JOURNAL OF ALGORITHMS 6, 163–180 (1985)

Dynamic Huffman Coding

DONALD E. KNUTH*

Department of Computer Science, Stanford University, Stanford, California 94305

Received July 2, 1981; revised December 2, 1983

This note shows how to maintain a prefix code that remains optimum as the weights change. A Huffman tree with nonnegative integer weights can be represented in such a way that any weight w at level l can be increased or decreased by unity in $O(l)$ steps, preserving minimality of the weighted path length. One-pass algorithms for file compression can be based on such a representation.

1. INTRODUCTION

Given nonnegative weights $(w_1, \ldots, w_n)$, the well-known algorithm of Huffman [2] can be used to construct a binary tree with n external nodes and $n - 1$ internal nodes, where the external nodes are labeled with the weights $(w_1, \ldots, w_n)$ in some order. Huffman's tree has the minimum value of $w_1 l_1 + \cdots + w_n l_n$ over all such binary trees, where l_j is the level at which w_j occurs in the tree.

Binary trees with n external nodes are in one-to-one correspondence with sets of n strings on $\{0, 1\}$ that form a "minimal prefix code." A prefix code is a set of strings in which no string is a proper prefix of another; and a minimal prefix code is a prefix code such that, if α is a proper prefix of some string in the set, then $\alpha 0$ is either in the set or a proper prefix of some string in the set, and so is $\alpha 1$. The correspondence between trees and codes is simply to represent the path from the root to each external node as a string of 0's and 1's, where 0 corresponds to a left branch and 1 to a right branch. An external node at level l corresponds in this way to a string of

*This research was supported in part by National Science Foundation Grant MCS 83-00984 and by Office of Naval Research contract N00014-81-K-0269. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

0196-6774/85 $3.00

length l. For example, the binary tree in Fig. 1 corresponds to the minimal prefix code $\{0, 100, 101, 11\}$.

If the strings $\{\alpha_1, \ldots, \alpha_n\}$ form a prefix code, it is easy to factor any string $\alpha_{i_1}\alpha_{i_2} \cdots \alpha_{i_m}$ into its component α's, by reading the 0's and 1's from left to right, traversing the binary tree until an external node is reached. Therefore if we have a string $a_{i_1}a_{i_2} \cdots a_{i_m}$ in some n-letter alphabet $\{a_1, \ldots, a_n\}$, we can encode that string as $\alpha_{i_1}\alpha_{i_2} \cdots \alpha_{i_m}$, a sequence of 0's and 1's.

Suppose the letter a_j occurs w_j times in the original string $a_{i_1}a_{i_2} \cdots a_{i_m}$; then the total length of $\alpha_{i_1}\alpha_{i_2} \cdots \alpha_{i_m}$ after encoding is $w_1l_1 + \cdots + w_nl_n$, where l_j is the length of α_j. Hence Huffman's algorithm produces a prefix code such that the resulting sequence of 0's and 1's has minimum length.

All of this is, of course, well known. But in practice there are times when the weights w_j are not given, since the string $a_{i_1}a_{i_2} \cdots a_{i_m}$ is to be encoded "on line." In this case we can imagine a dynamically changing prefix code in which the encoding of a_{i_k} is based on a Huffman tree for the frequencies of occurrence in the previously encoded portion $a_{i_1} \cdots a_{i_{k-1}}$; the encoding process thereby "learns" the frequencies of the encoded message as it proceeds. When the code is changing, we might wonder how the output could be unraveled later; but the decoding process can also be learning how the code is evolving, in exactly the same way as the encoding process does, since the decoder also knows the frequencies of occurrence in $a_{i_1} \cdots a_{i_{k-1}}$ when it is decoding a_{i_k}. Thus, by continually updating a Huffman code, both sender and receiver can keep synchronized with each other; a nearly optimum encoding will be obtained, assuming that the transmission is error-free.

In particular, this scheme avoids the need to transmit any information about the weights w_j. Such additional transmission of weights would be necessary if a two-pass strategy were adopted under which the sender first counted the frequencies of each letter and then used a globally optimum Huffman code.

Section 2 of this paper develops the basic ideas by which continuously varying Huffman trees can be maintained, where the updating is done in

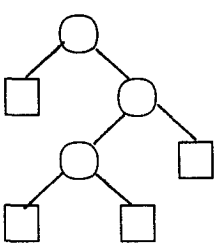

FIGURE 1

real time (i.e., in time proportional to the length of transmission). Section 3 presents an example, and detailed algorithms for the encoding and decoding appear in Sections 4, 5, and 6. The concluding section presents empirical results and discusses various ways to modify the procedure.

The basic principles of adaptive Huffman coding were discovered by Gallager [1]. This paper extends Gallager's work in three ways: (1) Pointer manipulations necessary for real-time operation are described explicitly. (2) Zero-weight letters are taken into account. (3) Positive weights may be decreased as well as increased.

2. A Strategy

Recall that Huffman's algorithm combines the two smallest weights w_i and w_j, replacing them by their sum $w_i + w_j$, and repeats this process until only one weight is left. For example, given the weights $(2, 3, 4, 5)$, the first step combines $2 + 3 = 5$; the remaining weights are $(4, 5, 5)$. The next step combines $4 + 5 = 9$, and then comes $5 + 9 = 14$. There is some ambiguity about which 5 is which, so this procedure might form two different trees (see Fig. 2), depending on where the 5 from '$2 + 3 = 5$' is placed. Both of these trees are optimum for the given weights, since $5 \cdot 1 + 4 \cdot 2 + 2 \cdot 3 + 3 \cdot 3 = 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 2 + 5 \cdot 2$. However, if we now increase the given weight 5 to 6, the left-hand tree is better, while if we increase the given weight 2 to 3 the right-hand tree is better. A procedure for updating Huffman trees dynamically must therefore be able to convert from each of these possibilities to the other.

The weighted combination process of Huffman's algorithm leads to a nondecreasing sequence

$$(x_1, x_2, \ldots, x_{2n-1})$$

of node weights for the internal and external nodes, and this sequence is the same for all Huffman trees on the given weights $(w_1, \ldots, w_n)$. For example,

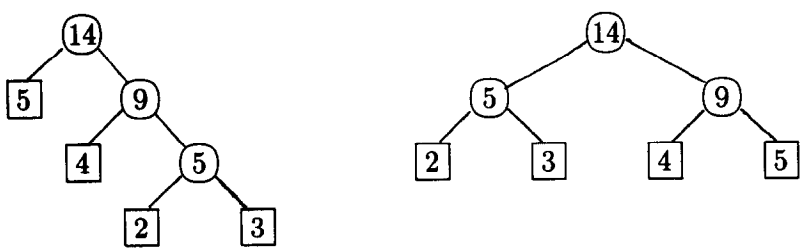

Figure 2

both of the trees above correspond to the x-sequence (2, 3, 4, 5, 5, 9, 14). The weighted path length $w_1 l_1 + \cdots + w_n l_n$ is easily seen to be equal to the sum $x_1 + \cdots + x_{2n-2}$.

The $n - 1$ internal nodes of each Huffman tree that correspond to a particular sequence $(x_1, x_2, \ldots, x_{2n-1})$ have the weights $(x_1 + x_2, x_3 + x_4, \ldots, x_{2n-3} + x_{2n-2})$. Conversely, a binary tree is a Huffman tree on the weights $(w_1, \ldots, w_n)$ if it satisfies the following properties:

(i) The n external nodes have been assigned the weights $(w_1, \ldots, w_n)$ in some order, and each internal node has been assigned a weight equal to the sum of the weights of its two children;

(ii) The $2n - 1$ nodes (external and internal) can be arranged in a sequence $y_1, \ldots, y_{2n-1}$ such that if x_j is the weight of node y_j we have $x_1 \leqslant \cdots \leqslant x_{2n-1}$, and such that nodes y_{2j-1} and y_{2j} are siblings (children of the same parent), for $1 \leqslant j < n$, where this common parent node does not precede y_{2j-1} and y_{2j} in the sequence.

For we can show inductively that such a tree is one that Huffman's algorithm might construct: The values of x_1 and x_2 must equal the values of the smallest two weights of $(w_1, \ldots, w_n)$; and the remainder of the tree satisfies the same condition on $n - 1$ weights, if these two weights are replaced by $x_1 + x_2$ and if the parent of y_1 and y_2 is regarded as an external node.

Given a tree satisfying (i) and (ii), let $y_{i_0}, y_{i_1}, \ldots, y_{i_l}$ be a sequence of nodes leading from some external node of weight w to the root. If w is replaced by $w + 1$, we must increase each of $x_{i_0}, x_{i_1}, \ldots, x_{i_l}$ by unity; the resulting tree will still satisfy (i) and (ii), provided that we had

$$x_{i_j} < x_{i_j+1} \qquad \text{for } 0 \leqslant j < l, \tag{*}$$

in the original tree. Thus, the same tree will be optimum both for w and $w + 1$, whenever condition (*) holds.

Suppose that all the weights w_i are positive. Then we can always transform a given Huffman tree into another one that satisfies condition (*), by interchanging subtrees of equal weight: First let i'_0 be maximum such that $x_{i'_0} = x_{i_0}$, and when i'_k has been defined let i'_{k+1} be maximum such that $x_{i'_{k+1}}$ has the weight of the parent of $y_{i'_k}$. (If $i'_k = 2n - 1$, however, let $l' = k$ and terminate the construction.) Now the tree can be permuted by interchanging the subtree rooted at y_{i_0} with the subtree rooted at $y_{i'_0}$, then interchanging the subtree rooted at the parent of $y_{i'_0}$ with the subtree rooted at $y_{i'_1}$, and so on. We obtain a tree satisfying (i) and (ii), where the path from w to the root is $y_{i'_0}, y_{i'_1}, \ldots, y_{i'_{l'}}$, and where $x_{i'_j} < x_{i'_j+1}$ for $0 \leqslant j < l'$. It is not difficult to verify by induction that $i_j \leqslant i'_j$ for $0 \leqslant j \leqslant l'$, hence $l' \leqslant l$; in other words, at most l interchanges are necessary to obtain a Huffman tree satisfying (*).

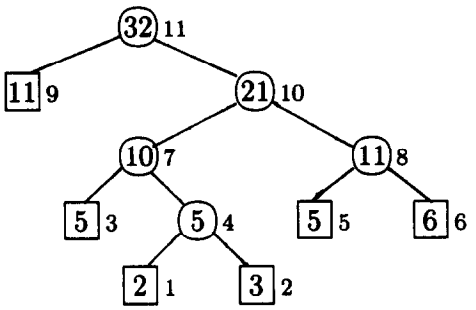

FIGURE 3

The construction in the preceding paragraph is the key to an efficient algorithm for maintaining optimal Huffman trees, so it will be helpful to illustrate it with an example. Let $(w_1, \ldots, w_6) = (2, 3, 5, 5, 6, 11)$, and consider the Huffman tree in Fig. 3 in which the nodes have been labeled with their weights and with indices from 1 to 11. The path from the weight-3 node to the root is defined by $(i_0, i_1, i_2, i_3, i_4) = (2, 4, 7, 10, 11)$; the construction in the previous paragraph yields $(i'_0, i'_1, i'_2, i'_3) = (2, 5, 9, 11)$ and $l' = 3$. After carrying out appropriate interchanges of subtrees, we obtain two Huffman trees (see Fig. 4) that are identical in structure, but the weight of 3 has been increased to 4 in the right-hand tree.

A similar procedure can be used when weights of 0 are present, but the maneuvering is more delicate because of the chance that a node and its parent both have the same weight. We can assume that there is at most one weight equal to 0, because Huffman's algorithm will begin by putting all of the 0-weight nodes into a subtree of total weight 0.

When the weights are all 0, every binary tree is "optimum"; but in our application it is more appropriate to make the external node levels be as equal as possible so that encoded first appearance of each letter will not be too long. Therefore we shall use the following minimal prefix code, when there are m letters $(a_1, \ldots, a_m)$ of weight zero, assuming that $m = 2^e + r$ and that $0 \leqslant r < 2^e$: Letter a_k is encoded as the $(e + 1)$-bit binary repre-

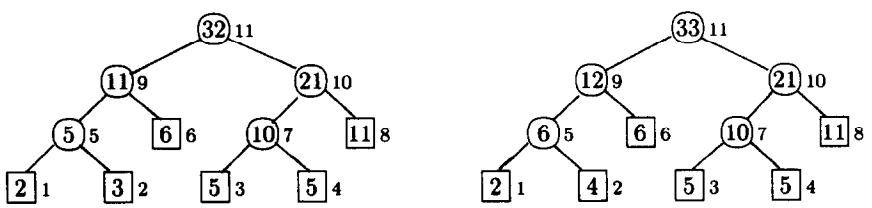

FIGURE 4

sentation of $k - 1$, if $1 \leqslant k \leqslant 2r$, otherwise as the e-bit binary representation of $k - r - 1$. For example, let $m = 5$; then $e = 2$, $r = 1$, and the encoding is

$$a_1 \to 000, \qquad a_2 \to 001, \qquad a_3 \to 01, \qquad a_4 \to 10, \qquad a_5 \to 11.$$

This encoding is optimum when all letters have the same weight ϵ, for any $\epsilon > 0$.

3. An Example

The methods sketched above lead to a real-time algorithm for maintaining Huffman trees as the weights change, as we shall see in Sections 4 and 5. But first it will be useful to study a worked example of dynamic Huffman coding, so that the detailed constructions are more readily understood.

Before transmission begins, both sender and receiver know only the size n of the alphabet being encoded. Let us assume for now that there are just 27 characters, namely **a**, **b**, ..., **z**, and !; the last symbol will be used only as the final character of the message. Since the adaptive encoding scheme seems to have a somewhat magic flavor, we shall attempt to encode the message **abracadabra**!.

Initially all weights are zero, so the first letter is encoded by the 0-weight scheme described at the end of Section 2, where $m = 27 = 2^4 + 11$:

$$\mathbf{a} \to 00000.$$

Now **a** has weight 1 and the other letters (!, **b**, ..., **z**) have weight 0. In the list of remaining letters, ! has swapped places with **a**, so that minimal changes need to be made to the data structure. The coding scheme at this point is represented by the tree in Fig. 5, hence the second letter of our message would be encoded simply as 1 if it were another **a**. However, it is a **b**, which comes out as an encoded 0-weight letter, prefixed by 0:

$$\mathbf{b} \to 000001.$$

At this point the Huffman tree is shown by Fig. 6. The third letter is therefore encoded as

$$\mathbf{r} \to 0010001,$$

after which the Huffman tree has changed to Fig. 7, assuming that the

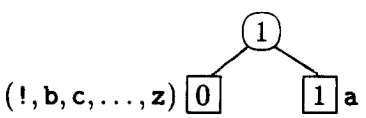

FIGURE 5

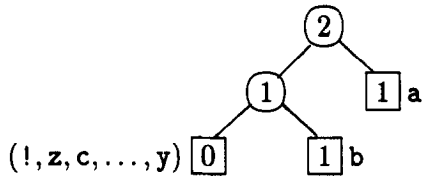

FIGURE 6

algorithms of Section 5 have been used. The encodings continue as

a → 0
c → 10000010
a → 0
d → 110000011
a → 0
b → 110
r → 110
a → 0

and by this time the tree has grown to Fig. 8. The final "shriek" is now transmitted:

! → 100000000.

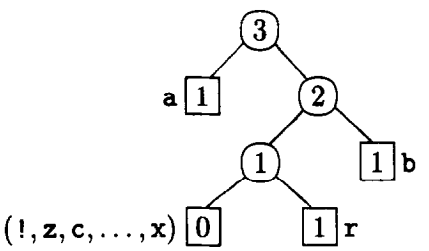

FIGURE 7

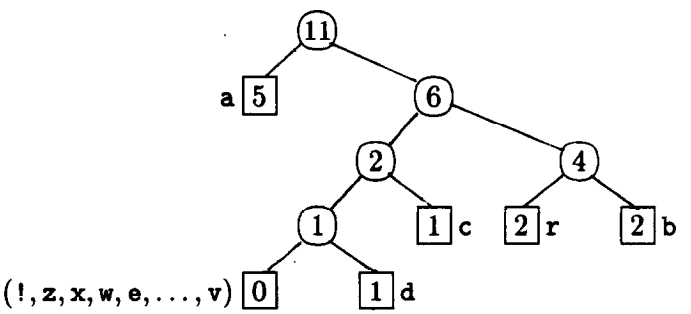

FIGURE 8

4. Data Structures

In order to update the Huffman tree and to encode and decode messages, we need to do the following things:

(a) Represent a binary tree with weights in each node.

(b) Maintain a linear list of nodes, in nondecreasing order by weight.

(c) Find the last node in this linear list that has the same weight as a given node.

(d) Interchange two subtrees of the same weight.

(e) Increase the weight of the last node in some block by unity.

(f) Represent the correspondence between letters and external nodes.

The zero-weight letters must also be dealt with. Each of the necessary operations can be done in real time with the following combination of data structures:

boolean array $S[1:n-1]$; a stack that holds bits before they are transmitted.

integer M, E, R; zero-weight counters. The number of zero-weight elements is $M = 2^E + R$ where $0 \leqslant R < 2^E$, except that $M = 0$ implies $R = -1$.

integer array $P[1:n]$; pointers to the parents of nodes. The parent of nodes $2j-1$ and $2j$ is node $P[j]$.

integer array $C[1:2n-1]$; pointers to the children of nodes. If node k is internal, and if its children are nodes $2j-1$ and $2j$, then $C[k] = j$, while if node k is external it represents letter $a_{C[k]}$.

integer array $A[0:n]$; representation of the alphabet. If letter a_k is currently the jth letter of weight zero, then $A[k] = j \leqslant M$; otherwise $A[k]$ is the number of the external node corresponding to a_k.

integer array $B[1:2n-1]$; pointers to the blocks of nodes. All nodes j of a given weight have the same value of $B[j]$.

integer array $W[1:2n-1]$; the weights. Block k has weight $W[k]$.

integer array $L[1:2n-1]$; left pointer for blocks. The nearest block whose weight is less than that of block k is block $L[k]$, unless block k has the smallest weight, in which case $L[k]$ is the block of largest weight.

integer array $G[1:2n-1]$; right pointer for blocks. The nearest block whose weight is greater than that of block k is block $G[k]$, unless block k has the largest weight, in which case $G[k]$ is the block of smallest weight.

integer H; pointer to the block of smallest weight.

integer array $D[1:2n-1]$; pointer to the largest node number in a given block.

integer V; pointer to the head of the list of array positions not currently used as block numbers. The available positions are $V, G[V], G[G[V]]$, etc.

integer Z; constant equal to $2n-1$. This also points to the root of the tree.

These data structures combine a variety of standard ideas. For example, the L, W, and G arrays define a doubly linked circular list ordered on the W field; the list head is H, and V is the available space pointer. The C and P arrays define the tree structure, taking advantage of the sequential nature of the nodes. The B and D arrays connect the tree to the list of weights.

If there are $M > 1$ letters of weight zero, the nodes of the trees will be in positions $2M-1$ through $Z = 2n-1$, inclusive. The node in position $k \geq 2M-1$ is external if and only if $A[C[k]] = k$; we have $A[0] = 2M-1$ and $C[2M-1] = 0$, to represent the zero-weight nodes. Nodes are in order by weight (i.e., $W[B[2M-1]] \leq W[B[2M]] \leq \cdots \leq W[B[Z]]$), and the root of the tree is node Z. Blocks are also in order by weight (i.e., $W[H] < W[G[H]] < W[G[G[H]]] < \cdots < W[B[Z]]$). If letter a_k is currently the jth letter of weight zero, we have $A[k] = j$ and $C[j] = k$. When there is at most one letter of weight zero, the nodes are in positions 1 through $2n-1$ and essentially the same conventions hold; but $A[0]$ will be 1 and $C[1]$ will not be zero. There is a convenient ambiguity when $M = 1$, which allows a simple transition between $M = 0$ and $M = 1$.

Note that most of the integer values in the data structures are nonnegative and less than $2n$, so we need just $\lceil \lg n \rceil + 1$ bits to represent them. The only exceptions are the integer weights in the W table, which may be as large as the length of the message being transmitted. Thus, the total memory requirements are about $12n \lg n + 2n \lg w$ bits, where w is a bound on the total weight of all n letters.

If a node and its parent are in the same block, the parent node should immediately follow its children in the node sequence.

5. Algorithms

Given the data structures specified in Section 4, it is not difficult to design algorithms that maintain the necessary invariants. The algorithms will be presented here in an Algol-like language that uses "**fi**" and "**od**" to close "**if**" and "**do**", so that other delimiters like "**begin**" and "**end**" are not needed very often.

At the beginning we can get all the data structures off to a good start as follows:

```
procedure initialize;
begin integer i;
M ← 0; E ← 0; R ← −1; Z ← 2 × n − 1;
for i ← 1 to n do
    M ← M + 1; R ← R + 1;
    if 2 × R = M then E ← E + 1; R ← 0 fi;
    A[i] ← C[i] ← i od;
H ← 1; L[H] ← G[H] ← H; W[H] ← 0; D[H] ← A[0] ← Z;
V ← 2; for i ← V to Z − 1 do G[i] ← i + 1 od; G[Z] ← 0;
P[n] ← 0; C[Z] ← 0; B[Z] ← H;
end;
```

Let us consider next a procedure that encodes the letter a_k and sends out the corresponding bits, based on the present state of the tree.

```
procedure encode (integer k);
begin integer i, j, q, t;
i ← 0; q ← A[k];
if q ≤ M then comment encode zero weight;
    q ← q − 1;
    if q < 2 × R then t ← E + 1 else q ← q − R; t ← E fi;
    for j ← 1 to t do i ← i + 1; S[i] ← q mod 2; q ← q div 2 od;
    q ← A[0] fi;
while q < Z do
    i ← i + 1; S[i] ← (q + 1) mod 2;
    q ← P[(q + 1) div 2] od;
while i > 0 do transmit (S[i]); i ← i − 1 od;
end;
```

Here *transmit* is a system output procedure that sends one bit.

The decoding process does essentially the same thing as the encoder, but backwards. It uses a system procedure *receive* that reads one bit of input and returns that value:

```
integer procedure decode;
begin integer j, q;
q ← Z;
while A[C[q]] ≠ q do q ← 2 × C[q] − 1 + receive od;
if C[q] = 0 then comment decode zero weight;
    q ← 0;
    for j ← 1 to E do q ← 2 × q + receive od;
    if q < R then q ← 2 × q + receive else q ← q + R fi;
    q ← q + 1 fi;
return C[q];
end;
```

The following subroutine interchanges two subtrees of the same weight, assuming that neither is the child of the other:

```
procedure exchange (integer q, t);
begin integer ct, cq, acq;
ct ← C[t]; cq ← C[q]; acq ← A[cq];
if A[ct] ≠ t then P[ct] ← q else A[ct] ← q fi;
if acq ≠ q then P[cq] ← t else A[cq] ← t fi;
C[t] ← cq; C[q] ← ct;
end;
```

The heart of dynamic Huffman tree processing is the *update* procedure, which is used by both sender and receiver after a_k has been encoded or decoded. It has the following overall form:

```
procedure update (integer k);
⟨Set q to the external node whose weight should increase⟩;
while q > 0 do
    ⟨Move q to the right of its block⟩;
    ⟨Transfer q to the next block, with weight one higher⟩;
    q ← P[(q + 1) div 2] od;
end;
```

Thus, *update* performs three main functions. The first of these is quite simple unless the letter a_k is appearing for the first time. In the latter case a new external node of zero weight is appended to the tree, together with a new internal node:

```
⟨Set q to the external node whose weight should increase⟩ =
 begin q ← A[k];
 if q≤ M then comment a zero weight will become positive;
     A[C[M]] ← q; C[q] ← C[M];
     if R = 0 then R ← M div 2; if R > 0 then E ← E − 1 fi fi;
     M ← M − 1; R ← R − 1;
     if M> 0 then
         q ← A[0] − 1; A[0] ← q − 1; A[k] ← q; C[q] ← k;
         if M > 1 then C[q − 1] ← 0 fi;
         P[M] ← q + 1; C[q + 1] ← M;
         B[q] ← B[q − 1] ← H fi;
     fi;
  end
```

The next portion of *update* moves node q to the right of its block, unless both q and its parent are already at the right of the block.

```
⟨Move q to the right of its block⟩ =
   if q< D[B[q]] and D[B[P[(q + 1) div 2]]] > q + 1 then
      exchange (q, D[B[q]]); q ← D[B[q]] fi
```

Finally, the remaining job is to augment the weight of q (and to augment the weight of q's parent as well, if they currently have the same weight):

```
⟨Transfer q to the next block, with weight one higher⟩ =
 begin integer j, u, gu, lu, x, t, qq;
 u ← B[q]; gu ← G[u]; lu ← L[u]; x ← W[u]; qq ← D[u];
 if W[gu] = x + 1 then
      B[q] ← B[qq] ← gu;
      if D[lu] = q − 1 or (u = H and q = A[0])
      then comment block u disappears;
           G[lu] ← gu; L[gu] ← lu; if H = u then H ← gu fi;
           G[u] ← V; V ← u;
      else D[u] ← q − 1 fi;
  else if D[lu] = q − 1 or (u = H and q = A[0]) then W[u] ← x + 1;
       else comment a new block appears;
            t ← V; V ← G[V];
            L[t] ← u; G[t] ← gu; L[gu] ← G[u] ← t;
            W[t] ← x + 1; D[t] ← D[u]; D[u] ← q − 1;
            B[q] ← B[qq] ← t fi;
 fi;
 q ← qq;
 end
```

The total time is $O(l)$ when updating an element at level l of the tree. It is important to realize that the algorithm does not assume that the parent of nodes $2j - 1$ and $2j$ be less than the parent of nodes $2j + 1$ and $2j + 2$. Such monotonicity cannot be guaranteed without violating the $O(l)$ time bound; fortunately it is not necessary.

6. Decreasing a Weight

The data structures developed in Section 4 for adding unity to a nonnegative weight can also be used to subtract unity from a positive weight. Since the ideas are similar to those discussed above, it suffices to present the algorithm here without further comment.

```
procedure downdate (integer k);
begin integer q;
q ← A[k]; comment q > M;
while q > 0 do
    ⟨Move q to the left of its block⟩;
    ⟨Transfer q to the previous block, with weight one lower⟩;
    q ← P[(q + 1) div 2] od;
```

```
if W[B[A[k]]] = 0 then ⟨Absorb zero-weight letter a_k⟩ fi;
end;

⟨Move q to the left of its block⟩ =
begin integer u, t;
u ← B[q]; if u = H then t ← 1 else t ← D[L[u]] + 1 fi;
if q ≠ t then exchange (q, t); q ← t fi;
end

⟨Transfer q to the previous block, with weight one lower⟩ =
begin integer j, u, lu, x, t;
u ← B[q]; lu ← L[u]; x ← W[u];
if W[lu] = x − 1 then
    B[q] ← lu; D[lu] ← q;
    if D[u] = q then comment block u disappears;
        G[lu] ← G[u]; L[G[u]] ← lu;
        G[u] ← V; V ← u fi;
else if D[u] = q then W[u] ← x − 1
    else comment a new block appears;
        t ← V; V ← G[V]; if u = H then H ← t fi;
        L[t] ← lu; G[t] ← u; G[lu] ← L[u] ← t;
        W[t] ← x − 1; D[t] ← q; B[q] ← t fi;
fi;
if A[C[q]] ≠ q and B[q] = B[2 × C[q]] and q > 2 × C[q] + 1 then
    exchange (2 × C[q] + 1, q) fi;
end

⟨Absorb zero-weight letter a_k⟩ =
begin integer q;
M ← M + 1; R ← R + 1;
if M > 1 then comment there are tree nodes of weight zero;
    if 2 × R = M then E ← E + 1; R ← 0 fi;
    q ← A[0] + 1; A[0] ← q + 1; C[q + 1] ← 0;
    A[k] ← M; C[M] ← k fi;
end
```

It is well known (and easy to prove directly) that the levels of two equal-weight nodes in a Huffman tree cannot differ by more than one, if the nodes have positive weight. Hence the number of iterations of "downdate" cannot be more than twice the original level of the external node that corresponds to a_k. For example, if a_k is at level 4, two steps of exchanging with the leftmost member of its block and moving to the parent node must lead to a node on level $\leqslant 3$. Two more steps lead to level $\leqslant 2$, and so on.

7. Empirical Tests

The algorithms above were applied to three kinds of data, and in each case the on-line approach to encoding produced results comparable to the "optimum" results that would be obtainable from two passes over the data.

The **abracadabra!** example of Section 3 can be used to illustrate the criteria by which we can judge larger examples: A total of 54 bits was produced by the on-line encoding of **abracadabra!**, while the optimum weighted path length for the frequencies of letters $(1, 1, 1, 2, 2, 5)$ is 28. It is not fair simply to compare 54 to 28, since the 28-bit encoding would be achieved only if the receiver knew the optimum code; further bits must be added to the 28 in order to specify what code is being used. Since it takes about $2k$ bits to specify a binary tree on k external nodes, and since each of the 27 possible letters of this example requires at least 4 bits for identification, we might expect to transmit at least $6k$ bits to identify an optimum coding scheme that uses k of the letters. In the example, $k = 6$ and $28 + 6k = 64$; so the 54 bits of the on-line method look quite good.

The following statistics appear to be the most relevant numbers governing the performance of dynamic Huffman encoding, based on these considerations: $\sum b$ (the total number of bits sent to encode a file); $\sum b_{\text{opt}}$ (the minimum weighted path length for the file); and the "overhead ratio"

$$\rho = \left(\sum b - \sum b_{\text{opt}}\right)/k - 2,$$

where k is the number of letters with nonzero weight. For example, the **abracadabra!** statistics are $\sum b = 54$, $\sum b_{\text{opt}} = 28$, and $\rho = 2.333$. If ρ is less than the binary logarithm of the alphabet size n, or if the ratio $\sum b / \sum b_{\text{opt}}$ is near 1, we can say that the on-line method is doing well.

The first file subjected to experiments was textual data from Grimm's first ten *Fairy Tales*, since the Grimm file has been available for many years at Stanford. In this case the relevant alphabet size is $n = 128$, because the data appears on the disk in 7-bit ASCII form. The carriage returns and line-feeds at the end of each line, and the form-feeds at the end of each Tale, were encoded together with the text itself; the text, of course, consisted primarily of upper and lower case letters together with blank spaces and a few punctuation marks. Only 58 of the 128 possible codes actually appeared in the file.

After 1000 characters had been transmitted, the statistics were

$$\sum b = 4558, \qquad \sum b_{\text{opt}} = 4297, \qquad \rho = 5.68;$$

after the first 10000 characters, the statistics were

$$\sum b = 44733, \qquad \sum b_{\text{opt}} = 44272, \qquad \rho = 6.70;$$

and after the first 100000 characters, they were

$$\sum b = 440164, \qquad \sum b_{\text{opt}} = 439613, \qquad \rho = 7.50.$$

As the file gets larger, the overhead ratio grows to the point where a two-pass scheme would transmit fewer bits, yet the on-line method is not far from optimum.

Note that the Grimm data have about 4.4 bits of information per character, when we consider each character separately. Another experiment was made on the same file, but with 14-bit character pairs replacing the individual 7-bit characters. The Grimm tales used 680 of the $n = 16384$ possible character pairs in this "alphabet"; and the statistics corresponding to the three sets of numbers above were

$$\begin{aligned} \sum b &= 5981, & \sum b_{\text{opt}} &= 3447, & \rho &= 12.40; \\ \sum b &= 44704, & \sum b_{\text{opt}} &= 38124, & \rho &= 13.06; \\ \sum b &= 393969, & \sum b_{\text{opt}} &= 383264, & \rho &= 13.95. \end{aligned}$$

Thus the larger alphabet size meant that the one-pass scheme had $\rho < \lg n$ for the entire file; but this improvement was probably not enough to justify the 128-fold increase in memory requirements.

A second set of experiments was based on the computer representation of a technical book. This file differed from the Grimm text primarily because it contained mathematical symbols and numerical data, as well as special formatting characters to control the typesetting; 106 of the 128 possible ASCII characters were present. The respective statistics corresponding to the first 1000, 10000, and 100000 characters of this file were

$$\begin{aligned} \sum b &= 5913, & \sum b_{\text{opt}} &= 5321, & \rho &= 6.0; \\ \sum b &= 50440, & \sum b_{\text{opt}} &= 49619, & \rho &= 7.33; \\ \sum b &= 519561, & \sum b_{\text{opt}} &= 518361, & \rho &= 9.32. \end{aligned}$$

When 14-bit character pairs were considered, the numbers were

$$\begin{aligned} \sum b &= 7899, & \sum b_{\text{opt}} &= 3874, & \rho &= 12.32; \\ \sum b &= 53204, & \sum b_{\text{opt}} &= 41970, & \rho &= 12.80; \\ \sum b &= 472534, & \sum b_{\text{opt}} &= 442564, & \rho &= 13.23; \end{aligned}$$

and 1968 different character pairs were present. A technical file like this is clearly more difficult to compress than a file of *Fairy Tales*.

The third set of experiments was based on graphical data for the boundaries of alphabetic characters that were about 500 pixels tall. Each boundary was encoded as a sequence of "king moves," where each move represents a choice between seven possibilities: the boundary either continues in the previous direction, or it turns $\pm 45°$, $\pm 90°$, or $\pm 135°$. The king-moves were grouped into triples, so the effective alphabet size was $7^3 = 343$. Seven different boundaries were studied, having respective lengths of 2508, 2736, 2172, 1236, 1008, 1632, and 612 moves; each boundary was treated as a separate file, with the tree initialized to zero at the beginning. The results were

	$\sum b$	$\sum b_{\text{opt}}$	ρ	$\sum b$/(total moves)
Boundary 1	3048	2877	10.21	1.22
Boundary 2	3284	3110	8.88	1.20
Boundary 3	2448	2278	9.30	1.13
Boundary 4	1495	1350	10.02	1.21
Boundary 5	1155	1020	8.38	1.15
Boundary 6	1258	1093	7.71	0.77
Boundary 7	490	394	7.60	0.80

The experiment was also carried out with quadruples instead of triples, using an alphabet of size $7^4 = 2401$, with the following results:

	$\sum b$	$\sum b_{\text{opt}}$	ρ	$\sum b$/(total moves)
Boundary 1	2935	2598	9.62	1.17
Boundary 2	3179	2805	10.90	1.16
Boundary 3	2369	2044	10.50	1.09
Boundary 4	1501	1205	10.87	1.21
Boundary 5	1191	916	10.50	1.18
Boundary 6	1205	993	9.83	0.74
Boundary 7	449	315	9.17	0.73

The last two boundaries consisted mostly of straight diagonal lines; the number of different king-move patterns of length t that appear on a straight line is at most $t + 1$ (see [3]), hence the encoding was (predictably) more efficient in this case.

However, the overhead ratio in these experiments was higher than expected, and the reason seems to be that comparatively few of the "letters" actually occurred in the character boundaries. Only 25 of 343 triples and 39

of 2401 quadruples were present, so it is clear that bits were being wasted to identify the triples that occurred. There is no necessity for an on-line Huffman algorithm to start out with a *tabula rasa* in which all weights are zero, so another experiment was attempted in which the 25 triples and 39 quadruples were initially given weight 1. (In other words, each boundary had effectively been preceded by a message containing the "important" letters, but this preamble message did not need to be sent because the decoder already knew what it was.) The resulting numbers of bits per king move were thereby reduced as follows:

	Triples	Quadruples
Boundary 1	1.19	1.10
Boundary 2	1.16	1.08
Boundary 3	1.10	1.00
Boundary 4	1.16	1.06
Boundary 5	1.10	1.01
Boundary 6	0.73	0.65
Boundary 7	0.79	0.72

The algorithm of Section 6, which allows weights to decrease as well as to increase, suggests another dynamic coding scheme based on "local" properties of the source text: Given a message $a_{i_1}a_{i_2}\ldots$, and a buffer size d, we can increase the weight of a_{i_k} by 1 and decrease the weight of $a_{i_{k-d}}$ by 1 just after encoding a_{i_k}, so that the coding scheme for $a_{i_{k+1}}$ depends on the letter frequencies in the previous d characters $a_{i_{k-d+1}}\cdots a_{i_k}$. The encoder and decoder now need to maintain a buffer of length d in order to stay synchronized, and the computing time is approximately doubled. But this "d-window method" has two advantages that might compensate for the extra effort: (a) The weights will be bounded by d, so there will be no chance of overflow in the W array. (b) The windowing method gets rid of the "garbage" that occasionally appears, minimizing the effect of anomalous weights that might otherwise penalize an entire file.

A d-window method seems especially suited to the boundary data, since the path around a character shape typically runs through distinct phases. For example, Boundaries 1 and 2 in the author's experiments were curves that sometimes went left for awhile then changed to rightward-turning curves; Boundary 3 was a combination of curves and straight lines. However, closer examination revealed that windowing helps only if the phases are substantially longer than d, since it is necessary for the tree to go through a somewhat traumatic transitional period between phases. Furthermore d needs to be large enough that the tree weights have a significant

effect on the encoding. Therefore the windowing method actually had a negative effect on the boundary data, except in the case of Boundary 3 which was very slightly improved.

Windowing was of no use with the Grimm text either, since that data was quite uniform in character. A small improvement was noted when windowing was applied to the technical file, however, because its beginning portions involved material of somewhat special nature; unfortunately the early advantage of windowing disappeared later when a more uniform part of the file was encountered. Here are the Σb figures for the first 10000, 25000, and 100000 characters, with various window sizes:

	$d = 500$	$d = 750$	$d = 1000$	$d = 1250$	$d = 1500$	$d = \infty$
10000	50823	50456	50257	50248	50256	50440
25000	129083	128202	127900	128045	128375	129120
100000	526736	523541	521102	520248	520706	519561

The methods of this paper work in "real time," but the constant of proportionality is rather large, so they would be most advantageous if embedded in hardware. Their one-pass character means that the number of disk accesses is cut in half by comparison with two-pass schemes; in many applications, this fact outweighs the time for internal processing.

ACKNOWLEDGMENT

I wish to thank Joel Bion, whose term paper suggesting the use of dynamically changing codes for computer-to-computer file transfers led me to think of the problem treated here.

REFERENCES

1. R. G. GALLAGER, Variations on a theme by Huffman, *IEEE Trans. Inform. Theory* **IT-24** (1978), 668–674.
2. D. A. HUFFMAN, A method for the construction of minimum redundancy codes, *Proc. IRE* **40** (1951), 1098–1101.
3. D. E. KNUTH, Solution to problem E2307, *Amer. Math. Monthly* **79** (1972), 773–774.