```
C  SET UP FIRST INTERVAL FOR SIMPSON RULE
  100 FS(2)=FC(.5*T)*COS(.5*WT)
      FS(3) = FC(T)* COS(WT)
      A=T
      GO TO 105
C  ADJUST UPPER LIMIT
  101 NP= IFIX(ALOG(WT/PI256)/ALN2)+1
      TA=  2**NP* PI256/W
      A=TA
C  SET UP FIRST INTERVAL FOR FILON RULE
      FS(2)= FC(.5*TA)
      FS(3)= FC(TA)
C  TAKE LAST INTERVAL FROM LIST
  105 A=AS(N)
      HI=B-A
      WHI=W*HI
      N2=2*N
      F1= FS(N2-1)
      F2= FS(N2)
      F3= FS(N2+1)
      XQ= B-.75*HI
      XQ3 = B-.25*HI
C  TEST TO DETERMINE WHICH QUADRATURE RULE IS APPLICABLE
      IF( WHI  - PI256 -ROC  ) 110,110,111
  110 IF( WHI  - PI256 +ROC  ) 200,200,201
  111 IF( WHI  - PI2   -ROC  ) 220,220,230
C  ESTIMATE BY SIMPSON RULE
  200 FQ= FC(XQ)*COS(W*XQ)
      FQ3=FC(XQ3)*COS(W*XQ3)
      VNEW1= HI*(F1+4.*FQ+F2)/12.
      VNEW2= HI*(F2+4.*FQ3+F3)/12.
      VNEW= VNEW1+VNEW2
      ERR= (PVAL(N)-VNEW1)/15.
      GO TO 300
C  SWITCH FROM FILON TO SIMPSON RULE
  201 F1 = F1* COS(W*A)
      F2 = F2* COS(W*(B-.5*HI))
      F3 = F3* COS(W*B)
      PVAL(N)= HI*(F1+4.*F2+F3)/6.
      GO TO 200
C  ESTIMATE BY FILON2
  220 H=.25*HI
      FQ= FC(XQ)
      FQ3= FC(XQ3)
      NH= IFIX(ALOG(PI2/WHI)/ALN2+ROC)+1
      W1= W1C(NH)
      W2=-W2C(NH)
      W3= W3C(NH)
      WA=W*A
      WA1=W*(B-.5*HI)
      WB=W*B
      CO1= COS(WA1)
      SI1= SIN(WA1)
      VNEW1 = H*((W1*COS(WA)+W2*SIN(WA))*F1 + W3*COS(W*XQ)*
     $          FQ+(W1*CO1-W2*SI1)*F2)
      VNEW2 = H*((W1*CO1 +W2*SI1)*F2 + W3*COS(W*XQ3)*FQ3
     $         +(W1*COS(WB) -W2*SIN(WB))*F3)
      VNEW=VNEW1+VNEW2
      ERT= ER(NH)
      ERR = ERT*(PVAL(N)-VNEW)/(1.-ERT)
C  SKIP CONVERGENCE TEST IF INTERVAL= ONE PERIOD
     IF(WHI- PI2+ ROC  ) 300,300,400
C  ESTIMATE BY FILON1
  230 FQ=FC(XQ)
      FQ3=FC(XQ3)
      W2=W*W
      CONST=8./(W2*HI)
      VNEW1= CONST*(F1-2.*FQ+F2)
      VNEW2= CONST*(F2-2.*FQ3+F3)
      VNEW= VNEW1+VNEW2
      W2=6./W2
      W3=HI*HI
      ERT=(W3/32.-W2)/(W3/8.-W2)
      ERR= ERT*(PVAL(N)-VNEW)/(1.-ERT)
C  CONVERGENCE TEST
C  SKIP CONVERGENCE TEST IF HI.GT.HL
  300 IF(HI- HL) 301,301,400
  301 IF(ABS(ERR)-EPS*HI/B) 500,500,400
C  CONVERGENCE NOT OBTAINED -SPLIT INTERVAL AND ADD TO LIST
C  TEST FOR POSSIBLE LIST OVERFLOW
  400 IF(N-30) 401,600,600
  401 FS(N2+3)= F3
      FS(N2+2)= FQ3
      FS(N2+1)= F2
      FS(N2)= FQ
      AS(N+1)=A+.5*HI
      PVAL(N)=VNEW1
      PVAL(N+1)=VNEW2
      N=N+1
      GO TO 105
C  CONVERGENCE OBTAINED -ADD EXTRAPOLATED PARTIAL SUM TO
C  TOTAL--ADJUST ERROR AND INTERVAL
  500 VAL= VAL +VNEW-ERR
      EPS = EPS-ABS(ERR)
      N=N-1
      B=A
      IF(N) 700,700,105
C  CONVERGENCE FAILURE -ROUTINE RETURNS ERC=1.E+30
C  OPTIONAL ERROR MESSAGE MAY BE INSERTED HERE
  600 FRCOS=ERC
      RETURN
C  COMPUTATIONS COMPLETED SUCCESSFULLY
  700 FRCOS= VAL
      RETURN
      END
```

Algorithm 428

Hu-Tucker Minimum Redundancy Alphabetic Coding Method [Z]

J.M. Yohe* [Recd. 2 January 1970, 12 February 1971, and 21 June 1971]
Mathematics Research Center, University of Wisconsin, Madison, WI 53706

Key Words and Phrases: information theory, coding theory, Hu-Tucker method, minimum redundancy coding
CR Categories: 5.6

Description

This algorithm implements the Hu-Tucker method of variable length, minimum redundancy alphabetic binary encoding [1]. The symbols of the alphabet are considered to be an ordered forest of n terminal nodes. Two nodes in an ordered forest are said to be tentative-connecting if the sequence of nodes between the two given nodes is either empty or consists entirely of nonterminal nodes.

An interval of nodes each pair of which is a tentative-connecting pair is called a tentative connecting string.

Given an ordered forest, we create a new ordered forest with one less tree by combining a pair of tentative-connecting nodes i_1, i_2 such that $Q[i_1] + Q[i_2]$ is minimal. Such a pair is said to have minimal weight sum. The old nodes i_1 and i_2 are eliminated, and the new node replaces the first of the former nodes in the ordering. Its weight is the sum of the weights of the former nodes.

The original forest will, after a finite number of steps, be connected into a single tree. This tree will not, in general, satisfy the order-preserving requirement. However, it is shown in [1] that the path lengths are feasible for the construction of a tree which does satisfy this requirement and is, moreover, minimal in cost.

The present procedure finds a minimal cost tree whose longest path length and total path length are minimal. This was done for the nonalphabetic case by Schwartz [3], and his work carries over directly to the alphabetic case by virtue of the fact that any optimal alphabetic encoding can be constructed by the Hu-Tucker method, simply by modifying the choice of which tentative-connecting nodes are combined. This procedure therefore represents a modification of the Hu-Tucker algorithm to incorporate these ideas of Schwartz.

During the procedure, the array L is used to determine which roots are tentative-connecting. If L is initially filled with 1's instead of 0's, any pair of nodes will be considered tentative-connecting, and the procedure will implement Huffman's method [2], giving the same results as the "bottom merging" method of Schwartz and Kallick [4]. This is because this procedure picks, among those pairs with minimal weight sum, the first pair having minimal length sum.

Modifying the procedure to pick the first pair having maximal length sum would be equivalent to the "top merging" method of Schwartz and Kallick, and would maximize the total number of digits and the maximal length of the code in alphabetic case (and in the nonalphabetic case, if the L-array is initially filled with 1's).

The decision tree may be obtained from the branch lengths by combining the first node of maximal path length with the second

* Sponsored by the United States Army under Contract No.: DA-31-124-ARO-D-462.

node of maximal path length to form a new node with path length one less than that of the original nodes, iterating the procedure until only one node (the root) remains. The code can then be constructed by assigning the value 0 to the first node on the next level from the root and 1 to the second node, appending 0 or respectively 1 to the ith level encoding of a node to obtain the encoding for the first or second son on the $(i + 1)$-th level.

References

1. Hu, T.C., and Tucker, A.C. Optimal computer search trees and variable-length alphabetical codes. *SIAM J. Appl. Math.* (to appear).
2. Huffman, David A. A method for the construction of minimum-redundancy codes. *Proc. I.R.E. 40* (1952), 1098–1101.
3. Schwartz, Eugene S. An optimum encoding with minimum longest code and total number of digits. *Inform. Contr. 7* (1964), 37–44.
4. Schwartz, Eugene S., and Kallick, Bruce. Generating a canonical prefix encoding. *Comm. ACM 7* (1964), 166–169.

Algorithm

```
procedure Hutree (n, Q, L);
  value n; integer n; integer array Q, L;
comment n is the number of symbols in the alphabet, and Q is a
  vector of length n. Q[i] is the weight to be attached to the ith
  symbol in the alphabet.
    The output of the procedure is the vector L of length n. L[i] is
  the length of the path to the ith symbol of the alphabet in a tree of
  minimal cost (i.e. the sum of the Q[i] × L[i] is minimal) which has
  the further property that, subject to minimality of cost, the sum of
  the L[i] and max L[i] are minimal;
begin
  integer maxn, m, i;
  integer array P[1 : n], s[1 : n − 1], d[1 : n − 1];
  comment P is used to hold the weights of the trees in the ordered
    forest, beginning with the alphabet at the start of the procedure
    and ending with the tree at the conclusion of the procedure. L is
    used during the procedure to hold information relating to the
    length sums. At the conclusion of the procedure, L is used to
    return the path lengths.
      If i1 < i2 and nodes i1 and i2 are connected on the mth pass
    through the body of the algorithm, then P[i1] will be set equal
    to P[i1] + P[i2], which is the weight of the new node, and P[i2]
    will be set to zero to indicate that node i2 is no longer a partici-
    pating node. L[i1] is set equal to L[i1] + L[i2] + 1, which is one
    less than the number of terminal nodes which are descended
    from the new node. This number is also one less than the incre-
    ment to the total path length which would result from connect-
    ing the new node i1 in a subsequent pass through the body of
    the algorithm. The value of L[i2] is irrelevant during the re-
    mainder of the procedure. The s and d vectors are used to
    record connections of tentative-connecting nodes. s[m] is set to
    i1 , which is both the ordered position of the leftmost node and
    the ordered position of the new node, and d[m] is set to i2 ,
    which is the ordered position of the rightmost node.
      The variable maxn is set to a number which is larger than the
    sum of the elements of Q.
      The following simple example should be of some assistance
    in understanding the procedure. Assume the procedure is called
    with n = 5 and Q = (3, 1, 1, 1, 3). The evolution of the vectors
    P, L, s, and d is shown in the following table. Values which are
    not relevant are indicated by dashes.

      m     0  1  2  3  4
      P[1]  3  3  3  6  9
      P[2]  1  2  3  3  0
      P[3]  1  0  0  0  0
      P[4]  1  1  0  0  0
      P[5]  3  3  3  0  0
      L[1]  0  0  0  1  4
      L[2]  0  1  2  2  –
      L[3]  0  –  –  –  –
      L[4]  0  0  –  –  –
      L[5]  0  0  0  –  –
      s[m]     2  2  1  1
      d[m]     3  4  5  2;

  maxn := 1;
  for i := 1 step 1 until n do
  begin
    L[i] := 0; P[i] := Q[i];
    maxn := maxn + Q[i];
  end
  comment Since there are n terminal nodes in the original forest, we
    must make exactly n − 1 connections. On each pass through
    the body of this procedure we will determine the next optimal
    connection. We initialize by setting the minimum weight to a
    large value to insure that any valid connection chosen will
    replace the bogus connection initially indicated;
  for m := 1 step 1 until n − 1 do
  begin
    integer j, j1, min1, minL1, j2, min2, minL2, pt, pmin, sumLt,
      sumL, i1, i2;
    i := 0;
    pmin := maxn;
B1:
    i := i + 1;
    comment At B2 we begin our scan of the next tentative-connect-
      ing string to find the most desirable pair in the string. If
      necessary, we skip over any previously absorbed nodes. We
      initialize the most desirable node to the first in the tentative-
      connecting string, and the record of the second most desirable
      node is initialized to reflect a very large minimum. This
      insures that any participating node will be more desirable and
      that valid information will replace the bogus information as
      soon as the next participating node is encountered. If the
      first participating node is the last node in the forest, or if no
      further nodes are participating nodes, then we have com-
      pleted our scan for the next tentative-connecting pair and we
      go to E1 to make the optimal connection;
B2:
    if i1 ≥ n then go to E1 else
    if P[i] = 0 then go to B1;
    min2 := maxn;
    j1 := i;
    minL1 := L[i]; min1 := P[i];
    comment We now begin our scan of all remaining nodes in the
      current tentative-connecting string. The string will end as
      soon as we have examined a participating node which has not
      previously been combined. The purpose of this scan is to
      locate the optimal tentative-connecting pair in the tentative-
      connecting string. The optimal pair is defined to be that pair
      with minimal weight and minimal length sum which occurs
      first in the tentative-connecting string;
    for j := i + 1 step 1 until n do
    begin
    comment We check for P[j] > 0 to see whether the jth node is a
      participating node. If P[j] = 0, the node has previously been
      absorbed and we pass over the empty space;
```

```
    if P[j] > 0 then
      begin
        if P[j] < min1 ∨ (P[j] = min1 ∧ L[j] < minL1) then
        begin
        comment If the jth node is "more desirable" than either of
          the previously most desirable tentative-connecting nodes,
          we record the previous most desirable node as the second
          most desirable node and record the jth node as being
          most desirable;
          min2 := min1; j2 := j1; minL2 := minL1;
          min1 := P[j]; j1 := j; minL1 := L[j];
        end
        else if P[j] < min2 ∨ (P[j] = min2 ∧ L[j] < minL2) then
        begin
        comment If the jth node was not more desirable than the
          previous most desirable node, but is more desirable
          than the previous second most desirable node, we record
          the jth node as being second most desirable;
          min2 := P[j]; j2 := j; minL2 := L[j];
        end;
        if L[j] = 0 then go to E2;
        comment If L[j] = 0 then we have reached the end of the
          current tentative-connecting string, and we have found
          the most desirable pair in that string. We now go to
          compare it with the previous most desirable pair in the
          forest;
      end
    end;
E2: pt := P[j1] + P[j2];
    sumLt := L[j1] + L[j2];
    comment We have now found the next tentative-connecting
      pair, namely the j1 and j2 nodes. Here, we test this new pair
      against the previous minimal pair to see whether the new pair
      is more desirable. The new pair is more desirable if its weight is
      less than that of the previous pair, or if its weight is equal to
      that of the previous pair and its length sum is smaller;
    if pt < pmin ∨ (pt = pmin ∧ sumLt < sumL) then
    begin
      pmin := pt;
      i1 := j1; i2 := j2;
      sumL := sumLt;
    end;
    comment The next tentative-connecting string begins with the
      last participating node in the current tentative-connecting
      string. Hence we replace i by j and return to B2 to begin
      processing the next tentative-connecting string;
    i := j;
    go to B2;
    comment Upon reaching E1 the procedure has scanned all
      tentative-connecting pairs and the decision has been made to
      connect nodes in order positions i1 and i2. We switch i1 and
      i2 if necessary to insure that i1 < i2. We record the connec-
      tion by setting s[m] := i1 and d[m] := i2. The weight of the
      new node is placed in the weight table in position i1 (the
      order position of the new node). The weight in the order
      position of the second combined node is set to zero to indi-
      cate that the node has now been absorbed and no longer
      participates in the scan. L[i1] is set to one less than the incre-
      ment to the path length sum which would result from con-
      necting the new node;
E1: if i1 > i2 then
    begin
      j1 := i1; i1 := i2; i2 := j1;
    end;
    s[m] := i1; d[m] := i2;
    P[i1] := pmin; P[i2] := 0;
    L[i1] := sumL + 1;
  end;
  comment s[n − 1] gives the ordered location of the root of the
    tentative tree. We now generate the path lengths as follows:
    the path length to the root is zero, and if the path length to any
    node is i, then the path length to each of its sons is i + 1. The
    two sons of the node whose order position is given in s[m] lie in
    the order positions given in s[m] and d[m]. Moreover, if an
    order position is given in s[m] for m < n − 1 then that order
    position must be listed in s[j] or d[j] for some j > m, so the path
    lengths obtained by this algorithm are well defined.
```

Returning to our example, we now trace the construction of the vector of path lengths. This is shown in the following table. For the sake of clarity, the vectors s and d are now shown in reverse order.

m		4	3	2	1
$L[1]$	0	1	2	2	2
$L[2]$	–	1	1	2	3
$L[3]$	–	–	–	–	3
$L[4]$	–	–	–	2	2
$L[5]$	–	–	2	2	2
$s[m]$		1	1	2	2
$d[m]$		2	5	4	3

Thus the final value of the vector L is (2, 3, 3, 2, 2);

```
  L[s[n − 1]] := 0;
  for m := n − 1 step −1 until 1 do
  L[s[m]] := L[d[m]] := L[s[m]] + 1;
end;
```