TECHNICAL MANUSCRIPT 10

A METHOD FOR SYSTEMATIC ERROR ANALYSIS OF DIGITAL COMPUTER PROGRAMS

AUGUST 1962

UNITED STATES ARMY
BIOLOGICAL LABORATORIES
FORT DETRICK

U.S. ARMY CHEMICAL-BIOLOGICAL-RADIOLOGICAL AGENCY
BIOLOGICAL LABORATORIES
Fort Detrick, Maryland

The work reported here was performed under Project 4X99-26-001, Task -02, Basic Biomathematics Research. The expenditure order was 2038.

Joan C. Miller

Biomathematics Division
DIRECTOR OF TECHNICAL SERVICES

Project 4X99-26-001 August 1962

ACKNOWLEDGMENT

The author wishes to express sincere appreciation to Dr. Clifford J. Maloney, Chief, Biomathematics Division, for his guidance and encouragement throughout the course of this project and for his valuable contributions to the revision of the manuscript. Gratitude is also extended to Mr. Truman May, whose computer program was used as a basis for the experimentation involved in developing the method presented in this paper.

ABSTRACT

A method is presented that will afford the programmer a systematic way in which to generate a test deck containing all possible combinations of input to a given problem. After consideration of various alternative representative forms for the logical development of the problem, the logical tree is chosen as most suitable for manual generation of a test deck that will explore every branch of the program. In addition, the tree is shown to give considerable assistance to the programmer in locating the cause of any errors that occur in the test run. The type of analysis recommended here can be applied to both scientific and business applications of computer programming.

CONTENTS

Acknowledgment 3
Abstract 3

I. INTRODUCTION 5

II. BACKGROUND 6

III. NOTATIONAL ALTERNATIVES 6

IV. EFFECT OF LOOPS ON THE DEBUGGING SCHEME 14

V. CHOICE OF MATHEMATICAL MODEL 15

VI. TEST-DECK GENERATION 15

VII. DEBUGGING 17

VIII. CHARACTERISTICS TYPICAL OF SCIENTIFIC PROBLEMS 18

IX. CONCLUSION 19

Literature Cited 20
References 21

FIGURES

1. Hypothetical Flow Chart 7
2. Modified Flow Chart, Omitting Numerical Calculations 8
3. Logical Tree Corresponding to Flow Chart in Figure 1 9
4. Calculation Sequence of Figure 1 Shown in Outline Form . . . 10
5. Tabular Form of Sample Computation 11
6. Symbolic Path List of Sample Computation 11
7. Boolean Matrix of Sample Calculation 12
8. Path Matrix of Basic Computation 13

I. INTRODUCTION

Effective program check-out is imperative to any complex computer program. One or more test cases are always run for a program before it is considered ready for application to the actual problem. Each test case checks that portion of the program actually used in its computation. Too often, however, errors show up as late as several months after a program has been put into operation. This is an indication that portions of the program called upon only by rarely occurring input conditions have not been tested during check-out.

In order for a person to rely with confidence upon any particular program, it is not sufficient for him to know that the program works most of the time or even that, so far, it has never made a mistake. The real question is whether it can be counted upon to fulfill its functional specifications successfully every single time. This means that after a program has passed the check-out stage, there should be no possibility that an unusual combination of input data or conditions may bring to light an unexpected bug in the program. Every portion of the program must be utilized during check-out in order that its accuracy may be confirmed.

The purpose of the work reported here has been to develop a systematic way in which a programmer may test all realistic combinations of input data, and hence all portions of a given program. Although at first this seems to be an arduous task, its handling is simplified by an orderly approach, and its reward is a greater assurance of the accuracy and reliability of the program. The potential number of test cases is given by 2^B, where B is the number of branch points in the flow chart. The computer could be programmed to generate all of these cases automatically. However, if the number of branch points is at all large, the total number of potential cases would become impractically large, though most of these would be unrealistic or identical to others of the cases, and hence unnecessary to test. The analysis set forth in this paper has as its object the determination by straightforward means of the full list of possible—as opposed to potential—test cases, their method of convenient generation, and their use in the location of specific errors in the program.

The approach taken here of using varied input conditions as the means to insure thorough testing of the program is particularly applicable to business problems, because the wide variety of input is one of the major aspects of such a problem. The method can be extended, however, to deal effectively with scientific problems, even though some of their branch points will not be dependent upon the input data. The procedure should, therefore, be useful as a method for systematic error analysis of computer programs irrespective of the type of program to be analyzed.

II. BACKGROUND

Probably the most common method of error detection is the running of a test case whose answers have already been calculated manually for comparison. Carefully chosen test data can be quite successful in pointing up the major errors. Other methods of error prevention that can be effective are (a) to code into the program certain checking devices, such as intermediate check sums, and (b) to write the code in such a way that it will be easy to check, even at the expense of certain sophistications and time-saving devices.[1]* It is always helpful to make a detailed manual check of the code as written prior to the first machine run. In addition to reviewing the coding, a second flow chart can be drawn, this time prepared directly from the coding, to determine whether the program executes its instructions according to the original plan.

An increasing number of check-out methods making use of the facilities of the machine itself are being employed. Some of the more common methods are manual step-by-step operation, dumping of memory, a tracing program with automatic skips, and the use of the break-jump switch, which will stop the program and jump to any specified location. Jacoby and Layton[2] have reported on an automated debugging program that runs the program to be tested, creating a trace record for later analysis by the debugging program itself. Lois Haibt[3] has described a very interesting program that enables the computer to draw multilevel flow charts directly from the coding of the problem. A particularly promising approach to checking is that reported by Senko.[4] The control system for logical block diagnosis that he describes will test a sub-routine or logical block of instructions as a separate entity, executing in one run all the tests necessary to explore each branch of the sub-routine. The significance of this method is that it allows the programmer to supply selected test data at different points throughout the program without requiring him to complicate his own program with the coding that would have been required for incorporating loaders into the original program.

III. NOTATIONAL ALTERNATIVES

As an aid to investigating the possibility of building a logically complete test deck for a given problem, various ways were considered by which the logical development of the problem might be displayed. In addition to the flow chart that is generally employed, a number of alternate methods of representation were discovered, including matrix, table, outline, logical tree, and symbolic path list. Karp[5] has discussed the application of graph theory to digital computer programming, and Voorhees[6] has taken a somewhat similar approach in considering an algebraic formulation of flow diagrams. Since the notation required by the latter methods differs slightly from the notation chosen here, no attempt will be made to illustrate their similarity to the other representative forms. Including

* See Literature Cited.

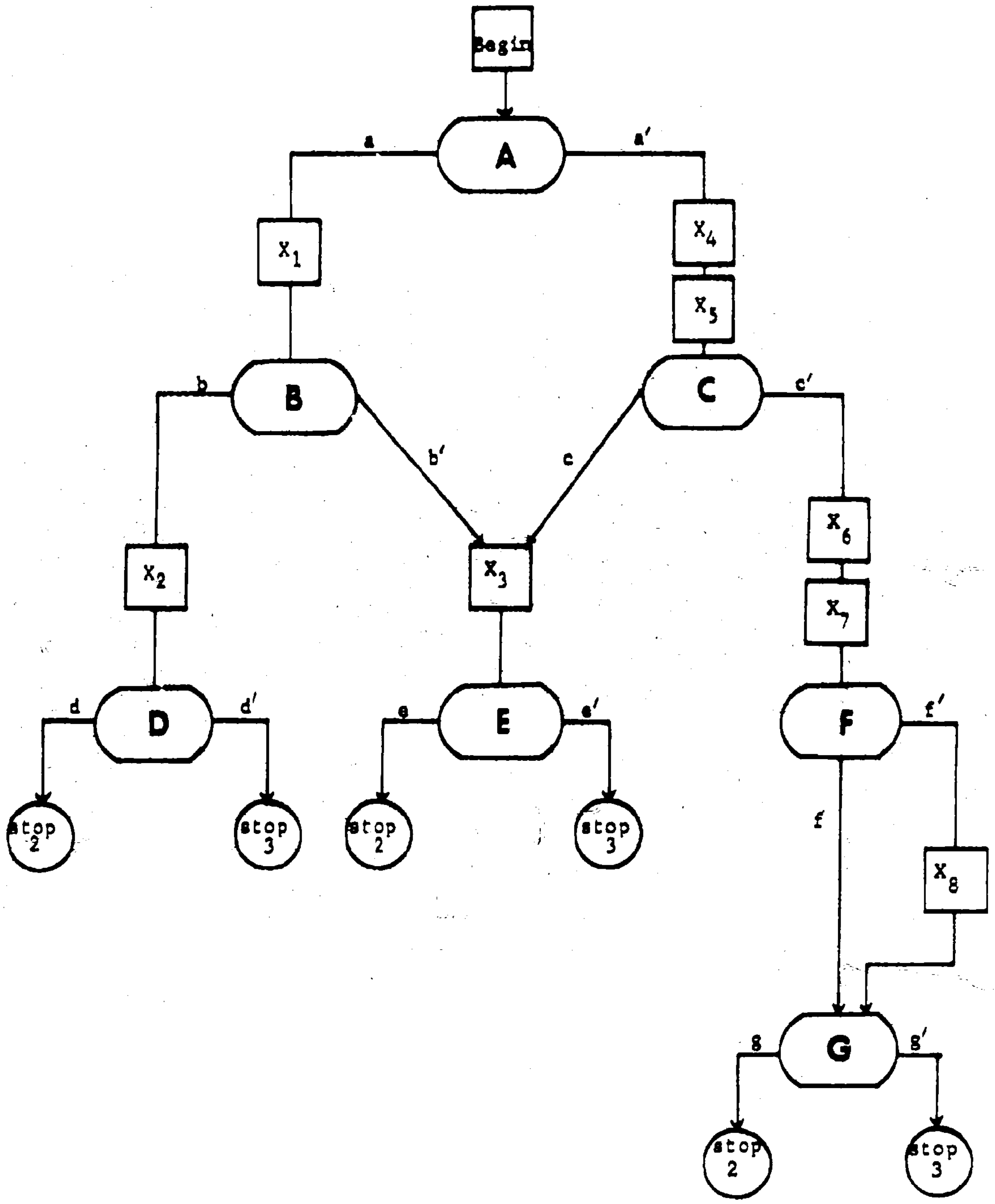

Figure 1. Hypothetical Flow Chart.

the flow chart, nine methods of representation will be described. It is possible to build those forms either directly from the original specification of the problem or from its flow chart. For the purpose of illustration, each of these models will be derived from an illustrative flow chart.

In the flow chart of Figure 1, the course of the program proceeds in sequential order from each step to the following step except at each branch point, where one of two available paths is selected according to the results of the decision at that point. The decision elements are shown as oval boxes and are designated by capital letters; the alternative results of the decisions are given by small letters. For example, if "A" symbolizes the decision element [is $x > y$?], then "a" indicates the path taken if $x > y$, and "a'" shows the path followed if $x \leq y$. Numerical calculations are represented by rectangular boxes.

Since the logical development of the problem is affected only when the program is required to select between two possible paths, it is only the decision elements that need concern us in our present analysis. For this reason, it is advantageous to consider representations of the problem flow that omit the descriptions of the numerical steps.

A modified flow chart from which the numerical calculations have been deleted is shown in Figure 2. The formulation is somewhat similar to the graph theory form used by Karp.[5]

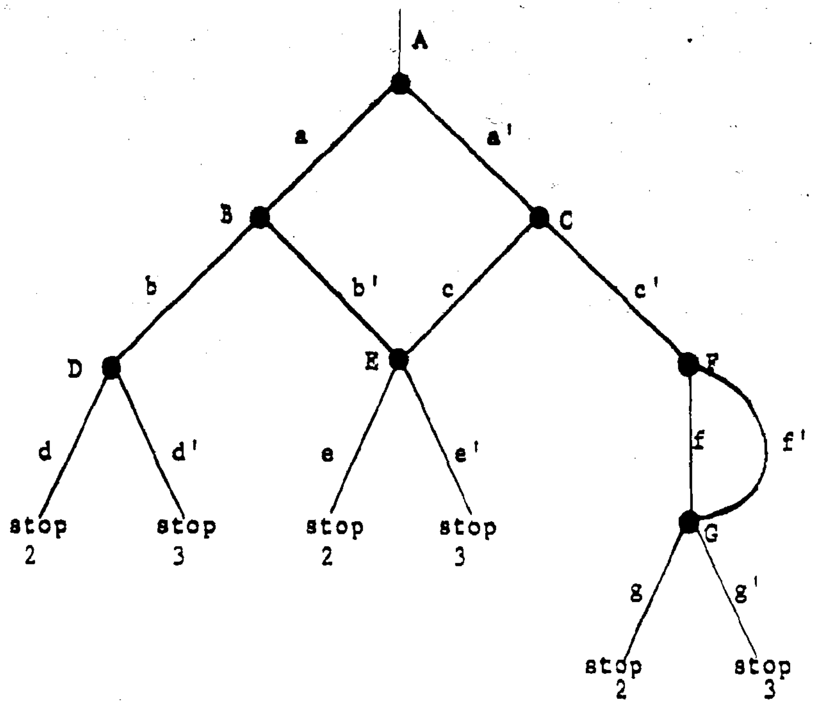

Figure 2. Modified Flow Chart, Omitting Numerical Calculations.

A logical tree is similar to the modified flow chart, with the distinction that a tree is defined as "a directed graph that has at most one branch entering each node and contains no circuits.7/" Since decision elements E and G can each be entered by two different paths in Figure 2, the results of those decisions each appear twice in the tree of Figure 3. Although the tree is slightly more cumbersome to draw, it has the advantage that each path is clearly visible, and the number of paths can be determined simply by counting the terminal nodes of the tree.

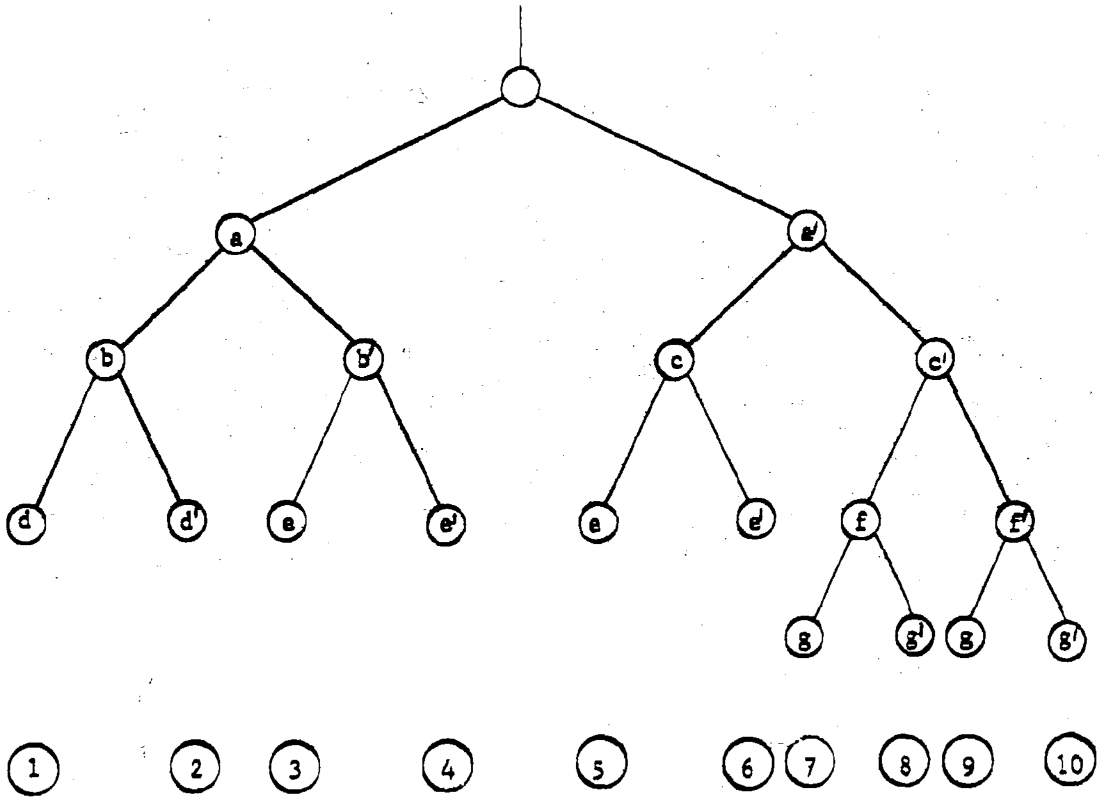

Figure 3. Logical Tree Corresponding to Flow Chart of Figure 1.

Outline form can also be used to depict the decisions (Figure 4). Each level of the outline is considered to be a particular decision element, and the two entries are mutually exclusive, in that exactly one of the two must be chosen. For example, decision element A is represented by the Roman numerals I and II. If the program does not select I, which is path ', then its only other choice is to select II, which is path "a'." At each step of the outline, the appropriate branch choice is indicated by its symbol corresponding to a path of the flow chart of Figure 1.

I.				a
	A.			b
		1.		d
		2.		d'
	B.			b'
		1.		e
		2.		e'
II.				a'
	A.			c
		1.		e
		2.		e'
	B.			c'
		1.		f
			a.	g
			b.	g'
		2.		f'
			a.	g
			b.	g'

Figure 4. Calculation Sequence of Figure 1 Shown in Outline Form.

In the table of Figure 5, the decision elements and their corresponding branch choices are listed sequentially by rows. A dash in any position indicates that there is no test at that point because of the impossibility of reaching the decision element for the row by the path selected in the preceding rows. Each column of the table then represents one particular path of program flow. This tabular representation can be used in connection with the Logic-Structure tables of the TABSOL System[8] of program analysis. Since the logical development of the program is completely designated by the top half of the TABSOL table in conjunction with the "go to" statements at the end of the columns, the third and fourth quadrants of TABSOL may be omitted when the tabular form is used for error analysis.

A	a	a	a	a	a'	a'	a'	a'	a'	a'
B	b	b	b'	b'	...	...	...	...	...	...
C	...	...	...	...	c	c	c'	c'	c'	c'
D	d	d'	...	...	...	...	...	...	...	...
E	...	...	e	e'	e	e'	...	...	...	...
F	...	...	...	...	...	...	f	f	f'	f'
G	...	...	...	...	...	...	g	g'	g	g'

Figure 5. Tabular Form of Sample Computation.

A symbolic path list (Figure 6) is simply a statement of each individual path that is a part of the program. It can be written directly from the flow chart or, even more quickly, from the tree, outline, or table.

1) abd
2) abd'
3) ab'e
4) ab'e'
5) a'ce
6) a'ce'
7) a'c'fg
8) a'c'fg'
9) a'c'f'g
10) a'c'f'g'

Figure 6. Symbolic Path List of Sample Computation.

The format of symbolic logic may be used to describe these paths in one statement, but the result seems to be too compact to be of much help in error analysis. Writing "v" for the logical "or" and the absence of a sign for the logical "and," this flow chart would reduce to

$$a[b(dvd')vb'(eve')]va'[c(eve')vc'(fgvfg'vf'gvf'g')].$$

A Boolean matrix (Figure 7), $A = (a_{ij})$, can be very helpful in describing the flow chart. This type of matrix consists entirely of 0's and 1's. In the <u>connectivity matrix</u>[9] for the flow diagram, a_{ij} will equal 1 if decision element j is the next consecutive decision element, and 0 otherwise. In this matrix, we will consider the decision elements to be represented, in alphabetical order, by the successive rows followed finally by Stop 2 and then Stop 3. (For the reader's convenience, the appropriate headings for rows and columns have been displayed with the matrix.)

$$\begin{array}{c|ccccccccc}
 & & & & & & & & \multicolumn{2}{c}{\text{stop}} \\
 & A & B & C & D & E & F & G & 2 & 3 \\
\hline
A & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
B & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
C & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
F & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
G & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{stop } 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{stop } 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 7. Boolean Matrix of Sample Calculation.

A limitation of use of the Boolean matrix, when considering decision elements only, is that two different paths from one decision element to the next one cannot be illustrated. Such is the case in our matrix for the two paths between F and G. In the matrix, all that is shown for these two paths is a single 1 in row F column G. A refinement of the Boolean matrix can be accomplished by replacing the 1's by the notation for the actual path followed, in a manner similar to that used by Karp.[5] In the path matrix, (Figure 8) the entry in row F column G is fvf', indicating that the program can arrive at G by one of two alternative paths.

$$\begin{array}{cc|ccccccccc}
 & & A & B & C & D & E & F & G & \text{stop } 2 & \text{stop } 3 \\
\hline
 & A & 0 & a & a' & 0 & 0 & 0 & 0 & 0 & 0 \\
 & B & 0 & 0 & 0 & b & b' & 0 & 0 & 0 & 0 \\
 & C & 0 & 0 & 0 & 0 & c & c' & 0 & 0 & 0 \\
 & D & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d & d' \\
 & E & 0 & 0 & 0 & 0 & 0 & 0 & 0 & e & e' \\
 & F & 0 & 0 & 0 & 0 & 0 & 0 & f v f' & 0 & 0 \\
 & G & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g & g' \\
\text{stop} & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 8. Path Matrix of Basic Computation.

IV. EFFECT OF LOOPS ON THE DEBUGGING SCHEME

Noticeably missing from our example flow chart (Figure 1) is any loop that causes a series of steps to be repeated. Such circuits occur frequently in flow charts, but their discussion has purposely been delayed until now. Before proceeding to their consideration, it is necessary to clarify the aims and corresponding requirements of our error analysis. The approach to be taken in forming a complete test deck is to consider conditions such that every path will be traversed at least once and, furthermore, that it will be entered from every possible entry point. If these conditions are satisfied, we shall know that the work performed along a path is correct and that the conditions at each of its entry points are the proper ones for successful completion of the path.

Under this premise, it would be unnecessary to go through each loop more than once, because a single traverse establishes its correctness. This means that if we have specified conditions to cause the program to exit once from each side of a branch point, it will be inconsequential to us whether the program subsequently loops back to a previous point in the program or goes to a stop. Consider, for example, that at branch point D of the flow chart in Figure 1, path d loops back to branch point A, rather than going to Stop 2. If we simply form a set of conditions that will cause the program to exit from D on path d, the program will automatically execute this loop. Furthermore, all of the paths it can take subsequent to this loop will be identical to ones we have already accounted for with the other sets of input conditions that we have planned. Ideally, then, we would wish to have a set of conditions that would cause the program to exit from D by path d' the first time it arrives at D, thus omitting the loop from its sequence of steps, and a second set of conditions that would send the program on path d through the loop exactly once before it exits from D by path d'. Then correct answers for the first set and incorrect answers for the second set of conditions will indicate that the loop itself is at fault, whereas incorrect answers for both sets will indicate that the error probably occurs on the path common to both sets before the point of entry to the loop or later in the program.

It is interesting to observe the effect that the introduction of the loop at D has on each of our forms representative of the flow chart. It is most readily shown in the two forms of the matrix (Figures 7 and 8), where in row D the entries at the intersection with columns H and A are simply interchanged. In the other forms of representation (because of the understanding that a loop may be considered covered, for purposes of debugging, if its path has been entered) it is not necessary to make any change whatever, for all steps preceding the loop branch were traversed in getting to it. The mere fact that path d appears as one of the paths in the representative form indicates that the loop will be entered. Hence, in the tree, path list, and outline, a path that leads into a loop will appear the same as one that leads directly to a stop, so far as the debugging requirements are concerned.

V. CHOICE OF MATHEMATICAL MODEL

Careful study of each of the forms of representation of problem flow suggests that the matrix and tree can be most helpful in error analysis. Further, the tree seems to be the simplest form for the programmer to use in the manual generation of a complete test deck, since it is similar to the original flow chart with which he is accustomed to dealing. By writing the program in terms of a tree with each loop, as well as each exit to a program stop, designated as a terminal node, the number of test cases required for exhaustive check-out can then be determined by simply counting the terminal nodes of the tree. The programming notation for trees presented by Iverson,10/ and the ideas used by Sauder11/ in his general test-deck generator for COBOL, encourage the possibility of mechanization of this method of error analysis. Matrix representation may be found preferable for machine use because of the ease with which a matrix can be stored in the machine and the availability of the laws of matrix algebra that would be advantageous in the analysis. Since it is a mechanical matter to translate from tree form to matrix form, the programmer could still do his part using the tree, and then ask the machine to translate the tree into a matrix and proceed with its operation.

VI. TEST-DECK GENERATION

The potential number of test cases would be generated by considering the effect of the interaction of each branch point with every other branch point in the program. This number will never become a reality, however, because it will always be possible to eliminate the consideration of the interaction of branch points between which there is no possible path, such as points E and F in the flow chart of Figure 1. The logical tree, drawn either directly from the original specification of the problem or from the flow chart, will portray the complete set of possible interactions among the branch points, each different combination of input conditions being represented by a distinct path from the initial node to a terminal node of the tree.

The mere drawing of the tree automatically simplifies the problem in three ways. It omits, for the purposes of test-deck generation, all numerical calculations, all paths that the program would ordinarily follow subsequent to entering a loop, and all unrealistic interactions between branch-point conditions. In addition, the tree itself may often be further simplified for this specific purpose by giving attention to branches for which the input would not differ from the input to some of the other branches, i.e., where the same test criterion is used two or more times in the program. For a set of branches having identical entry conditions, all but the earliest of these branches may be omitted from the tree, because test-deck data that will cause entry into that one branch will automatically cause entry into each of the others. For example, if Figure 1 depicts the flow chart for a business problem, the decision elements at A and E might both be testing whether the person is a salaried or an hourly employee, the paths a and e being the ones that will be chosen for an hourly employee. In such a case, input conditions that will cause exit from A by path a will also, without exception, cause exit from E by path e, so that the decision at E will always be dependent upon the decision at point A. This means that in determining test-deck specifications, paths e and e' may be deleted from the tree in Figure 3, thus leaving b' and c as terminal nodes and reducing the number of test cases by two. Such repetition of test conditions throughout a program is not uncommon, since it is often desired to inform the computer at different points in the program of the type of data being handled in order that appropriate computations may be performed. Before reducing the tree in this manner, however, one must make certain that the input conditions are not altered in some way during the course of the program, so that the input to the branches under consideration is not, in fact, identical. After simplification of the tree has been achieved, if the tree is still too large to be workable, it is helpful to postpone the consideration of the problem as a whole, in preference to working separately at first with significant portions of the tree.

When the working portion of the tree has finally been selected, test-deck generation is begun. Each terminal node of the tree stands for a different test case that should be run with the program for check-out. Test data are selected that will cause the program to arrive at each of these end points exactly once. In our tree, for instance, there are ten terminal nodes, indicating that there will be ten test cases for our problem. The conditions needed for any particular case are exactly those on the path from the beginning of the tree to the particular end point. Test Case 1 will contain data that complies with conditions a, b, and d; Test Case 2 will meet conditions a, b, and d', etc. (Figure 3).

VII. DEBUGGING

When a list of all possible input conditions is secured, a test deck containing an input card for each condition is prepared and run on the machine with the actual program. After running this test deck on the machine, comparison of the machine answers with ones that have previously been hand-calculated will indicate whether there were any errors and for which input data they occurred. Next, a quick glance at the tree will often give a very good idea of the location in the coding to begin searching for the error. For example, if the answers for test cases 3 and 4 were the only incorrect ones, the error is very likely to be somewhere in path b', whereas if 1, 2, 3, and 4 were wrong, the error is probably in path a. Likewise, if 3 and 5 are both wrong, but all others right, path e may be at fault because that is the only path common to these two test cases. The aid that this type of analysis gives to pinpointing an error is particularly advantageous to the programmer who seeks to find the error by examining a monitor sheet showing each step the program followed for a particular case. With the aid of the tree he knows where in the monitor to begin searching for the error, and if there are two errors that seem to have the same cause, such as 3 and 5, he can compare the monitor sheets for those two cases to determine whether the errors truly are of the same type.

If it is necessary to work with only a portion of the main tree, it will be found that after this portion has been thoroughly checked out, its test deck can be used in conjunction with other portions of the tree until the testing of the entire tree has been completed. The use of this systematic approach simplifies the work considerably. Since each test case will differ from the next only slightly, many of the hand calculations will not have to be repeated for each new case. It would be most valuable to choose the test data close to, or right at, the limits of the decision points in order to test whether the branching is effected exactly as it was meant to be.

VIII. CHARACTERISTICS TYPICAL OF SCIENTIFIC PROBLEMS

The method of test data formulation described was developed originally in connection with the checking of a business application of computer programming. The method and this type of problem are definitely best suited for each other. In a business problem, there is usually a wide variety of input, so that the combination of varied input conditions is one of the major problems in the debugging of the program. In addition, since most of the branch points are directly or indirectly influenced by the input data, a test deck with input data varied according to the method described in this paper will effectively test all paths of the program. In a scientific problem, on the other hand, the input data generally do not vary in this way, and the majority of the branch points are influenced not by the input conditions, but by the inherent properties of a problem. A loop may be executed a certain number of times before the program exits from it, and the number of times may not be influenced in any way by the input. For example, in computing $\sum_{i=1}^{10} x_i$, the loop that accomplishes this summation will be repeated exactly ten times before the program exits from the loop, regardless of how the input to the problem is varied. If, on the other hand $\sum_{i=1}^{N} x_i$ is desired, and N is variable, then the value of N will be included in the input data, and the frequency of the loop will be controlled directly by the input conditions.

For some scientific cases, the running of one test case will completely check out the problem. In such a case, there would be no point in using the tree method of analysis. For a problem that is to a certain extent influenced by the input, there are two possible methods of dealing with the loops that are independent of input in order that this method of analysis may be applied. One solution is to omit from the tree the decision elements for such loops. The test cases would then be limited strictly to the diverse combinations of input data. The assumption here would be that since the loops that have been omitted from consideration will automatically be executed in the process of the program, the answers to the test data will indicate whether these loops are correct and are traversed the proper number of times. A second possibility is to load with the test deck a set of pseudo-operations that will, for the sake of the test run, afford control over the branch points that are not normally dependent upon the input. In the loop for $\sum_{i=1}^{10} x_i$, for instance, a pseudo-operation that alters the branching instruction to test whether x_1 has been summed will cause the program to proceed immediately upon arrival at the branch point, and a second test case containing a pseudo-operation that tests whether x_2 has been summed will allow the program to repeat the loop exactly once before proceeding. The answers for the two test cases will determine, respectively, whether the summation procedure itself is correct and whether the looping is achieved successfully. The advantage of running two such cases, rather than checking the summation and the looping mechanism simultaneously by running only the second, is that, with two individual cases, the exact trouble spot in the program is immediately indicated by an incorrect answer to one of the test cases and a correct answer to the other.

With the addition of pseudo-operations, as described above, the method of error analysis using a logical tree is no longer limited to input-dependent decision points. If the complete test deck, including all test data and pseudo-operations indicated by the tree, is used for the checkout of the problem, there will be exactly as many test cases as there are terminal nodes of the tree. The incorrect answers that appear when the complete test deck is run will enable the programmer to pinpoint rapidly, using the tree as a guide, the specific points of the program that are in error. If the programmer prefers to spend less time amassing the original test deck, an alternative method would be to use only the various combinations of input data, omitting any employment of pseudo-operations for the first run. Then if errors appear for which he cannot determine the cause, he can, in a second run, use some pseudo-operations for the area of the program that he suspects to be at fault, in order to search more thoroughly for the error.

IX. CONCLUSION

The logical tree can be invaluable as an aid to sorting out the various combinations of input data that the program must be equipped to handle, testing these cases, and subsequently assisting the programmer in locating the cause of errors that occur. Systematic formulation of the test deck will eliminate the inadvertent duplication of test situations that often occurs in program check-out, and will significantly increase the number of different types of cases tested. It is, therefore, an effective aid in the formulation of a test deck that will contain, for a given problem, all combinations of input that can be expected to occur.

LITERATURE CITED

1. McCracken, D.D.: Digital Computer Programming, John Wiley and Sons, New York, 159-169, 1957.

2. Jacoby, K., and Layton, H.: "Automation of Program Debugging," paper presented at the 16th National Conference of the Association for Computing Machinery, Los Angeles, September 8, 1961. Abstract in Comm ACM, 4:7, 1961.

3. Haibt, Lois M.: "A Program to Draw Multilevel Flow Charts," Proc. Western Joint Computer Conference, San Francisco, March, 1959.

4. Senko, M.E.: "A Control System for Logical Block Diagnosis with Data Loading," Comm ACM, 3:4:236-240, 1960.

5. Karp, R.M.: "A Note on the Application of Graph Theory to Digital Computer Programming," Information and Control, 3:2, 1960.

6. Voorhees, E.A.: "Algebraic Formulation of Flow Diagrams," Comm ACM, 1:6:4-8, 1958.

7. Salton, G.: "Manipulation of Trees in Information Retrieval," Comm ACM, 5:2:103-114, 1962.

8. Kavanaugh, T.F.: "TABSOL, The Language of Decision Making," Comp and Aut., 10:9, 1961.

9. Prosser, R.T.: "Applications of Boolean Matrices to the Analysis of Flow Diagrams," Technical Report 217, MIT Lincoln Laboratory, Lexington, Mass., 1960.

10. Iverson, K.E.: "A Programming Notation for Trees," IBM Research Report RC-390, 1961.

11. Sauder, R.L.: "A General Test Data Generator for COBOL," Report presented at the AFIPS Spring Joint Computer Conference, San Francisco, May, 1962.

REFERENCES

1. Berkeley, E.C.: "Boolean Algebra and Applications to Insurance," reprint by Berkeley and Associates, New York, 1952.

2. Cantrell, H.N.; King, J.; and King, F.E.H.; "Logic-Structure Tables," Comm ACM, 4:6:272-275, 1961.

3. Copi, I.M.: Symbolic Logic, Macmillan, New York, 1954.

4. Curtis, H.A.: "A Generalized Tree Circuit," J ACM, 8:4:484-496, 1961.

5. Hickerson, R.C.: "An Engineering Application of Logic-Structure Tables," Comm ACM, 4:11:516-520, 1961.

6. Roth, J.P., and Wagner, E.G.: "Algebraic Topological Methods for the Synthesis of Switching Systems, Part III: Minimization of Non-Singular Boolean Trees," IBM J Res and Dev, 3:4:326-344, 1959.