ldots x_k(n_k))$ be the binary code word for letter $a_k$, $1 \leq k \leq K$. Here $x_k(i)$ is a binary digit, $1 \leq i \leq n_k$, and $n_k$ is called the length of the code word. A binary code for A is the set of code words plus the mapping that maps $a_k \rightarrow x_k$ for $1 \leq k \leq K$. A code word $x_k$ is called a prefix of a code word $x_j$ if $n_k \leq n_j$ and $x_k(i) = x_j(i)$ for each i, $1 \leq i \leq n_k$. A prefix condition code is a code with the property that no

code word is a prefix of any other code word. A prefix condition code can be conveniently represented as a rooted binary code tree (see Figure 1) in which each source letter corresponds to a leaf on the tree, and the associated code word is the sequence of labels on the path from root to leaf. If two nodes are adjacent on a path from the root to a leaf, we say that the one closer to the root is the parent of the other, which is called the child of the parent. Two nodes with a common parent are called siblings.

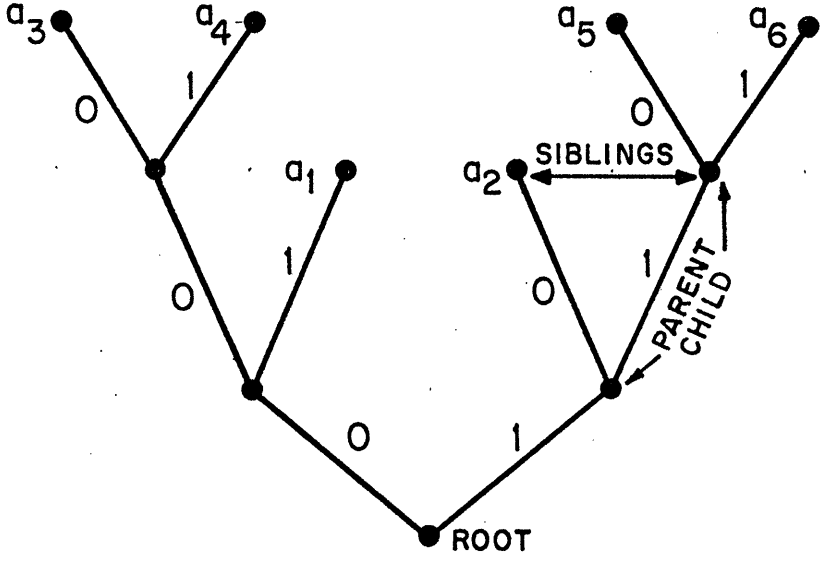
It is well known [2] that an arbitrary concatenation of code words from a prefix condition code can be uniquely decoded into the corresponding source letters. Furthermore, every code that can be uniquely decoded has lengths satisfying the Kraft inequality,

$$\sum_{k=1}^{K} 2^{-n_k} \leq 1 \tag{1}$$

and prefix condition codes can be constructed with any set of lengths satisfying (1).

Huffman [1] developed an algorithm that generates, for any such source, a prefix condition code that is optimum in the sense of minimizing the expected code word length,

$$E(n) = \sum_{k=1}^{K} P_k \, n_k \tag{2}$$




$x_1$ = (01)

$x_2$ = (10)

$x_3$ = (000)

$x_4$ = (001)

$x_5$ = (110)

$x_6$ = (111)

ROOTED BINARY CODE TREE

Figure 1

The algorithm, illustrated in Figure 2, is most easily viewed as starting with the leaves of a rooted tree, and iteratively generating the intermediate nodes. The algorithm is given below:

1) Let L be a list of the probabilities of the source letters corresponding to the leaves of the tree.

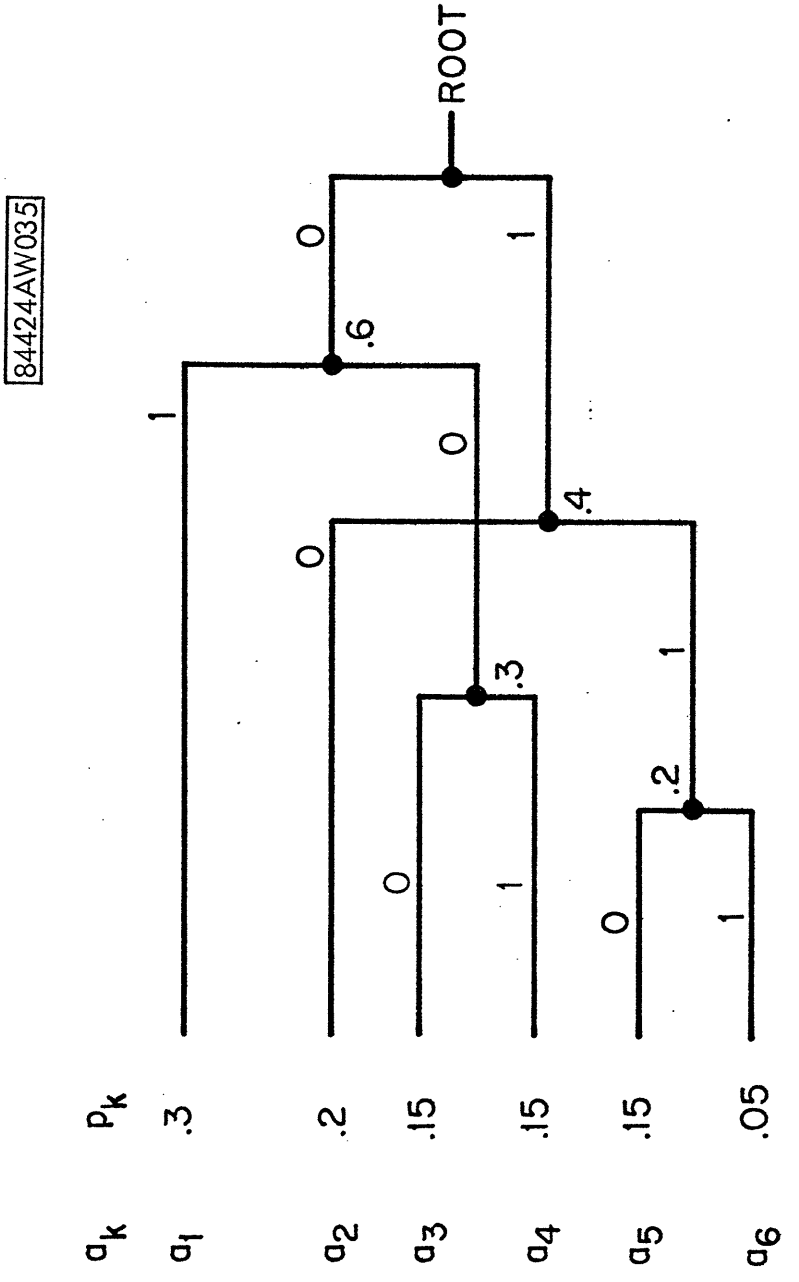
2) Take two smallest probabilities in L, make the corresponding nodes siblings, generate an intermediate node as their parent, and label the link from parent to one child with 0 and the other with 1.

3) Replace the above two probabilities in L with their sum, associated with the new intermediate node. If the new L contains one element, stop, and otherwise return to step 2.

The codes generated by this algorithm are called Huffman codes. Our first objective is to give a structural characterization of Huffman codes, as opposed to the algorithmic characterization just given.

Each of the leaves of a code tree has a probability assigned to it, namely the probability of the corresponding source letter. We also assign a probability to each intermediate node, defined recursively as the sum of the probabilities of its children. An equivalent non-recursive definition is that the probability of an intermediate node is the sum of the probabilities of all leaves for which the path from root to leaf passes through the given intermediate node.

Definition: A binary code tree has the sibling property if each node (except the root) has a sibling and if the nodes can be listed in order of non-increasing probability with each node being adjacent in the list to its sibling.




HUFFMAN ENCODING ALGORITHM

(SAME CODE AS FIGURE 1)

Figure 2

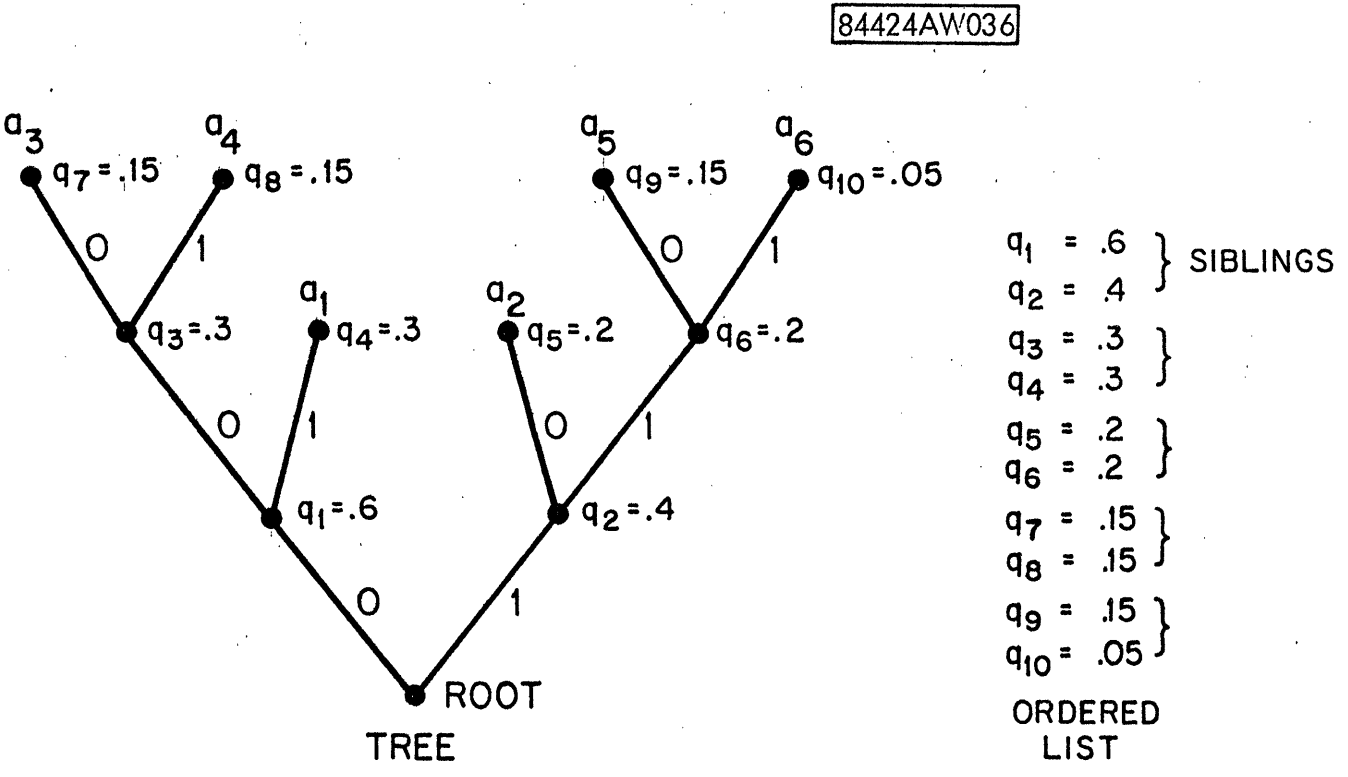
Figure 3 illustrates the sibling property. Note that if several nodes have the same probability, as in Figure 3, the list in order of non-increasing probability is non-unique, and the definition only requires that there be some such list with each node being adjacent to its sibling. Note also (by a simple inductive argument) that the list (excluding the root) must contain 2K-2 elements, and that for each k, $1 \leq k \leq K-1$, the $2k^{th}$ and $2k-1^{th}$ elements on the list must be siblings for the sibling property to hold.

Theorem 1: A binary prefix condition code is a Huffman code iff the code tree has the sibling property.

Proof: First assume that a code tree has the sibling property. Then the last two elements on the ordered list are siblings, and in addition they must be leaves, for if one were an intermediate node, at least one of the children of that intermediate node would have a smaller probability than the intermediate node[1], which is impossible because of the ordering property. Thus these nodes correspond to two smallest probability source letters, and can be made siblings in the first execution of step 2 in the Huffman algorithm. Now remove these siblings from the code tree, removing also the last two elements from the ordered list. The resulting reduced code

---

[1]This is where we use the assumption that at most one source letter have zero probability. The theorem is true without this restriction, but the proof is harder and the restriction is of no importance.




THE SIBLING PROPERTY

Figure 3

tree still has the sibling property and the leaves of the reduced code tree correspond to the list L in the Huffman algorithm after the first execution of step 3. Thus the above argument can be repeated; at each step the Huffman algorithm chooses, as siblings, elements which are siblings in the original code tree. By matching the link labels in the Huffman code to those in the original code tree, the two codes are seen to be identical. Next assume that a binary code tree is generated by the Huffman algorithm, and assume that each time the algorithm executes step 2, we add the two nodes defined as siblings to the top of an initially empty list, putting the less probable below the more probable. The list so generated clearly has each node adjacent to its sibling, so to establish the sibling property, we simply have to show that the list is non-increasing in order of probability. This is trivial, however, since at each iteration, the two elements added to the list have probabilities less than or equal to that of each element in the new L of the Huffman algorithm, and the next two elements added to the list are chosen from this new L. QED

Next define the level of a node as the number of links on the path from the root to the node. It is clear from the optimality of Huffman codes that for each $\ell \geq 1$, the probability of each node at level $\ell$ is less than or equal to the probability of each node at level $\ell$-1. For the purist, this property can be directly derived from the sibling property using induction on $\ell$. Define an ordered Huffman code as a Huffman code in which when two nodes are defined as siblings, the label 0 is assigned to the link going to the more probable of the siblings. Also define a lexicographically ordered code tree as a tree in which, for each $\ell \geq 1$, the

probability of each node at level $\ell$ is less than or equal to the probability of each node at level $\ell$-1, and the probabilities of nodes at level $\ell$ are monotonic non-increasing in the binary number corresponding to their path names from the root.

Corollary: A binary prefix condition code is an ordered Huffman code iff the code tree is lexicographically ordered.

Proof: Lexicographic ordering implies the sibling property, which implies that the code is a Huffman code. Lexicographic ordering also implies that the link from a parent to the more probable of the siblings is labelled 0, implying an ordered Huffman code. Now assume an ordered Huffman code and use induction on the level $\ell$. The nodes at level 1 are lexicographically ordered by construction. Assume for any $\ell > 1$ that the nodes at level $\ell$-1 are lexicographically ordered. By the sibling property, if the probability of one parent is greater than or equal to that of another, the children are correspondingly ordered. This shows that non-sibling nodes at level $\ell$ have the correct ordering. Siblings, however, are correctly ordered by the construction. QED

3) The Redundancy of Huffman Codes

The redundancy r of a source code is defined to be the expected length of the code words minus the binary entropy, $H(P_1,\ldots,P_K)$, of the source probabilities,

$$r = \sum_{k=1}^{K} P_k n_k - H(P_1,\ldots,P_K) \tag{3}$$

where $H(P_1,\ldots,P_K) = -\sum_k P_k \log P_k$.

It is well known [2] that for optimal codes, the redundancy always lies between 0 and 1. The upper limit, 1, is reached by a source with two letters, of probabilities 0 and 1, or more strictly, approached as $\varepsilon \to 0$ by a source with probabilities $1 - \varepsilon$ and $\varepsilon$. Our purpose here is to show that when the most probable letter in a source has a probability much less than 1, then the upper limit on r can be greatly improved upon.

Suppose we have a Huffman code, and using the sibling property, we number all of the nodes (except the root node) in order of decreasing probability and increasing level so that for each k, $1 \leq k \leq K-1$, nodes 2k and 2k-1 are siblings. Let $q_k$ be the probability of the $k^{th}$ node on the list, $1 \leq k \leq 2K-2$. The expected length of the code can be written as

$$E(n) = \sum_{k=1}^{2K-2} q_k \tag{4}$$

In order to see this, perform the conceptual experiment of writing each $q_k$ in (4) as the sum of the leaf probabilities of leaves whose path from the root passes through k. Then a code word i of length $n_i$ has its probabilitiy $P_i$ written in $n_i$ of these sums, showing the equivalence of (4) and (2). We can also rewrite the entropy as

$$H(P_1,\ldots,P_K) = \sum_{k=1}^{K-1} (q_{2k-1} + q_{2k}) H\left(\frac{q_{2k}}{q_{2k-1} + q_{2k}}\right) \tag{5}$$

where $\mathcal{H}$ is the binary entropy function

$$\mathcal{H}(x) = -x \log_2 x - (1-x)\log_2(1-x) \tag{6}$$

Each term in (5) is the probability that a given parent in the code tree will occur times the entropy of the choice of its child. Formally (5) can be established by induction on reduced trees. Combining (5) and (4), we have

$$r = \sum_{k=1}^{K-1} (q_{2k-1} + q_{2k}) \left[1 - \mathcal{H}\left(\frac{q_{2k}}{q_{2k-1} + q_{2k}}\right)\right] \tag{7}$$

Finally let $\ell \geq 1$ be some level at which the tree is full (i.e., for which the tree has $L = 2^{\ell}$ nodes of level $\ell$), but for which there are also nodes at level $\ell + 1$. For $K > 2$, such an $\ell$ must exist. Let m be the smallest integer for which node $2m-1$ is at level $\ell + 1$, and let $q'_1,\ldots,q'_L$ be the probabilities of the nodes at level $\ell$. Splitting the sum in (7) into $k < m$ and $k \geq m$, and writing the terms for $k < m$ in the form of (4), we have

$$r = \ell - H(q'_1,\ldots,q'_L) + \sum_{k=m}^{K-1} (q_{2k-1} + q_{2k}) \left[1 - \mathcal{H}\left(\frac{q_{2k}}{q_{2k-1} + q_{2k}}\right)\right] \tag{8}$$

For K=2, (8) is valid for $\ell = 1$ with the final term omitted.

Theorem 2: Let $P_1$ be the probability of the most likely letter in a finite discrete source. Then the redundancy of the Huffman code for the source satisfies

$$r \leq P_1 + \sigma \tag{9}$$

where $\sigma = 1 - \log_2 e + \log_2(\log_2 e) \approx .086$. For $P_1 \geq 1/2$,

$$r \leq 2 - \mathcal{H}(P_1) - P_1 \leq P_1 \tag{10}$$

Proof: For $0 \leq x \leq 1/2$, $\mathcal{H}(x) \geq 2x$. Using this inequality, the final term in (8) is upper bounded by

$$\sum_{k=m}^{K-1} q_{2k-1} - q_{2k}$$

Since the sequence $q_k$ is non-increasing, this is further upper bounded by $q_{2m-1}$, so that

$$r \leq \ell - H(q_1', \ldots, q_2') + q_{2m-1} \tag{11}$$

Let $n_1$ be the length of the shortest code word, which must correspond to a source letter of probability $P_1$. First assume $P_1 \geq 1/2$. Then $n_1 = 1$, and taking $\ell = 1$ in (11), we obtain

$$r \leq 1 - \mathcal{H}(P_1) + q_{2m-1} \tag{12}$$

Since $q_{2m-1} \leq 1 - P_1$, (12) implies the first inequality in (10). The second inequality is satisfied with equality at $P_1 = 1/2$ and $P_1 = 1$, and is satisfied in between because of convexity. This also establishes (9) for $P_1 \geq 1/2$, and thus whenever $K = 2$. If all the code words are of the same length, $n_1 > 1$, choose $\ell$ in (8) to be $n_1 - 1$, and otherwise choose $\ell = n_1$. In both cases, $q_{2m-1} \leq P_1$, and in both cases, we can order $q'_1,\ldots,q'_L$ to satisfy $q'_1 \geq q'_2 \geq \ldots \geq q'_L \geq q'_1/2$. Let $Q$ be the set of choices for $q'_1,\ldots,q'_L$ that satisfy the above linear inequality constraints along with $\sum q'_i = 1$. Then

$$r \leq \ell - \min_{\{q'_i\} \in Q} H(q'_1,\ldots,q'_L) + P_1 \tag{13}$$

Since H is convex $\cap$, the minimum above must occur at an extreme point of $Q$; the extreme points of $Q$ are those for which, for some n, $1 \leq n \leq L$, $q'_i = q'_1$ for $i \leq n$, and $q'_i = q'_1/2$ for $n < i \leq L$. For a given n, then, $q'_1 = 2/(L+n)$, and

$$\min_{\{q'_i\} \in Q} H(q'_1,\ldots,q'_L) = \min_{1 \leq n \leq L} \left[-\log \frac{2}{L+n} + \frac{(L-n)}{L+n}\right]$$

Further lower bounding by allowing n to take on non-integer values, we have an elementary calculus problem with the minimum value $\ell - \sigma$ where $\sigma = 1 - \log_2 e + \log_2(\log_2 e)$. Substituting this in (13) completes the proof.

The bound on redundancy here is quite tight. For all $P_1 \geq 1/2$, a source with probabilities $(P_1, 1-P_1, 0)$ satisfies the bound with equality. Also, the source with probabilities (1/3, 1/3, 1/3, 0) has a redundancy of .415, whereas the bound for $P_1 = 1/3$ is .419. It can also be shown by tedious calculation that for all $P_1$, there exist sources with $r \geq \sigma$, which shows that the bound is tight in the limit $P_1 \to 0$.

Next we take a somewhat different approach to redundancy. Suppose we want to reserve one or more code words for control or protocol purposes. One could regard these control messages as having probabilities like everything else, but it is frequently more convenient to regard them separately.

Theorem 3: For every finite discrete source there exists a prefix condition code with an unused code word of length 2 and with redundancy, $r \leq 1$.

Proof: We shall construct the desired code by first constructing the Huffman code for the source. We then take the less probable node at level 1, say node 2, and move it to level 2, making it the sibling of a newly created reserved word, and making the parent the sibling of the other level one node; all other parent-children relationships are unchanged. In effect we have lengthened by one each code word stemming from node 2. Let $q_1$ and $q_2$ be the probabilities of the original level one nodes, $q_1 \geq q_2$. Then if r is the redundancy of the original code and r' that of the modified code, $r' = r + q_2$. From (11), using $\ell = 1$, we have

$$r \leq 1 - \mathcal{H}(q_2) + q_2$$

$$r' \leq 1 - \mathcal{H}(q_2) + 2q_2 \leq 1$$

where in the final inequality we have used the fact that $q_2 \leq 1/2$.

Note that for a source with probabilities (1/2, 1/2, 0), this bound is met with equality. It also can be shown that the above procedure for choosing an unused word of length 2 is optimal in the sense of minimizing r'.

4) Adaptive Huffman Codes

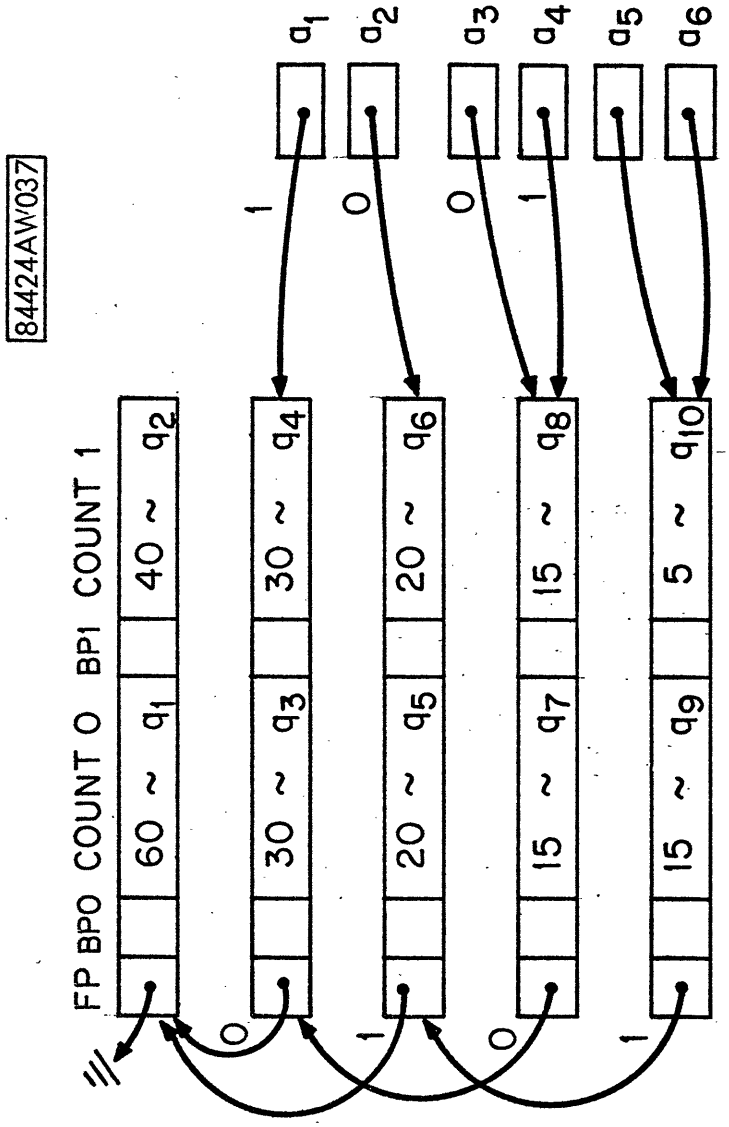
In this section we are interested in Huffman encoders which maintain a running estimate of the source letter probabilities; as these estimates change, the code will change, remaining optimal for the current estimates. Our primary concern is with the algorithm to modify the code rather than the problem of estimating the probabilities. In fact, the method to be used to estimate the letter probabilities is almost trivial. Simply maintain a counter for each letter of the source alphabet, and increment the counter each time that letter occurs. Periodically, say after each N'th letter in the source sequence, multiply each of the letter counts by some fixed number $\alpha < 1$. The current estimate for the probability of a source letter is the current count for that letter divided by the sum of all the counts. Since the counts are proportional to the probability estimates, the algorithm to be described operates directly on the counts, and never needs to calculate the probability estimates.

The choice of $\alpha$ and N determine how quickly the estimates can change, and it can be seen that the time constant for the number of letters entering into the estimate is $N/(1-\alpha)$. As this time constant increases, the adaptation becomes slower, yielding better estimates for slowly varying statistics, but more irrelevant estimates for rapidly varying statistics. For small time constants, of course, the estimates will be noisy, whether the statistics are slowly or quickly varying. For a given time constant, it seems appropriate to chose $\alpha = 1/2$, since this makes the multiplication by $\alpha$ simple and keeps N relatively large.

The algorithm will keep a count for each node in the current code tree. By the sibling property, if the nodes can be listed by decreasing counts so that each node is adjacent to its sibling, then the code is optimal for the current probability estimates.

We can implement this strategy, in a micro-computer, say, by maintaining a fixed list of sibling pairs in storage. For a K letter alphabet, there are K-1 such sibling pairs. The storage location for each sibling pair will contain 5 components, two of which are the current counts for the sibling nodes, and three of which are pointers to be described later. The structure will be maintained in such a way that each count for the top sibling pair will be greater than or equal to each count for the next pair, and so forth down to the bottom of the list.

The structure of the code tree is maintained in the sibling list by a set of forward pointers, indicated by FP in Figure 4. The FP pointer for a given sibling pair points to the parent of the pair, or more strictly to the sibling pair containing the parent, with an extra bit to indicate whether the parent is the 0 sibling or the 1 sibling. The source letters themselves have a separate storage area containing only a pointer to the letter's current location in the sibling list. For example, in Figure 4,




DATA STRUCTURE FOR ADAPTIVE HUFFMAN CODER

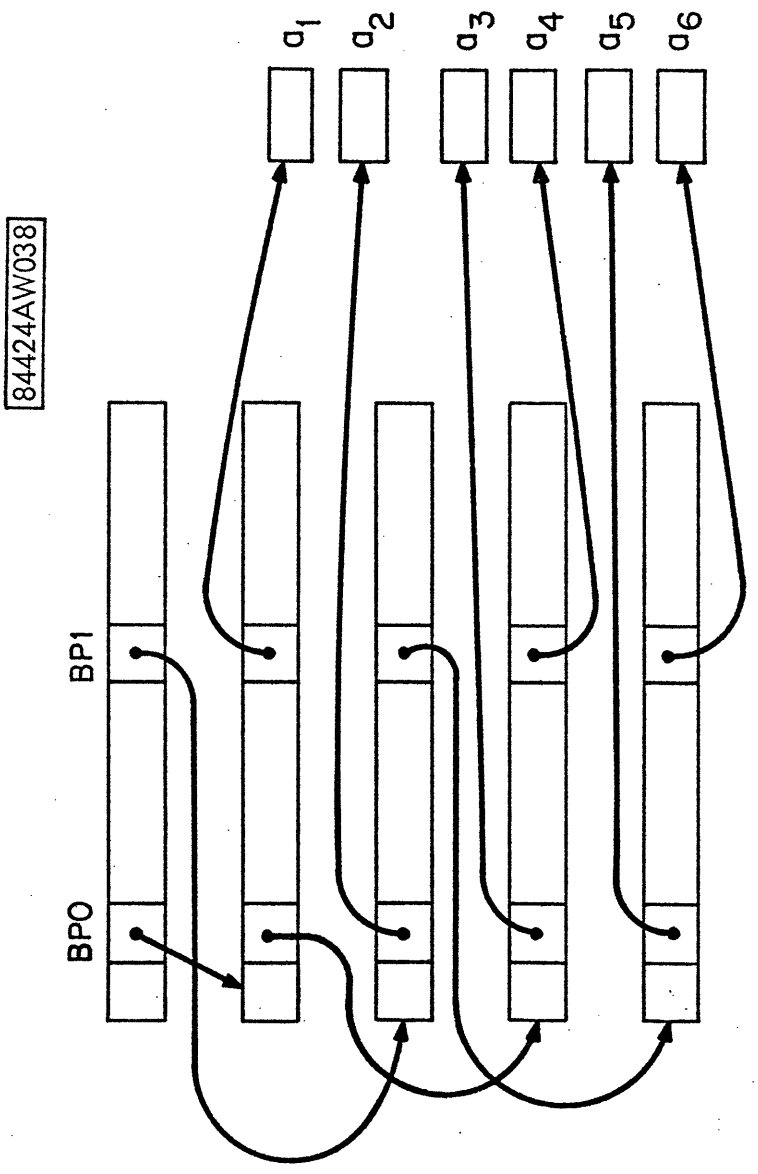
Figure 4

if letter $a_1$ occured, the last digit of code word $x_1$ would be determined as 1, since the pointer from $a_1$ goes to the 1 side of the second sibling pair. Since the FP pointer from this pair goes to the 0 side of the first sibling pair, the code word is determined as 01. The first sibling pair always corresponds to the level 1 nodes and its FP pointer is nil.

We have now seen how code words are generated and next take up the problem of updating the counts and perhaps changing the code. It is important to recognize that a code word must be generated before any changes to the code are allowed, since the decoder must decode the code word to obtain the information on which changes are based. After generating a code word, the counts are incremented, one by one, on the nodes on the path from leaf node to root. For the example of letter $a_1$, first the 1-count in the second sibling pair is incremented from 30 to 31 and then the 0-count in the first sibling pair is incremented from 60 to 61. Each time a count is incremented, the count must be compared with the counts of the next higher sibling pair, and if it exceeds one of these counts, the two nodes must be interchanged, which means that the forward pointers into those nodes must be switched. The purpose of the backward pointers, BP0 and BP1 in Figure 5, is to find these forward pointers without a search. These pointers also allow the decoder to decode easily. We see that a code change is made by changing two FP pointers and two BP pointers. The point of changing the counts one at a time is that if a code change is made when one count is changed, it is the new path to the root rather than the old path that must have its counts incremented. For example if $a_5$ occurs, then $a_3$ and $a_5$ would be interchanged in the tree. The 0-count in the fourth sibling pair would become 16, the 0-count in the second sibling pair would become 31, and the 0-count of the first 61.

It is possible for several code changes to occur for one source digit, but there is at most one change for each count incremented, and thus at most one change for each encoded bit. The entire computation, corresponding to an encoded bit then, is one memory access to find the bit and the location of the next bit, one memory access to find the count to be incremented, two accesses and comparisons to see if the code must be changed, and 4 pointer changes if the code is changed (it is not necessary to interchange the counts, since a comparison for equality can be done before the incrementing, and then the new count can be incremented). Thus, in summary, the computational load is independent of the alphabet size and proportional to the code bit generation rate.

In the above description, we have left out one annoying detail. Many counts in the list could be the same. If one of those counts were to be incremented, the interchange would have to be with the first sibling pair containing that number. The search to find this first sibling pair can be avoided, at the expense of storage, by having a storage location for each such list. Each element on the list has a pointer to that location, and that location contains a pointer to the top of the list (which could contain just one element). Since elements join such a list from the bottom (in terms of the ordered sibling pairs), and leave from the top (after the code switch), no variable length searches are required.




BACK POINTERS FOR STRUCTURE OF FIGURE 4

Figure 5

## Appendix

We now generalize the previous results to the case where the code alphabet is of arbitrary size D rather than binary. We allow D-1 of the source messages to have 0 probability; because of this, we can assume without loss of generality that the size of the message set, K, is $c(D-1) + 1$ for some integer c. The Huffman coding algorithm is then changed by replacing the word two, in the algorithm as given, by the letter D. A code tree has the sibling property if the nodes, excluding the root, can be listed in order of non-increasing probability such that for each i, $1 \leq i \leq c$, nodes iD, iD-1,...,iD - D + 1 are all siblings of each other. The proof of Theorem 1, that a prefix condition code is a Huffman code iff the code tree has the sibling property, is the same as the original proof, with "two" replaced by "D". The lexicographic property also follows in an obvious fashion.

We define the redundancy r of a source code, with a code alphabet size of D, as the expected code word length minus the entropy, in base D, of the source probabilities. If we let $\ell$ be some level at which the code tree is full, let $L = D^{\ell}$ nodes, number the nodes in order of non-increasing probability, and let m be the smallest integer for which node Dm - (D-1) is at level $\ell + 1$, then, as in (8), we have

$$r = \ell - H(q'_1,\ldots,q'_L) + \sum_{k=m}^{c} t_k \left[1 - H(\vec{q}_k)\right] \qquad (A1)$$

when $q'_i$, $1 \leq i \leq L$, is the probability of the $i^{th}$ node in level $\ell$, $t_k = q_{Dk} + q_{Dk-1} + \ldots + q_{Dk-(D+1)}$, and $\vec{q}_k$ is a probability vector, with components $(q_{Dk}/t_k, q_{Dk-1}/t_k, \ldots q_{Dk-D+1}/t_k)$. The final term is 0 if all

terminal nodes have level $\ell$.

The final term in (A1) is more difficult to handle than the corresponding term in the binary case. We need the following lemma:

Lemma: Let $x_1 \geq x_2 \geq \ldots \geq x_D$ be probabilities, $\sum_{i=1}^{D} x_i = 1$, and let $H(x_1,\ldots,x_D)$ be the entropy base D. Then

$$1 - H(x_1,\ldots,x_D) \leq (x_1-x_D)D/\ln D \tag{A2}$$

Proof: By definition,

$$1 - H(x_1,\ldots,x_D) = \sum_{i=1}^{D} x_i \log_D\left(x_i\ D\right)$$

Choose $\lambda_i$, $1 \leq i \leq D$ such that $x_i = \lambda_i x_1 + (1-\lambda_i)x_D$. Then

$$\sum_{i=1}^{D} x_i \log_D\left(x_i\ D\right) \leq \sum_{i=1}^{D} \lambda_i\ x_1 \log_D\left(x_1 D\right) + \sum_{i=1}^{D} (1-\lambda_i)x_D \log_D\left(x_D\ D\right) \tag{A3}$$

Using the fact that $\sum x_i = 1$, we see that $\sum \lambda_i = (1-x_D\ D)/(x_1 - x_D)$, and the right hand side of (A3) is equal to

$$\frac{(1 - x_D\ D)x_1}{x_1 - x_D} \log_D\left(x_1\ D\right) + \frac{(x_1\ D - 1)x_D}{x_1 - x_D} \log_D\left(x_D\ D\right)$$

Since $x_D\ D \leq 1$ and $x_1\ D \geq 1$, we can upper bound this with the inequality $\log_D x \leq (x-1)/\ln D$.

$$\sum_{i=1}^{D} x_i \log_D\left(x_i\ D\right) \leq (1-x_D D)(x_1 D-1)/\ln D$$

$$\leq (1-x_D D)x_1 D/\ln D \leq (x_1 D-x_D D)/\ln D$$

It is likely that the bound in (A2) could be improved somewhat, but it is necessary that the bound increase with increasing D as D/ln D. Substituting (A2) into (A1),

$$r \leq \ell - H(q_1',\ldots q_L') + q_{mD-(D-1)}\ D/\ln D \tag{A4}$$

Assuming that level $\ell + 1$ is non empty, $q_L' \geq q_1'/D$ for an optimum code, and we can lower bound the entropy term in (A4), as in Theorem 2, by

$$H(q_1',\ldots,q_L') \geq \ell - \sigma_D \tag{A5}$$

$$\sigma_D = \log_D(D-1) + \log_D(\log_D e) - \log_D e + \frac{1}{D-1} \tag{A6}$$

Thus the generalization of Theorem 2 is given by (A7) below

$$r \leq \sigma_D + P_1\ D/\ln D \tag{A7}$$

Unfortunately, as D gets large, $\sigma_D \to 1$, but the approach is not rapid.

For example, $\sigma_3 = .135$, $\sigma_5 = .194$, $\sigma_{10} = .269$, $\sigma_{20} = .335$. As in the case of Theorem 2, for any given D, there are sources with P arbitrarily small, for which r is arbitrarily close to $\sigma_D$.

Next we extend Theorem 3 by showing that with a code alphabet size D, it is always possible to have D-1 unused code words of length 2 while still maintaining a redundancy $r' \leq 1$. The strategy is the same as before; construct a Huffman code for the source and then lengthen by one each code word emanating from the least likely level one node, leaving D-1 unused level 2 nodes. Let $\vec{q} = (q_1, q_2, \ldots q_D)$ be the probabilities of the level one nodes, $q_1 \geq q_2 \geq \ldots \geq q_D$ and let r be the redundancy of the original code; then $r' \leq r + q_D$. By considering a line from the probability vector $(1/D, 1/D, \ldots 1/D)$ through $\vec{q}$ to the point $\vec{q}'$ where $q'_D = 0$ and using convexity, we find that

$$H(\vec{q}) \geq D\, q_D$$

Substituting this into (A4) at $\ell=1$, we have

$$r \leq 1 - D\, q_D + D\, q_D/\ln D \tag{A9}$$

$$r' \leq 1 - D\, q_D(1 - 1/(\ln D) - 1/D) \tag{A10}$$

The term in parentheses is positive for $D \geq 4$, leaving us only the case D=3 to consider. First assume that the Huffman code tree has only level 1 nodes. Then the final term in (A9) vanishes and $r' \leq 1$. Next we use

(A1) for $\ell=0$, using the lemma to bound all nodes on the third level or more.

$$r \leq [1 - H(\vec{q})] + \sum_{k=1}^{3} q_k [1 - H(\vec{q}_k)] + 3q'/\ln 3 \qquad \text{(A11)}$$

where $q'$ is the probability of the most probable third level node and is 0 if no third level nodes exist, and $\vec{q}_k$ is the set of conditional probabilities for the second level nodes emanating from the $k^{th}$ first level node; if no second level nodes emanate from node k, we take $[1 - H(\vec{q}_k)] = 0$.

First assume that third level nodes exist. Then $q' \leq 1/9$, and for each first and second order node, the least likely sibling has at least $1/D$ the probability of the most likely so that (A5) applies. Thus

$$r \leq 2\sigma_3 + (3/9)/\ln 3$$

Since $q_3 \leq 1/3$, this implies $r' \leq 1$. Finally assume $q' = 0$, and let $k'$ be the highest number first level node for which second level nodes exist. For $k < k'$, (A5) applies if second level nodes exist for node k, and

$$r' \leq \sigma_3 + \sum_{k=1}^{k'-1} q_k \sigma_3 + q_{k'} + q_3 \qquad \text{(A12)}$$

Since second level nodes exist by assumption, $q_1 \leq q_3/3$, so that $q_1 + q_3 \leq .8$. Using this, it is easy to verify that $r' \leq 1$ for $k' = 1$, 2, 3, completing the proof.

Adaptive Huffman coding for an alphabet size of $D > 2$ is essentially

the same as for D=2. For a source alphabet of $K = c(D-1) + 1$ letters, we need c storage locations, each containing 2D+1 components; there are D counts, one for each sibling, D backpointers, and one forward pointer. For $D > 2$, and perhaps also for D=2, it is desirable to keep the counts ordered within the storage locations, thus requiring only one comparison instead of D to see if the code must be changed for each encoded letter. The trade off here is between number of comparisons and number of code changes, which is really an implementation detail.

## References

1) D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proc. IRE, 40, pp. 1098-1101, 1952.

2) R. G. Gallager, Information Theory and Reliable Communication, Wiley & Sons, New York, New York, 1968.