ALGORITHM 673
Dynamic Huffman Coding

JEFFREY SCOTT VITTER
Brown University

We present a Pascal implementation of the one-pass algorithm for constructing dynamic Huffman codes that is described and analyzed in a companion paper [3]. The program runs in real time; that is, the processing time for each letter of the message is proportional to the length of its codeword. The number of bits used to encode a message of t letters is less than t bits more than that used by the well-known two-pass algorithm. This is best possible for any one-pass Huffman scheme. In practice, it uses fewer bits than all other Huffman schemes. The algorithm has applications in file compression and network transmission.

Categories and Subject Descriptors: C.2.0 [**Computer Communication Networks**]: General—*data communications*; E.1 [**Data**]: Data Structures—*trees*; E.4 [**Data**]: Coding and Information Theory—*data compaction and compression, nonsecret encoding schemes*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*trees*; H.1.1 [**Models and Principles**]: Systems and Information Theory—*value of information*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Huffman codes, networks

1. INTRODUCTION

In this paper we present a Pascal implementation of Algorithm Λ, the one-pass algorithm for constructing dynamic Huffman codes that is described and analyzed in the companion paper [3]. The dynamic code used to process the $(t + 1)$st letter in the message is a Huffman code based upon the first t letters. The processing time for each letter is proportional to the length of its encoding, so the program runs in real time. It is shown in [3] that the number of bits used by Algorithm Λ to encode a message of t letters is less than t bits more than that used by the well-known two-pass algorithm developed by D. A. Huffman [1]. Algorithm Λ is

Support was provided in part by National Science Foundation research grant DCR-84-03613, by an NSF Presidential Young Investigator Award with matching funds from an IBM Faculty Development Award and an AT&T research grant, by an IBM research contract, and by a Guggenheim Fellowship. Part of this research was done while the author was at the Mathematical Sciences Research Institute in Berkeley, Calif., at INRIA in Rocquencourt, France, and at École Normale Supérieure in Paris, France.

Author's current address: Department of Computer Science, Brown University, Providence, R.I. 02912.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0098-3500/89/0600-0158 $01.50

optimum in this respect among all one-pass Huffman schemes. Experiments indicate that Algorithm Λ typically uses fewer bits than Huffman's algorithm and other one-pass Huffman methods. The algorithm has applications in file compression and network transmission.

We shall use the following terminology in our discussion:

n = alphabet size;

a_j = jth letter in the alphabet;

t = number of letters in the message processed so far;

k = number of distinct letters processed so far.

Each letter in the message is encoded by the path from the root to its leaf node in the current version of the Huffman tree. The weight of each leaf is the number of times the corresponding letter has appeared previously in the message. When $k < n$, a 0-node is used to represent the $n - k$ unused letters in the message. When a letter appears in the message for the first time, additional bits are put into the encoding to specify which of the unused letters it is, and if there are still unused letters remaining, the 0-node is split to create an extra leaf to accommodate the new letter.

One of the main features of Algorithm Λ is its use of *implicit numbering* in which the nodes in the Huffman tree are numbered in increasing order from bottom to top and at each level in increasing order from left to right. Another main feature is the invariant that all leaves of a given weight precede in the implicit numbering all internal nodes of the same weight. These two features are shown in [3] to guarantee good coding efficiency.

In this paper we present a detailed implementation for Algorithm Λ. The data structure is described in Section 2, and Section 3 contains the Pascal code.

2. DATA STRUCTURE

We shall denote all of the leaves of a given weight as a *leaf block* and all the internal nodes of a given weight as an *internal block*. The largest numbered node in a block is called the *leader* of the block. Our invariant implies that the blocks are linked together in increasing order by weight and that a leaf block always precedes an internal block of the same weight. The main operations that must be supported by the data structure for Algorithm Λ are

—Represent a binary Huffman tree with nonnegative weights that maintains the invariant.

—Store a contiguous list of internal tree nodes in nondecreasing order by weight; internal nodes of the same weight are ordered with respect to the implicit numbering. A similar list is stored for leaf nodes.

—Find the leader of a node's block, for any given node, based upon the implicit numbering.

—Interchange the contents of two leaves of the same weight.

—Increment the weight of the leader of a block by 1, which can cause the node's implicit numbering to "slide" past the implicit numberings of the nodes in the next block and cause their implicit numberings to each decrease by 1.

—Represent the correspondence between the k letters of the alphabet that have appeared in the message and the positive-weight leaves in the tree.

—Represent the $n - k$ letters in the alphabet that have not yet appeared in the message by a single leaf 0-node in the Huffman tree.

The components of the data structure are listed below. The number of leaves of zero weight are specified by integer variables M, E, and R.

$$\begin{aligned} M &= n - k = \text{the number of zero-weight letters in the alphabet} \\ &= 2^E + R, \quad \text{where } 0 \le R < 2^E, \quad \text{except that } R = -1 \text{ when } M = 0. \end{aligned}$$

The data structure makes use of an *explicit numbering*, which corresponds to the physical storage locations used to store information about the nodes. This is not to be confused with the implicit numbering defined in the last section. Unless otherwise stated, all references to node numberings in this section are based upon the explicit numbering. Leaf nodes are explicitly numbered $1, \ldots, n$ in contiguous locations in physical memory, and internal nodes are explicitly numbered $n + \max\{1, M\}, \ldots, 2n - 1$ in contiguous locations in memory. Node q is a leaf node if and only if $q \le n$. When $k < n$ (that is, when $M > 0$), the 0-node in the Huffman tree is node M, and the positive-weight leaves are nodes $M + 1, \ldots, n$. Nodes $1, \ldots, M$ represent letters of zero weight, though only node M actually appears in the Huffman tree. When $k > 1$ (that is, when $M < n$), the root of the tree is the internal node $2n - 1$; otherwise, we have $M = n$ and the root of the tree is node n, the 0-node.

There is a close relationship between the explicit and implicit numberings, as specified in the second operation listed above: For two internal nodes p and q, we have $p < q$ in the explicit numbering if and only if $p < q$ in the implicit numbering; the same holds for two leaf nodes p and q.

The tree data structure is called a "floating tree" because the parent and child pointers for the nodes are not maintained explicitly. Instead, each block has a *parent* pointer and a *rtChild* pointer that points to the parent and right child of the leader of the block. This allows a node to slide over an entire block without having to update more than a constant number of pointers. Because of the contiguous storage of leaves and of internal nodes, the locations of the parents and children of the other nodes in the block can be computed in constant time via an offset calculation from the block's *parent* and *rtChild* pointer.

The correspondence between leaf nodes and the letters they represent is given by the following arrays *alpha* and *rep*:

$$\begin{aligned} &\mathit{alpha}[q] = j, \quad 1 \le j \le n, \\ &\qquad \text{if and only if } a_j \text{ is the letter represented by node } q, \quad 1 \le q \le n. \\ &\mathit{rep}[j] = q, \quad 1 \le q \le n, \\ &\qquad \text{if and only if node } q \text{ corresponds to letter } a_j, \quad 1 \le j \le n. \end{aligned}$$

The main entity in the floating tree representation is the block, as defined earlier. Blocks are numbered in the range $1, \ldots, 2n - 1$ in no particular order. The mapping between nodes and blocks is given by

$$\mathit{block}[q] = \text{block number of node } q \quad \text{for } \max\{1, M\} \le q \le n \text{ or } n + \max\{1, M\} \le q \le 2n - 1.$$

The following eight arrays of integers are each indexed by a block number b in the range $1 \le b \le 2n - 1$.

weight$[b]$ = weight of each node in block b.
parent$[b]$ = the parent node of the leader of block b, if it exists; and 0 otherwise.
parity$[b]$ = 0 if the leader of block b is a left child or the root of the Huffman tree; and 1 otherwise.
rtChild$[b]$ = q if b is a block of internal nodes and node q is the right child of the leader of block b.
first$[b]$ = q if node q is the leader of block b.
last$[b]$ = q if node q is the smallest numbered node in block b.
prevBlock$[b]$ = previous block on the circularly linked list of blocks.
nextBlock$[b]$ = next block on the circularly linked list of blocks.

Each slot in the array *weight* must be capable of storing any integer in the range $[0, t]$. The unused blocks are linked together using *nextBlock* in a list headed by

availBlock = first block in the available-block list if the list is nonempty; and 0 otherwise.

The final component of the data structure is the following array indexed by $1 \le i \le n$:

stack$[i]$ = ith-to-last bit of the encoding of the current letter being processed.

Except for the elements of the array *weight*, each integer variable can take on at most n or $2n - 1$ values, which requires either $\lceil \log_2 n \rceil$ or $\lceil \log_2 n \rceil + 1$ bits of storage. The total amount of storage (in bits) needed for the data structure is

$$\begin{aligned} &2\lceil \log_2 n \rceil + \lceil \log_2 \log_2 n \rceil + 2n\lceil \log_2 n \rceil \\ &\quad + (2n - 1)(\lceil \log_2 t \rceil + 7\lceil \log_2 n \rceil + 7) + \lceil \log_2 n \rceil + n \\ &\quad \approx 16n\lceil \log_2 n \rceil + 15n + 2n\lceil \log_2 t \rceil, \end{aligned}$$

which is only about $4n\lceil \log_2 n \rceil$ more bits of storage than used by Algorithm FGK. The storage requirement can be reduced by roughly $n\lceil \log_2 n \rceil$ bits if separate available-block lists are kept for internal nodes and leaf nodes since leaf blocks do not need a *rtChild* value. If storage is dynamically allocated, as opposed to preallocated via arrays, it is typically much less.

3. PASCAL CODE

The basic implementation of Algorithm Λ is along the lines of the implementation of Algorithm FGK in [2]. The primary difference between the two is that Algorithm Λ uses the implicit node numbering and the floating tree data structure in order to maintain the invariant defined in Section 1.

The basic loop for encoding and transmitting a message is

```
*Initialize*;
**while** there are more letters to encode **do begin**
    Let $a_j$ be the next letter to encode;
    *EncodeAndTransmit*($j$);
    *Update*($j$)
    **end**;
```

The corresponding loop for receiving and decoding a message is

```
*Initialize*;
**while** there is more to decode **do begin**
    $j$ := *ReceiveAndDecode*;
    *Output*($a_j$);
    *Update*($j$)
    **end**;
```

The *Initialize* procedure forms an initial Huffman tree consisting of a single leaf 0-node. The global variable Z is always equal to $2n - 1$.

```
**procedure** *Initialize*;
**var** $i$: *integer*;
**begin**
$M$ := 0; $E$ := 0; $R$ := −1; $Z$ := $2 \times n - 1$;
**for** $i$ := 1 **to** $n$ **do begin**
    $M$ := $M + 1$; $R$ := $R + 1$;
    **if** $2 \times R = M$ **then begin** $E$ := $E + 1$; $R$ := 0 **end**;
    *alpha*[$i$] := $i$; *rep*[$i$] := $i$
    **end**;
{Initialize node $n$ as the 0-node}
*block*[$n$] := 1; *prevBlock*[1] := 1; *nextBlock*[1] := 1; *weight*[1] := 0;
*first*[1] := $n$; *last*[1] := $n$; *parity*[1] := 0; *parent*[1] := 0;
{Initialize available block list}
*availBlock* := 2;
**for** $i$ := *availBlock* **to** $Z - 1$ **do**
    *nextBlock*[$i$] := $i + 1$;
*nextBlock*[$Z$] := 0
**end**;
```

The *EncodeAndTransmit* procedure determines the encoding of letter a_j on the basis of the path from the root of the Huffman tree to a_j's leaf, using the convention that "0" means "go to the left child" and "1" means "go to the right child." If a_j has not appeared previously in the message, extra bits are sent to specify which one of the M zero-weight letters has been encountered. These extra bits are computed by the following minimum prefix code: If $1 \le j \le 2R$, then a_j is specified by the $(E + 1)$-bit binary representation of $j - 1$; otherwise, a_j is specified by the E-bit binary representation of $j - R - 1$. The system procedure *Transmit* is called for each bit in the encoding to send it to the receiver.

```
**procedure** *EncodeAndTransmit*($j$: *integer*);
**var** $i$, $ii$, $q$, $t$, *root*: *integer*;
**begin**
$q$ := *rep*[$j$]; $i$ := 0;
**if** $q \le M$ **then begin**    {Encode letter of zero weight}
    $q$ := $q - 1$;
    **if** $q < 2 \times R$ **then** $t$ := $E + 1$ **else begin** $q$ := $q - R$; $t$ := $E$ **end**;
```

```
        for ii := 1 to t do begin
          i := i + 1; stack[i] := q mod 2;
          q := q div 2
          end
        q := M;
        end;
    if M = n then root := n else root := Z;
    while q ≠ root do begin   {Traverse up the tree}
      i := i + 1; stack[i] := (first[block[q]] − q + parity[block[q]]) mod 2;
      q := parent[block[q]] − (first[block[q]] − q + 1 − parity[block[q]]) div 2
      end;
    for ii := i downto 1 do Transmit(stack[ii])
    end;
```

The *ReceiveAndDecode* function repeatedly calls a system function *Receive* to read one more bit of input until the inputed sequence of 0's and 1's has specified a path to a leaf node in the Huffman tree. Extra bits are read when $k < n - 1$ and a 0-node is reached in order to determine which zero-weight letter is being transmitted.

```
    function ReceiveAndDecode: integer;
    var i, q: integer;
    begin
    if M = n then q := n else q := Z;   {Set q to the root node}
    while q > n do   {Traverse down the tree}
      q := FindChild(q, Receive);
    if q = M then begin   {Decode 0-node}
      q := 0;
      for i := 1 to E do q := 2 × q + Receive;
      if q < R then q := 2 × q + Receive else q := q + R;
      q := q + 1
      end;
    Decode := alpha[q]
    end;
```

The function *FindChild* returns the node number of either the left or right child of node *j*, depending on whether the parity parameter is set to 0 or 1.

```
    function FindChild(j, parity: integer): integer;
    var delta, right, gap: integer;
    begin
    delta := 2 × (first[block[j]] − j) + 1 − parity;
    right := rtChild[block[j]]; gap := right − last[block[right]];
    if delta ≤ gap then FindChild := right − delta
    else begin
      delta := delta − gap − 1;
      right := first[prevBlock[block[right]]]; gap := right − last[block[right]];
      if delta ≤ gap then FindChild := right − delta
      else FindChild := first[prevBlock[block[right]]] − delta + gap + 1
      end
    end;
```

The last (inner) **else**-clause is never executed when the Huffman tree is well formed, such as when *FindChild* is called by *Decode*, but it is needed when *FindChild* is called by *Update* during the modification of the tree.

The procedure *InterchangeLeaves* interchanges the contents of two leaf nodes $e1$ and $e2$ in the Huffman tree.

```
procedure InterchangeLeaves(e1, e2: integer);
var temp: integer;
begin
rep[alpha[e1]] := e2; rep[alpha[e2]] := e1;
temp := alpha[e1]; alpha[e1] := alpha[e2]; alpha[e2] := temp
end;
```

The procedure *Update* is the main component of the algorithm. It is called by both *EncodeAndTransmit* and *ReceiveAndDecode* in order to modify the dynamic Huffman tree to account for the letter just processed.

```
procedure Update(k: integer);
var q, leafToIncrement, bq, b, oldParent, oldParity, nbq, par, bpar: integer;
  slide: boolean;
begin
{Set q to the node whose weight should increase}
FindNode;
while q > 0 do
  {At this point, q is the first node in its block. Increment q's weight by 1 and slide q
   if necessary over the next block to maintain the invariant. Then set q to the node
   one level higher that needs incrementing next}
  SlideAndIncrement;
{Finish up some special cases involving the 0-node}
if leafToIncrement ≠ 0 then
  begin q := leafToIncrement; SlideAndIncrement end
end;
```

The procedure *Update* calls two internal procedures: *FindNode* and *Slide-AndIncrement*. Both procedures have no local variables and do not take parameters. The *FindNode* procedure sets q to point to the leaf to process. If that leaf is the 0-node, which corresponds to the transmission of a letter that has not been transmitted earlier in the message, the 0-node is split to form an extra leaf if there is still an untransmitted letter left in the alphabet. Otherwise, q is interchanged with the leader of its block.

```
procedure FindNode;
begin
q := rep[k]; leafToIncrement := 0;
if q ≤ M then begin   {A zero weight becomes positive}
  InterchangeLeaves(q, M);
  if R = 0 then begin R := M div 2; if R > 0 then E := E − 1 end;
  M := M − 1; R := R − 1; q := M + 1; bq := block[q];
  if M > 0 then begin
    {Split the 0-node into an internal node with two children. The new 0-node is
     node M; the old 0-node is node M + 1; the new parent of nodes M and M + 1 is
     node M + n}
    block[M] := bq; last[bq] := M; oldParent := parent[bq];
    parent[bq] := M + n; parity[bq] := 1;
    {Create a new internal block of zero weight for node M + n}
    b := availBlock; availBlock := nextBlock[availBlock];
    prevBlock[b] := bq; nextBlock[b] := nextBlock[bq];
    prevBlock[nextBlock[bq]] := b; nextBlock[bq] := b;
    parent[b] := oldParent; parity[b] := 0; rtChild[b] := q;
    block[M + n] := b; weight[b] := 0;
```

```
      first[b] := M + n; last[b] := M + n;
      leafToIncrement := q; q := M + n
      end
    end
  else begin  {Interchange q with the first node in q's block}
    InterchangeLeaves(q, first[block[q]]);
    q := first[block[q]];
    if (q = M + 1) and (M > 0) then
      begin leafToIncrement := q; q := parent[block[q]] end
    end
  end;
```

The *SlideAndIncrement* procedure increments the weight of node q by 1 and adjusts the tree pointers to reflect the new implicit numbering. Finally, q is set to point to the node one level higher in the tree that needs incrementing next.

```
  procedure SlideAndIncrement;
  begin  {q is currently the first node in its block}
  bq := block[q]; nbq := nextBlock[bq];
  par := parent[bq]; oldParent := par; oldParity := parity[bq];
  if ((q ≤ n) and (first[nbq] > n) and (weight[nbq] = weight[bq]))
    or ((q > n) and (first[nbq] ≤ n) and (weight[nbq] = weight[bq] + 1)) then
      begin  {Slide q over the next block}
      slide := true;
      oldParent := parent[nbq]; oldParity := parity[nbq];
      {Adjust child pointers for next higher level in tree}
      if par > 0 then begin
        bpar := block[par];
        if rtChild[bpar] = q then rtChild[bpar] := last[nbq]
        else if rtChild[bpar] = first[nbq] then rtChild[bpar] := q
        else rtChild[bpar] := rtChild[bpar] + 1;
        if par ≠ Z then
          if block[par + 1] ≠ bpar then
            if rtChild[block[par + 1]] = first[nbq] then
              rtChild[block[par + 1]] := q
            else if block[rtChild[block[par + 1]]] = nbq then
              rtChild[block[par + 1]] := rtChild[block[par + 1]] + 1
        end;
      {Adjust parent pointers for block nbq}
      parent[nbq] := parent[nbq] − 1 + parity[nbq]; parity[nbq] := 1 − parity[nbq];
      nbq := nextBlock[nbq];
      end;
  else slide := false;
  if (((q ≤ n) and (first[nbq] ≤ n)) or ((q > n) and (first[nbq] > n)))
    and (weight[nbq] = weight[bq] + 1) then
      begin  {Merge q into the block of weight one higher}
      block[q] := nbq; last[nbq] := q;
      if last[bq] = q then begin  {q's old block disappears}
        nextBlock[prevBlock[bq]] := nextBlock[bq];
        prevBlock[nextBlock[bq]] := prevBlock[bq];
        nextBlock[bq] := availBlock; availBlock := bq
        end
      else begin
        if q > n then rtChild[bq] := FindChild(q − 1, 1);
        if parity[bq] = 0 then parent[bq] := parent[bq] − 1;
        parity[bq] := 1 − parity[bq];
        first[bq] := q − 1
        end
        end
```

```
else if $last[bq] = q$ then begin
    if $slide$ then begin  {$q$'s block is slid forward in the block list}
        $prevBlock[nextBlock[bq]] := prevBlock[bq]$;
        $nextBlock[prevBlock[bq]] := nextBlock[bq]$;
        $prevBlock[bq] := prevBlock[nbq]$; $nextBlock[bq] := nbq$;
        $prevBlock[nbq] := bq$; $nextBlock[prevBlock[bq]] := bq$;
        $parent[bq] := oldParent$; $parity[bq] := oldParity$
        end;
    $weight[bq] := weight[bq] + 1$
    end
else begin  {A new block is created for $q$}
    $b := availBlock$; $availBlock := nextBlock[availBlock]$;
    $block[q] := b$; $first[b] := q$; $last[b] := q$;
    if $q > n$ then begin
        $rtChild[b] := rtChild[bq]$;
        $rtChild[bq] := FindChild(q - 1, 1)$;
        if $rtChild[b] = q - 1$ then $parent[bq] := q$
        else if $parity[bq] = 0$ then $parent[bq] := parent[bq] - 1$
        end
    else if $parity[bq] = 0$ then $parent[bq] := parent[bq] - 1$;
    $first[bq] := q - 1$; $parity[bq] := 1 - parity[bq]$;
    {Insert $q$'s block in its proper place in the block list}
    $prevBlock[b] := prevBlock[nbq]$; $nextBlock[b] := nbq$;
    $prevBlock[nbq] := b$; $nextBlock[prevBlock[b]] := b$;
    $weight[b] := weight[bq] + 1$;
    $parent[b] := oldParent$; $parity[b] := oldParity$
    end;
{Move $q$ one level higher in the tree}
if $q \leq n$ then $q := oldParent$ else $q := par$
end;
```

The processing in Algorithm Λ is dominated by the calls to *SlideAndIncrement* made by *Update*. Roughly speaking, *SlideAndIncrement* is called once for each level in the tree above the leaf node for the current letter being processed. Each execution of *SlideAndIncrement* involves sliding the current node and then moving up one level in the tree. The floating tree data structure clearly supports these operations in constant time. The net result is that Algorithm Λ does encoding and decoding in real time; that is, the processing time for each letter in the message is proportional to the length of the letter's encoding.

The running time of our implementation is comparable to that of Algorithm FGK. In more than 95 percent of the time that *SlideAndIncrement* is executed in practice, the node q is neither slid over the next block (that is, we have *slide* = **false**) nor merged into the next block, so the observed execution time is fast. Faster running times can be obtained by replacing the procedures with macros and by breaking up *SlideAndIncrement* into two separate macros, one for leaves and one for internal nodes.

One nice feature of a floating tree, due to the use of implicit numbering, is that the parent of nodes $2j - 1$ and $2j$ is less than the parent of nodes $2j + 1$ and $2j + 2$ in both the implicit and explicit numberings. Such an invariant is not maintained by the data structure in [2], for example.

REFERENCES

1. HUFFMAN, D. A. A method for the construction of minimum redundancy codes. In the *Proceedings of the Institute of Radio Engineers 40* (1951), pp. 1098–1101.
2. KNUTH, D. E. Dynamic Huffman coding. *J. Algorithms 6* (1985), 163–180.
3. VITTER, J. S. Design and analysis of dynamic Huffman codes. *J. ACM 34*, 4 (Oct. 1987), 825–845.

Received March 1987; revised May 1988; accepted October 1988