Optimal Computer Search Trees and Variable-Length Alphabetical Codes

T. C. Hu; A. C. Tucker

SIAM Journal on Applied Mathematics, Vol. 21, No. 4 (Dec., 1971), 514-532.

Stable URL:
http://links.jstor.org/sici?sici=0036-1399%28197112%2921%3A4%3C514%3AOCSTAV%3E2.0.CO%3B2-R

SIAM Journal on Applied Mathematics is currently published by Society for Industrial and Applied Mathematics.

SIAM J. APPL. MATH.
Vol. 21, No. 4, December 1971

OPTIMAL COMPUTER SEARCH TREES AND VARIABLE-LENGTH ALPHABETICAL CODES*

T. C. HU AND A. C. TUCKER†

Abstract. An algorithm is given for constructing an alphabetic binary tree of minimum weighted path length (for short, an optimal alphabetic tree). The algorithm needs $4n^2 + 2n$ operations and $4n$ storage locations, where n is the number of terminal nodes in the tree. A given binary tree corresponds to a computer search procedure, where the given files or letters (represented by terminal nodes) are partitioned into two parts successively until a particular file or letter is finally identified. If the files or letters are listed alphabetically, such as a dictionary, then the binary tree must have, from left to right, the terminal nodes consecutively. Since different letters have different frequencies (weights) of occurring, an alphabetic tree of minimum weighted path length corresponds to a computer search tree with minimum-mean search time. A binary tree also represents a (variable-length) binary code. In an alphabetic binary code, the numerical binary order of the code words corresponds to the alphabetical order of the encoded letters. An optimal alphabetic tree corresponds to an optimal alphabetic binary code.

1. Introduction. One problem of many applications is to construct an optimal binary tree, that is, a binary tree of minimum weighted path length (see Knuth [3, pp. 399–415]). An elegant algorithm for finding such a tree has been given by D. A. Huffman [2]. In this paper we examine a variation of this problem in which an order restriction on the terminal nodes is added. A computer search procedure to identify an unknown letter is of the form: is the letter before m; if so, is it before f; etc. Such a search procedure corresponds to an alphabetic binary tree, where the terminal nodes of the tree ordered from left to right correspond to the letters in alphabetic order. Since different letters have different frequencies (weights) of occurring, an optimal alphabetic tree corresponds to a computer search tree with minimum-mean-search time. A binary tree also represents a (variable-length) binary code. In an alphabetic binary code, the numerical binary order of the code words corresponds to the alphabetical order of the encoded letters. An optimal alphabetic tree corresponds to an optimal alphabetic binary code. In 1959, Gilbert and Moore [1] gave an algorithm for constructing an optimal alphabetic tree. The number of operations in the algorithm is proportional to $(n^3 - n)/6$, where n is the number of terminal nodes in the tree. Recently, Knuth [4] solved the alphabetic tree problem using $O(n^2)$ operations and $3n^2$ storage locations. In the present paper, we give a different and faster algorithm also using $O(n^2)$ operations but only $4n$ storage locations. The present algorithm also solves a more general order-preserving problem.

* Received by the editors September 8, 1970, and in revised form May 3, 1971.

† Mathematics Research Center, University of Wisconsin, Madison, Wisconsin 53706. This research was supported in part by the United States Army under Contract DA-31-124-ARO-D-462, and in part by the National Science Foundation under Grant GJ-28339.

The following notation is used:

l_j:	the path length of the node j.
n:	the number of terminal nodes.
S:	the initial sequence of terminal nodes.
S^*:	the initial sequence of terminal and internal nodes.
$C(S)$:	the class of T-C level-by-level trees built on an initial sequence S.
T:	a tree in general (a forest or a tree in the class $C(S)$ from § 4 on).
$\|T\|$:	the weighted path length or the cost of the tree T.
T':	the tree built by the T-C algorithm.
$T'(m)$:	the forest built by the T-C algorithm in m additions.
T_N:	the normalized tree of a tree in $C(S)$.
V_j:	the terminal node j.
v_j:	the internal node j, or a generic node j.
w_j:	the weight of the node j.

2. Definitions. In this section, we give definitions of terms used in this paper. Unless stated explicitly to the contrary, all definitions and notations are the same as those used in the book by Knuth [3, vol. 1, pp. 362–405].

We consider a binary tree (called an extended binary tree by Knuth) as a node (called the *root*) with its two disjoint binary trees called the left and right subtrees of the root. For our purpose, we consider a binary tree with n terminal nodes (a *terminal node* has no sons, while the other nodes, called *internal nodes*, each have two other nodes as their sons). It is well known that the number of internal nodes is always one less than the number of terminal nodes. The internal nodes will be denoted by v_i, $i = 1, \cdots, n-1$, and are represented by circles in Fig. 1. The terminal nodes will be denoted by $V_j, j = 1, \cdots, n$, and are represented by squares in the figure. In this paper, when no distinction between terminal nodes and internal nodes is necessary, we shall use the word "node" to mean either a terminal node or an internal node and denote it by a lower-case v.

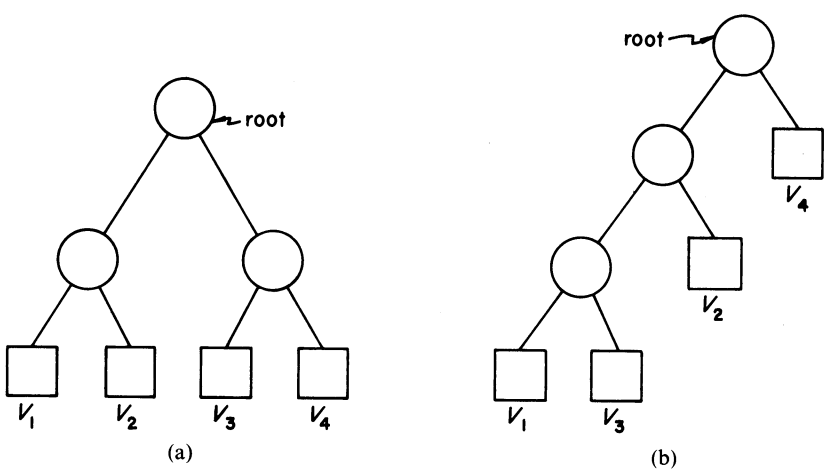

FIG. 1

There is a unique path from the root to every node (internal or terminal), and the *path length* of a node is the length (number of arcs) of the path from the root to that node. For example, in Fig. 1(a) the path length of every terminal node is 2, and in Fig. 1(b) the path length of V_1 is 3. We shall denote the path length of V_j by l_j. The *path length of a tree* is the sum of all path lengths of all the terminal nodes. In Fig. 1(a), the path length of the tree is $l_1 + l_2 + l_3 + l_4 = 2 + 2 + 2 + 2 = 8$, and in Fig. 1(b), the path length of the tree is $l_1 + l_2 + l_3 + l_4 = 3 + 2 + 3 + 1 = 9$. In many problems, every terminal node V_j has associated with it a positive weight w_j. The *weighted path length of a tree*, denoted by $|T|$, is defined to be $\sum w_j l_j$. For example, in Fig. 1(b), if $w_1 = 1$, $w_2 = 3$, $w_3 = 5$ and $w_4 = 4$, then the weighted path length of the tree is $1 \times 3 + 3 \times 2 + 5 \times 3 + 4 \times 1 = 28$. Assume that we are given n terminal nodes V_j, $j = 1, \cdots, n$, with weights w_j, $j = 1, \cdots, n$. Our problem is to construct a binary tree having these terminal nodes such that the weighted path length $\sum w_j l_j$ is minimum. Furthermore, the binary tree constructed must be *alphabetic*, that is, the terminal nodes $V_1, V_2, \cdots, V_n$ must be in left-to-right order. For example, the binary tree in Fig. 1(a) satisfies the order restriction, while the binary tree in Fig. 1(b) does not since the terminal nodes left to right are V_1, V_3, V_2, V_4.

It is well known that a binary tree corresponds to a binary code and an alphabetic tree corresponds to an alphabetical code (see Gilbert and Moore [1]).

We say a node is *at level i* if the path length of the node is i (level is just another way of saying the path length). Thus the root is at level zero, and is considered to be the *highest* level.

We say that a node v_i *dominates* a node v_j at a lower level if there is a path descending from v_i to v_j. Thus the root dominates every other node.

Given a binary tree T with n terminal nodes, we can write down the path lengths of the n terminal nodes from the left to the right as a sequence of n positive integers. On the other hand, if we are given an arbitrary sequence of n positive integers, there may not exist a binary tree T whose n terminal nodes have these positive integers as their path lengths from the left to right. For example, it is impossible to have a binary tree of three terminal nodes with path lengths corresponding to 2, 1, 2. But it is possible to have a binary tree with three terminal nodes with path lengths corresponding to 1, 2, 2, or 2, 2, 1.

Thus, we define a sequence of n positive integers to be a *feasible sequence* if there exists a binary tree with n ordered terminal nodes with path lengths corresponding to these n positive integers from left to right.

LEMMA 2.1. *A finite sequence of positive integers is a feasible sequence if and only if the following three conditions are satisfied*:

(i) *If the largest integer in the sequence is q, then there must be an even number of q's and such q's always occur in consecutive sets of even length.*

(ii) *If we form a reduced sequence from the original sequence by successively replacing (from left to right) every two consecutive q's by one occurrence of the integer* $q - 1$, *then the reduced sequence again satisfies* (i).

(iii) *If the process of* (ii) *is repeated by considering the reduced sequence as the original sequence,* (i) *is still satisfied until finally a reduced sequence of* 1, 1 *is formed.*

Proof. Because all the nodes at the level q must be the sons of nodes at the level $q - 1$, this implies that the nodes at the level q must occur in pairs. So (i) is clearly necessary. If we erase all the nodes at the level q, then we have a binary tree with the lowest level $q - 1$. The fathers of the nodes in the level q now become terminal nodes themselves. And the necessary condition (i) becomes the condition (ii). The necessity of (iii) is obvious.

If conditions (i), (ii) and (iii) are satisfied, then we can build the binary tree from the bottom up (i.e., from the lowest level) and finally obtain the root which is the father of two nodes at the level 1.

LEMMA 2.2. *The total number of internal nodes in a binary tree is one less than the total number of terminal nodes.*

Proof. (See Knuth [3, p. 399].)

Let us consider an alternate way of calculating the weighted path length of a tree. First we write the weight of every terminal node inside the square node which represents it. Then we write the sum of the two weights of the sons inside the circular node which is the father. Whenever two nodes have weights written inside their nodes, the weight associated with the father is the sum of the two weights. In this way, every circular node gets a weight also. We claim that the sum of all weights of the $n - 1$ circular nodes is the weighted path length $\sum w_j l_j$ of the binary tree. This claim is certainly true for a binary tree with two terminal nodes. And if the claim is true for both the left and right subtrees of a binary tree, then it is also true for the whole binary tree. We state this as a lemma. (See Knuth [3, p. 405].)

LEMMA 2.3. *The sum of $n - 1$ weights in the $n - 1$ circular nodes is the weighted path length of the binary tree.*

Instead of using the long term "weighted path length of a tree" we shall use *cost of a tree* from now on. A minimum-cost tree is called *optimal.*

Given n terminal nodes, a circular node corresponds to an addition of two other nodes. A binary tree corresponds to a way of bracketing the weights. For example, the binary tree in Fig. 1(a) can be represented by the nonassociative addition $((w_1 + w_2) + (w_3 + w_4))$, and the binary tree in Fig. 1(b) can be represented by $(((w_1 + w_3) + w_2) + w_4)$.

Thus by Lemma 2.2, we can construct a binary tree with $n - 1$ *nonassociative* additions and, at the same time, calculate its costs by Lemma 2.3.

3. Huffman's tree. In this section, we consider some results related to the construction of Huffman's tree where no restriction is imposed on the ordering of the terminal nodes. Huffman's algorithm is as follows: First find the two nodes with smallest weights, say w_1 and w_2; then these two nodes are combined (they have a common father); next replace the subtree formed by the two nodes and their father by a new terminal node having weight $w_1 + w_2$; and repeat the same procedure on the reduced problem of $n - 1$ terminal nodes with weights $w_1 + w_2$, $w_3, \cdots, w_n$. From now on the process of letting two nodes have a common father will be referred to as *combining* the two nodes or *adding* the two nodes.

Huffman's algorithm constructs an optimal tree from a given set of n weights. The cost of the tree can be expressed as the total of $n - 1$ sums (Lemma 2.3) and the algorithm is a way of performing $n - 1$ sums sequentially. Suppose that

we stop Huffman's algorithm after m additions ($1 \leqq m < n - 1$). We shall prove that the sum of the m weights in the m circular nodes constructed so far is also a minimum compared with all other ways of performing m additions among the given w_j. In other words, Huffman's algorithm gives an optimal forest. (A forest is a set of trees.)

LEMMA 3.1. *Huffman's algorithm gives an optimal m-sum forest.*

Proof. The proof is by induction. It is clear that Huffman's algorithm gives an optimal 1-sum forest. Assume that Huffman's algorithm is correct for $m < m_0$ ($m_0 > 1$), and consider an optimal m_0-sum forest on weights $w_1, w_2, w_3, \cdots, w_k$ ($k > m_0$). In this forest, there exists at least one *internal* node v_i of maximal path length among all internal nodes. If the two sons of v_i do not have the two smallest weights (let these be w_1 and w_2), then we can interchange the two sons of v_i with the terminal nodes with w_1 and w_2 without increasing the cost of the forests.

Now observe that if an m_0-sum forest on $w_1, w_2, \cdots, w_k$ combines w_1 and w_2, then this forest is optimal if and only if the other $m_0 - 1$ combinations in this forest form an optimal forest on $w_1 + w_2, w_3, \cdots, w_k$. However, by induction, Huffman's algorithm generates an optimal $(m_0 - 1)$-sum forest.

4. Alphabetic trees and the tentative-connecting algorithm. Now we consider alphabetic binary trees. Then, as one scans the bottom of the tree from left to right, the terminal nodes must occur in the given order. Such a tree is called an *alphabetic* tree for the given ordered set of terminal nodes. Any alphabetic tree can be built as follows: Start with an *initial sequence*, $V_1, V_2, V_3, \cdots, V_n$, having the terminal nodes in their given order; combine some adjacent pair of nodes V_i, V_{i+1} and form a new sequence $V_1, V_2, \cdots, V_{i-1}, v_{(i,i+1)}, V_{i+2}, \cdots, V_n$, where $v_{(i,i+1)}$ is an internal node (the father of V_i and V_{i+1}) and the other nodes are still terminal; now combine some adjacent pair in this new sequence and replace the combined pair by their father in the sequence; and so on. Any of the intermediate sequences, as well as the initial sequence, is called a *construction sequence.* The nodes in a construction sequence are either terminal nodes or the roots of previously constructed feasible subtrees. By Lemma 2.2, $n - 1$ combinations (n is the length of the initial sequence) are needed to form an alphabetic tree; if fewer than $n - 1$ combinations are made in the above construction, an *alphabetic forest* results.

Ideally, for a given initial sequence of terminal nodes, we would like to construct an optimal alphabetic tree (or forest) by simply combining the minimum-weight adjacent pair in successive construction sequences in a manner analogous to the Huffman algorithm in the unrestricted case (throughout this paper, each father is assumed to have a weight equal to the sum of its sons' weights). Figure 2 shows that this *is not* possible, for the tree in Fig. 2 is an optimal alphabetic tree and yet the minimum-weight adjacent pair in the initial sequence is not combined (the weights of the terminal nodes are given inside their squares).

We shall introduce an algorithm for constructing optimal alphabetic trees (also forests). The algorithm has two parts. The first part constructs an optimal tree T' which *does not* satisfy the ordering restriction. Then it will be shown that T' can be converted into another binary tree T'_N with the same cost which *does* satisfy the ordering restriction. We shall first describe the algorithm for constructing

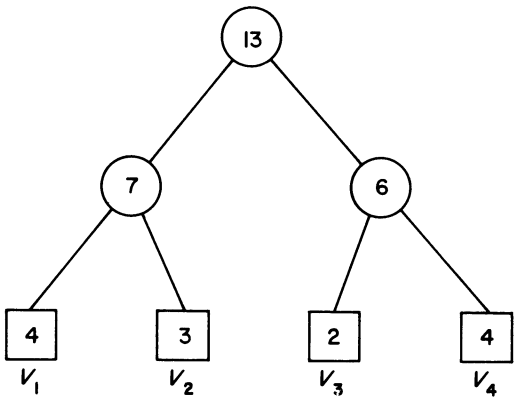

FIG. 2

T'. In the process of building a tree from a given initial sequence, if we combine two nonadjacent nodes in a construction sequence, let the father take the position of its left son in the resultant construction sequence (and of course the right son no longer appears). Two nodes in a given construction sequence are called *tenative-connecting* (T-C, for short) if the sequence of nodes between the two nodes is either empty or consists entirely of internal nodes (roots of subtrees). In Fig. 3, we illustrate the definition of tentative-connecting. Figure 3 shows a construction sequence and any two nodes joined by solid lines are T-C in this sequence, while two nodes joined by broken lines are not T-C. If V_2 and v_4 were combined, then in the resultant construction sequence v_1 and v_5 would be T-C, as would v_1 (or v_5) and the father of V_2 and v_4. If V_7 and V_{10} are first combined, then v_6 and v_8 become T-C.

T-C ALGORITHM (for constructing optimal tree T'). In each successive construction sequence (starting with the initial sequence), combine the pair of T-C nodes with the minimum sum of weights (in case of a tie, combine the pair with the leftmost node—if several pairs have the same leftmost node, pick the pair with leftmost second node).

Figure 4 illustrates the T-C algorithm (the initial weights are in the square nodes). The T-C algorithm starts by combining the minimum-weight adjacent pair of terminal nodes (the rightmost terminal pair with weights 1 and 3 in Fig. 4) to get a new (internal) node with weight 4. It proceeds to get the circular nodes

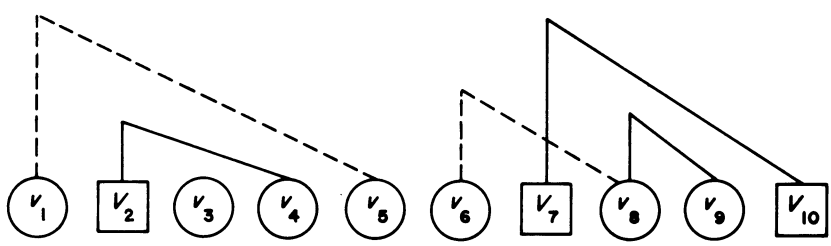

FIG. 3

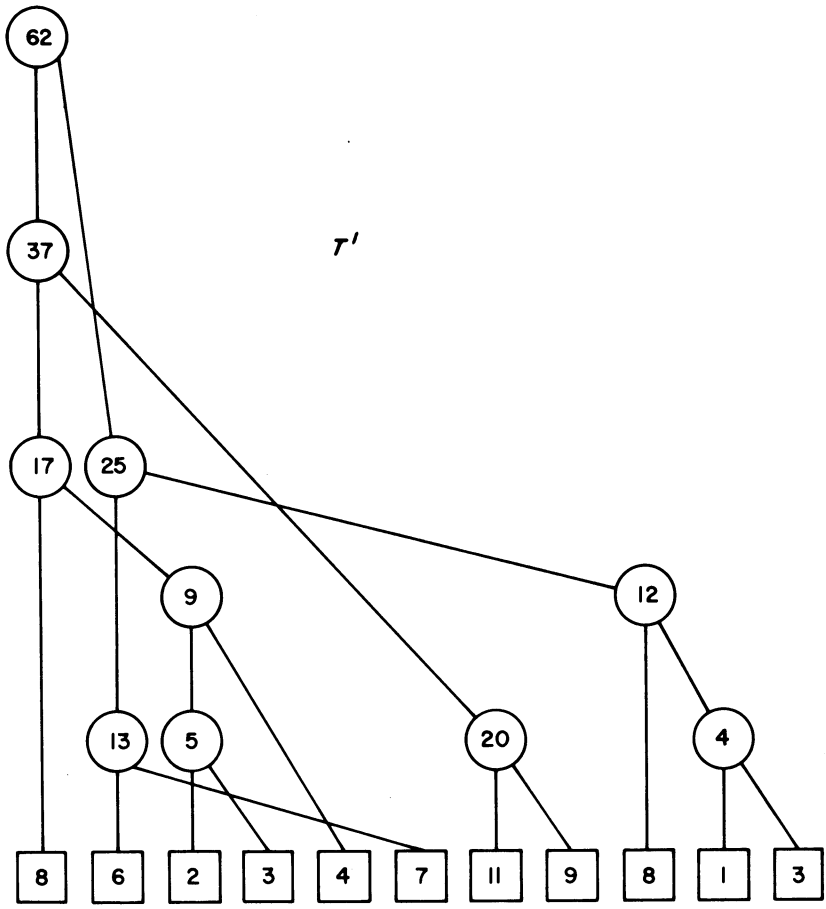

FIG. 4

with weights 5, 9, 12, 13, 17, 25, 37, 62 (we place a father over its left son to indicate its position in the successive construction sequences).

At this stage, the reader should familiarize himself with this algorithm and try to build a tree T' from some other sequences of terminal nodes. Although this algorithm is simple to state, the proof that the T' can be converted into an optimal alphabetic tree T'_N is somewhat complicated. (In Fig. 6, we show the corresponding T'_N of Fig. 4.) Most of the following theorems involve T-C level-by-level trees (T' is such a tree). First we define this type of tree. Then we present the reader with our strategy for the rest of the paper.

A *T-C tree* (built on a given initial sequence) is a tree which can be built up in successive construction sequences such that each successively combined pair of nodes is tentative-connecting in its construction sequence when combined. The tree T' is certainly a T-C tree.

For a *given* T-C tree, a *T-C level-by-level construction* of the T-C tree combines all nodes on the *lowest* level of the T-C tree *first*, then all nodes on the next-to-lowest level, and so on (all combined pairs must still be T-C when combined in

their construction sequence). For example, if the T-C tree is as shown in Fig. 4, a T-C level-by-level construction would create the internal node with weight 5 first, then internal nodes of weights 9 and 4, then internal nodes of weights 17, 13, 20 and 12, and finally internal nodes of weights 37 and 25. It is not necessary to create all internal nodes from the left to right at a given level, but no tentative-connecting pair should be combined until all tentative-connecting pairs at lower levels have been combined.

Consider the sequence of terminal nodes with weights 2, 3, 1, 1, 3, 2. The tree T' built by the T-C algorithm is shown in Fig. 5(a), and a *T-C tree* is shown in Fig. 5(b). The reader should verify that there is a *T-C level-by-level construction* for the tree in Fig. 5(a) but not for the tree in Fig. 5(b). Those T-C trees for which the T-C level-by-level construction is possible are called *T-C level-by-level trees.* A *T-C forest* and *T-C level-by-level forest* are similarly defined. (The level of a node in a forest is the level of the node in the tree to which the node belongs. All isolated terminal nodes and roots of trees are at level zero.)

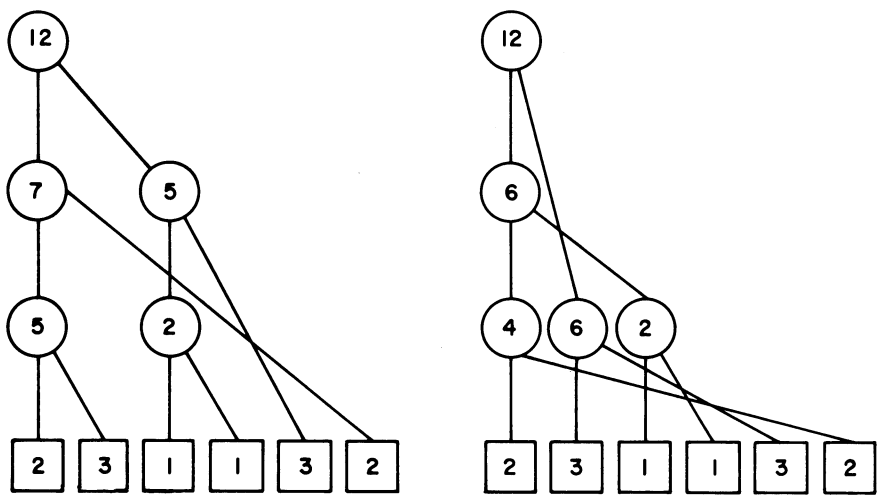

FIG. 5

We shall denote an initial sequence of terminal nodes by S, *and let* $C(S)$ *be the class of all T-C level-by-level forests (including trees) built on an initial sequence* S. Since an adjacent pair is, by definition, tentative-connecting, *all alphabetic forests built on* S *are in* $C(S)$.

Because the proof is long, we shall outline the general approach of the proof.

First we shall consider the class of T-C level-by-level forests (or trees). This class clearly includes all alphabetic forests (or trees) since an alphabetic forest (or tree) always has a T-C level-by-level construction. We shall show that for every forest (or tree) in $C(S)$, there is an alphabetic forest (or tree) of the same cost. In § 5, we shall prove that the tree T' is in this class. This would mean that there exists an alphabetic tree T'_N of the same cost as T'. In § 6, we shall prove that T' is optimal in this class.

Since we shall be dealing with trees and forests in $C(S)$ from now on, we shall use T to denote a forest in $C(S)$.

It will be shown that every forest T in $C(S)$ has an associated alphabetic forest in $C(S)$ of the same cost. We call this associated forest the *normalized form* of T, written T_N. T_N is a forest obtained from T as follows: Let level q be the lowest level in T. There are an even number of nodes, say $2k$, on this level (as on any level below level 0, since each such node has a unique brother). Reassign the father-son relationships between the k fathers on level $q-1$ and the $2k$ sons on level q so that the leftmost father has as sons the two leftmost nodes on level q and so on (from left to right). This reassignment does not change the path length of any initial nodes (nodes in the initial sequence), i.e., the nodes on level q remain on level q. Now repeat the reassignment on successively higher levels of T. Again at each level, the reassignments do not change the path length of any initial nodes. Figure 6 shows the normalized form of the tree in Fig. 4.

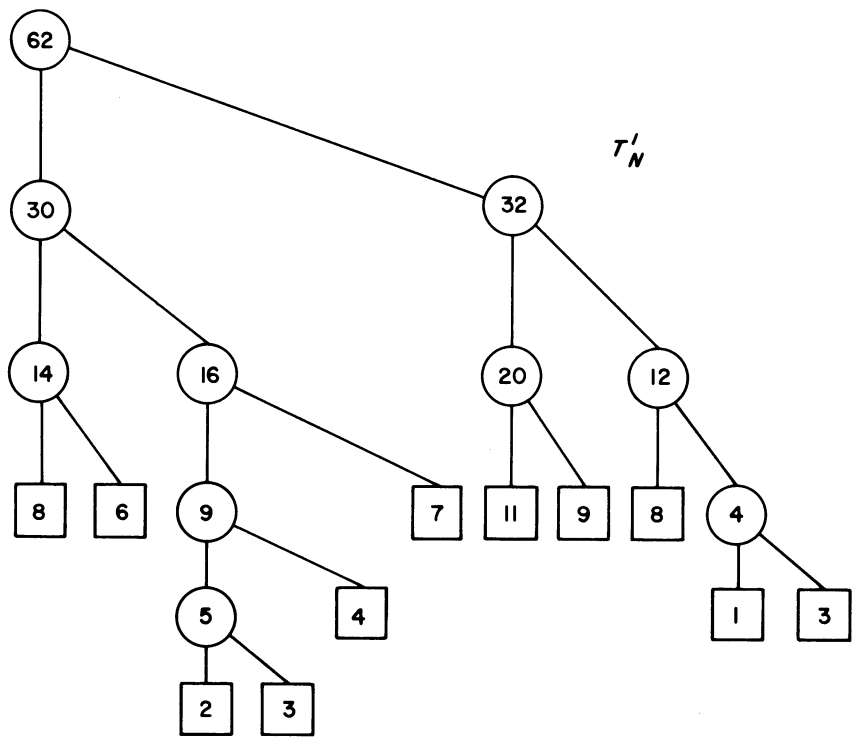

FIG. 6

THEOREM 4.1. *For all T in $C(S)$, T_N is an alphabetic forest in $C(S)$ and $|T| = |T_N|$ (they cost the same).*

Proof. The fact that $|T_N| = |T|$ follows from the properties of the reassignments mentioned above. It is also not hard to see that all the combinations in T_N are tentative-connecting. Consider any level k where the reassignments are made. If there are some terminal nodes positioned between the nodes on level k and those terminal nodes are at higher levels, then there must be an even number (perhaps zero) of nodes at level k between any two such terminal nodes (since

T is a T-C level-by-level forest). Then in T_N, there is a consecutive T-C pairing of all the nodes on level k positioned between two terminal nodes on higher levels. It remains to show that T_N is alphabetic.

Let q be the lowest level of T and suppose T_N combines two (consecutive) nodes on level q which are not adjacent in S, the initial sequence. Then between the two nodes is a terminal node which necessarily is on a higher level of T_N (or the given two nodes would not be consecutive on level q). But then this given pair is not T-C on level q of T_N—a contradiction. Now assume all internal nodes at level i or lower ($0 < i < q$) in T_N are roots of alphabetic subtrees. Then the preceding argument shows that the nodes on level i are alphabetically combined in T_N so that all internal nodes on level $i - 1$ are roots of alphabetic subtrees. An inductive argument completes the proof.

For a given initial sequence S, let $T'(m)$ denote the m-sum forest generated by m steps of the T-C algorithm (and T' the finished tree). If $T'(m)$ and T' are optimal (or minimal cost) in $C(S)$, then so are the alphabetic $T'(m)_N$ and T'_N by Theorem 4.1.

It remains to prove the optimality of the T-C algorithm (which we do in § 6). However, there is another equally pressing problem which the reader may have noticed—namely, the construction of $T'(m)$ by the T-C algorithm is not likely to be level-by-level. For example, in Fig. 4 the first combination made by the T-C algorithm is not at the lowest level. In the next section we prove that there exists a way to build $T'(m)$ in the manner of a T-C level-by-level construction.

5. $T'(m)$ is a T-C level-by-level forest. In this section, we prove that $T'(m)$ is in $C(S)$.

We shall use the symbol $\sim$ to denote that two nodes are tentative-connecting in a given construction sequence.

LEMMA 5.1. *The weights associated with the internal nodes created successively by the T-C algorithm are monotonically increasing.*

Proof. We want to prove that if an internal node v_i is created just before another internal node v_j, then $w_i \leqq w_j$. In other words, if v_i is the father of v_b and v_c and v_j is the father of v_a and v_d, then

$$w_b + w_c \leqq w_a + w_d. \tag{1}$$

If v_a or v_d coincides with v_i, then (1) holds. Therefore, we shall assume that there are four distinct nodes v_a, v_b, v_c and v_d involved in a construction sequence. Renaming the four nodes from the left to right as v_1, v_2, v_3 and v_4, and letting $w_{1,2} = w_1 + w_2$, $w_{3,4} = w_3 + w_4$, we have

$$w_i + w_j = (w_b + w_c) + (w_a + w_d) = w_{1,2} + w_{3,4}. \tag{2}$$

If (1) does not hold and

$$w_i > w_j, \tag{3}$$

then it follows from (2) that

$$w_i > \min(w_{1,2}, w_{3,4}). \tag{4}$$

Note that v_1 is one of the four nodes v_a, v_b, v_c or v_d which is tentative-connecting to one of the three nodes to its right in creating v_i or v_j, hence $v_1 \sim v_2$. The same

argument shows that $v_3 \sim v_4$. But $v_{1,2}$ and $v_{3,4}$ in (4) can be created independently in a construction sequence. Thus (4) implies there exists at least one T-C pair ($v_{1,2}$ or $v_{3,4}$) with weight less than w_i, contradicting the T-C algorithm. This completes the proof.

We define a *noninitial* node in a forest to be a node not in the initial construction sequence. If the initial construction sequence consists of terminal nodes, then noninitial nodes are synonymous with internal nodes. In the next section, we shall work with an initial construction sequence with both internal and terminal nodes. Then noninitial nodes will mean those internal nodes which are not in the initial construction sequence. The theorems and lemmas in this section are valid for these more general forests built from a construction sequence of internal and terminal nodes.

LEMMA 5.2. *If v_i and v_j are two equal-weight noninitial nodes in a construction sequence of T', and if v_i was formed first (by the T-C algorithm), then v_i is to the left of v_j in the construction sequence.*

Proof. We can assume v_j was formed immediately after v_i, for if the T-C algorithm formed other nodes $v_{i_1}, v_{i_2}, \cdots$ between v_i and v_j, then:

(a) by Lemma 5.1, the v_{i_k}'s are of the same weight as v_i and v_j, and

(b) proving the lemma true for each two successive equal-weight nodes implies it is true for v_i and v_j.

Let v_i be the father of v_b and v_c and v_j be the father of v_a and v_d say. Rename the four distinct nodes in the construction sequence from the left to right as v_1, v_2, v_3 and v_4. By the same argument as used in the proof of Lemma 5.1, we have

$$v_1 \sim v_2 \quad \text{and} \quad v_3 \sim v_4. \tag{5}$$

Either $w_1 + w_2 < w_3 + w_4$ or $w_1 + w_2 > w_3 + w_4$ will contradict the T-C algorithm. So we have

$$w_1 + w_2 = w_3 + w_4. \tag{6}$$

If v_i is the node $v_{1,2}$, $v_{1,3}$, or $v_{1,4}$, then the theorem is proved. If v_i is the node $v_{3,4}$, then it contradicts the T-C algorithm. Therefore, the only possibilities left are

$$v_i = v_{2,3} \quad \text{or} \quad v_i = v_{2,4}.$$

If $v_i = v_{2,3}$, then by assumption, $w_i = w_2 + w_3 = w_1 + w_4 = w_j$, and from (6), we have

$$w_1 = w_3. \tag{7}$$

From (7) we have $w_1 + w_2 = w_2 + w_3 = w_i$, which contradicts the fact that the T-C algorithm picks the leftmost pair.

If $v_i = v_{2,4}$, then $w_i = w_2 + w_4 = w_1 + w_3 = v_j$ and from (6) we have

$$w_1 = w_4. \tag{8}$$

But (8) implies $w_1 + w_2 = w_2 + w_4 = w_i$, which again contradicts the fact that the T-C algorithm picks the leftmost pair.

LEMMA 5.3. *Suppose the two noninitial nodes v_i and v_j are tentative-connecting in some construction sequence of $T'(m)$. If $w_i = w_j$ and v_i is to the left of v_j, or if $w_i < w_j$, then $l_i \geqq l_j$ in $T'(m)$ (l_k is the level of node v_k in $T'(m)$).*

Proof. Our proof is by induction on $k = l_j$. If $k = 0$, the result is trivial. Assume the theorem is true for $k < m$ ($m \geqq 1$) and consider the case where $k = m$. Observe v_i is combined before (or at the same time as) v_j if $w_i < w_j$, and the same is true if $w_i = w_j$ and v_i is to the left of v_j since the T-C algorithm works from left to right. If v_{i^*} and v_{j^*} are the fathers of v_i and v_j, respectively, then $w_{i^*} \leqq w_{j^*}$, by Lemma 5.1, and further if $w_{i^*} = w_{j^*}$, then v_{i^*} is to the left of v_{j^*} by Lemma 5.2 (if v_j is combined with v_i or v_{i^*}, the lemma is trivial). Now $l_{j^*} = k - 1 < m$ implies $l_{i^*} \geqq l_{j^*}$ by induction. So $l_i \geqq l_j$.

THEOREM 5.1. *$T'(m)$ is in $C(S)$.*

Proof. We wish to show that any forest or tree built by the T-C algorithm has a T-C level-by-level construction. Consider first the nodes in the lowest level of $T'(m)$. We shall do all the combinations made by the T-C algorithm on this level in the order of the weights of the internal nodes created (in case of ties, we go from left to right as in the T-C algorithm). Then we shall do the same for nodes in the next-to-lowest level. For example, in Fig. 4, we would create the internal nodes with weights 5, 4, 9, 12, 13, 17, 20, 25, and 37 successively. Since all pairs combined by the T-C algorithm are T-C, we now have a T-C level-by-level construction of $T'(m)$ unless the kth combined pair in the T-C algorithm is not tentative-connecting at its level because some ith pair (for some $i < k$) contains a terminal node which is at a higher level and is positioned between the kth pair. We shall prove that such a situation cannot happen.

We shall use S_k to denote the construction sequence resulting after combining the kth pair in the T-C algorithm. (By this notation, the initial sequence S should be denoted by S_0.) Consider the construction sequence S_k and let v_k be the kth internal node created by the T-C algorithm and v_i the node in S_k dominating (or perhaps equal to) the ith internal node created by the T-C algorithm. Since at least one son of the ith internal node is assumed to be between the two sons of v_k, it follows that v_k and v_i are T-C in S_k. Since v_i is created before v_k in the T-C algorithm, $w_i \leqq w_k$ by Lemma 5.1. If $w_i = w_j$, then v_i is to the left of v_k by Lemma 5.2. From Lemma 5.3, we know that $l_k \leqq l_i$, which contradicts the assumption that the kth combination was at a lower level than the ith combination.

6. The optimality of $T'(m)$. We have shown in § 5 that T' can be converted into an alphabetic tree of the same cost. We also know that the class $C(S)$ includes the class of alphabetic trees. If we can prove that T' is an optimal tree in $C(S)$, then we shall have proved the optimality of the T-C algorithm. Instead of proving that T' is an optimal tree in $C(S)$, we shall prove that the T-C algorithm generates an optimal tree for a more general problem. Recall that $C(S)$ is the class of T-C level-by-level forests (trees) that can be built from an initial sequence S of terminal nodes. Now we introduce the notion of a generalized initial sequence. A *generalized initial sequence* is a sequence of internal and terminal nodes. We shall denote a generalized initial sequence by S^* and the class of T-C level-by-level forests built on S^* by $C(S^*)$. We introduce the notion of a generalized initial sequence so that a T-C level-by-level forest can be built from any construction sequence

(irrespective of how the construction sequence was obtained). If one scans the bottom of an alphabetic tree built on S^*, then the bottom nodes of the tree must occur, from left to right, in the order of the generalized initial sequence S^*. We call the nodes of a forest T in $C(S^*)$ which are in the (generalized) initial sequence *initial* nodes. Then the *noninitial* nodes of T are the internal nodes of T not in S^*. Theorem 4.1 is valid for generalized initial sequences, i.e., the normalized form T_N of a forest T in $C(S^*)$ is also in $C(S^*)$ and $|T_N| = |T|$. All the lemmas and theorems in § 5 are valid with generalized initial sequences. However, T_N need no longer be alphabetic for every T in $C(S^*)$ when S^* contains internal nodes. Take Fig. 7, for example. A generalized initial sequence $S^* = \boxed{3}\ \textcircled{4}\ \boxed{3}\ \textcircled{4}\ \boxed{3}$ is given and a T-C level-by-level tree built on S^* is shown. The path lengths of the five nodes in the generalized initial sequence are 3, 2, 3, 2, 2 respectively. But there is no alphabetic tree built on a generalized initial sequence of five nodes with path lengths 3, 2, 3, 2, 2. Since we are *only* to prove that T' is an optimal tree (built on S^*) in $C(S^*)$, we are not concerned whether every tree in $C(S^*)$ can be converted into an alphabetic tree.

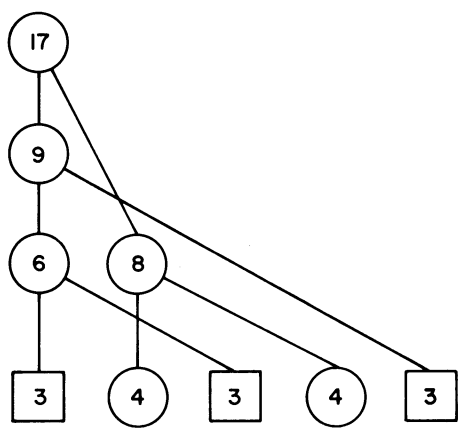

FIG. 7

The optimality of T' will ultimately follow from the same inductive argument that was used to show the optimality of Huffman's algorithm (Theorem 3.1)—namely, the minimum-weight T-C pair of S^* must be on the same level in any optimal m-sum forest in $C(S^*)$; from this fact, we obtain the following result.

THEOREM 6.1. *For any S^*, there is an optimal m-sum forest in $C(S^*)$ which combines the minimum-weight T-C pair in S^*.*

(The proof appears later in this section.)

Once we prove this theorem, we can prove the optimality of the T-C algorithm by applying this theorem to any initial sequence S^* and to the successive construction sequences (this is just like the proof of the Huffman algorithm). A formal proof appears later. The key to proving Theorem 6.1 is to show that an optimum m-sum forest of $C(S^*)$ must have the minimum-weight T-C pair of S^* at the same level.

To prove Theorem 6.1, we need the following operation. Suppose that v_a and v_b are nodes in the normalized forest T_N in $C(S^*)$ such that:

(i) for some integer k, v_a is at level k and v_b is at level $k+1$; and

(ii) node v_a and the *father of* v_b are *T-C on level* k, i.e., T-C in the construction sequence of T_N obtained after all combinations of nodes below level k in T_N are made.

For any such v_a and v_b, we define the *level interchange* of v_a and v_b to mean: (a) moving v_a and the subtree of which it is the root down one level in the forest; moving v_b and its dominated subtree up one level (put v_b just to the right of its previous father, v_{b+}. The node v_{b+} remains on level k with temporarily one son, v_b's brother); (b) letting v_a become a son of v_{b+} and letting v_b be a son of the previous father of v_a. (In other words, v_a and v_b interchange their fathers; see Fig. 8.) Then we renormalize the resulting forest from the lowest level up through level k (by renormalizing we mean the level-by-level reassignment of the father-son relationship used in obtaining the normalized form of a forest).

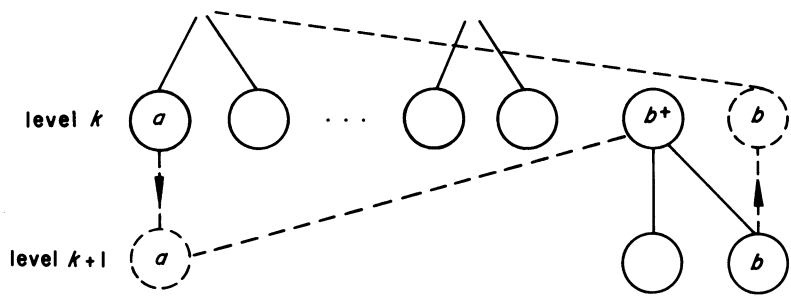

FIG. 8

LEMMA 6.1. *Given a normalized forest* T_N *in* $C(S^*)$, *let nodes* v_a *and* v_b *in* T_N *satisfy* (i) *and* (ii) *in the last paragraph. Further, assume that* v_a *is an initial node, and that if* v_b *is a terminal node, then* v_b*'s brother is on the same side of* v_b *as* v_a *is. Then the level interchange of* v_a *and* v_b *produces a normalized forest in* $C(S^*)$. *Furthermore, if* $w_a < w_b$ ($w_a \leqq w_b$), *then the resulting normalized forest costs less than* (*less than or equal to*) T_N.

Proof. After lowering v_a and raising v_b and its subtree, we make v_a the other son of v_{b+}, the previous father of v_b (see Fig. 8). Now we have a forest and renormalizing is well-defined. Since the final resulting tree, call it T_N^i, is normalized, it remains to show that T_N^i is indeed in $C(S^*)$. Because renormalizing involves the combination of consecutive pairs of nodes, it is sufficient for us to show that after lowering v_a and raising v_b and its subtree, there are an even number of nodes on level j to the left of any terminal node V_f on a higher level than j.

Case a. If $j < k$, level j is unaffected by the lowering and raising. Thus, as in T_N, there is an even number of nodes on level j to the left of any such V_f after the lowering and raising.

Case b. Suppose $j = k$. Of course, in T_N, there must be an even number of nodes on level k to the left of V_f. If there is an odd number of nodes on level k to the left of V_f after the lowering and raising, then clearly V_f must be between v_a and v_{b+} in T_N. But this would contradict the fact that v_a and v_{b+} are T-C on level k of T_N.

Case c. Suppose $j = k + 1$. If $V_f \neq V_b$, the argument in Case b applies. If $V_f = V_b$ (i.e., v_b is a terminal node), then suppose v_a is to the left of V_b and hence the brother of V_b, by assumption, is also to the left of V_b (a similar argument works if v_a is on the right of V_b). In T_N there are an even number of nodes on level $k + 1$ to the right of V_b and an odd number to the left of V_b, since V_b's brother is on its left. After the lowering and raising there are still an even number of nodes on level $k + 1$ to the right of V_b (which is now on level k), and, in addition, there are now an even number of nodes on level $k + 1$ to the left of V_b, since v_a was lowered to level $k + 1$ on the left of V_b.

Case d. Suppose $j > k + 1$. In T_N there are clearly an even number of nodes to the left of V_f on level j. Also in T_N, v_b dominates an even number (perhaps zero) of nodes to the left of V_f on level j. After raising v_b and its subtree one level, each node on level j dominated by v_b is replaced by two nodes or no nodes, and hence after the raising there are an even number of nodes to the left of V_f on level j.

The interchange increases the path length of v_a by one and decreases by one the path lengths of nodes with total weight w_b. If $w_a < w_b$, then $|T_N^i| < |T_N|$ (or $w_a \leqq w_b$ implies $|T_N^i| \leqq |T_N|$). This completes the proof.

The T-C algorithm uses the idea of always making the cheapest (lightest weight) T-C combination possible. An alternative approach might start by making some expensive (heavy) combinations in order to make possible some very cheap combinations later. The following lemma shows that this alternative cannot be optimal in $C(S^*)$.

LEMMA 6.2. *Let d be the minimum weight of any T-C pair in the (generalized) initial sequence* S^*. *If* T_N^0 *is a normalized optimal m-sum forest in* $C(S^*)$, *then, all noninitial nodes in* T_N^0 *have weight greater than or equal to d.*

Proof. Suppose there exists the noninitial node v_s in T_N^0 with $w_s < d$. If there is more than one possible choice of v_s, let the chosen v_s be at as low a level as possible. Let v_a and v_b be the left and right sons of v_s, respectively. It follows that v_a and v_b must both be initial nodes, i.e., in S^*. Since $w_a + w_b < d$, v_a and v_b are not T-C in S^*. Hence there exists in S^* a terminal node V_f between v_a and v_b. Nodes v_a and v_b must be brothers on some level, call it level k, of T_N^0 since they are combined in T_N^0. Then clearly V_f is on a lower level of T_N^0. Suppose V_f is on level $k + 1$. Then switch the positions of v_a and V_f in T_N^0. It is easy to check that the new forest, call it T_N^{00}, is a normalized forest in $C(S^*)$ (see Fig. 9). Now V_f and v_b

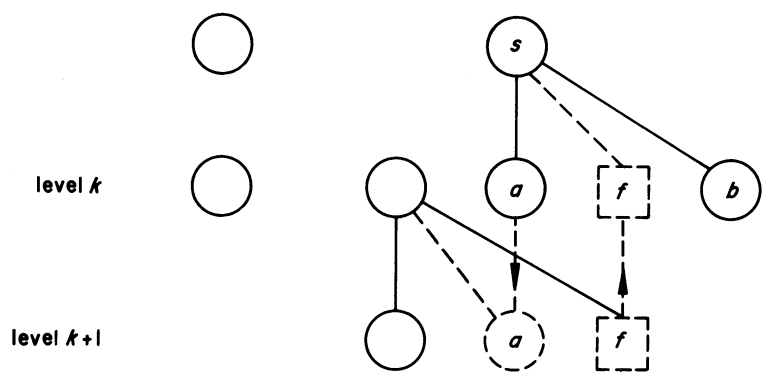

FIG. 9

are consecutive at level k of T_N^{00} since there were no nodes between v_a and v_b. Also we see that V_f and v_b are consecutive in S^* (because any initial node between V_f and v_b must be dominated at level k of T_N^0 by a node between v_a and v_b—but there were no nodes between v_a and v_b in T_N^0). Since V_f and v_b are consecutive in S^*, $w_f + w_b \geqq d$. Since $d > w_s = w_a + w_b$, we have $w_f + w_b \geqq d = w_a + w_b$ or $w_f > w_a$. This implies $|T_N^{00}| < |T_N^0|$, a contradiction.

Thus V_f must be below level $k + 1$ of T_N^0. Consequently there is a noninitial node v_e on level $k + 1$ of T_N^0 which dominates V_f, and the father of v_e is seen to be T-C with v_a on level k. We now perform a level interchange of v_a and v_e. The resulting tree T_N^{00} is in $C(S^*)$ by Lemma 6.1. Also by Lemma 6.1,

$$w_e \geqq d > w_a + w_b > w_a$$

implies $|T_N^{00}| < |T_N^0|$, a contradiction.

THEOREM 6.1. *For any S^*, there is an optimal m-sum forest in $C(S^*)$ which combines the minimum-weight T-C pair in S^*.*

Proof. Let T_N^0 be an optimal normalized m-sum forest in $C(S^*)$. Let v_a and v_b be a minimum-weight T-C pair in S^*. Either node may be terminal. We shall obtain another optimal m-sum forest in $C(S^*)$ which combines v_a and v_b together.

Case a. Suppose v_a and v_b are on the same level of T_N^0, say level $k > 0$. Then let us reassign the relationships between the fathers on level $k - 1$ and the sons on level k: first combine v_a and v_b (they are T-C on level k, since they are T-C in S^*), then do the regular reassignment with the rest of the sons—successively combine the two leftmost remaining nodes on level k (the father of v_a and v_b is fitted in the proper position among the other fathers on level $k - 1$). The resulting forest (which is seen to be in $C(S^*)$) satisfies the conclusion of this theorem. Suppose v_a and v_b are both on level 0, i.e., neither is used in T_N^0. Then take any noninitial node, v_f, on level 0 in T_N^0 and delete it. Then the nodes dominated by v_f are moved up one level and the forest is renormalized level-by-level (as in a level interchange). The result is an $(m - 1)$-sum normalized forest in $C(S^*)$ (this follows by the same argument as in Case d in the proof of Lemma 6.1). Now combine v_a and v_b. Since $w_f \geqq w_a + w_b$ by Lemma 6.2, the resulting m-sum forest is at least as cheap as T_N^0.

Case b. Now suppose v_a is on level i and v_b is on level j, where $i > j$ (a similar argument holds if $j > i$). Let v_d be the node on level j dominating v_a. Since v_a and v_b are T-C in S^*, then v_d and v_b must be T-C on level j of T_N^0. Let v_e be the son of v_d which dominates v_a, and let v_f be the son which does not (see Fig. 10). If v_e (or v_f) is a noninitial node, then perform a level interchange of v_b and v_e (or v_f). By Lemma 6.1, the resulting forest is in $C(S^*)$ and costs less than T_N^0, since by Lemma 6.2, w_e (or w_f) $\geqq w_a + w_b > w_b$. This is impossible (T_N^0 is optimal) and so v_e and v_f must be initial nodes (and hence $v_e = v_a$). Since $w_a + w_b \leqq w_a + w_f$ by Lemma 6.2, then $w_b \leqq w_f$. Note that if v_f is a terminal node, it cannot be between v_a and v_b in S^*. Then v_b and v_f satisfy the hypotheses of Lemma 6.1. So a level interchange of v_b and v_f produces a normalized forest which, by Lemma 6.1, is in $C(S^*)$ and costs no more than T_N^0. The new forest has v_a and v_b on the same level and is optimal. Now apply the argument in Case a to this new forest.

THEOREM 6.2. *For a given generalized initial sequence S^*, $T'(m)$ is an optimal m-sum forest in $C(S^*)$.*

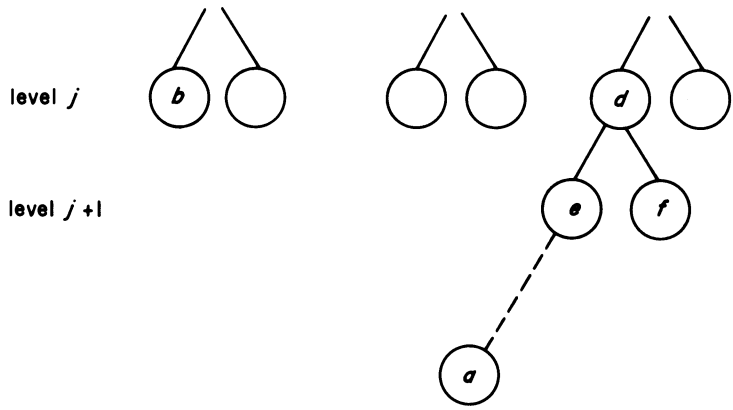

FIG. 10

Proof. The proof is by induction on m. If $m = 1$, the theorem is obvious. Assume for any generalized sequence S^* the theorem is valid for all $m < m_0$ $(1 < m_0)$. Let S^* be a (generalized) initial sequence with more than m_0 nodes (with m_0 or fewer nodes there is nothing to prove). By Theorem 6.1, there is an optimal m_0-sum forest in $C(S^*)$ which combines the minimum-weight T-C pair v_a and v_b in S^*. Let S_1^* be the resulting construction sequence after combining v_a and v_b. Clearly an optimal m_0-sum forest of $C(S^*)$ which combines v_a and v_b costs at least as much as the cost of a minimal-cost $(m_0 - 1)$-sum forest of $C(S_1^*)$ plus the cost (i.e., weight) of v_a and v_b. Note that $T'(m_0)$ built on S^* contains the $(m_0 - 1)$-sum forest built on S_1^* by the T-C algorithm. Moreover, this $(m_0 - 1)$-sum forest on S_1^* is optimal by induction. Thus $T'(m_0)$, which is in $C(S^*)$ by the generalized form of Theorem 5.1, is an optimal m-sum forest of $C(S^*)$.

COROLLARY 6.1. *For a given initial sequence S of terminal nodes, the normalized form of the m-sum forest (tree) generated by the T-C algorithm is an optimal alphabetic m-sum forest (tree).*

Proof. It follows from Theorems 4.1 and 5.1 that $T'(m)$ can be converted into an alphabetic forest. It follows from Theorem 6.2 that $T'(m)$ is optimal.

7. Implementation of the T-C algorithm. We define *Huffman's set* to be a maximal set of consecutive nodes in a construction sequence such that any nodes in the set are tentative-connecting. It is easy to see that a Huffman's set contains at most two terminal nodes. A typical Huffman's set consists of $V_i, v_{i+1}, \cdots, v_{j-1}, V_j$, i.e., two terminal nodes at the ends with several internal nodes between them. A special case is that the set of internal nodes in the middle is empty. Also, one or both of the terminal nodes may be absent. At any given stage of constructing $T'(m)$, for a given initial sequence we have k $(1 \leqq k \leqq n - 1)$ Huffman's sets which are ordered from the left to right.

An ALGOL program for the T-C algorithm has been written by Yohe [7]. The program needs $O(n^2)$ arithmetic operations for computing and $4n$ storage locations. Here we give additional comments to improve computing. In the beginning, there are n terminal nodes which are ordered from left to right. These n

terminal nodes can also be regarded as $n - 1$ Huffman's sets ordered from left to right. Let L, M and R be any three successive Huffman's sets from left to right and let W_L, W_M, W_R be the minimum weights of the tentative-connecting pairs in each of the three sets L, M, and R respectively. If $W_L > W_M \leqq W_R$, then the minimum weight tentative-connecting pair in M can be combined. (If L is the leftmost Huffman's set and $W_L \leqq W_M$, then the pair in L can be combined and similarly if R is the rightmost set and $W_M > W_R$. This follows from Lemma 5.1 and the fact that the T-C algorithm works from left to right in case of ties. In other words, if a tentative-connecting pair is not the pair with absolute minimum weight among all the Huffman's sets but is only of relative minimum weight (relative to the sets to its left and right), then it is all right to combine that pair. This means that we do not have to search the entire list to find the pair with absolute minimum weight. In general, let us assume that there are m Huffman's sets each with k_i nodes ($i = 1, \cdots, m$); then

$$\sum_{i=1}^{m} k_i \leqq 2(n - 1).$$

Since it takes $2k_i$ operations to find the tentative-connecting pair with minimum weight in the ith Huffman's set, it takes at most $2\sum k_i$ or $4(n - 1)$ operations to find the pair with absolute minimum weight. Since there are $n - 1$ combinations in T', we need at most $(n - 1)4(n - 2) = 4n^2$ operations to get the tree T'. After the tree T' is obtained it takes another $2n$ operations to get the path lengths of all terminal nodes.

After the path length of every terminal node is obtained, we can get the alphabetic tree T^* as follows. Assume the path lengths of terminal nodes are ordered from left to right successively. We always combine the two adjacent nodes of maximum path length, say q, and replace the two nodes by a node with path length $q - 1$. This procedure is repeated until a node (the root) of length zero is obtained.

8. Related problems. We have discussed the problem of optimal binary tree as a coding problem. There are many other problems in operations research and information retrieval which can also be formulated as optimum binary tree type problems. A very common problem is to identify unknown objects by a sequence of tests. For example, a coke machine has to identify the coin that is being put into the machine. Let us assume that the unknown object must belong to one of n kinds, in the case of a coin, it must be a penny, a nickel, a dime, or a quarter, say. Furthermore, the jth object, $j = 1, \cdots, n$, has a probability w_j of occurring, and these probabilities are assumed to be known in advance. There are tests T_i, $i = 1, \cdots, m$, available for identifying the objects. Each test has the effect of partitioning the n objects into two complementary sets and asserting the unknown object to be in one of the two sets. If it costs c_i dollars to perform the ith test T_i, what is the optimum sequence of performing the tests such that the expected cost is a minimum?

If all c_i are the same, and all 2^n tests are available (i.e., all partitionings of n objects), then the problem becomes a problem of constructing Huffman's code.

If all objects are different in length and the tests are to find if the length of the unknown object is greater or less than a fixed length, so that there are $n - 1$ tests available, then this is the problem of alphabetical code which we have considered in this paper.

We have mentioned two cases; in one case all the tests are available, and in the other case, the tests are specially restricted. If we consider objects as points in space and tests as hyperplanes, then there are many other problems besides the two cases that need to be solved.

Some related problems are as follows. If the incidence relationships between tests and objects are given as a zero–one matrix, are the given m tests sufficient to identify the objects? If yes, what is the degree of redundancy of identifying the jth object? What would be a nonredundant subset of tests which will give the minimum expected costs? What happens if a test has the effect of partitioning the n objects into three or more subsets? What happens if several tests can be given simultaneously?

Acknowledgment. The authors wish to thank Dr. Mike Yohe for stimulating discussions.

Note added in proof. Professor D. E. Knuth informed us that a better implementation of our algorithm needs only $O(n \log n)$ operations when suitable data structures are employed.

REFERENCES

[1] E. N. GILBERT AND E. F. MOORE, *Variable-length binary encodings*, Bell System Tech. J., 38 (1959), pp. 933–968.

[2] D. A. HUFFMAN, *A method for the construction of minimum-redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.

[3] D. E. KNUTH, *The Art of Computer Programming*, vol. 1, Addison-Wesley, New York, 1968.

[4] ———, *Optimum binary search trees*, Computer Science Dept. Rep. C. S. 149, Stanford University, Stanford, Calif., 1970.

[5] E. S. SCHWARTZ, *An optimum encoding with minimum longest code and total number of digits*, Information and Control, 7 (1964), pp. 37–44.

[6] E. S. SCHWARTZ AND B. KALLICK, *Generating a canonical prefix encoding*, Comm. ACM, 7 (1964), pp. 166–169.

[7] M. YOHE, *An Algol procedure for the Hu-Tucker minimum redundancy alphabetic coding method*, submitted to Comm. ACM.