For example, Fig. 12 shows the tree which results when the most common 31 words of English are entered in decreasing order of frequency. The relative frequency is shown with each word [cf. *Cryptanalysis* by H. F. Gaines (New York: Dover, 1956), p. 226]. The average number of comparisons for a successful search in this tree is 4.042; the corresponding binary search, using Algorithm 6.2.1B or C, would require 4.393 comparisons.

Optimum binary search trees. These considerations make it natural to ask about the best possible tree for searching a table of keys with given frequencies. For example, the optimum tree for the 31 most common English words is shown in Fig. 13; it requires only 3.437 comparisons for an average successful search.

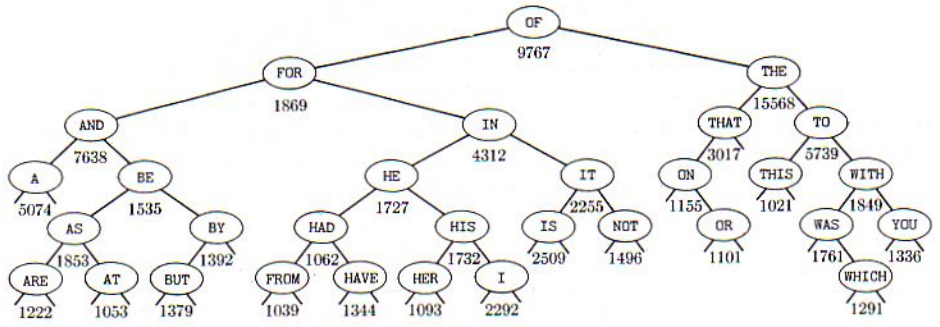
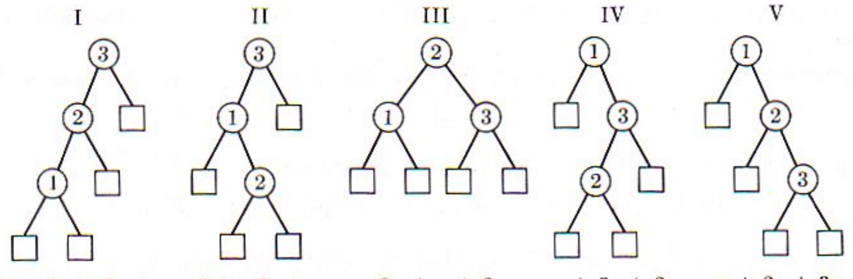


Fig. 13. Optimum search tree for the 31 most common English words.

Let us now explore the problem of finding the optimum tree. When $N = 3$, for example, let us assume that the keys $K_1 < K_2 < K_3$ have respective probabilities p, q, r. There are five possible trees:



(13)

Cost: $3p + 2q + r$ $\quad 2p + 3q + r$ $\quad 2p + q + 2r$ $\quad p + 3q + 2r$ $\quad p + 2q + 3r$

Figure 14 shows the ranges of p, q, r for which each tree is optimum; the balanced tree is best about 45 percent of the time, if we choose p, q, r at random (see exercise 21).

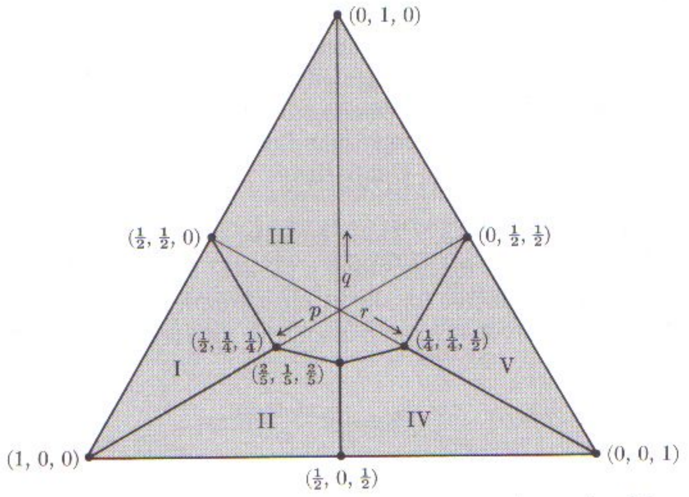


Fig. 14. If the relative frequencies of (K_1, K_2, K_3) are (p, q, r), this graph shows which of the five trees in (13) is best. The fact that $p + q + r = 1$ makes the graph two-dimensional although there are three coordinates.

Unfortunately, when N is large there are

$$\binom{2N}{N}\Big/(N+1) \approx 4^N/(\sqrt{\pi}\,N^{3/2})$$

binary trees, so we can't just try them all and see which is best. Let us therefore study the properties of optimum binary search trees more closely, in order to discover a better way to find them.

So far we have considered only the probabilities for a successful search; in practice, the unsuccessful case must usually be considered as well. For example, the 31 words in Fig. 13 account for only about 36 percent of typical English text; the other 64 percent will certainly influence the structure of the optimum search tree.

Therefore let us set the problem up in the following way: We are given $2n + 1$ probabilities $p_1, p_2, \ldots, p_n$ and $q_0, q_1, \ldots, q_n$, where

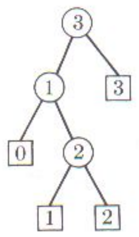
p_i = probability that K_i is the search argument;

q_i = probability that the search argument lies between K_i and K_{i+1}.

(By convention, q_0 is the probability that the search argument is less than K_1, and q_n is the probability that the search argument is greater than K_n.) Thus, $p_1 + p_2 + \cdots + p_n + q_0 + q_1 + \cdots + q_n = 1$, and we want to find a binary tree which minimizes the expected number of comparisons in the search, namely

$$\sum_{1\le j\le n} p_j(\mathrm{level}(\textcircled{j}) + 1) + \sum_{0\le k\le n} q_k\,\mathrm{level}(\boxed{k}), \qquad (14)$$

where $\textcircled{j}$ is the jth internal node in symmetric order and $\boxed{k}$ is the $(k+1)$st external node, and where the root has level zero. Thus the expected number of comparisons for the binary tree


(15)

is $2q_0 + 2p_1 + 3q_1 + 3p_2 + 3q_2 + p_3 + q_3$. Let us call this the *cost* of the tree; and let us say that a minimum-cost tree is *optimum*. In this definition there is no need to require that the p's and q's sum to unity, we can ask for a minimum-cost tree with any given sequence of "weights" $(p_1, \ldots, p_n; q_0, \ldots, q_n)$.

We have studied Huffman's procedure for constructing trees with minimum weighted path length, in Section 2.3.4.5; but that method requires all the p's to be zero, and the tree it produces will usually not have the external node weights $(q_0, \ldots, q_n)$ in the proper symmetric order from left to right. Therefore we need another approach.

The principle which saves us is that *all subtrees of an optimum tree are optimum.* For example if (15) is an optimum tree for the weights $(p_1, p_2, p_3; q_0, q_1, q_2, q_3)$, then the left subtree of the root must be optimum for $(p_1, p_2; q_0, q_1, q_2)$; any improvement to a subtree leads to an improvement in the whole tree.

This principle suggests a computation procedure which systematically finds larger and larger optimum subtrees. We have used much the same idea in Section 5.4.9 to construct optimum merge patterns; the general approach is known as "dynamic programming," and we shall consider it further in Chapter 7.

Let $c(i, j)$ be the cost of an optimum subtree with weights $(p_{i+1}, \ldots, p_j; q_i, \ldots, q_j)$; and let $w(i, j) = p_{i+1} + \cdots + p_j + q_i + \cdots + q_j$ be the sum of all those weights; $c(i, j)$ and $w(i, j)$ are defined for $0 \le i \le j \le n$. It follows that

$I < K \leq J$

$$\begin{aligned} c(i, i) &= 0, \\ c(i, j) &= w(i, j) + \min_{i \le k < j} \big(c(i, k-1) + c(k, j)\big), \qquad \text{for} \quad i < j, \end{aligned} \tag{16}$$

since the minimum possible cost of a tree with root $\textcircled{k}$ is $w(i, j) + c(i, k-1) + c(k, j)$. When $i < j$, let $R(i, j)$ be the set of all k for which the minimum is achieved in (16); this set specifies the possible roots of the optimum trees.

Equation (16) makes it possible to evaluate $c(i, j)$ for $j - i = 1, 2, 3, \ldots, n$; there are about $\frac{1}{2}n^2$ such values, and the minimization operation is carried out for about $\frac{1}{6}n^3$ values of k. This means we can determine an optimum tree in $O(n^3)$ units of time, using $O(n^2)$ cells of memory.

TSNS

A factor of n can actually be removed from the running time if we make use of a "monotonicity" property. Let $r(i, j)$ denote an element of $R(i, j)$; we need not compute the entire set $R(i, j)$, a single representative is sufficient. Once we have found $r(i, j-1)$ and $r(i+1, j)$, the result of exercise 27 proves that we may always assume that

$$r(i, j-1) \leq r(i, j) \leq r(i+1, j) \tag{17}$$

when the weights are nonnegative. This limits the search for the minimum, since only $r(i+1, j) - r(i, j-1) + 1$ values of k need to be examined in (16) instead of $j - i$. The total amount of work when $j - i = d$ is now bounded by the telescoping series

$$\sum_{\substack{d \leq j \leq n \\ i=j-d}} \bigl(r(i+1, j) - r(i, j-1) + 1\bigr) = r(n-d+1, n) - r(0, d-1) + n - d + 1 < 2n,$$

hence the total running time is reduced to $O(n^2)$.

The following algorithm describes this procedure in detail.

Algorithm K (*Find optimum binary search trees*). Given $2n + 1$ nonnegative weights $(p_1, \ldots, p_n; q_0, \ldots, q_n)$, this algorithm constructs binary trees $t(i, j)$ which have minimum cost for the weights $(p_{i+1}, \ldots, p_j; q_i, \ldots, q_j)$ in the sense defined above. Three arrays are computed, namely

$c[i, j]$,	for	$0 \leq i \leq j \leq n$,	the cost of $t(i, j)$;
$r[i, j]$,	for	$0 \leq i \leq j \leq n$,	the root of $t(i, j)$;
$w[i, j]$,	for	$0 \leq i \leq j \leq n$,	the total weight of $t(i, j)$.

The results of the algorithm are specified by the r array: If $i = j$, $t(i, j)$ is null; else its left subtree is $\underline{t(i, r[i, j] - 1)}$ and its right subtree is $\underline{t(r[i, j], j)}$.

K1. [Initialize.] For $0 \leq i \leq n$, set $c[i, i] \leftarrow 0$ and $w[i, i] \leftarrow q_i$ and $w[i, j] \leftarrow w[i, j-1] + p_j + q_j$ for $j = i+1, \ldots, n$. Then for $1 \leq j \leq n$ set $c[j-1, j] \leftarrow w[j-1, j]$ and $r[j-1, j] \leftarrow j$. (This determines all the 1-node optimum trees.)

K2. [Loop on d.] Do step K3 for $d = 2, 3, \ldots, n$, then terminate the algorithm.

K3. [Loop on j.] (We have already determined the optimum trees of less than d nodes. This step determines all the d-node optimum trees.) Do step K4 for $j = d, d+1, \ldots, n$.

K4. [Find $c[i, j]$, $r[i, j]$.] Set $i \leftarrow j - d$. Then set

$$c[i, j] \leftarrow w[i, j] + \min_{r[i,j-1] \leq k \leq r[i+1,j]} (c[i, k-1] + c[k, j]),$$

and set $r[i, j]$ to a value of k for which the minimum occurs. (Exercise 22 proves that $r[i, j-1] \leq r[i+1, j]$.) ▮

As an example of Algorithm K, consider Fig. 15, which is based on a "key-word-in-context" (KWIC) indexing application. The titles of all articles in the first ten volumes of the ACM *Journal* were sorted to prepare a concordance in which there is one line for every word of every title. However, certain words like "THE" and "EQUATION" were felt to be sufficiently uninformative that they were left out of the index. These special words and their frequency of occurrence are shown in the internal nodes of Fig. 15. Note that a title such as "On the solution of an equation for a certain new problem" would be so uninformative, it wouldn't appear in the index at all! The idea of KWIC indexing is due to H. P. Luhn, *Amer. Documentation* **11** (1960), 288–295. (See W. W. Youden, *JACM* **10** (1963), 583–646, where the full KWIC index appears.)

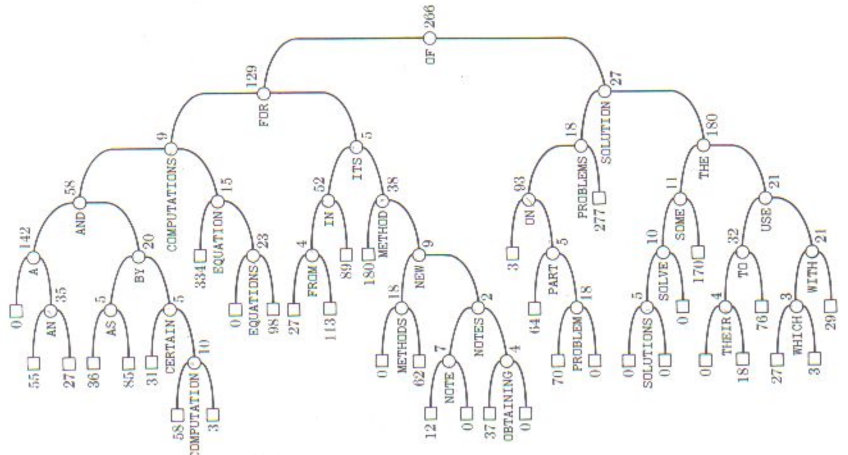
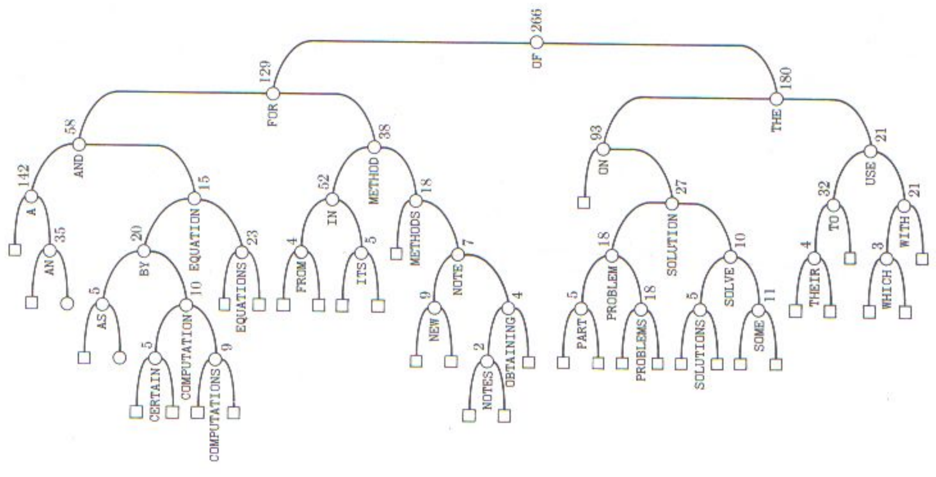


Fig. 15. An optimum binary search tree for a KWIC indexing application.

When preparing a KWIC index file for sorting, we might want to use a binary search tree in order to test whether or not each particular word is to be indexed. The other words fall between two of the unindexed words, with the frequencies shown in the external nodes of Fig. 15; thus, exactly 277 words which are alphabetically between "PROBLEMS" and "SOLUTION" appeared in the *JACM* titles during 1954–1963.

Figure 15 shows the optimum tree obtained by Algorithm K, with $n = 35$. The computed values of $r[0, j]$ for $j = 1, 2, \ldots, 35$ are (1, 1, 2, 3, 3, 3, 3, 8, 8, 8, 8, 8, 8, 11, 11, $\ldots$, 11, 21, 21, 21, 21, 21, 21); the values of $r[i, 35]$ for $i = 0$, 1, $\ldots$, 34 are (21, 21, $\ldots$, 21, 25, 25, 25, 25, 25, 25, 26, 26, 26, 30, 30, 30, 30, 30, 30, 30, 33, 33, 33, 35, 35).

The "betweenness frequencies" q_j have a noticeable effect on the optimum tree structure; Fig. 16(a) shows the optimum tree that would have been obtained with the q_j set to zero. Similarly, the internal frequencies p_i are important: Fig. 16(b) shows the optimum tree when the p_i are set to zero. Considering



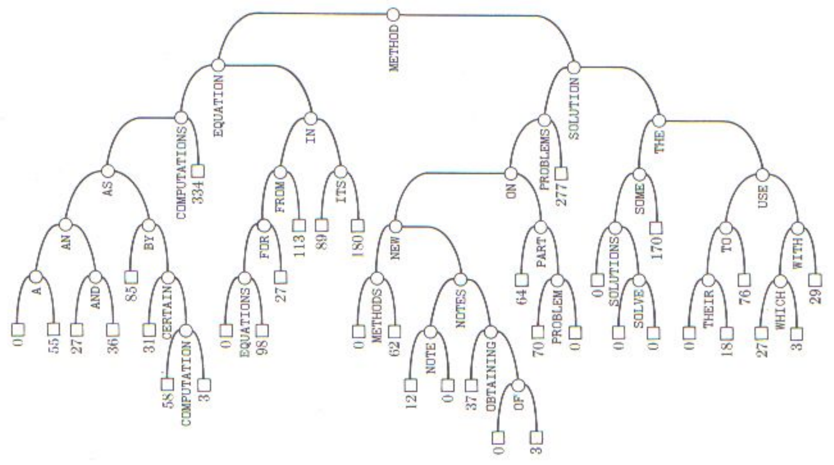


Fig. 16. Optimum binary search trees based on half of the data of Fig. 15; (a) external frequencies suppressed, (b) internal frequencies suppressed.

the full set of frequencies, the tree of Fig. 15 requires only 4.75 comparisons, on the average, while the trees of Fig. 16 require, respectively, 5.29 and 5.32. (A straight binary search would have been better than the trees of Fig. 16, in this example.)

Since Algorithm K requires time and space proportional to n^2, it becomes impractical to use it when n is very large. Of course we may not really want to use binary search trees for large n, in view of the other search techniques to be discussed later in this chapter; but let's assume anyway that we want to find an optimum or nearly optimum tree when n is large.

We have seen that the idea of inserting the keys in order of decreasing frequency can tend to make a fairly good tree, on the average; but it can also be very bad (see exercise 20), and it is not usually very near the optimum, since it makes no use of the q_j weights. Another approach is to choose the root k so that the weights $w(0, k-1)$ and $w(k, n)$ of the resulting subtrees are as near to being equal as possible. This approach also fails, because it may choose a node with very small p_k to be the root.

A fairly satisfactory procedure can be obtained by combining these two methods, as suggested by W. A. Walker and C. C. Gotlieb [*Graph Theory and Computing* (Academic Press, 1972)]: Try to equalize the left-hand and right-hand weights, but be prepared to move the root a few steps to the left or right to find a node with relatively large p_k. Figure 17 shows why this method is reasonable: If we plot $c(0, k-1) + c(k, n)$ as a function of k, for the KWIC data of Fig. 15, we see that the result is quite sensitive to the magnitude of p_k.

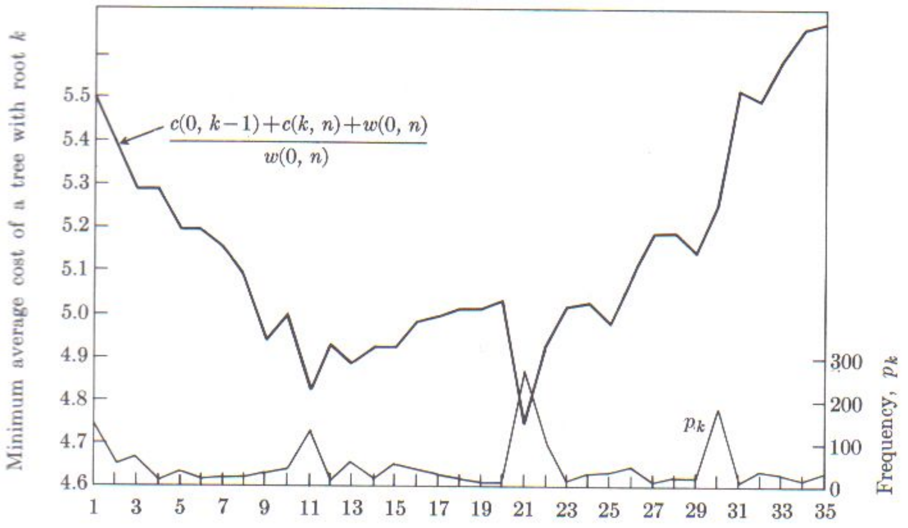


Fig. 17. Behavior of the cost as a function of the root, k.

A "top-down" method such as this can be used for large n to choose the root and then to work on the left and the right subtrees. When we get down to a sufficiently small subtree we can apply Algorithm K. The resulting method yields fairly good trees (reportedly within 2 or 3 percent of the optimum), and it requires only $O(n)$ units of space, $O(n \log n)$ units of time.

***The Hu-Tucker algorithm.** In the special case that all the p's are zero, T. C. Hu and A. C. Tucker have discovered a remarkable "bottom-up" way to construct optimum trees; if appropriate data structures are used, their method requires $O(n)$ units of space and $O(n \log n)$ units of time, and it constructs a tree which is *really* optimum (not just approximately so).

The Hu-Tucker algorithm can be described as follows.

*

n log n

• PHASE 1, Combination. Start with the "working sequence" of weights written inside of *external* nodes,

$$\boxed{q_0}\quad\boxed{q_1}\quad\boxed{q_2}\quad\cdots\quad\boxed{q_n}\,. \tag{18}$$

Then repeatedly combine two weights q_i and q_j for $i < j$ into a single weight $q_i + q_j$, deleting the node containing q_j from the working sequence and replacing the node containing q_i by the *internal* node

$$\enclose{circle}{q_i + q_j}\,. \tag{19}$$

This combination is to be done on the unique pair of weights (q_i, q_j) satisfying the following rules:

i) No external nodes occur between q_i and q_j. (This is the most important rule which distinguishes the algorithm from Huffman's method.)

ii) The sum $q_i + q_j$ is minimum over all (q_i, q_j) satisfying rule (i).

iii) The index i is minimum over all (q_i, q_j) satisfying rules (i), (ii).

iv) The index j is minimum over all (q_i, q_j) satisfying rules (i), (ii), (iii).

• PHASE 2, Level assignment. When Phase 1 ends, there is a single node left in the working sequence. Mark it with level number 0. Then undo the steps of Phase 1 in reverse order, marking level numbers of the corresponding tree; if (19) has level l, the nodes containing q_i and q_j which formed it are marked with level $l + 1$.

• PHASE 3, Recombination. Now we have the working sequence of external nodes and levels

$$\begin{array}{ccccc} \boxed{q_0} & \boxed{q_1} & \boxed{q_2} & \cdots & \boxed{q_n}\,. \\ l_1 & l_2 & l_3 & & l_n \end{array}$$

The internal nodes used in Phases 1 and 2 are now discarded, we shall create new ones by combining weights (q_i, q_j) according to the following new rules:

i′) The nodes containing q_i and q_j must be adjacent in the working sequence.

ii′) The levels l_i and l_j must both be the maximum among all remaining levels.

iii′) The index i must be minimum over all (q_i, q_j) satisfying (i′), (ii′).

The new node (19) is assigned level $l_i - 1$. The binary tree formed during this phase has minimum weighted path length over all binary trees whose external nodes are weighted $q_0, q_1, \ldots, q_n$ from left to right.

Figure 18 shows an example of this algorithm; the weights q_i are the relative frequencies of the letters ␣, A, B, . . . , Z in English text. During Phase 1, the

[illegible]

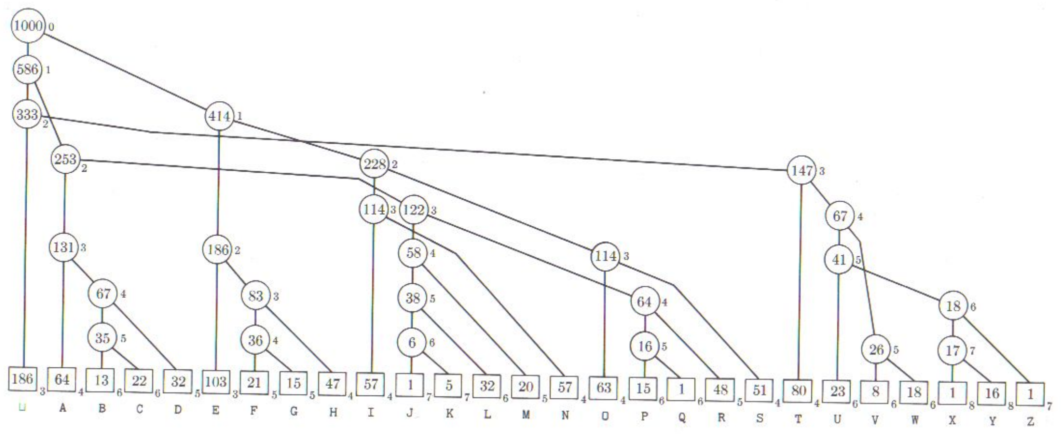


Fig. 18. The Hu-Tucker algorithm applied to alphabetic frequency data: Phases 1 and 2.

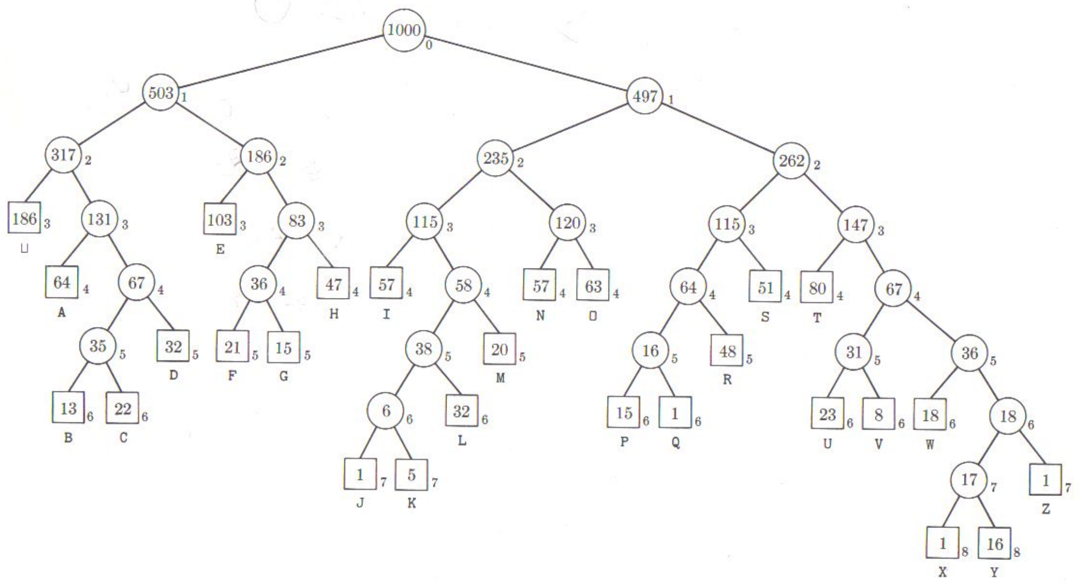


Fig. 19. The Hu-Tucker algorithm applied to alphabetic frequency data: Phase 3.

first node formed is $\textcircled{6}$, combining the J and K frequencies; then the node $\textcircled{16}$ is formed (combining P and Q), then

$$\textcircled{17},\ \textcircled{18},\ \textcircled{26},\ \textcircled{35},\ \textcircled{36},\ \textcircled{38},\ \textcircled{41},\ \textcircled{58},\ \textcircled{64},\ \textcircled{67},\ \textcircled{67},\ \textcircled{83};$$

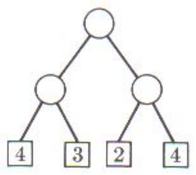
at this point we have the working sequence

$$\boxed{186}\ \boxed{64}\ \textcircled{67}\ \boxed{103}\ \textcircled{83}\ \boxed{57}\ \textcircled{58}\ \boxed{57}\ \boxed{63}\ \textcircled{64}\ \boxed{51}\ \boxed{80}\ \textcircled{67}\,. \qquad (20)$$

Rule (i) allows us to combine nonadjacent weights only if they are separated by internal nodes; so we can combine $57 + 57$, then $63 + 51$, then $58 + 64$, etc.

The level numbers assigned during Phase 2 appear at the right of each node in Fig. 18. The recombination during Phase 3 now yields the tree shown in Fig. 19; note that things must be associated differently in this tree than in Fig. 18, because Fig. 18 does not preserve the left-to-right ordering. But Fig. 19 has the same cost as Fig. 18, since the external nodes appear at the same levels in both trees.

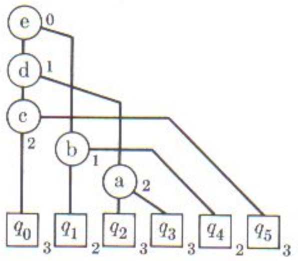
Consider a simple example where the weights are 4, 3, 2, 4; the unique optimum tree is easily shown to be



(21)

This example shows that the two smallest weights, 2 and 3, should *not* always be combined in an optimum tree, even when they are adjacent; some recombination phase is needed.

It is beyond the scope of this book to give a proof that the Hu-Tucker algorithm is valid; no simple proof is known, and it is quite possible that no simple proof will ever be found! In order to illustrate the inherent complexities of the situation, note that Phase 3 must combine all nodes into a single tree, and this is not obviously possible. For example, suppose that Phases 1 and 2 were to construct the tree



(22)

by combining nodes $\textcircled{a}$, $\textcircled{b}$, $\textcircled{c}$, $\textcircled{d}$, $\textcircled{e}$ in this order; this accords with rule (i). Then Phase 3 will get stuck after forming

$$\begin{array}{cccccccccc} \boxed{q_0} & & \boxed{q_1} & & (q_2+q_3) & & \boxed{q_4} & & \boxed{q_5} & \\ & 3 & & 2 & & 2 & & 2 & & 3 \end{array}$$

because the two level-3 nodes are not adjacent! Rule (i) does not by itself guarantee that Phase 3 will be able to proceed, and it is necessary to prove that configurations like (22) will *never* be constructed during Phase 1.

When implementing the Hu-Tucker algorithm, we can maintain priority queues for the sets of node weights which are not separated by external nodes. For example, (20) could be represented by priority queues containing, respectively,

$$\begin{array}{cccccccc} 64 & 64 & 57 & 57 & 57 & 51 & 51 & 67 \\ 186 & 67 & 83 & 57 & 63 & 63 & 80 & 80 \\ & 103 & 103 & 58 & & 64 & & \end{array} \qquad (23)$$

plus information about which of these is external, and an indication of left-to-right order for breaking ties by rules (iii) and (iv). Another "master" priority queue can keep track of the sums of the two least elements in the other queues. The creation of the new node $57 + 57$ causes three of the above priority queues to be merged. When priority queues are represented as leftist trees (cf. Section 5.2.3), each combination step of Phase 1 requires at most $O(\log n)$ operations; thus $O(n \log n)$ operations suffice as $n \to \infty$. Of course for small n it will be more efficient to use a comparatively straightforward $O(n^2)$ method of implementation.

The optimum binary tree in Fig. 19 has an interesting application to coding theory as well as to searching: Using 0 to stand for a left branch in the tree and 1 to stand for a right branch, we obtain the following variable-length codewords:

␣	000	I	1000	R	11001
A	0010	J	1001000	S	1101
B	001100	K	1001001	T	1110
C	001101	L	100101	U	111100
D	00111	M	10011	V	111101
E	010	N	1010	W	111110
F	01100	O	1011	X	11111100
G	01101	P	110000	Y	11111101
H	0111	Q	110001	Z	1111111

(24)

Thus a message like "`RIGHT ON`" would be encoded by the string

$$1100110000110101111110000101111010.$$

Note that decoding from left to right is easy, in spite of the variable length of the codewords, because the tree structure tells us when one codeword ends and another begins. This method of coding preserves the alphabetical order of messages, and it uses an average of about 4.2 bits per letter. Thus the code could be used to compress data files, without destroying lexicographic order of alphabetic information. (The figure of 4.2 bits per letter is minimum over all binary tree codes, although it could be reduced to 4.1 bits per letter if we disregarded the alphabetic ordering constraint. A further reduction, preserving alphabetic order, could be achieved if pairs of letters instead of single letters were encoded.)

An interesting asymptotic bound on the minimum weighted path length of search trees has been derived by E. N. Gilbert and E. F. Moore:

Theorem G. *If $p_1 = p_2 = \cdots = p_n = 0$, the weighted path length of an optimum binary search tree lies between*

$$\sum_{0\le i\le n} q_i \log_2 (Q/q_i) \qquad \textit{and} \qquad 2Q + \sum_{0\le i\le n} q_i \log_2 (Q/q_i),$$

where $Q = \sum_{0\le i\le n} q_i$.

Proof. To get the lower bound, we use induction on n. If $n > 0$ the weighted external path length is at least

$$Q + \sum_{0\le i<k} q_i \log_2 (Q_1/q_i) + \sum_{k\le i\le n} q_i \log_2 ((Q - Q_1)/q_i) \\ \ge \sum_{0\le i\le n} q_i \log_2 (Q/q_i) + f(Q_1),$$

for some k, where

$$Q_1 = \sum_{0\le i<k} q_i,$$

and

$$f(Q_1) = Q + Q_1 \log_2 Q_1 + (Q - Q_1) \log_2 (Q - Q_1) - Q \log_2 Q.$$

The function $f(Q_1)$ is nonnegative, and it takes its minimum value 0 when $Q_1 = \frac{1}{2}Q$.

To get the upper bound, we may assume that $Q = 1$. Let $e_0, \ldots, e_n$ be integers such that $2^{-e_i} \le q_i < 2^{1-e_i}$, for $0 \le i \le n$. Construct codewords C_i of 0's and 1's, by using the most significant $e_i + 1$ binary digits of the fraction $\sum_{0\le k<i} q_k + \frac{1}{2}q_i$, expressed in binary notation. Exercise 35 proves that C_i is never an initial substring of C_j when $i \ne j$; it follows that we can construct a binary search tree corresponding to these codewords. For example when the q's are the letter frequencies of Fig. 19, this construction gives $C_0 = 0001$,