MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 173 March 1969

A HEURISTIC PROGRAM THAT CONSTRUCTS DECISION TREES

Patrick Winston

INTRODUCTION

The Problem

Suppose there is a set of objects, $\{A,B,\ldots,Z\}$, and a set of tests, $\{T1,T2,\ldots,TN\}$. When a test is applied to an object, the result is either T or F. Assume the tests may vary in cost and the objects may vary in probability of occurence. One then hopes that an unknown object may be identified by applying a sequence of tests. The appropriate test at any point in the sequence in general should depend on the results of previous tests.

The problem is to construct a good test scheme using the test costs, the probabilities of occurence, and a table of test outcomes.

To put this in concrete terms, suppose an induction center wishes to deploy recruits into jobs appropriate to their qualifications. Further, suppose all men are known to fit one of five jobs: general, rifleman, typist, pilot, or spy. Possible tests might be:

I.Q.: $>100 \rightarrow$ T

vision: normal $\rightarrow$ T

dexterity: good $\rightarrow$ T

The cost-effectiveness oriented sargent constructs a table of qualifications such as shown in figure 1 (in general I call such tables, tables of outcomes). He then makes a list jobs

	general	rifleman	typist	pilot	spy
symbol	G	R	T	P	S
I. Q.	T	F	F	T	T
vision	F	F	F	T	T
dexterity	F	F	T	T	F

test name outcomes

Figure 1. A table of outcomes.

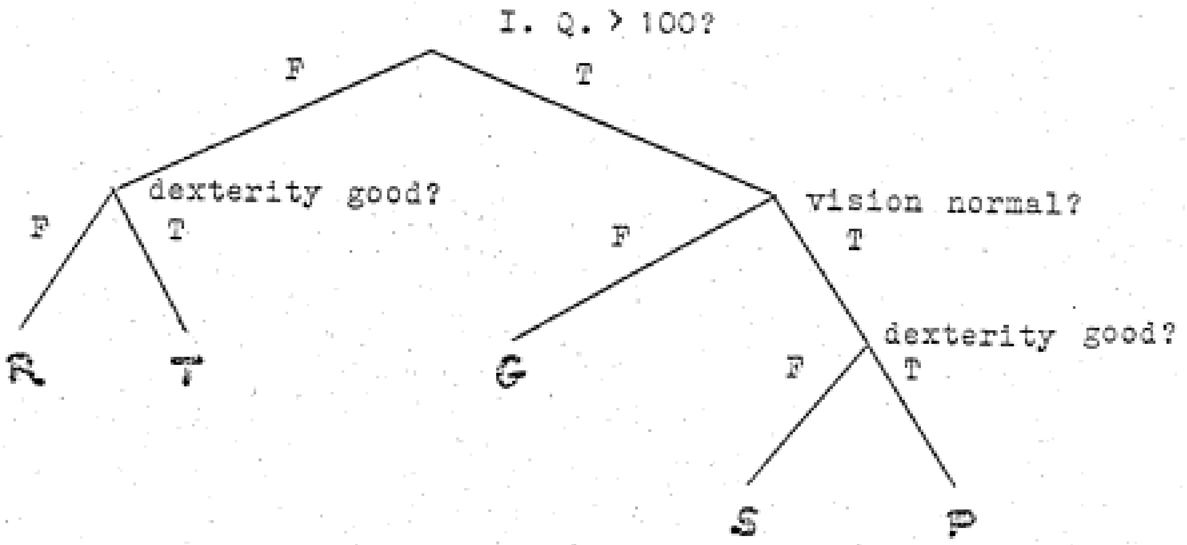


Figure 2. A possible test tree.

of test costs and a list of probabilities of job types. Finally he attempts to invent a good test tree. Figure 2 pictures one tree he might select.

A more serious possibility would involve construction of a diagnosis tree for a computerized doctor. Other applications of good tree building methods are not obvious. I hope to employ some of the ideas that have evolved in a program that searches semantic memories of the sort described by Quillian.(1)

Exhaustive Enumeration

One's first reaction might be to try all possible trees and select the best one. This, however, is impractical for even moderately large problems since the number of possible trees grows rapidly with the number of objects and tests:

Let the number of available tests be N. To simplify the calculation, we will consider only those trees of the form shown in figure 3a. In these trees any path descends down through d

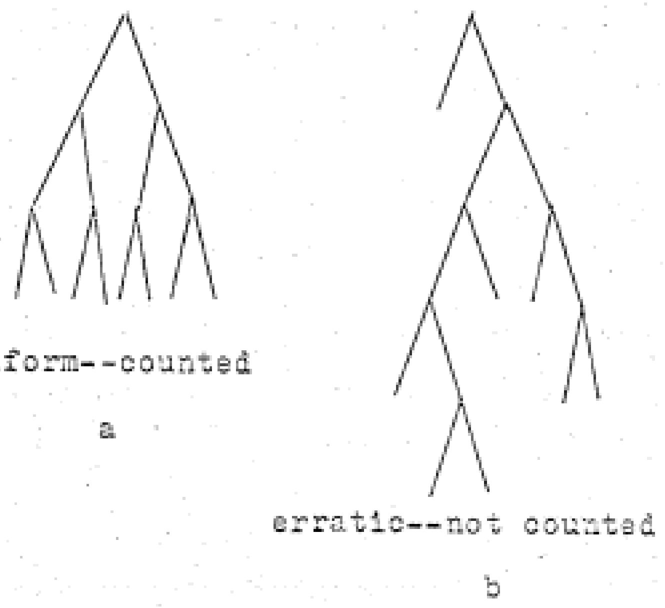


Figure 3. Tree forms.

tests, where that depth d is just sufficient to insure at least as many terminal branches as objects. Our number will be a lower bound, since trees that branch erratically as that of figure 3b will not be counted.

The number of possibilities for the first test will be N. The number of possibilities for the second test will be N-1, but we now have two tests to be fixed. For each of the N-1 ways the first of the two is fixed, there are N-1 ways to fix the second. Hence the total number of variations so far is $N\cdot(N-1)^2$. The number of tests now left unused at any tree tip is N-2 and there are 2^2 choices to be made. The number of variations is therefore increased by a third level of tests to $N\cdot(N-1)^{2^1}\cdot(N-2)^{2^2}$. Extending to d levels we have $N\cdot(N-1)^{2^1}\ldots(N-[d-1])^{2^{d-1}}$. This number rapidly becomes uncomfortable. For example, a sample problem worked in the <u>RESULTS</u> section involves only 5 tests and 8 objects. d is then 3 yielding:

$$\text{number of possible trees} > 5 \cdot 4^2 \cdot 3^4 = 6{,}480 \ .$$

In a second sample problem, there are 7 tests and 16 objects. In this case, d is 4 and the lower bound is:

$$\text{number of possible trees} > 7 \cdot 6^2 \cdot 5^4 \cdot 4^8 \simeq 1.3\times 10^{10}$$

Previous Work

In 1964 Slagle(2) wrote a paper on this problem, but he was concerned only with the special case in which objects belong to one of two categories, and the goal is only to decide to which category an unknown object belongs. Other restrictions on the nature of the table of outcomes further limit his discussion.*

A later paper by Pollack(3) offered an elementary heuristic approach, but he oversimplified the problem by assuming all tests to be of equal cost and all objects to be of equal probability.

His idea is simply to insert at each point that test which most nearly divides the object set into two groups of equal probability.

Finally, Reinwald and Soland(4) present an interesting method that at first glance seems to lead directly to an optimal solution of these test tree problems. On closer examination, however, it may be seen that their scheme is only slightly better than exhaustive enumeration for the class of problems of interest here. This is shown in the appendix.

A Heuristic Approach

Since no entirely satisfactory solution could be found in the literature, I developed a heuristic LISP program that seems to do a reasonable job on a number of test cases. The program, as described in the next two sections, exhibits several concepts found in intelligent programs: there is a simple form of planning; there is a focusing of attention; and there are a number of local heuristic operations. The length of the program was about 12 pages.

* See (4) for details.

THE HEURISTICS

Focusing Attention

The basic idea of the program is to repeatedly expand and improve a partially formed tree of tests. A function named FOREMAN is in charge of local operations and tries to apply heuristic transformations at that part of the tree to which it is directed by a higher level function named EXECUTIVE. The heuristic resources of FOREMAN are described in the following paragraphs. It should be noted that only the essentials of the functions are indicated, with no attention to the trouble the actual programs face in handling normal variants of the situations described as well as the many unusual cases.

Budding

Suppose the EXECUTIVE directs FOREMAN to work at the spot indicated in figure 4. Since a number of objects have yet to

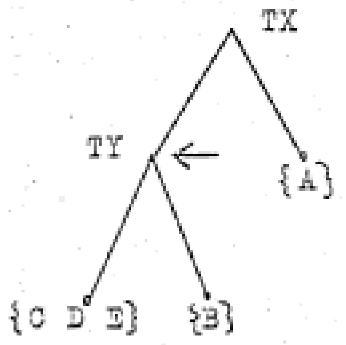


Figure 4. The use of BUD.

be separated, it is desirable to pick suitable tests from a list of those not previously used and graft them onto the tree. This is handled by the BUD function. Now it is clearly desirable that BUD select a low cost test. On the other hand, if all costs were the same, it would be a good idea to select the test that best divides the objects into equi-probable groups. If working out a few examples is not convincing enough, further justification for this method of selection may be found in Gallager$^{(5)}$; essentially the same problem is discussed in the context of minimizing the average length of the codewords in a communications system.

But since the test costs do vary, a compromise is necessary. The test actually used is the one which maximizes the following function of cost and splitting:

$$\text{Quality (test)} = \frac{(P_R \cdot P_L)^2}{T}$$

Where P_R, P_L are the total probabilities of objects sent down the right and left branches
and T is the test cost.

If the function evaluates to the same number for two different tests, the BUD function is said to be indifferent to which of the two is selected. Figure 5 shows " indifference curves" for the function and offers some insight into how the scheme works.* Figure 6 shows two other possibilities for the quality function, but the curves of figure 5 gave the compromise intuitively desired, and indeed the others proved inferior experimentally. This is admittedly flimsy justification, but intuition was exercised in order to expedite getting a working program. This will be seen again and the reader should realize that the unde-

* These curves are reminiscent of potential lines, and indeed, one may think of the process as selecting the test with highest "potential."

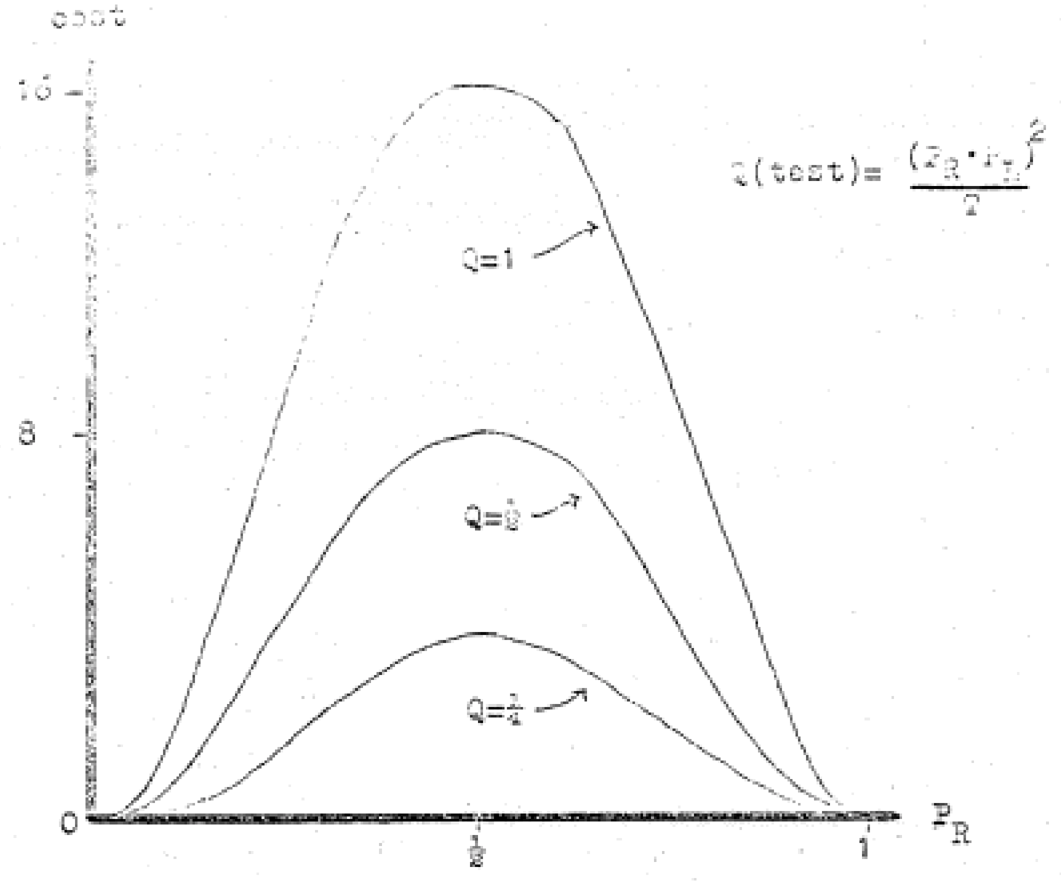


Figure 5. Lines of indifference.

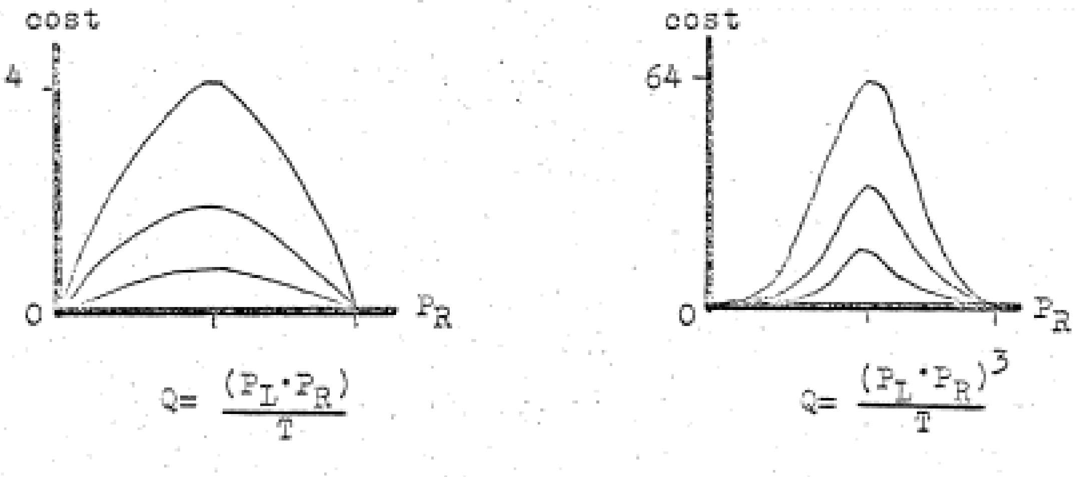


Figure 6. Lines of indifference for other quality functions.

fixed parameters which appear were set through intuitive judge-ments.

Substituting

It is easy to see how the BUD function can make mistakes. Suppose, for example, that three tests are available for grafting at some end point in a partial tree. Suppose further that they divide the probability as indicated somewhat abstractly by figure 7. BUD would obviously recommend that test T1 be followed by test T2 to produce the division of probability shown in figure 8a. But with only a slight increase in cost, one could get much superior overall probability splitting by using T3 followed by T2 as shown in figure 8b.

Of course one could merely complicate BUD by examing 2-level combinations of tests in about the same way, but the number of combinations increases more than linearly with the number of tests and is to be avoided. Instead, this program uses a function named SUBSTITUTE to look at a particular point in the tree with a view toward changing the test. The test is changed if another gives a better quartering of the probability with the <u>same</u> subsequent tests. Hence SUBSTITUTE would fix the error made by BUD in the current example by the change shown in figure 9.

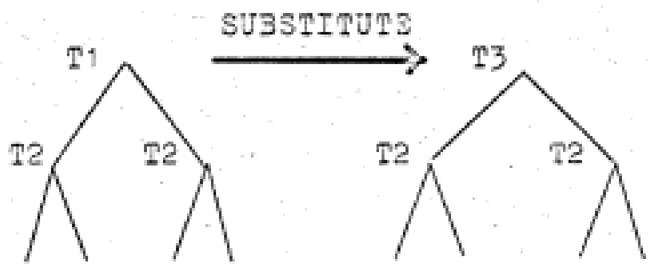


Figure 9. Change effected by SUBSTITUTE.

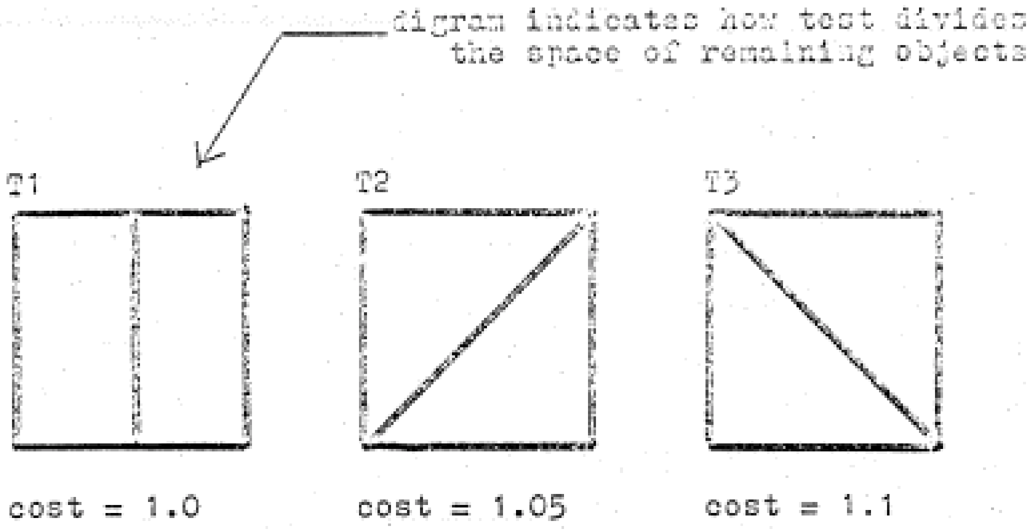

digram indicates how test divides the space of remaining objects
T1
T2
T3
cost = 1.0
cost = 1.05
cost = 1.1

Figure 7. Three available tests.

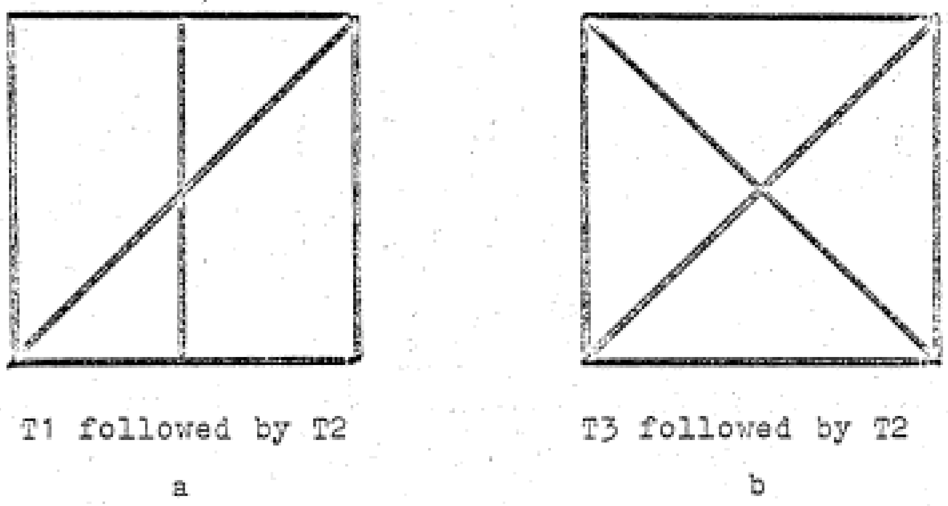


Figure 8. Alternative two-test partitions.

Again, of course, a compromise with cost is necessary and no change is made unless a different test yields a greater number in

$$\text{Quality}_2(\text{test}) = \frac{(P_{RR}\ P_{RL}\ P_{LR}\ P_{LL})^2}{T},$$

in which the parameters and rationale are analogous to those discussed under Budding.

Naturally this idea could be extended to still higher orders, but I doubt if any significant gain would be had.

Exchanging

Suppose now that one is examining a tree tip of the form in figure 10a. One wonders if perhaps a little alteration

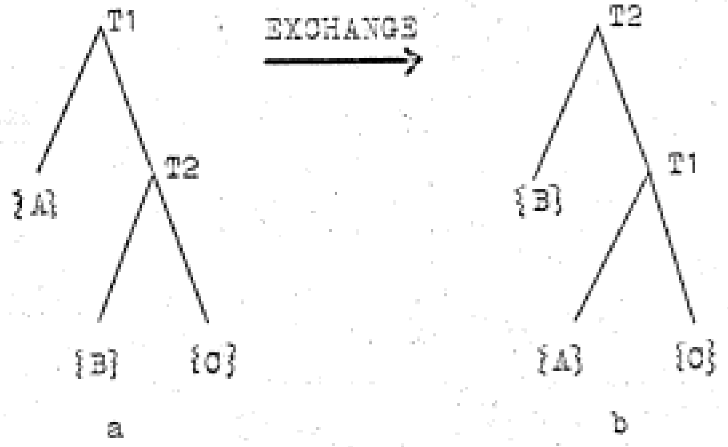


Figure 10. A possible exchange.

would be better, say to the form shown in figure 10b. Calculating expected costs it is clear that the change should be made if

C_{TX} = cost of test TX, $p = P(\{A\ B\ C\})$,

and

$$C_{T1}\cdot p + C_{T2}\cdot[P(\{B\}) + P(\{C\})] > C_{T2}\cdot p + C_{T1}\cdot[P(\{A\}) + P(\{C\})] ,$$

or equivilently,

$$C_{T1} \cdot P(B) > C_{T2} \cdot P(A)$$

Generalized a bit, this idea can be applied anywhere in the tree and as such, constitutes the EXCHANGE function. One need only think of A, B and C in the above paragraphs as sets of objects instead of simple objects. Now, however, the function must be prepared to do some more extensive work on the tree if subsequent tests are already in place. Figure 11 helps indicate what should be done. Notice that the objects of set C remain together in the new structure. So do those of B. One expects that the subtrees used to divide up sets B and C are still good and should simply be moved to new locations, as in figure 12.

The trouble is that some of the objects of set A likely invade the branch of set B. Now EXCHANGE is not necessarily a good idea and errors are made. For this reason, the swapping condition was altered so that swapping does not occur if possible gains are slight. This is done through the introduction of a parameter α:

$$C_{T1} \cdot P(\{B\}) > \alpha C_{T2} \cdot P(\{A\}) \rightarrow \text{swap.}$$

α was set at 1.25 with seemingly satisfactory results.

<u>Sweeping</u>

Consider the situation of figure 13a. Sweep will transform the tree to the form shown in figure 13b if $C_{T2} < C_{T1}$. No immediate change in expected cost results, but it seems a good idea to "sweep" longer tests down the tree structure, for it may well be that they can be eliminated in some of the sebsequent paths and there result in an eventual saving.

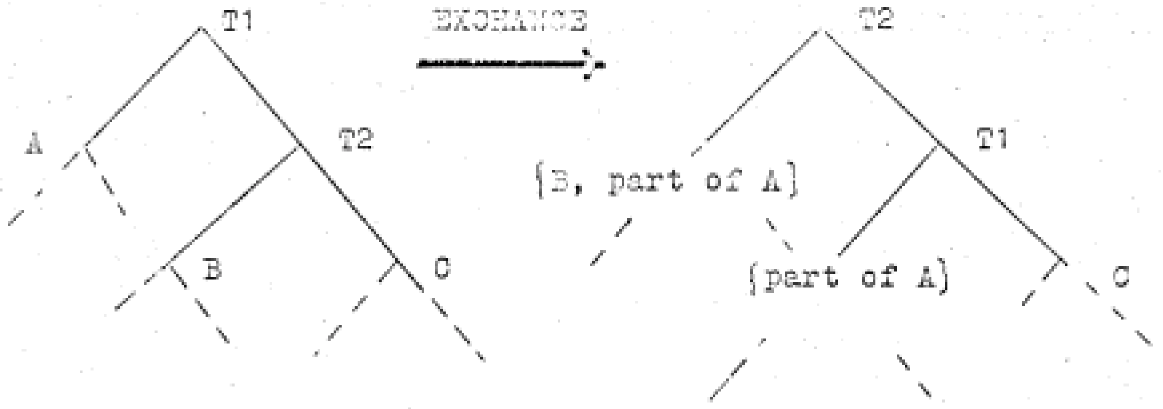


Figure 11. A generalized exchange.
A, B and C are sets.

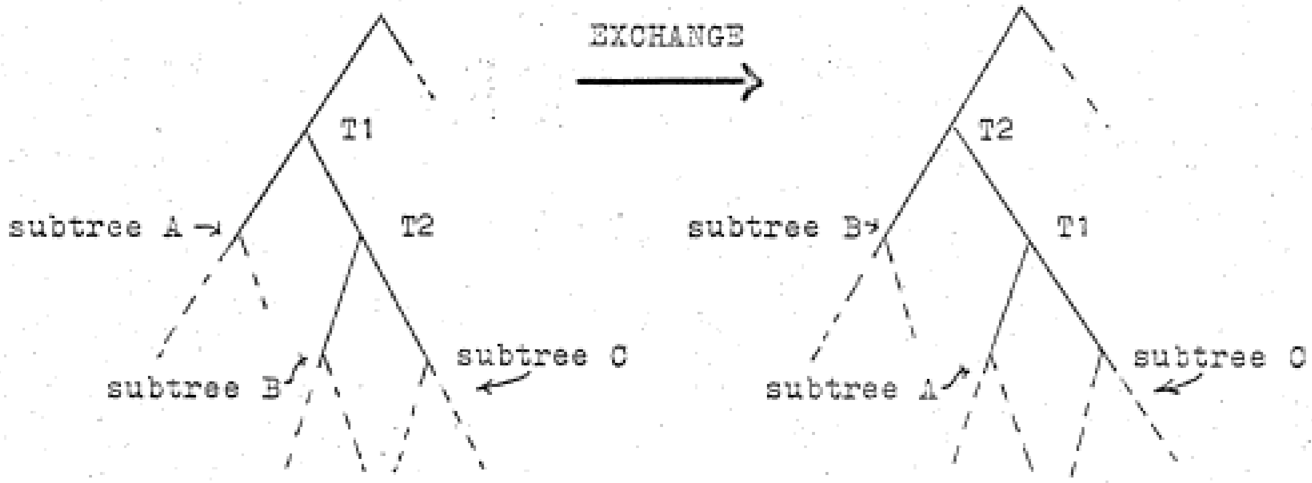


Figure 12. Modifications appropriate for
generalized application of EXCHANGE.

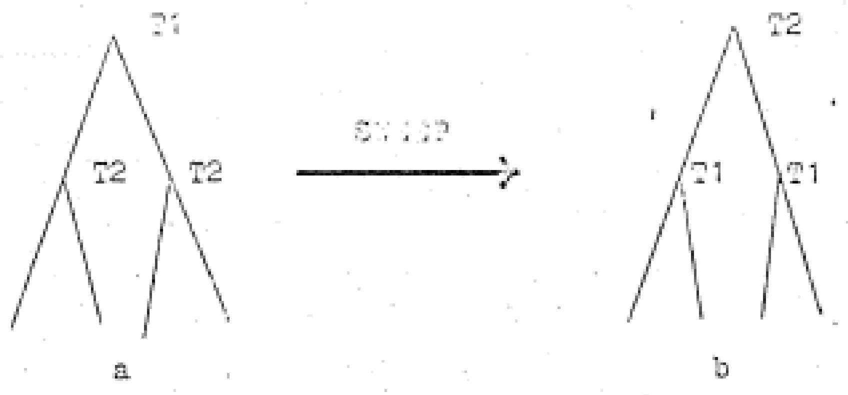


Figure 13. A sweep.

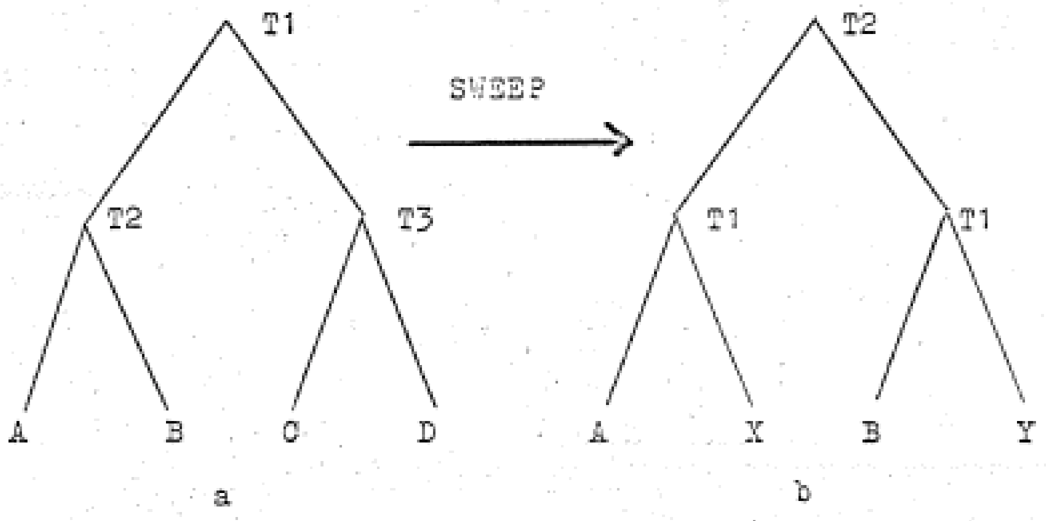


Figure 14. A more complicated sweep.
A, B, C, D, X and Y are sets.

Like EXCHANGE, the SWEEP function in use is a generalization. It will take the tree of figure 14a and transform it to that of figure 14b if certain conditions are met. For one thing $C_{T1} > C_{T2}$ as before. But also notice that only the sets A and B are preserved; C and D will be in general divided between X and Y. In order to avoid a complete mangling, the function refuses to effect the change unless D matches X fairly well, and C matches Y fairly well (or if C with X and D with Y). "Match fairly well" means 65% or more of the objects in the first set are found in the second.

SWEEP may also be used even if $C_{T1} < C_{T2}$ if $C_{T3} >> C_{T2}$ and the other conditions hold. This second criterion was implemented since in such cases the elimination of T3 seems like a good deal ($>>$ means more than 3 times here).

Naturally SWEEP also looks at the tree with the above roles of T2 and T3 reversed.

Pruning

EXCHANGE and SWEEP cause many objects to course down different paths. The result is that "dead" tests, or tests that cause no splitting, may be found here and there in the tree. When FOREMAN encounters such a test, it uses PRUNE to eliminate it as in figure 15. PRUNE is repeated if the test structure it elevates is also headed by a dead test.

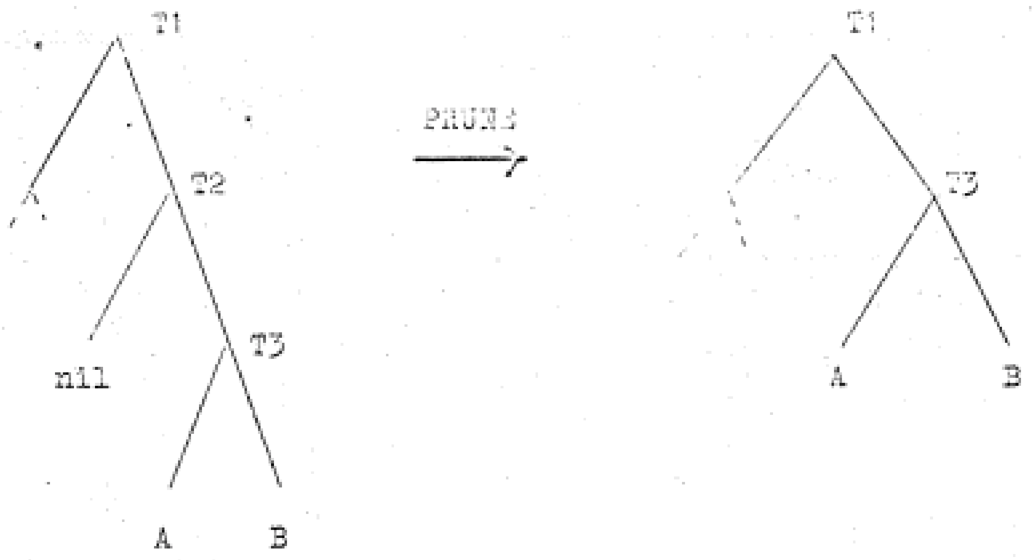


Figure 15. The use of PRUNE.

The Executive Function

With the armamentarium of heuristic operations just reviewed, one could assemble a simple supervisory program. The program could simply work its way down the growing tree by calling itself recursively on the branches it generates.

Notice that a substitution at one point might well set up the higher test point for a sweep or other change. But in the simple scheme mentioned above, there is no way for the program to back up and review when such changes make reconsiderations of a higher level choice a good idea. This consideration led to the development of EXECUTIVE, a somewhat more elegant supervisory program. It may be helpful to refer frequently to the flow chart of figure 16, as the program is explained in the following paragraphs.

Goals of EXECUTIVE

The job of EXECUTIVE is to focus the program's attention at an appropriate point in the partially constructed tree. In diagrams the convention will be to indicate the center of attention by a small arrow ($\rightarrow$). Two tests immediately following a given test are called its daughters. The given test is then referred to as the parent of its daughters.

EXECUTIVE must be able to do more than just back up and review an old choice. To avoid wasted effort it should not work on a test point if a parent of that test is to be reviewed, because a change may in fact completely change the nature of subsequent problems. In figure 17, for example the problem

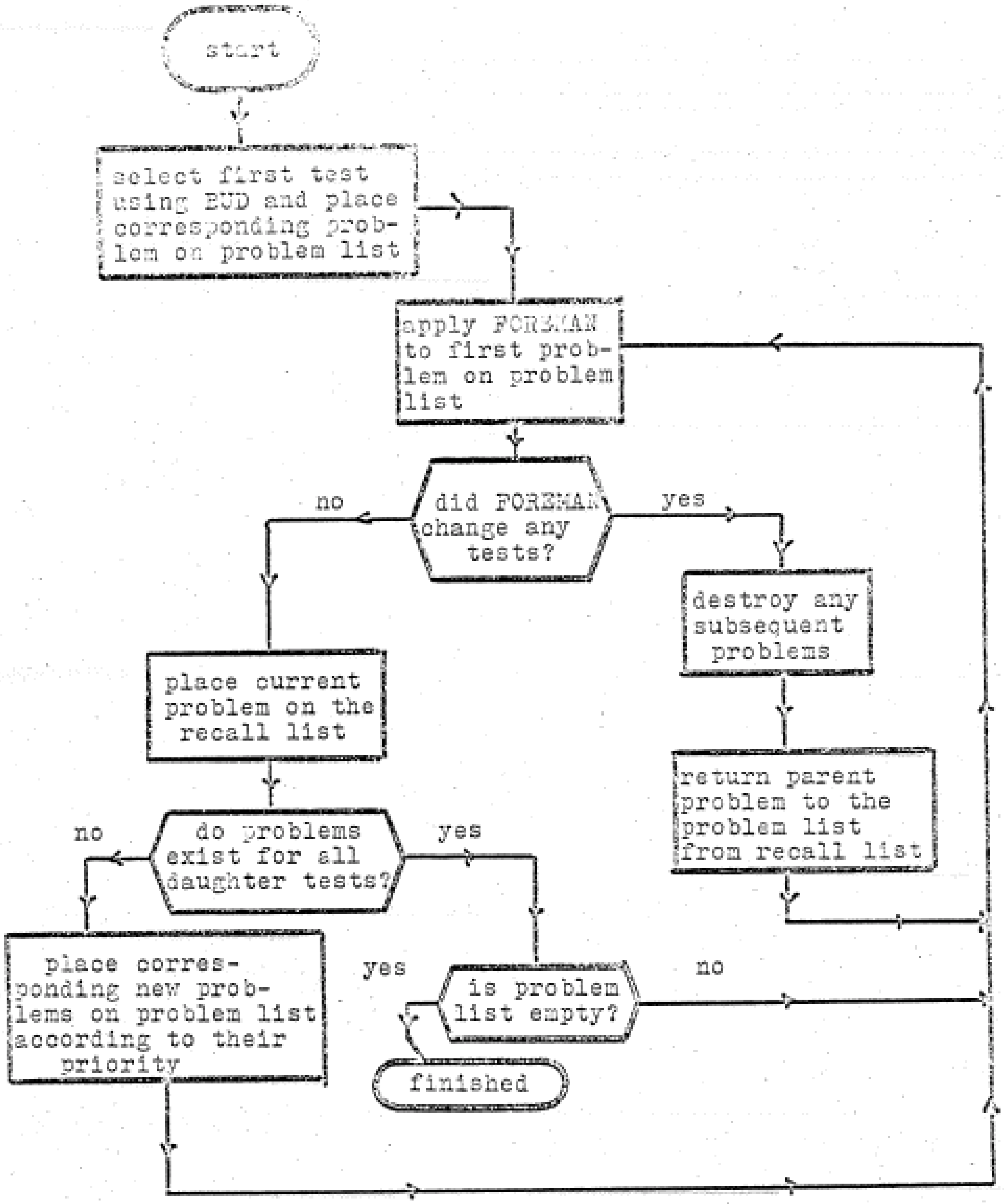

start
select first test using EUD and place corresponding problem on problem list
apply FOREMAN to first problem on problem list
did FOREMAN change any tests?
no
yes
destroy any subsequent problems
return parent problem to the problem list from recall list
place current problem on the recall list
do problems exist for all daughter tests?
no
yes
place corresponding new problems on problem list according to their priority
is problem list empty?
yes
no
finished

Figure 16. EXECUTIVE.

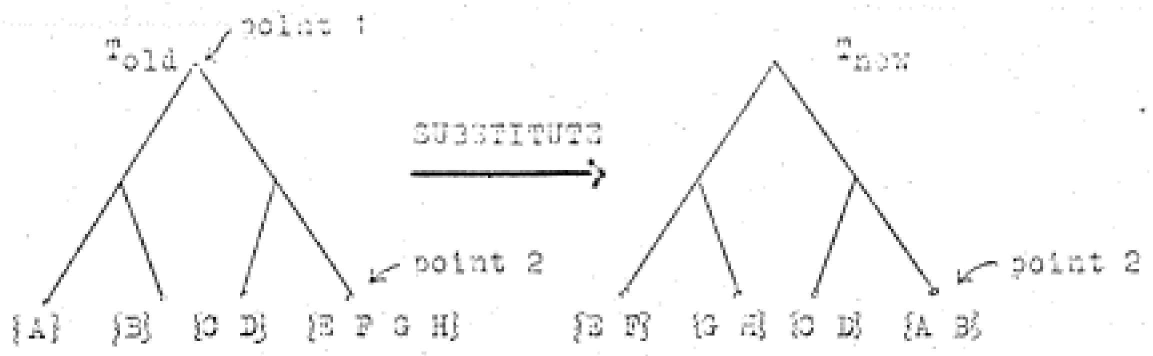


Figure 17. Execution of parent problem before daughter problem.

of point 2 has a different set of objects to consider as a result of a higher level operation at point 1. EXECUTIVE should similarly avoid deep extension of one tree branch while neglecting another. Otherwise, a review and subsequent alteration generated by a later look at the neglected branch might well render most of the work useless. See figure 18. Finally, EXECUTIVE should have some kind of cycling protection so that chains of circular operations are not repeated indefinitely.

Before explaining how these things are done, the role of FOREMAN will be discussed.

The Foreman

Once EXECUTIVE has picked a center of attention, FOREMAN cleans up the area with PRUNE and, if necessary, extends the tree with BUD; it tries to apply SUBSTITUTE, EXCHANGE, or SWEEP in turn; and finally it uses PRUNE and BUD in case a transformation has left some garbage in the immediate tree structure.

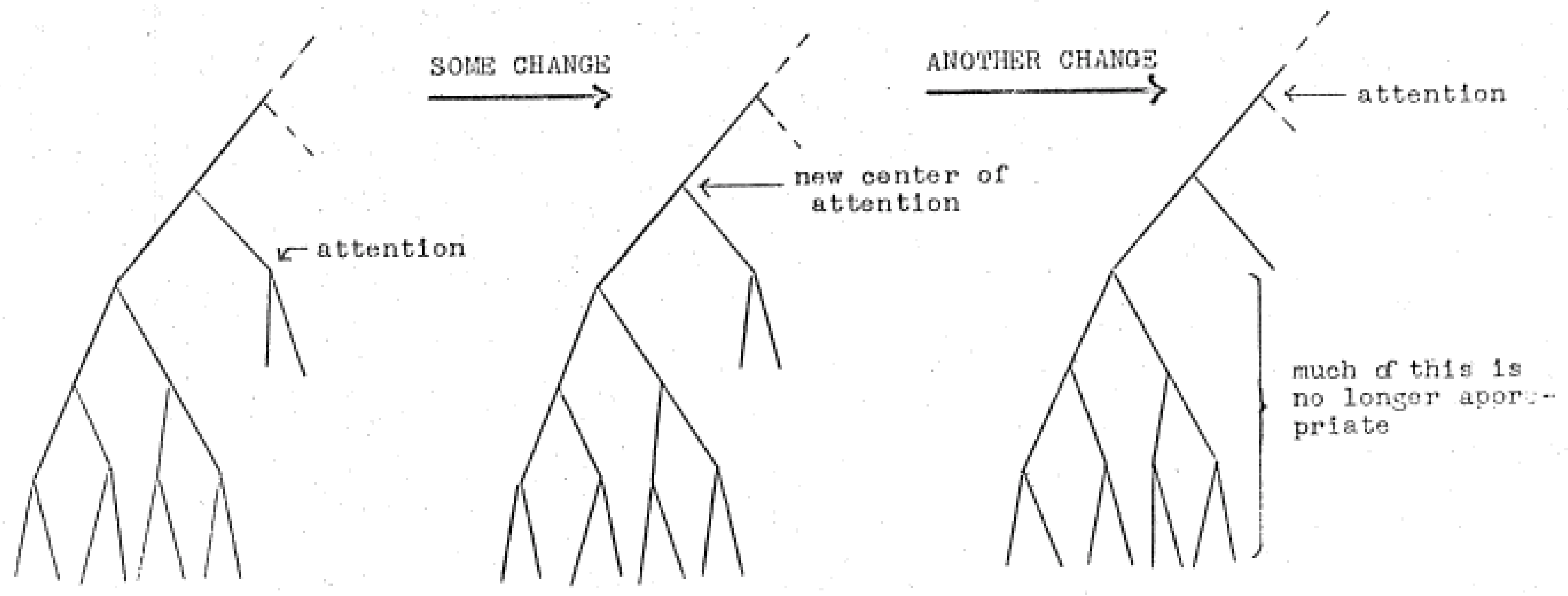


Figure 18. Result of uneven tree extension.

Note that the *** test-switching operations can be applied only if the test at the center of attention has daughters in each of its two branches. A flow chart for FOREMAN is in figure 19.

<u>The Problem List; Planning</u>

Once EXECUTIVE has picked an initial test, it places a packet of information called a <u>problem</u> on a list called the <u>problem list</u>. A problem consists of 1) a pointer to the test it is associated with in the tree, 2) a number indicating its priority, and 3) other lists that aid the operation of FOREMAN, but are of no concern here. As can be seen in the flow chart of figure 16, EXECUTIVE enters a loop around which it circles until the tree is complete.

The first step in the loop is to apply FOREMAN to the first entry (the one with highest priority) in the problem list. Of couse, when entering for the first time, there is only one entry.

If FOREMAN only extends the tree with BUD, or does nothing, EXECUTIVE simply creates new problems associated with any daughter tests otherwise without associated problems. It then removes the current problem from consideration by placing it on a list of old problems, the <u>recall list</u>. The use of the recall list will be explained later.

The new problems are inserted into the problem list according to a priority, which is simply the number of objects that terminate paths from the associated test. See figure 20 for some

*** Insert <u>SWEEP and SUBSTITUTE</u> in this position.

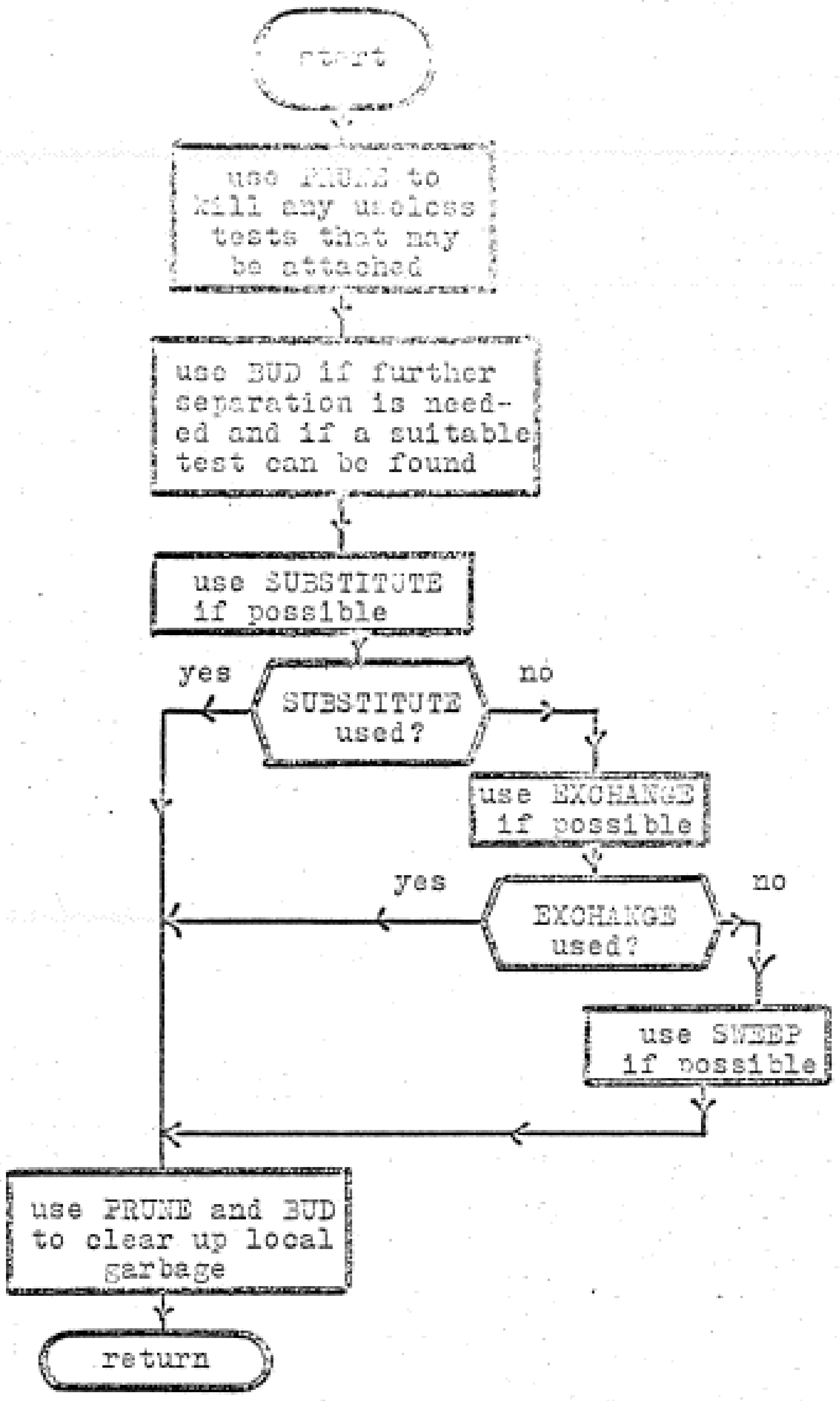

start
use PRUNE to kill any useless tests that may be attached
use BUD if further separation is needed and if a suitable test can be found
use SUBSTITUTE if possible
SUBSTITUTE used?
yes
no
use EXCHANGE if possible
EXCHANGE used?
yes
no
use SWEEP if possible
use PRUNE and BUD to clear up local garbage
return

Figure 19. FOREMAN.

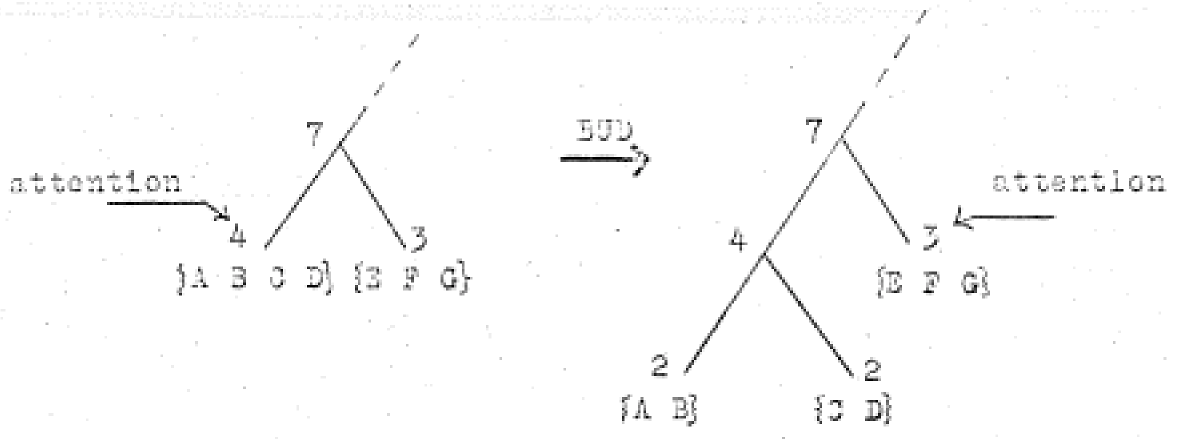


Figure 20. Priorities and the movement of attention.

examples.

In establishing this priority, the program may be said to exercise a simple sort of planning.

This priority scheme assures that a problem will not be considered if a problem associated with an ancestor is also on the problem list. This is true since any test in the tree certainly has more objects coursing through it than have any of its progeny.

The priority defining scheme also helps avoid extensive growth of one branch with neglect of others. This follows since, in general, the repeated partitioning of the objects rapidly reduces the priorities of problems further down the tree. See figure 20 for an illustration.

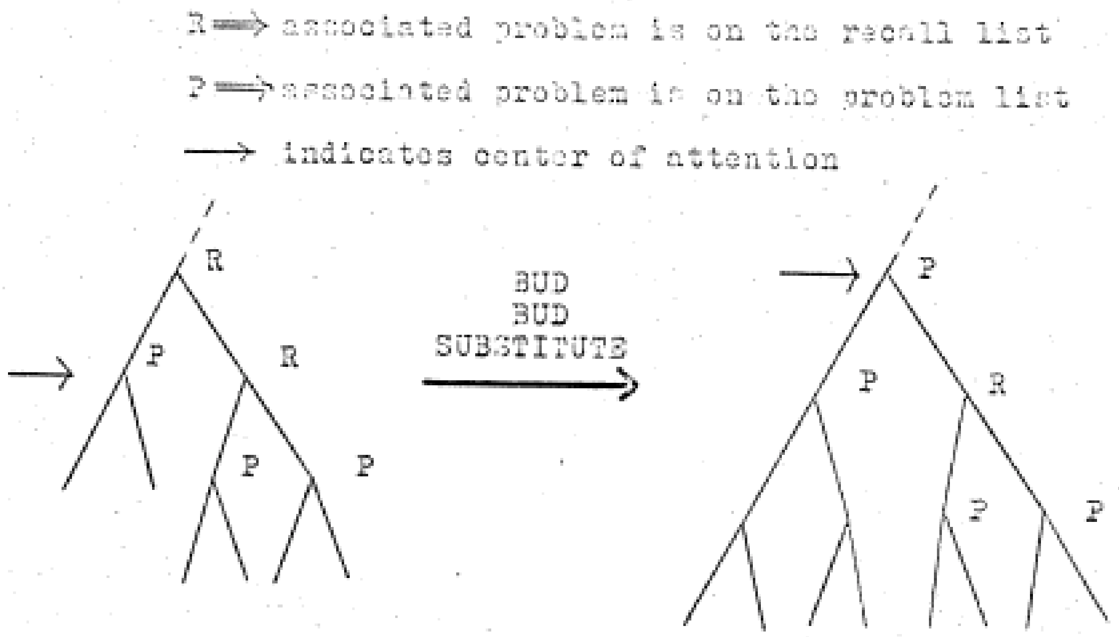
Finally, notice that, as in figure 20, the daughter in the branch with more objects is attacked before the other. This eems like a good idea because a change of attention back to a higher level problem is more likely to come from the more populated of two daughter branches. Work down the other branch may then be wasted. This movement of attention back up the tree seems more likely from a highly populated branch because more variations of object deployment are possible. This yields more values for the quality functions, increasing the chance that an improvement may be found.

The Recall List

Returning to the top of the loop, notice that if FOREMAN does in fact change a test by exercising EXCHANGE, SUBSTITUTE, or SWEEP, then the program works through a different set of operations. First, the auxilliary lists carried in problems indicate what tests have previously been used and what objects are to be handled. The test-swapping heuristics frequently change the course of objects through the tree structure and invalidate the auxilliary lists. Trouble is avoided simply by destroying any problems found subsequent to the change. Bear in mind that only problems are eliminated, not the tree structure itself.

The parent of a changed test should be re-examined since change in a daughter may in turn call for a transformation of the parent. Notice that the parent problem resides on the recall list. To initiate review, it is merely moved from the recall list to the problem list at the spot appropriate to its priority number.

Figure 21 illustrates these steps.



R ⟹ associated problem is on the recall list
P ⟹ associated problem is on the problem list
⟶ indicates center of attention
R
P
R
P
P
BUD
BUD
SUBSTITUTE
P
P
R
P
P

Figure 21. Recall of an old problem.

<u>Cycling</u>

The cycle protection gear is not included on the flow chart to avoid cluttering. The current scheme merely counts the number of times a problem has been reviewed and, if more than twice, further change is not allowed.

<u>Termination</u>

The problem list will be empty when the tree is fully grown and all necessary decisions have been reviewed. EXECUTIVE then exits from the loop and announces that it is finished.

RESULTS

An Example

The completed program was applied to 12 situations of which two are reviewed in this section. The first has the table of outcomes, probabilities, and costs shown in table I. Notice

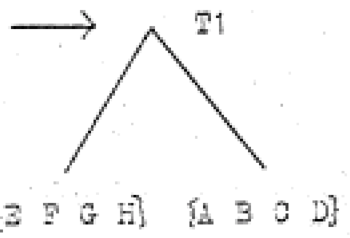
tests ↘ objects →	A	B	C	D	E	F	G	H	costs ↓
T1	T	T	T	T	F	F	F	F	1.1
T2	T	T	F	F	T	T	F	F	1.2
T3	T	F	T	F	T	F	T	F	1.65
T4	T	T	F	T	T	T	F	F	.98
T5	F	F	T	F	F	F	T	F	1.55

(all probabilities equal .125)

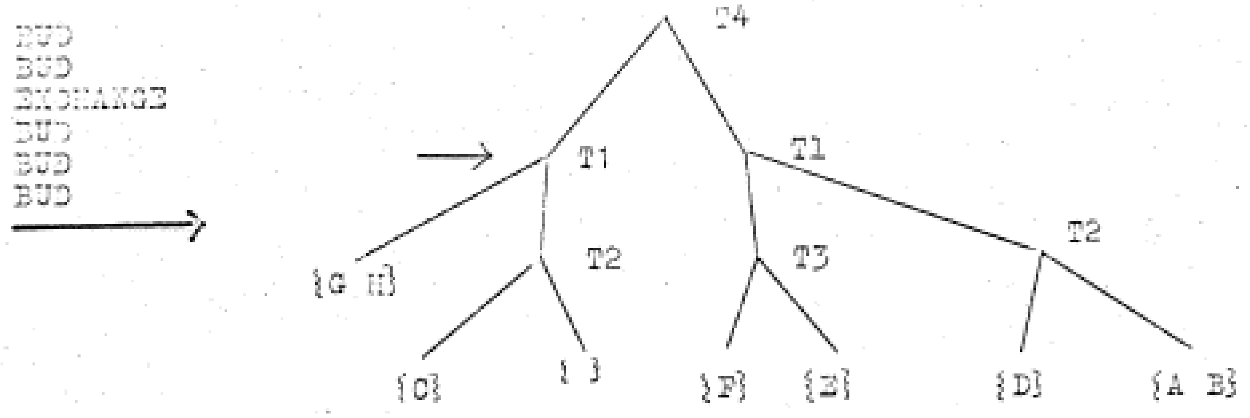
Table I

that all probabilities are the same and that the first three tests all divide the objects into equally probable groups. The remaining parameters were fixed more or less arbitrarily. The net result is a data set on which EXECUTIVE's activity is non-trivial, but easy to follow. As before, an arrow indicates the point of attention defined by the pointer in the first problem on the problem list.

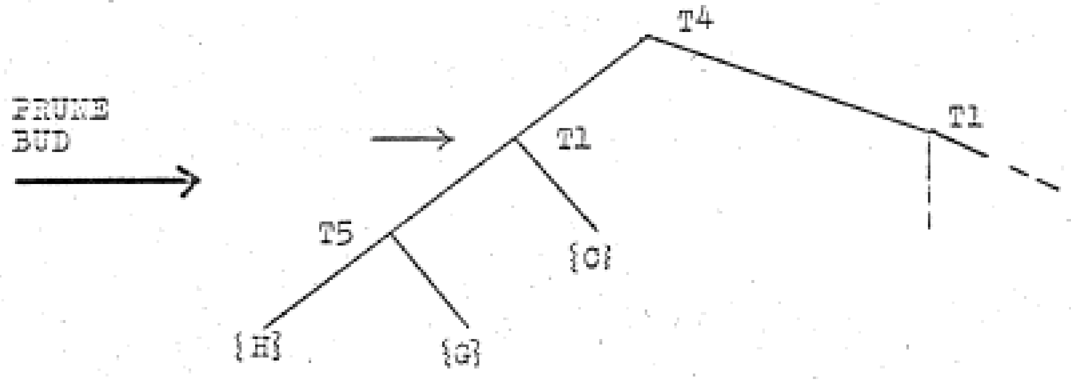
The first test selected was T1.



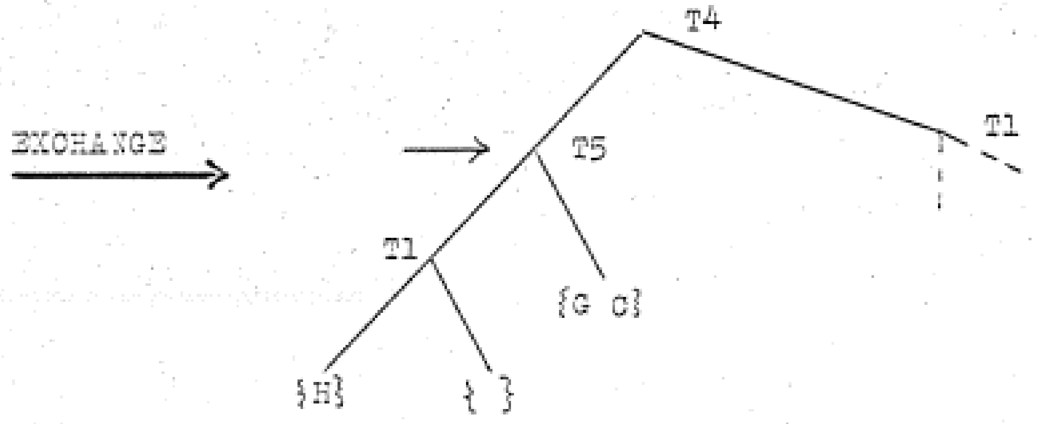
Two calls to FORMAN left the following:



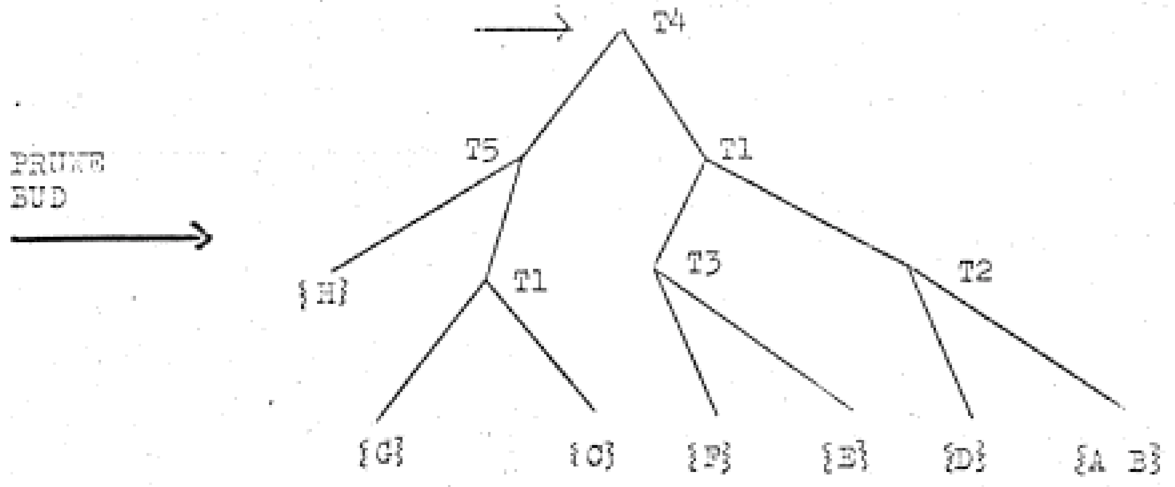
The obvious action here was to prune out the T2 and bud something in to separate G and H. Only T5 and T3 could separate them so T5, the cheaper, was used.



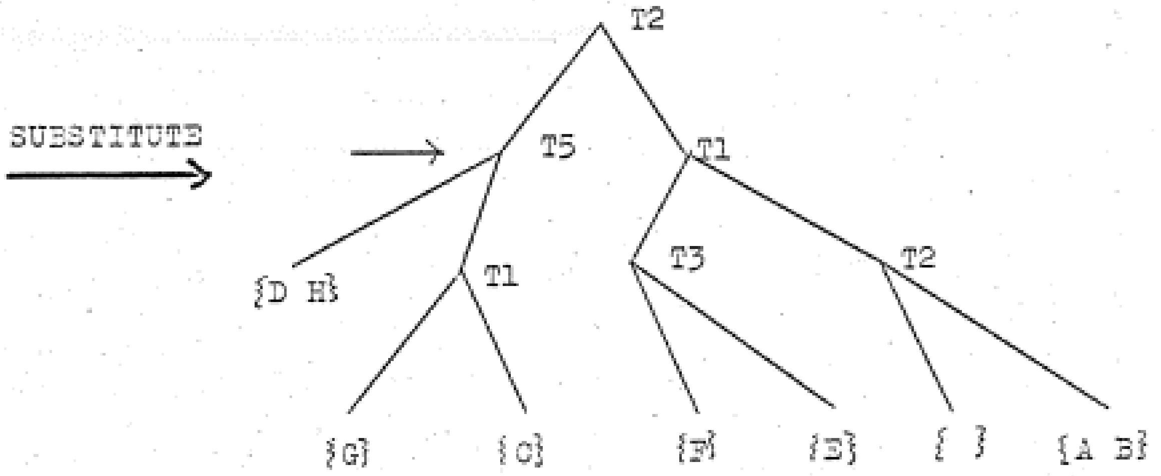
But at this point EXCHANGE made a mistake because of an obscure bug and switched T1 and T5.



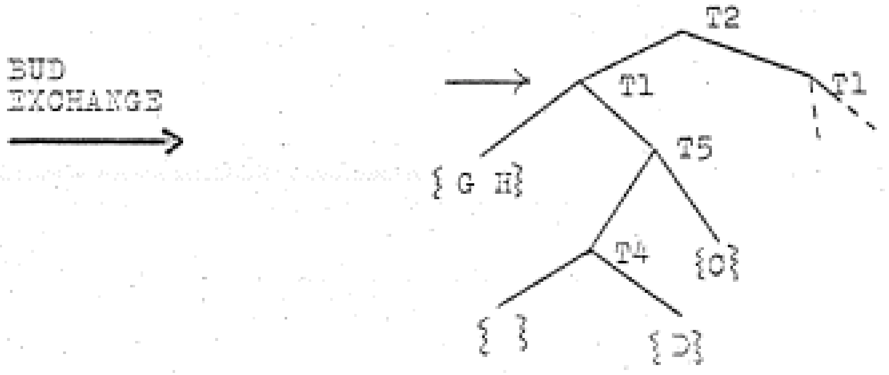
PRUNE and BUD were called and the problem associated with the T4 node was moved from the recall list to the problem list.



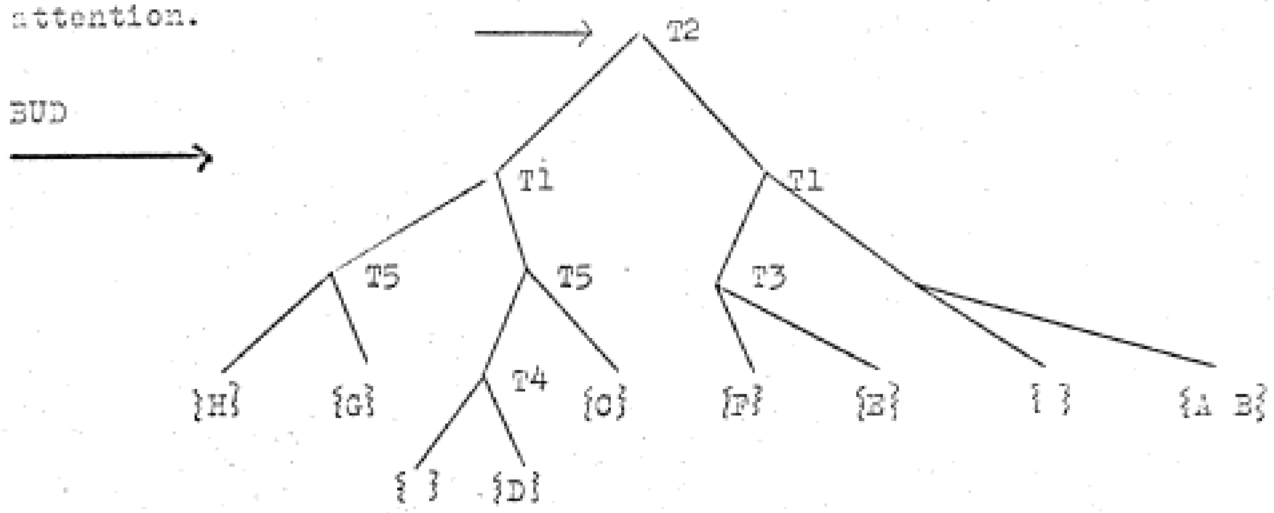
SUBSTITUTE then noticed that T2 yielded a perfect quartering although slightly more expensive than T4.



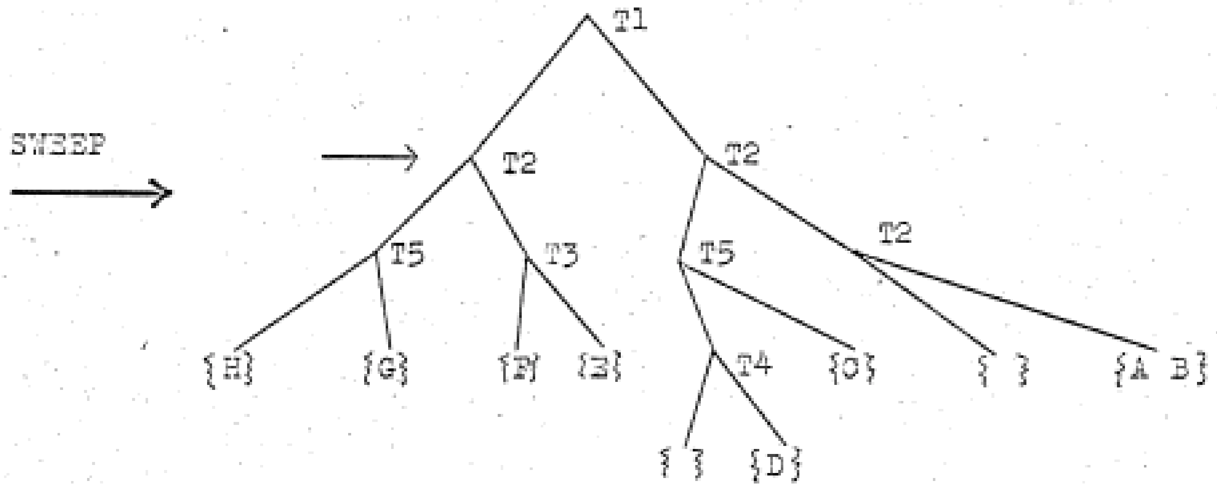
Now a BUD split D and H with T4 and was followed immediately by EXCHANGE switching T5 and T1.



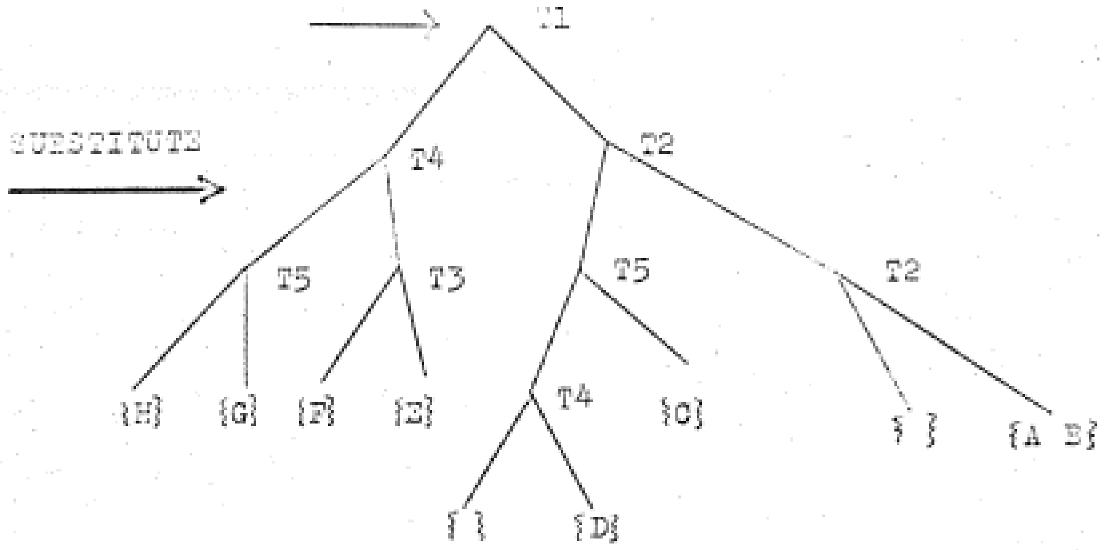
This time EXCHANGE did good work: the T1-T5 combination and the T5-T1 combination split the probability in the same way, while T1-T5 had the advantage of moving the more costly test down the tree. After a BUD, the top node was again the center of attention.



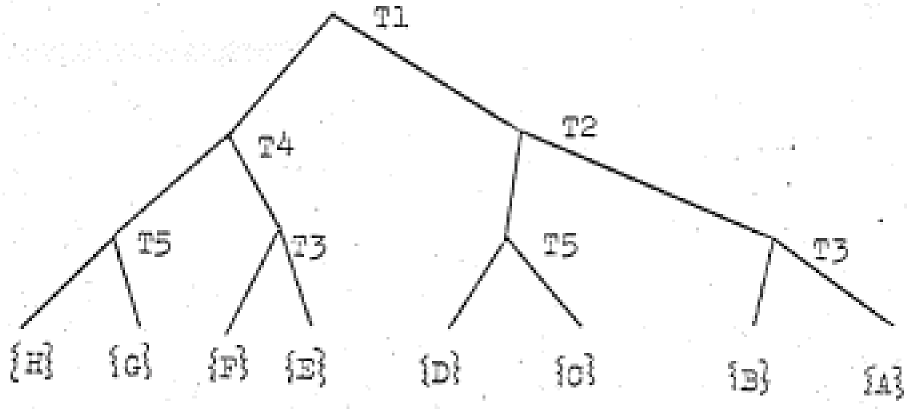
Since $C_{T1} < C_{T2}$, sweep was clearly in order.



At that point, T2 was replaced by the shorter T4



Anti-cycle protection excluded any activity at the top. Further operations involved only pruning out dead tests and separating A and B for a final average cost of 3.79 and a tree:



This tree seems reasonable in many respects. T1 is the least costly test that exactly divides the probability. T2 and T4 are also relatively short and together with T1 exactly quarter the probability. The longer tests are found only in the last row, where , with one exception, they are absolutely required to complete the separation. The exception is that T4 would be better for separating C and D than would T5. The reason for the error is that this instance of T5 was originally installed to separate G and H, but through a series of transformations resulting from EXCHANGE, SWEEP, and SUBSTITUTE, it found itself with an entirely different duty. If so unsuitable a test were to occur higher in the tree, SUBSTITUTE or another operation might well have excised it, but such transforms are powerless at the tips of the tree where there are no tests following. A special case transformation could easily be written to handle this sort of situation.

A Bigger Problem

In another experiment, twice as many objects were involved and the number of tests was expanded from 5 to 7. The parameters involved are shown in table II. In this case, however, the outcomes were set randomly with the aid of a coin, and the costs and probabilities similarly fixed with the aid of a Boston telephone book.

On this data the program rather uneventfully generated the tree of figure 22. It is instructive, however, to notice how the tests and objects are deployed:

Tests 4, 3, 7 and 5 split the objects into sets of nearly equal number. Moreover, 4, 3, and 7 cost less than any others. That they populate the highest regions of the tree is therefore not unexpected.

The expensive test 5 is found only in one place, and reference to the table of outcomes shows no other test can function as it does to separate M and N.

Note that two sets remain unseparated: {A F} and

A	.009	E	.016	I	.04	M	.006
B	.004	F	.005	J	.2	N	.024
C	.021	G	.051	K	.06	O	.09
D	.102	H	.081	L	.1	P	.19

probabilities

objects

tests	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	cost
T1	T	F	F	T	T	T	T	T	T	T	F	T	T	T	F	T	7.15
T2	T	T	F	T	F	T	T	T	F	T	F	T	T	T	T	T	6.94
T3	F	F	T	T	F	F	F	T	F	T	F	T	T	T	T	F	1.77
T4	T	T	T	F	T	T	F	T	F	F	F	T	F	F	F	T	3.48
T5	F	T	F	T	F	F	F	T	F	T	F	T	T	F	F	T	9.70
T6	T	T	T	F	T	T	F	T	T	F	T	T	T	T	T	F	4.96
T7	T	F	F	F	F	T	T	F	T	T	T	F	T	T	T	F	1.49

outcomes

Table II

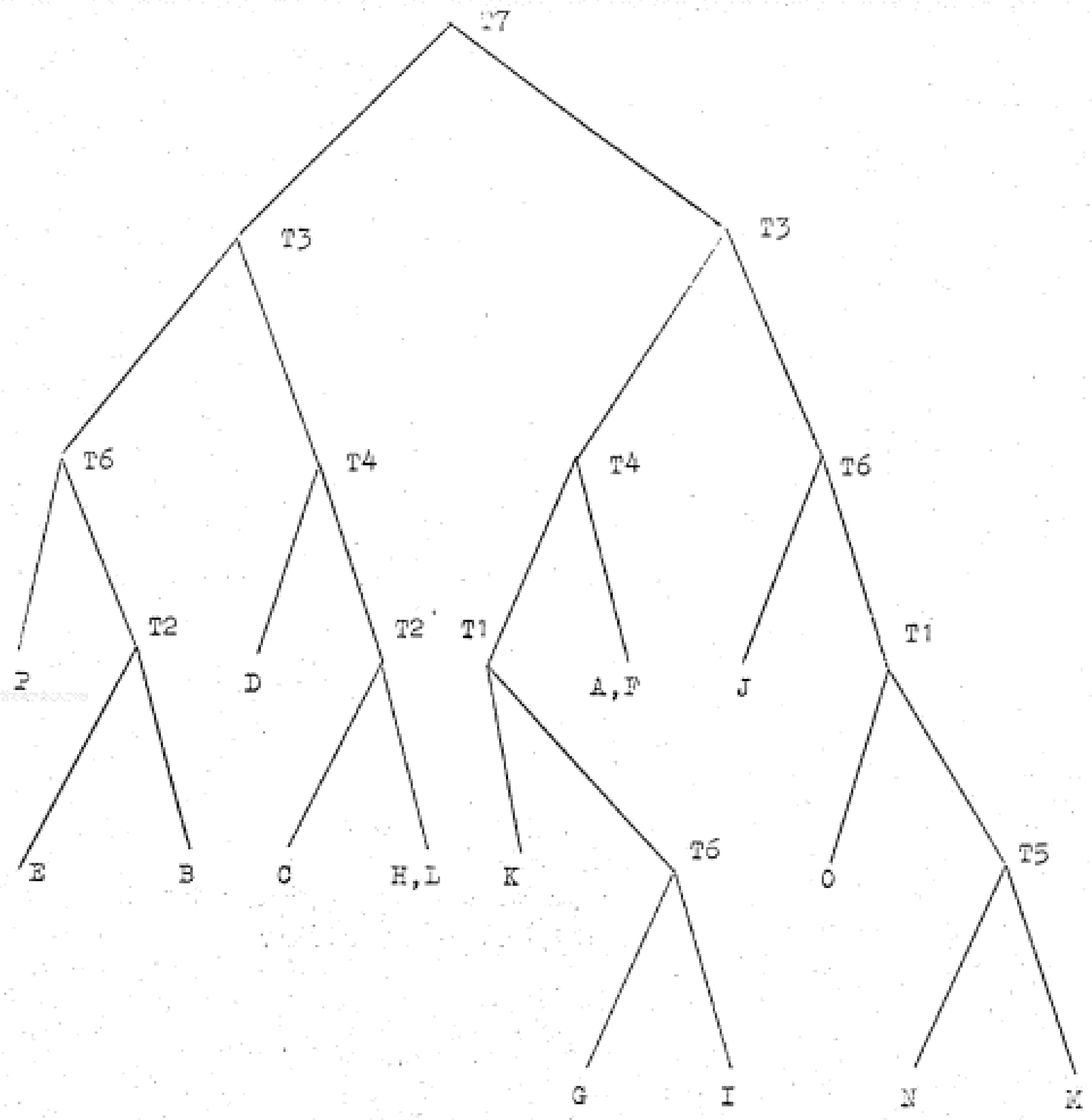


Figure 22. Tree erected for bigger problem.

{H L} . Inspection of the table shows they indeed cannot be separated.

One expects to find the more probable tests higher in the tree where fewer tests are required to identify them. The most probable objects are J, P, and D. Notice they are found in the highest populated level.

REMARKS

Conclusions

The program as it stands did some intelligent things with a number of situations to which it was addressed. On others, serious mistakes were made, but the overall performance seemed acceptable as a first try. Its successes indicate value in the heuristic approach, while its failures reveal new problems to be explored.

Refinement

Alterations to the present program would fall in three categories: bug removal, tuning, and major revision. Bug removal currently refers to the two faults observed in the RESULTS section: first, a subroutine is needed to insure that the tests at the tips of the tree are optimum; second, EXCHANGE blunders frequently and swaps tests when it should not. Correction of these weaknesses should involve no major difficulty.†

Tuning refers to improvement of the simple criteria the program uses to make its decisions. For example, there may be better ways to define the priority number or to determine the quality of a test from its cost and splitting ability. As was mentioned, many of these things were set somewhat arbitrarily.

Finally, major revision refers to substantial changes which would give the program more of the qualities characteristic of "intelligence." For one thing, the program now has no facility for retreating from blind alleys; the condition for change is "change if tests indicate that it looks like a good idea" instead of "change if the resultant structure does in fact

† See Appendix II

lead to better results."

A second major revision would involve revising FORMAN. As it stands, the function just checks to see if any transformations can be used and blindly uses the first that works. One alternative would try all possible changes and use the one that works "best." This, however, is a form of exhaustive enumeration. It would be more elegant to introduce a function that somehow would quickly examine tree structure in the vicinity, form an opinion on what it needs, and then check to see if a transformation appropriate to correction can be used.

Generalization

It is easy to think of generalized problems on which extended programs could be used. A few are listed below:

1. The test outcomes are not limited to just two possibilities.
2. The objects are partitioned into categories. The tester is concerned only with determining the category.
3. The test outcomes are probabilistic functions of the object.
4. Certain tests are inappropriate unless preceded by certain others.
5. Certain important objects must not require more than a certain cost before recognition. (A lion appearing before the reader is unlikely, but he should be glad to recognize it quickly).

APPENDIX I

The Reinwald-Soland Method

The Method

Since the Reinwald-Soland scheme is well presented in their paper,$^{(4)}$ only the gist of the idea will be reviewed here.

Suppose, given a partially constructed tree, it is possible to establish a lower bound on the expected cost of any complete tree formed by budding out the tree at hand. Let this lower bound to the expected cost be L_T where T is the name of the associated tree.

Then if a group of partial trees are candidates for the trunk of the final tree, it would be wise to try budding out the candidate whose L_T is smallest until the L_T of the derived tree exceeds that of some other candidate on the original list.

This general idea led to the flow chart of figure 23. The original set of partial trees consists of all trees containing just one test. These are ordered according to their associated lower bounds, the L_T's. The first member of the list is picked off and all trees that can be formed from it by a single test addition are placed on the partial tree list in the appropriate place. The algorithm terminates when a complete tree appears at the head of the list. This tree is optimal because its L_T is not only a lower bound, but is in fact the expected cost; and that cost must certainly be lower than that of any complete tree that evolves from partial trees whose L_T's are greater.

Now suppose we think of the N tests as forming N coordinates

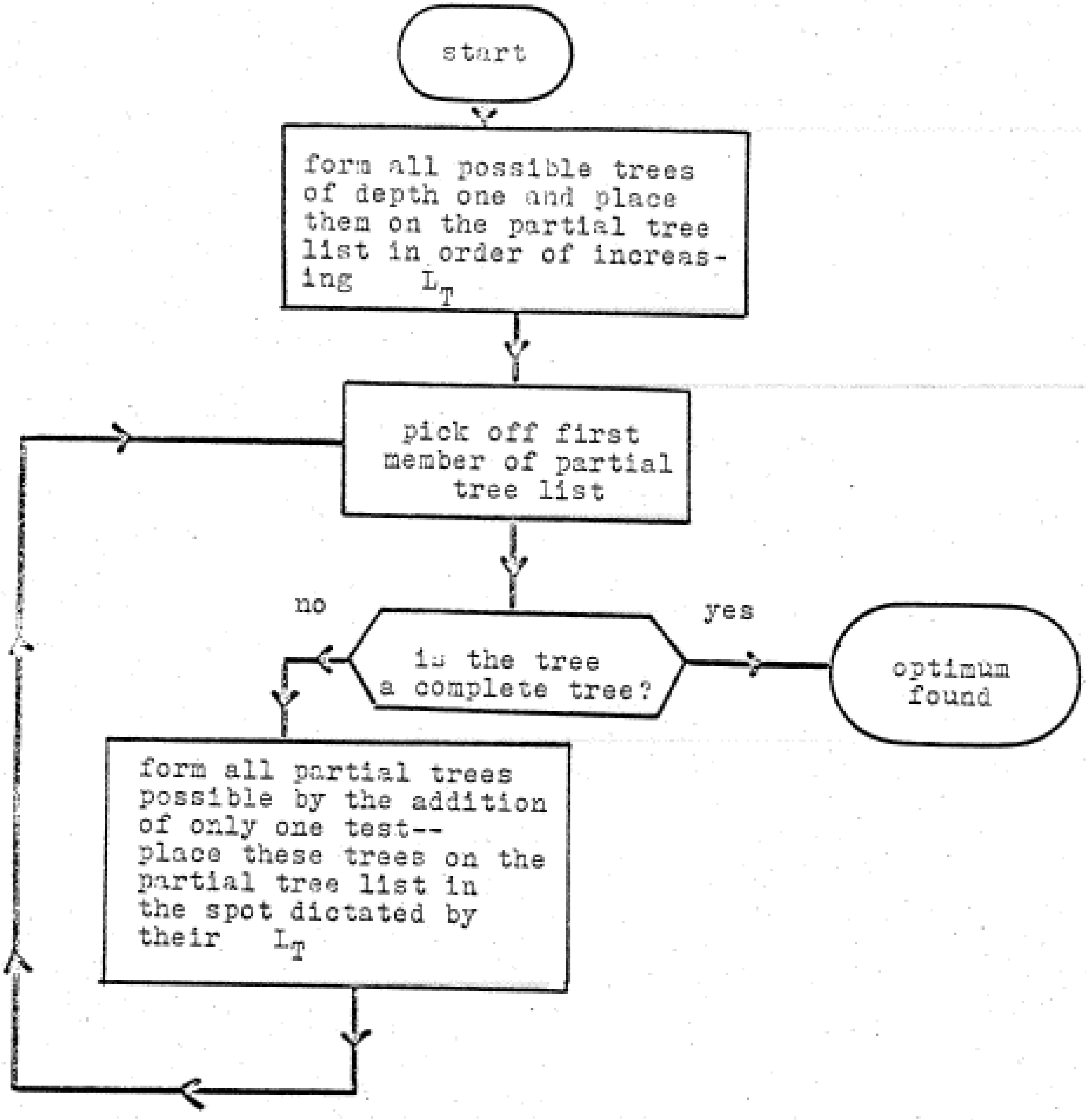

start
form all possible trees of depth one and place them on the partial tree list in order of increasing L_T
pick off first member of partial tree list
no
is the tree a complete tree?
yes
optimum found
form all partial trees possible by the addition of only one test-- place these trees on the partial tree list in the spot dictated by their L_T

Figure 23. The Reinwald-Soland Algorithm.

of a space. Then each object lies on some corner of the resultant N dimensional cube. A thin space will be defined informally as one in which few if any objects lie on adjacent corners for which only one coordinate differs.*

My contention is that for reasonably large, thin spaces, the Reinwald-Soland method amounts essentially to exhaustive enumeration of possible trees.

Degeneration of the Method

A study of the paper reveals that in thin spaces the rule for forming L_T simplifies to:

1) find the expected cost of each test by multiplying its cost by the total probability of the objects that course through it.

2) sum the results.

Thus it is clear that each time a complete layer is added, the lower bound is increased by the average costs of the tests in the layer. The lower bound, L_T, therefore increases steadily with the depth of a partial tree.

The reason for the failure of the method in thin spaces lies in this steady growth of L_T with tree depth: If any tree of a certain depth has been examined, it is likely that all trees with two or three fewer layers are on the partial tree list. Hence, before any complete tree can be formed, an enormous number of partial trees of only slightly less depth have been enumerated, evaluated, and placed on the partial tree list. But it has been shown that enumeration of all trees of any non-trivial

* Communications enthusiasts prefer to say that the minimum Hamming distance separating objects is 2.

depth is out of the question.

APPENDIX II

Bugs

The bugs mentioned on pages 27 and 31 have been somewhat alleviated since this paper was completed in 1967.

Blunders by the EXCHANGE heuristic are prevented by blocking action if substantial rerouting takes place. Refer to figure 11, page 13. EXCHANGE does not act unless

$$P(\{\text{part of } A\}) > \lambda \cdot P(A).$$

λ is currently .8 .

A function called NEWTIP eliminates the problem of inappropriate tests at the tree's tips. It simply checks to see if any other test can do the same final splitting job with lower cost. The FOREMAN machinery applies NEWTIP whenever a tip is encountered.

REFERENCES

1. Quillian, M. Ross:"Semantic Memory," Bolt Beranek, and Newman Inc. technical report no. AFCRL-66-189, 1966.

2. Slagle, James:"An efficient Algorithm for Finding Certain Minimum-Cost Procedures for Making Binary Decisions," JACM, July, 1964.

3. Pollack, Soloman:"Conversion of Limited-Entry Decision Tables to Computer Programs," CACM, November, 1965.

4. Reinwald, Lewis and Richard Soland:"Conversion of Limited-Entry Decision Tables to Optimum Computer Programs I: Minimum Average Processing Time," JACM, July 1966.

   see also

   "Conversion of Limited-Entry Decision Tables to Optimum Computer Programs II: Minimum Storage requirement," JACM, October 1967.

5. Gallager, Robert: Unpublished 6.574 class notes. Book in preparation. Pollack (above) also discusses this heuristic.