WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

# OPTIMIZING BINARY TREES GROWN WITH A SORTING ALGORITHM

W. A. MARTIN -- D. N. NESS

421-69

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

# OPTIMIZING BINARY TREES GROWN WITH A SORTING ALGORITHM

W. A. MARTIN -- D. N. NESS

421-69

HD28
.M414
no. 421-69

[illegible]

# OPTIMIZING BINARY TREES GROWN WITH A SORTING ALGORITHM

W. A. MARTIN -- D. N. NESS

## Tree Growing Processes

In this paper we consider a process which grows and uses a labeled binary tree structure. Each node in this structure has an item of information, an upward pointer, and a downward pointer. The upward and downward pointers may point to null elements.

At any node in the strcuture all items of information in the sub tree extending from that which are larger* than the item of information at that node will be in the subtree pointed to by the upward pointer. Similarly, all smaller items are in the sub-tree pointed to by the downward pointer. In this paper we will not consider contexts where multiple nodes have the same item value.

Such trees are easy to grow. The first element is placed at the root. Thereafter each new element is placed in the tree by comparing it with the root and moving up or down depending on whether the new element is larger or smaller. This process is repeated at each node until an attempt is made to move to a null node. The item is then placed at this point in the tree. (We call this algorithm "A" below.)

*Obviously the measure can be any kind of computation.

As an example, consider adding the item 40 to the tree:

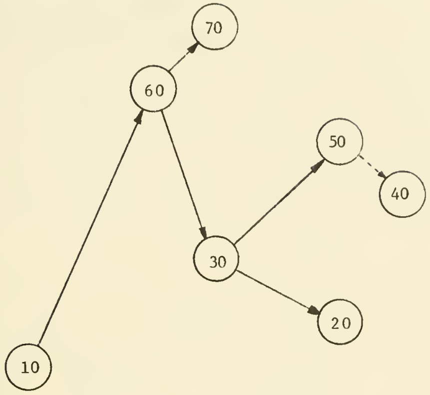


Figure 1

The following steps will be performed:

1) $40 > 10 \rightarrow$ move up

2) $40 < 60 \rightarrow$ move down

3) $40 > 30 \rightarrow$ move up

4) $40 < 50 \rightarrow$ move down

Since there is no item down from 50, 40 is attached at this point. This algorithm can be used for building symbol tables and it is closely related to the sorting algorithm, QUICKSORT (3,4).

Let us now consider some mathematical properties of the tree structures that are grown by this algorithm.

## Mathematical Characteristics

The shape of a tree containing a given set of n items depends on the order in which the items are encountered by algorithm A. For example, the tree in Figure 1 was formed by considering the items in the order 10, 60, 30, 70, 20, 50, 40. By considering the same items in the order 40, 20, 60, 10, 30, 50, 70, the algorithm forms the tree in Figure 2.

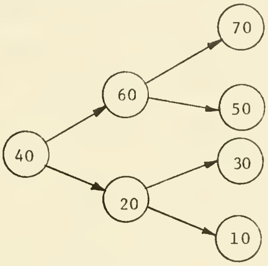


Figure 2

Algorithm A thus generates a tree for each of the n! possible arrangements of n items, but not all of these trees are distinct as can be seen from Figure 3.

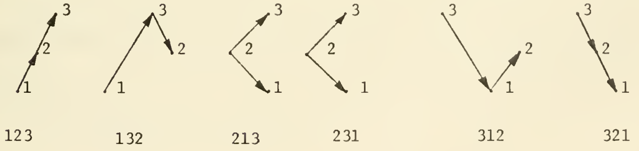


Figure 3

In the analysis to follow we will consider each of the n! permutations of the n items to be equally probable. Thus, some trees will be generated more often than others. We will also mention the results which obtain for the case where each distinct tree is taken as equally probable.

We can search for an item in a tree by following exactly the same steps used to insert the item while the tree is being constructed. It seems reasonable to assume that the time required is proportional to the number of nodes visited. (For example, see Figure 6) It is easy to see that one can find each of the items in the tree in Figure 2 by visiting a total of 17 (= 1*1 + 2*2 + 4*3) nodes, while one must visit 22 (= 1*1 + 1*2 + 2*3 + 2*4 + 1*5) nodes to find the same items in the tree in Figure 1. Clearly, the tree in Figure 2 is not only better, but optimum.* We have devised an algorithm (Algorithm B), presented below, which will convert any tree generated by Algorithm A into an optimum tree containing the same items. Algorithm B requires a time proportional to the number of items (see Figure 7). It is natural to ask whether the time thus saved in searching a reorganized tree is greater than the time required for the conversion from the non-optimal into the optimal form.

To this end, let us calculate the number $\overline{V}(n)$ of nodes visited per item, in finding each item in each of the n! trees containing n nodes. Let V(n) be the number of nodes visited in finding each item in each of

*See Reference (5), page 402.

the n! trees with n nodes. The root node of each tree will be accessed n times; the total number of root node accesses summed over all trees is thus $n \cdot n!$. Now, for $1 \leq k \leq n$ exactly $n!/n$ trees have item k at the root node. If item k is at the root node there is a tree of k-1 items below the root node and a tree of n-k items above the root node. In the (n-1)! trees with item k at the root node, each of the (k-1)! trees containing k-1 items occurs (n-1)!/(k-1)! times below the root node. Similarly, each of the (n-k)! trees containing n-k items occurs (n-1)!/(n-k)! times above the root node.

Therefore, we can write $V(n) = u(n, n!)$ where

$$u(n,m) = n \cdot m + 2\sum_{k=0}^{n-1} u(k,m/n)$$

u(n,m) represents the number of node visits summed over all trees of n nodes, where the forest (collection of trees) consists of m trees. The formula can be obtained by observing the root node is visited n times for each tree in the forest, or a total of $n \cdot m$ node visits. For each value of the root node (1/n of the trees in the forest have the same root nodes) the upper (and lower) trees are a forest of m/n trees, and consist of k (k = 0, . . ., n-1) nodes. Thus the rest of the upper (or lower) tree requires $\sum_{k=0}^{n-1} u(k,m/n)$ visits.

Since $\overline{V}(n) = \dfrac{u(n,n!)}{n!}$

we want to find $\overline{u}(n,m) = \dfrac{u(n,m)}{m} = n + \dfrac{2}{n}\sum_{k=0}^{n-1} \dfrac{u(k,m/n)}{m/n}$ . 2)*

*Note: $\overline{u}(0,m) = 0$

So we can see by induction that $\overline{u}$ is not a function of m and we can

write $$\overline{V}(n) = n + \frac{2}{n} \sum_{k=0}^{n-1} \overline{V}(k). \qquad 3)$$

From this we find
$$\overline{V}(n) = \frac{n+1}{n}\,\overline{V}(n-1) + 2 - \frac{1}{n} \qquad 4)$$

and
$$\overline{V}(n) = -n + 2(n+1) \sum_{i=2}^{n+1} \frac{1}{i} \qquad 5)^*$$

The sum in 5) can be bounded by the log function to give the bounds.

$$\overline{V}(n) < -n + 2(n+1)\log n \qquad 6)$$

$$\overline{V}(n) > -n + 2(n+1)\log \frac{n+1}{2}$$

Thus, for large n we have $\overline{V}(n) \simeq 2n\log n$. 8)

This result has been obtained by Hibbard (3).

Now, for comparison, we derive an expression for the number r(n) of node visits for an optimum tree of n items. If $2^j-1 \leq n \leq 2^{j+1}-1$ we have

$$r(n) = j2^j - 2^j + 1 + (j+1)(n - 2^j + 1) \qquad 9)$$

In order to compare the average trees with the optimum, the function $\dfrac{\overline{V}(n) - r(n)}{r(n)}$ is plotted in Figure 5. For the plotted values the optimum tree gives an improvement of 10 to 30%. To examine this improvement

*Note: $\overline{V}(1) = 0$

$$\sum_{i=0}^{j} x^i = \frac{x^{j+1}-1}{x-1}$$

$$\sum_{i=1}^{j} i x^i = \frac{(j+1)x^j}{x-1} - \frac{x^{j+1}-1}{(x-1)^2}$$

$$(j+1)2^j - 2\cdot 2^j + 1$$

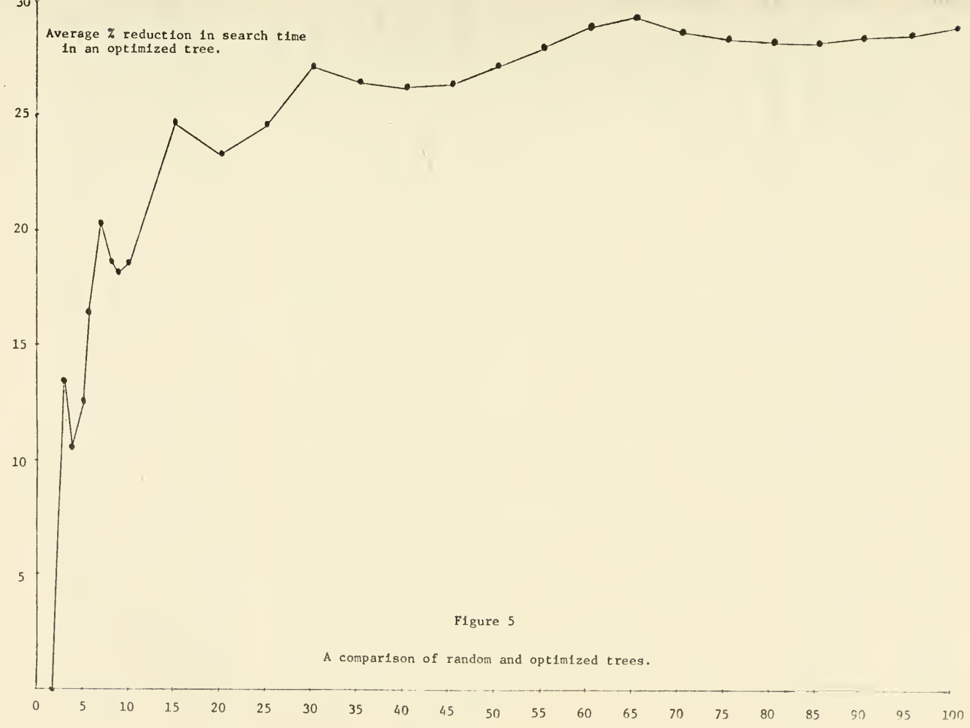


Figure 5

A comparison of random and optimized trees.

further note that for large n and for $n=2^j-1$, r(n) approaches $\frac{n \log(n)}{\log 2}$ .

The improvement is then ≃ .386 nlog n (.386 = 2 log 2 - 1).

Since the time it takes our Algorithm B to form an optimum tree is proportional to n, there is some n beyond which application of the algo- rithm must result (on the average) in a saving.

Referring to Knuth (5) we may also note that if we take all distinct labeled trees to be equally probable we obtain the result $\overline{V}(n) \simeq n\sqrt{n}$ for large n. This would make application of our algorithm even more profitable on the average. It suggests that those distinct labeled trees which are generated more than once by Algorithm A are often the "good" ones.

The above calculation suggests exactly how inefficient we can expect the "average" tree to be. It must be recognized, however, that the actual tree grown in any given case may depart even more substantially from the optimum. If the data were to be incorporated in ascending sequence, for example, each item would be placed "up" from all previous items, and our process would reduce to linear search with a time to access all items proportional to $n^2$.

It is obviously easy to extend the algorithm presented above to allow us to maintain information about the efficiency of the structure as it is being grown. If we start a counter at 1 and then increment it each time a comparison is performed in the process of placing that element in the tree, we obtain a count of the number of comparisons necessary to

find this element. If we accumulate this number as we bring in each new item we will always have the number of node visits to access all items available.

This information may suggest, then, that at some point it would be worth restructuring a tree as it is being grown or after it has been grown but before some searches are to be performed. We now present our algorithm for performing this restructuring.

## The Restructuring Algorithm (Algorithm B)

The algorithm is presented here in English; it is also presented in the Appendix in FORTRAN.

The procedure IBEST returns as its answer a pointer to the root node of the restructured tree. This procedure also establishes the environment for the other subroutines: IGROW and NEXT. IBEST computes IGROW(n), where n is the number of nodes in the tree to be restructured. It returns the result of this computation (which is the restructured tree) as its answer.

The procedure IGROW(n) is responsible for constructing an optimum tree containing n nodes. It must be recursive as it may call itself. It uses the procedure NEXT, which returns a pointer to the smallest node in the old tree the first time it is called and a pointer to the smallest node not previously returned on each successive call. IGROW(n) can take three courses of action:

1) If $n = 0$ , return a pointer to a null node.

2) If $n = 1$ , call NEXT and return its result.

3) If n > 1

    a) Call IGROW ($\lfloor(n-1)/2\rfloor$)

    b) Call NEXT

    c) Call IGROW ($\lceil(n-1)/2\rceil$)

    Then alter the node pointed to by the result of b) by replacing its up pointer with the result of a) and its down pointer with the result of c). Return the result of b), thus altered, as the answer.

The procedure NEXT moves through the original tree structure returning the nodes in ascending sequence. Given a tree, it returns the nodes in the lower branch by calling itself recursively with this branch as argument, then it returns the root node, and then the nodes in the upper branch. It is convenient to think of NEXT and IGROW as coroutines. It should be noted that IGROW does not need to modify the pointers of any node until NEXT is completely through with it. Thus the nodes of the original tree can be patched directly and no additional memory is required.

## Incremental Restructuring

If one wishes to form an optimum tree each time a new node is added, then it is not necessary to use a global restructuring method like Algorith B. A method which concentrates on those nodes visited during the addition of the new node can be used. We are not aware of such a method which is highly efficient. However, a good method is known for maintaining a tree structure which may not be optimum, but is very good (1,2).

This tree structure is characterized by the constraint that the maximum path length of the subtree above a given node cannot differ by more than one from the maximum path length of the subtree below that node. This constraint excludes the really bad trees formed by algorithm A and so the average number of nodes visited per item searched must be less than 2logn, but slightly more than for an optimum binary tree. This has been verified numerically by Foster (2). Such a tree is shown in Figure 4.

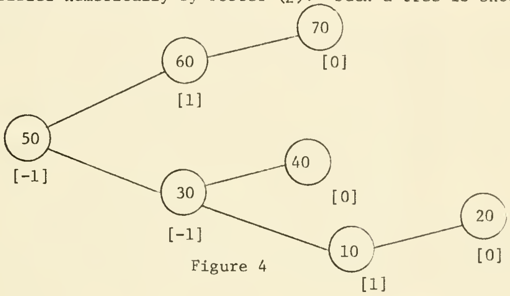


Figure 4

In order to make the method of adding a new node efficient, each node must have associated with it a number indicating the amount by which the maximum path length of the upper subtree exceeds the maximum path length of the lower subtree. In Figure 4 these numbers are shown in brackets next to each node. A new node is added to this structure in two steps. First, the node is added to the tree using algorithm A. However, a pointer to each node visited should be placed on a push-down-list so that the path can later be retraced to restructure the tree. (Note that

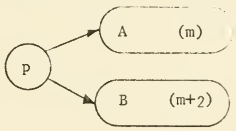
the tree cannot be altered to meet the path length constraint until the new node is in place, because the maximum path length of a subtree containing the new node depends on exactly where the new node is added.) The tree is then restructured by applying algorithm C. To apply this algorithm one traces back along the path followed in adding the new node. At each node along this path one of the following conditions will hold:

a. Either the upper or lower subtree was previously longer by one and now they are the same length. In this case no restructuring is needed and it is not necessary to move back toward the root node any farther since the length of the subtree extending from this node is unchanged.

b. Both subtrees were previously the same length and now one of them is longer by one. No restructuring is done but the path must be retraced farther, since the length of the subtree extending from this node has increased by one.

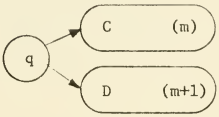
c. A subtree which was previously longer by one is now longer by two. The tree must be restructured at this node but the path need not be retraced any farther since the restructuring will return the subtree to its original length.

In order to explain the restructuring algorithm let us assume that the lower subtree is longer, so that the tree has the form:



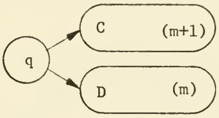
A and B are subtrees of length m and m+2 respectively. To restructure we save a pointer to node p and examine subtree B. It will either have the form:

I)



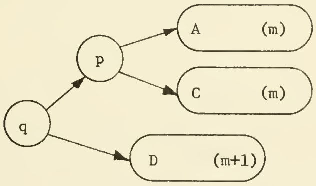
or the form:

II)



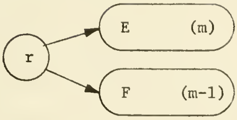
since it was of length m+1 before the new node was added, and the new node increased its length without unbalancing it so that it needed to be restructured.

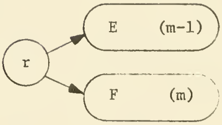
In case I) the restructuring is completed by forming the tree:



In case II) we examine subtree C. If m ≠ 0 it will have one of the forms:

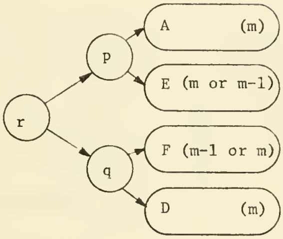
III)



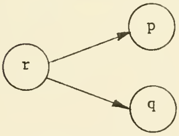


IV) For m = 0 subtree C will just be a single node.

In case III) the restructuring is completed by forming the tree:



Finally, in case IV) the final tree has the form:



One can now verify that each of these restructuring operations returns the subtree to its length before the new node was added. Furthermore, it is easy to see that the effort required to restructure once a node with unbalanced subtrees has been located is independent of the size of the entire tree.

No one has given an exact analysis of the effort required to add a new node to the tree and then retrace the path until a node in either condition a) or c) is found. Foster (2) argues that for large trees the average effort to retrace and the fractional number of times each case of restructuring is applied are independent of the size of the tree. If this is the case then the total average effort to apply algorithm C while building a large tree is proportioned to n, the number of nodes, just as it is for algorithm B.

## A Comparison of the Global and Incremental Restructuring Algorithms

We can then propose two strategies for building a tree of n items.

a. Apply algorithm A n times and then apply algorithm B once.

b. Apply algorithm A n times and apply algorithm C each time algorithm A is used.

Strategy a) will require an effort $A_1 n\log n + Bn$, for some constants $A_1$ and B. Strategy b) will require, under the assumptions of Foster, effort $A_2 n\log n + Cn$. Strategy b) keeps the tree organized at each step and this would tend to make $A_2$ less than $A_1$. On the other hand, strategy b) requires saving pointers to the nodes visited on a push-down-list and this would tend to make $A_2$ greater than $A_1$. Strategy a) always leaves us with an optimum tree while strategy b) leaves us with a tree which is slightly less than optimum. In addition, strategy b) requires slightly more storage.

## Experimental Results

Programs for strategy a) and strategy b) were coded in FORTRAN for the IBM 1130. Algorithm C is longer and more difficult to program than algorithm B. Figure 6 shows the average compution time/node visited as a function of the size of the tree being searched by algorithm A. Trees of a given size were generated at random; the same results also held for worst trees.

Figure 7 shows the average computation time/item for application of algorithm B to random trees of different sizes.

Figure 8 shows the average time/item to apply strategy a) and strategy b) as a function of the size of the tree. The solid line is fitted to the points for strategy a) and the broken line is fitted to the points for strategy b). It can be seen that strategy b) is slightly faster for trees of less than 1000 items.

## Conclusions

Trees grown without any reorganization are quite good on the average. Reorganization is a second order improvement, and so a decision to use algorithm B or algorithm C depends on how the information is to be accessed.

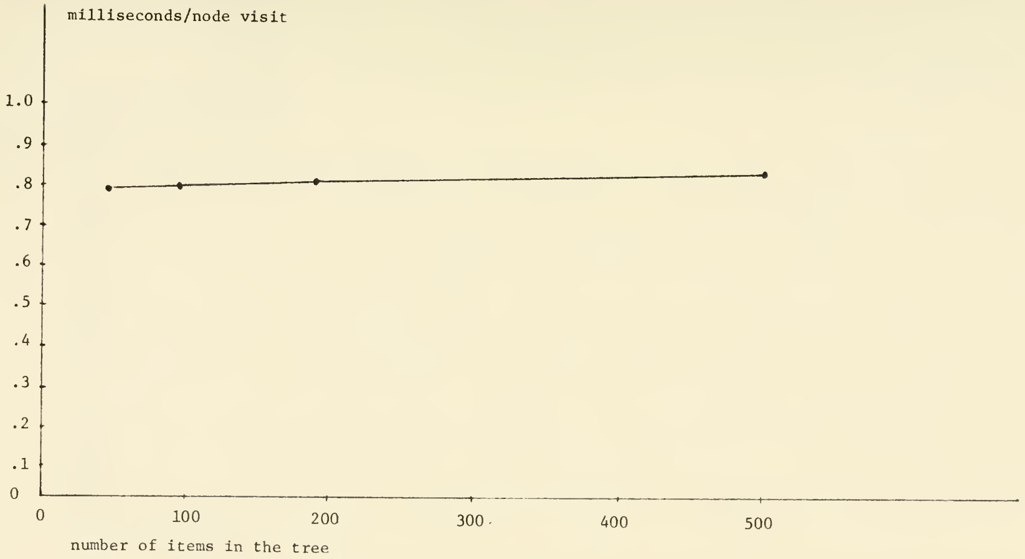


Figure 6

The time required for algorithm A to visit one node for random trees of different sizes. (The shape of the tree does not change the time within the accuracy of this graph.)

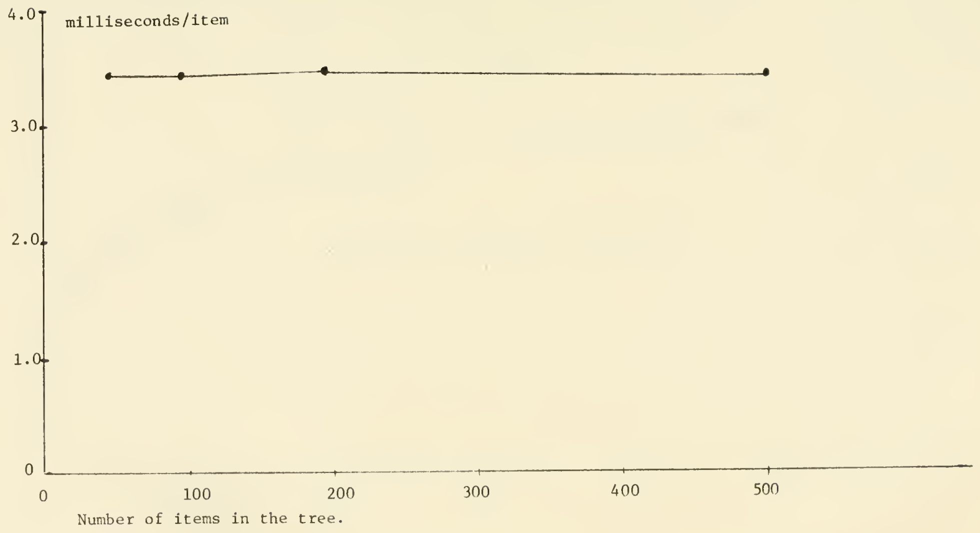


Figure 7

The time required to apply algorithm B.
(The shape of the tree does not change
the time within the accuracy of this graph.)

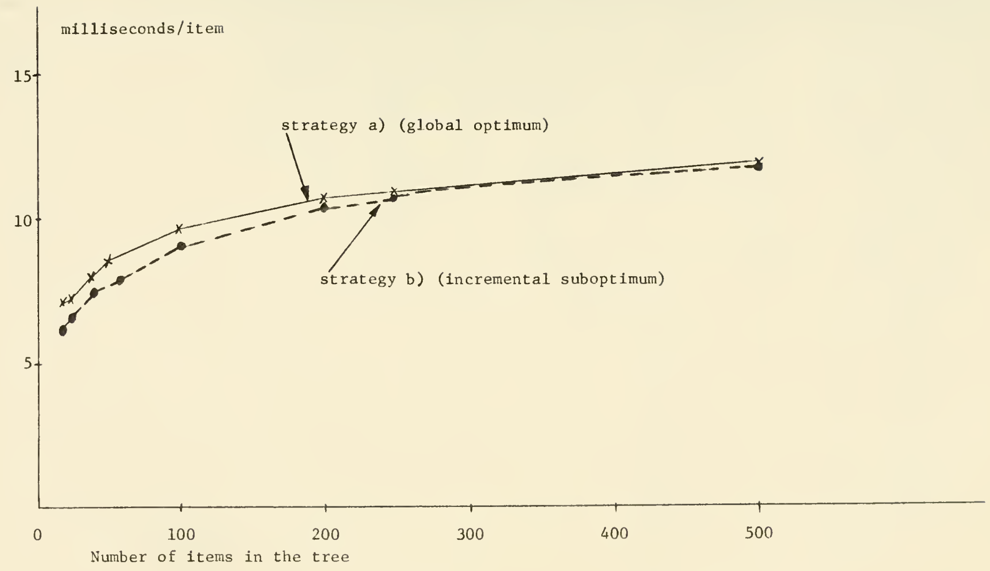


Figure 8

The time/item to grow and optimize a tree.

```
C       ***********************************************************************
C       ********** ALGORITHM A                                          *******
C       ***********************************************************************
C       DATA ITEMS IN 'ITEMS'
C       UPWARD POINTERS IN 'IUP'
C       DOWNWARD POINTERS IN 'IDOWN'
C       PUSH-DOWN LIST FOR NEXT IS NPOL, INDEX IS NPOLI
C       PUSH-DOWN LIST FOR GROW IS MPOL, INDEX IS MPOLI
C       CURRENT NUMBER OF ITEMS IN 'IN'
C       BEGINNING OF TREE IN 'IBEG'
C       IUP, IDOWN, ITEMS, NPOL AND MPOL MUST BE DIMENSIONED
C       INITIALIZATION
        IN=0
        IBEG=1
C       ...
C       PUT AWAY THE ITEM IN 'ITEM'
        IN=IN+1
        ITEMS(IN)=ITEM
        IUP(IN)=0
        IDOWN(IN)=0
C       IF IT IS FIRST ITEM IN LIST, RETURN (VIA STATEMENT 30)
        IF(IN-1)100,30,100
C       NOT FIRST ITEM IN LIST, PLACE IT
 100    ICUR=IBEG
 110    IF(ITEMS(ICUR)-ITEM) 120,120,150
C       GOES IN UPWARD TREE
 120    IF(IUP(ICUR)) 140,130,140
C       PLACE IF UPWARD TREE EMPTY
 130    IUP(ICUR)=IN
        GO TO 30
C       UP ANOTHER BRANCH
 140    ICUR=IUP(ICUR)
        GO TO 110
C       GOES IN DOWNWARD TREE
 150    IF(IDOWN(ICUR)) 170,160,170
C       PLACE IN DOWNWARD TREE
 160    IDOWN(ICUR)=IN
        GO TO 30
C       DOWN ANOTHER BRANCH
 170    ICUR=IDOWN(ICUR)
        GO TO 110
C       RETURN TO PROGRAM
 30     ...
```

```
C     ****************************************************************
C     ********** ALGORITHM B                                  ********
C     ****************************************************************
C     INITIALIZATION
      NPDLI=0
      MPDLI=0
      NPDLU=IBEG
C     CALL IGROW(IN)
      IG=IN
      IGR=1
      GO TO 800
 240  IBEG=IANS
C     'IBEG' NOW POINTS TO BEGINNING OF RESTRUCTURED TREE
C     RETURN TO MAIN COMPUTATION
      ...
C     ****************************************************************
C     ********** SUBROUTINE IGROW                             ********
C     ****************************************************************
C     ARGUMENT IN 'IG'
C     RETURN VIA 'IGN'
C     ANSWER TO 'IANS'
 800  IANS=0
      IF(IG-1) 830,810,840
C     IGROW(1), RETURN INEXT
 810  INR=1
      GO TO 900
 820  IUP(IANS)=0
      IDOWN(IANS)=0
C     GENERAL EXIT
 830  GO TO (240,850,870),IGR
C     IGROW(IG), IG 1, PUSH DOWN AND CALL
 840  IGN=(IG-1)/2
C     SAVE RETURN AND NUMBER TO CALL WITH NEXT TIME
      MPDLI=MPDLI+2
      MPDL(MPDLI-1)=IGR
      MPDL(MPDLI)=IG-1-IGN
C     CALL IGROW(IG) RECURSIVELY
      IG=IGN
      IGR=2
      GO TO 800
C     RETURN FROM RECURSIVE CALL, CALL NEXT
 850  IG=MPDL(MPDLI)
      MPDL(MPDLI)=IANS
```

```
      INR=2
      GO TO 900
C     RETURN FROM NEXT, CALL IGRCW RECURSIVELY AGAIN
 860  MPCLI=MPCLI+1
      MPDL(MPCLI)=IANS
      IGR=3
      GC TC 8CC
C     RESTRUCTURE AS IS APPRCPRIATE
 870  IG=MPCL(MPCLI)
      IUP(IG)=IANS
      IDOWN(IG)=MPCL(MPDLI-1)
      IANS=IG
C     POP THE STACK ANC RETURN
      IGR=MPDL(MPDLI-2)
      MPDLI=MPDLI-3
      GO TO 830
C     ****************************************************************
C     ********** SUBROUTINE NEXT                                ********
C     ****************************************************************
C     NPCLU CONTAINS NCDE TO BEGIN DOWNWARD PATH, UNLESS ZERO
 900  IF(NPDLU) 910,92C,91C
C     PUSH CURRENT NCDE AND CHAIN DOWNWARDS
 910  NPDLI=NPDLI+1
      NPDL(NPDLI)=NPDLU
      NPCLU=IDOWN(NPDLU)
      GO TO 9CC
C     NC FURTHER CHAIN, RETURN A VALUE AND POP STACK
 920  IANS=NPDL(NPDLI)
      NPCLI=NPDLI-1
      NPCLU=IUP(IANS)
C     EXIT
      GO TO (820,860),INR
      END
```

## REFERENCES

1. Adel'son-Vel'skiy, G. M., and Landis, Ye. M., "An Algorithm for the Organization of Information," Deklady Akademii Nauk USSR, Moscow, p. 263-266; Vol. 16, No. 2, 1962; also available in translation as U.S. Dept. of Commerce OTS, JPRS 17,137, Washington, D.C., and as NASA Document N63-11777.

2. Foster, C. C., "Information Storage and Retrieval Using AVL Trees," Proc. ACM 20th National Conference, (1965), p. 192-205.

3. Hibbard, T. N., "Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting," JACM 6 (May 1963), 206-213.

4. Hoare, C. A. R., "Algorithms 63, Partition, and 64, Quicksort," CACM 4 (July 1961), p. 321.

5. Knuth, D. E., The Art of Computer Programming, Vol. 1, Addison-Wesley, (1968), Section 2.3.4.5.

[illegible]