or more. Create the appropriate entry in the Payroll Systems Table for such a program (label it PR65).

3. If the format of the Payroll Register were changed, which program in the system would have to be changed?

4. According to the Program-File cross-reference table (Figure 2), program PR40 may use files PR0001, PR20RM, PR4002, PR4004 and PR40RM. If program PR40 were run twice in a row without an intervening regular processing cycle, what would be in the file identification of the Reconciliation Master File used as input to the *second* running of PR40, i.e., PR20RM or PR40RM?

5. How many elements of data are in the Taxable Limits Report-PR5902?

6. If the element of data HIRE-DATE were deleted from the system, how many files would be affected?

7. Which Payroll System file could you identify from an unlabeled listing with only these four elements of data:

01 EMPNUMBER, 04 CURGROSS, 11 DEPT,
16 WORK-COMP?

8. According to the Element of Data-File cross-reference table (Figure 4b) for the Payroll Register, is there room for an element of data called SPECDIST? (SPECDIST is five digits long; you can assume three spaces between each element of data and two lines for each employee on the register—120 character lines.)

9. If HIRE-DATE (a class 3 element) were deleted from the system, which rules in the decision tables might have to be changed?

10. If an additional function were added to the file maintenance program and called INA for inactivate (opposite of activate), then how many rules would have to be added to Decision Table 1?

TEST ANSWERS

1. No. Check PR20 entry in the Systems Table (Figure 1b).

2. PR65 QUARTERLY OVER $3.00 PRXXEM PR6502 QR PRINTER.

3. PR20. In the Program-File cross-reference table (Figure 2), note that the Payroll Register file (PR2002) goes only to the printer.

4. PR40RM. After the first running of PR40, the Reconciliation Master file would remain PR40RM until PR20 was run again. Therefore, the input to the second running of PR40 would be the latest PRXXRM, or PR40RM.

5. Ten. (See Note 2, accompanying Figure 2).

6. Five file types are affected: A, G, O, U and V (see Figure 4a), but actually eight programs would need to be changed, because file type G includes four files (PR10EM, PR20EM, PR72EM, and PR90EM).

7. PR1003 or PP Labor Distribution Transactions. In the Element of Data-File cross-reference (Figure 4a), note that file 'F' is the only entry with just those four elements.

8. Yes. Count the positions required in Figure 4B [226 = 158+5+3 (21)].

9. Decision Table 1, rules 5, 8, 9, 10 and 12; Decision Table 2, rule 2; Decision Table 3, rules 2, 3 and 4; Decision Table 6, rules 3, 4 and 5. (Check the Element of Data-Decision Rule cross-reference table, Figure 5).

10. Two: INA INA
— —
Y N

Acknowledgments. Some of the ideas expressed in this paper depend on extensive work with tables accomplished by personnel of the United States Air Force Logistics Command and the Sutherland Consulting Company.

RECEIVED FEBRUARY, 1965; REVISED AUGUST, 1965

Programming Decision Tables in FORTRAN, COBOL or ALGOL

CYRIL G. VEINOTT
Reliance Electric & Engineering Company, Cleveland, Ohio

A simple broad-based approach for programming decision tables in FORTRAN or COBOL is developed and presented. With inputs in standard form, as defined in the paper, the programming of any decision table can be done with one or two FORTRAN statements, or with two COBOL statements, if the COMPUTE verb is available in the COBOL processor. It is shown that the method is applicable even when there are more than two mutually exclusive states of one, two or more table conditions. It is further shown that multi-state conditions in decision tables can often simplify the programming. The method outlined has the further advantage that all possible combinations of conditions are considered. It is shown that the suggested procedure is easily implemented in ALGOL.

1. Introduction

Much has been written in the literature about the merits of decision tables in expressing complex logic. A recent article by Kirk [1] points out some of these merits and gives an elegant method for programming a decision table. The subject is also discussed in a very recent paper [2].

In this paper a very simple and broad-based approach to this problem is developed for programs written in FORTRAN or COBOL. It is shown that any decision table can be programmed by two statements in FORTRAN II, or by a single one in FORTRAN IV, so long as the two conditions are expressed in a standard form. In COBOL, two statements are sufficient if the COMPUTE verb is implemented in the COBOL processor being used. In ALGOL, a switch serves the purpose.

The approach here has been extended to cover decision tables where each condition can have two or more mutually exclusive states. It is shown that such tables can also be programmed with equal ease, that is, with two statements in either FORTRAN or COBOL. Moreover, the use of a plural number of states of conditions leads to simpler tables and simpler programs than adherence to decision tables where all conditions are limited to two states.

Also, it is shown how the approach of this paper can readily be extended to ALGOL.

Nature of Simple Decision Tables (Two-State Conditions). Table I is a typical decision table. It is, in fact, the one used by Kirk [1]. This table shows three different conditions, and calls for four courses of action, as expressed by 4 "Rules," depending upon particular *combinations* of the specified conditions. Each condition, in this case, is repre-

sented by one of two possible states, yes or no, true or false.

We ignore the fact that there are really only two different courses of action in the case of Table I because, in the general case, there may be more.

As a rule, it may be said that a decision table is merely a convenient form for expressing a multiple branch where the particular branch to be followed is dictated, not by one condition, but by a *certain combination of a number of conditions*. Flowcharts for such a case can get very involved, and can be very difficult to follow; they also involve testing for each condition more than once.

2. General Approach

As we have seen, a decision table represents a multiple branch in a program, depending upon a set of specified conditions; there can be as many branches as there are possible combinations of conditions. FORTRAN provides for a multiple branch by means of a computed GO TO statement. COBOL likewise provides for a multiple branch with its "GO TO ··· DEPENDING ON ···" statement. In either language, the current value of the branching variable determines which branch the program follows.

The general procedure followed in this paper is to set up a system for calculating a unique number for each possible combination of conditions. The unique numbers must be an unbroken series of consecutive numbers so that they can be used as a branching variable.

The logic involved may be easier to follow if it is applied to the simple case of Table I before generalizing.

Programming Table I. Suppose in Table I, we denote

"credit limit OK"	by a value of 0 or 1
"pay experience OK"	by a value of 0 or 2
"special clearance"	by a value of 0 or 4

TABLE I. CREDIT APPROVAL: TYPICAL LIMITED-ENTRY DECISION TABLE EXAMPLE

Condition	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*
Credit limit OK	Y	N	N	N
Pay experience favorable		Y	N	N
Special clearance obtained			Y	N
Action				
Do approve order	X	X	X	
Do not approve order				X

TABLE II

Condition	*Value*	*Rules*							
		0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Credit limit OK	1		X		X		X		X
Pay experience OK	2			X	X			X	X
Special clearance obtained	4					X	X	X	X
Action									
Do approve order			X	X		X			
Do not approve order		X							
Corresponding Rule Number, Table I		4	1	2	1	3	1	2	1

Now, a new Table to replace Table I is constructed, adding a "Value" column. Since there can be $2^3 = 8$ combinations of three conditions, let us provide 8 columns, one for each combination. This has been done in Table II. Let these 8 columns be numbered from 0 to 7, inclusive, as shown. Now, let X's be put in these columns in such a way that the corresponding "values" add to give the number at the top of the column concerned. Now then, this procedure gives (a) identification to all eight possible combinations of conditions, (b) a unique number for each combination, obtained by the simple process of adding respective values for the three conditions, and (c) consecutive order to unique numbers.

Since the series contains a zero, we need to add 1 so as to be able to use this number as a branching variable.

Ordinarily we prefer to denote a yes or no invariably by a 1 or a 0; if this is done consistently there are less likely to be errors in the input. Suppose, in Table II, we denote

I1 = credit limit OK — 1 = yes 0 = no
I2 = pay experience OK — 1 = yes 0 = no
I3 = special clearance obtained — 1 = yes 0 = no
N1 = statement number (FORTRAN) or procedure name (COBOL) initiating action to "not approve the order."
N2 = statement number (FORTRAN) or procedure name (COBOL) initiating action to "approve the order."

Now then, the FORTRAN program for Table II is:

```
JUMP = 1 + I1 + 2*I2 + 4*I3
GO TO (N1,N2,N2,N2,N2,N2,N2,N2), JUMP
```

Note. There are 8 statement numbers inside the ( ) since JUMP may have any value from 1 to 8. In this case, seven of the statement numbers are the same, but this would not generally be true.

If FORTRAN IV is used the expression for JUMP could be written in place of JUMP in the GO TO statement, so that only one FORTRAN statement would be needed. N1, N2, etc., represent the numbers of the statement to which control is to be transferred.

Similarly, the COBOL program for Table II would be:

```
COMPUTE JUMP = 1 + I1 + 2*I2 + 4*I3
GO TO N1 N2 N2 N2 N2 N2 N2 N2 DEPENDING ON JUMP
```

Note. As in FORTRAN, it is necessary to provide 8 procedure numbers, to take care of the 8 possible values of JUMP, even though the same procedure name is used more than once.

If the COMPUTE verb is not available, the operations indicated have to be performed by using the available verbs. N1, N2, etc., are, of course, the specific procedure names to which control is to be transferred.

Note that Table I shows only four of the eight possible combinations of conditions, whereas all eight are specificaly shown in Table II. It can be said that Rule 1 of Table I, by ignoring two of the three conditions, "covers," at least by implication, four of the combinations shown in Table II. It may be convenient to represent four combinations by a single rule, but the safety of such a practice in the general case leaves something to be desired. The format of Table II forces consideration of every possible combina-

tion; if some of these happen to be meaningless, or indication of an error in inputs, a diagnostic could be printed out.

Now, the beauty and power of the decision table is that it permits a multiple branch, based upon a *combination of conditions*. When it is possible to branch upon a *single* condition, it is probably better not to include that condition in a decision table, but to branch upon it directly beforehand. For example, the "credit limit" condition of Table I might better have been left out of the decision table itself.

In general, the engineering analyst, or the procedures analyst, may give only the rules of interest, in no particular order, as done in Table I. The programmer then needs to compute the value of JUMP for each of the rules indicated. He must then add all the other possible combinations. It will help to avoid errors if he lists the combinations in order of the magnitude of JUMP.

General Procedure for M Two-State Conditions. For M two-state conditions, let

```
I1 (condition Number 1)    1 = yes   0 = no
I2 (condition Number 2)    1 = yes   0 = no
⋮                          ⋮         ⋮
IM (condition Number M)    1 = yes   0 = no
KC = 2**(M-1)
```

In FORTRAN programs, N1, N2, ⋯, NM would represent statement numbers to which control would be transferred. In COBOL programs, these would represent procedure names to which control would be transferred.

Number of possible "rules" or branches = 2^M. (1)

The first step would be to rewrite or develop the decision table with 2^M columns, so that each combination of conditions was identified, and provision made for it. These columns do not have to be arranged consecutively in order of magnitude of JUMP, but it is probably safer and more convenient to do so.

The FORTRAN program would be:

```
JUMP = 1+I1 + 2*I2 + 4*I3 + 8*I4 + ⋯KC*IM
GO TO (N1, N2, ⋯NM), JUMP
```

The COBOL program would be:

```
COMPUTE JUMP = 1 + I1 + 2*I2 + 4*I3 + 8*I4 + ⋯KC*
  IM
GO TO N1 N2 ⋯NM DEPENDING ON JUMP
```

The GO TO statement, in either FORTRAN or COBOL has to have 2^M statement numbers or procedure names but the same statement number or procedure name may be repeated as many times as necessary. Such repetition is illustrated above in the FORTRAN and COBOL programs for Table II.

What is the decision table-size limitation on programming this way? This would be determined by how large a GO TO statement would be allowed by the particular FORTRAN or COBOL processor (compiler) used. In general, large decision tables should probably be avoided, for they can eat up memory (see Section 4).

Conditions Represented by More Than Two States. It may be desirable to represent one or more conditions by more than two states. For example, still using the example of Table I, we may wish to delineate different dollar limits for which the credit is OK, e.g.,

Condition 1. *Credit is OK*
State 1—Under no condition
State 2—For any amount less than $10,000
State 3—For any amount of $10,000 or more.

Now, let there again be M conditions, the state of each of which is indicated by the value of variables I1, I2, ⋯, IM. Let the various conditions have K1, K2, ⋯, KM mutually exclusive states. To clarify the above, consider the table below.

Condition Number	*Represented by Variable*	*Values of the Variable for Different States*
1	I1	0, 1, 2, ⋯⋯,K1−1
2	I2	0, 1, 2, ⋯⋯,K2−1
⋮	⋮	⋮
M	IM	0, 1, 2, ⋯,KM−1

That is the conditions themselves are represented by the I variables; each of these I variables can take on different values, starting from 0, to express the state of this particular condition. The number of states of any condition depends in no way upon the number of states of any other condition.

Since the states, for any condition, are mutually exclusive, by definition, only one state can exist at a time for any given condition.

It can then be shown that the number of combinations or "rules" that exist will be

Number of rules = (K1) (K2) ⋯ (KM) = R. (2)

For convenience, let KNL equal the number of states of the next-to-the last condition.

Now then, the procedure in programming such a table is to set up R columns or rules and identify each combination. To each combination a statement number (FORTRAN) or a procedure name (COBOL) must be assigned; some of these can be repeated if need be and some may lead only to error print-outs, but all combinations must be identified.

The FORTRAN program would be

```
JUMP = 1 + I1 + K1*I2 + K1*K2*I3 + ⋯ K1*K2*K3*I4 +
  (K1*K2*⋯*KNL)*IM
GO TO (N1, N2, N3, ⋯,NR), JUMP
```

The COBOL program would be

```
COMPUTE JUMP = 1 + I1 + K1*I2 + K1*K2*I3 + ⋯
  K1*K2*K3*I4 + (K1*K2*⋯KNL)*IM
GO TO N1 N2 N3 ⋯NR DEPENDING ON JUMP
```

Illustration of the General Case of Multi-State Conditions. To illustrate application of the preceding, let us assume that there are three conditions, as follows:

Conditions	*States*	
1	3	:K1 = 3
2	4	:K2 = 4
3	2	:K3 = 2

So, R, the number of rules that have to be considered, is $R = 3 \times 4 \times 2 = 24$.

Now, lay out Table III to provide lines for each state of

each condition with 24 columns for 24 rules, as shown. Also, provide three more columns, as shown, headed:

Local Value - This would be the value presumably stored in the computer to denote the state for the particular condition.

Multiplier—This is the multiplier by which the local value is multiplied. It is 1 for Condition 1, K1 for Condition 2, K1 × K2 for Condition 3, etc.

Net Value—This is the net value of the particular state.

Now, in the last line of the table, enter numbers from 0 to 23 in successive columns.

Next, insert X's in each column so that the net values opposite the X's add up to give the total in the last line.

We now have listed in Table III every possible combination of all states of the three given conditions.

The FORTRAN program for this table would be:

```
JUMP = 1 + I1 + 3*I2 + 12*I3
GO TO (N1, N2, N3, ···, N24), JUMP
```

The COBOL program for this table would be:

```
COMPUTE JUMP = 1 + I1 + 3*I2 + 12*I3
GO TO N1 N2 N3 ···N24 DEPENDING ON JUMP
```

Multi-State Conditions Versus Two-State Conditions. It is clear that the 9 states in Table III could have been represented by 9 two-state conditions. Would this be simpler or not is a natural question.

With nine two-state conditions, from (1)

$$\text{Number of rules} = 2^9 = 512.$$

With three conditions of 3, 4 and 2 states respectively, from (2)

$$\text{Number of rules} = 3 \times 4 \times 2 = 24.$$

Clearly then, it would be impracticable to program Table III as nine two-state conditions, but it is quite practicable to program the nine states as they were done.

Perhaps a reader may ask, how does it happen that one way of setting up the problem gives 24 alternatives, whereas a different way leads to 512 alternatives? The answer is fairly simple. Condition 1 is actually represented by *three* mutually exclusive states, so there are only three valid alternatives to represent this condition. If these states were set up as three two-state conditions, there would be $2\times2\times2=8$ alternatives, only three of which are valid; the additional five invalid alternatives serve but to confuse and complicate the picture. A similar situation exists for Conditions 2 and 3, where treating them as more two-state conditions introduces invalid and unwanted alternatives.

Quite obviously then the lesson is clear: When a condition can be represented by more than two *mutually exclusive* states, it should be done!

Local Value Numbers of Conditions. It was assumed in Table III, for example, that the "local values" for each condition were already available in storage. If this is not the case, the local value or "net value" has to be determined prior to taking the steps outlined in this paper. For example, only the amount of the order might be given. Rather than have a human determine which state it represented in condition 1, the programmer would develop and store either the corresponding local or net value. In like fashion, some or all of the other conditions might have to be examined and programmed to calculate the appropriate state for that condition.

3. Extension of Techniques to ALGOL Programming

The techniques just described are equally applicable to programs written in ALGOL. The procedure would be:

1. Set up the decision table as discussed above.
2. In the ALGOL program declare a switch

```
SWITCH DT1 = N1, N2, N3, N4, etc.
```

3. The table is then implemented by

```
JUMP := 1 + I1 + 2×I2 + 4×I3 + ···
GO TO DT1 (JUMP)
```

TABLE III

	Local Value	*Multiplier*	*Equal Net Value*																								
Condition 1																											
State 1	0		0	X			X			X			X			X			X			X			X		
State 2	1	1	1		X			X			X			X			X			X			X			X	
State 3	2		2			X			X			X			X			X			X			X			X
Condition 2																											
State 1	0		0	X	X	X									X	X	X										
State 2	1		3				X	X	X									X	X	X							
State 3	2	3	6							X	X	X									X	X	X				
State 4	3		9										X	X	X									X	X	X	
Condition 3																											
State 1	0	12	0	X	X	X	X	X	X	X	X	X	X	X	X												
State 2	1		12													X	X	X	X	X	X	X	X	X	X	X	X
"JUMP" = 1 +				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

4. Notes on Decision Table Sizes

It is quite obvious that the number of rules goes up very rapidly with an increase in the number of conditions. When the size of a table threatens to get unmanageably large, a number of courses of action are available.

1. Take care not to include any condition that is independent of all other conditions in the table. That is, do not include a condition unless it has to be considered *in combination* with other conditions *in this same table.*

2. If two or more conditions in the table are mutually exclusive of each other, represent them by different states of the same condition.

3. When a choice exists, always use more than two states of a given condition, rather than adding new conditions.

4. Break the decision table up into more, smaller tables.

5. If a limit is reached because of length of GO TO statement, two or more GO TO statements can be used. In this case, the value of JUMP would have to be tested to determine which GO TO statement would be used; also the value of JUMP would have to be adjusted to suit the computed GO TO to which control is about to be transferred.

5. Summary and Conclusions

A powerful tool for programming decision tables has been developed and presented.

1. The method is simple and applicable to FORTRAN or COBOL.

2. Any number of conditions may be specified.

3. Each condition can have two or more mutually exclusive states.

4. Each of the conditions can severally have different numbers of states.

5. The standard method of representing states is, for each condition, 0, 1, 2, 3, etc.

6. The method automatically develops all possible combinations of states of conditions and requires provision to be made for all.

7. In either FORTRAN or COBOL, a single data word completely specifies the branching required by the decision table.

8. Only one or two statements are needed in either FORTRAN or COBOL (if COMPUTE verb is available) to program almost any decision table, when inputs are of standard form.

9. The only limits on the size of table that can be programmed by this procedure is set by the maximum length of GO TO statements that can be handled by the processor being used or by available memory.

RECEIVED AUGUST, 1965

REFERENCES

1. KIRK, H. W. Use of decision tables in computer programming. *Comm. ACM 8* (Jan. 1965), 41–43.
2. PRESS, L. I. Conversion of decision tables to computer programs. *Comm. ACM. 8* (June 1965), 385–390.

Letters to the Editor

Remarks on a Computer Program for the Construction of School Timetables

Dear Editor:

In an earlier letter Duncan [1] has reported the results of a number of runs on an IBM 7090 with a program written for the solution of the timetable problem as outlined by Gotlieb and Csima [2]. In a recent review of this communication, Broder [3] calls attention to the fact that the computer time required for an $N \times N \times N$ problem is proportional to 2^N and concludes that, despite the introduction of clever programming improvements, the method is likely to prove impractical.

In his program, Mr. Duncan employed a tight-set search algorithm, as described by Gotlieb and Csima, in the subroutine used to reduce availability matrices; this algorithm is very inefficient and we can be virtually certain that, among the improvements envisaged by Mr. Duncan, he included the replacement of this algorithm by an efficient one. Indeed, in following up the work of Mr. Duncan, we have done just this with the result that the computer time for a problem involving 18 teachers was reduced from approximately 75 minutes to a few seconds and the computer time for a case involving 43 teachers was approximately 3 minutes, not $2^{43\text{-}16} = 2^{27}$ hours as suggested by Broder. (These computer times refer to a program written by us and used on an IBM 7094.)

The work of Mr. Duncan has been of great value to us in the development of our present program which is designed to deal with real problems as presented by the secondary schools in Ontario. We are currently using this program on an experimental basis in cooperation with representatives from a number of schools, including one school with approximately 100 teachers.

REFERENCES:

1. DUNCAN, A. K. Further results on computer construction of school timetables. *Comm. ACM 8*, 1 (Jan. 1965), 72.
2. GOTLIEB, C. C., AND CSIMA, J. Tests on a computer method for constructing school timetables. *Comm. ACM 7*, 3 (Mar. 1964), 160–163.
3. BRODER, S. Review 7846. *Comp. Rev.* (July–Aug. 1965), 236.

B. A. GRIFFITH
J. Kates & Associates
Toronto, Ontario, Canada

On the Confusion Between "0" and "O"

Dear Editor:

I should like to describe briefly a technique which has been in use at the Lewis Research Center of NASA for approximately ten years for resolving the confusion between the mark 0 intended to mean zero and the mark O intended to mean the character between N and P in the Latin alphabet.

As applied to the management of identifiers and numerical values in assembler or compiler languages, it has worked without failure and does not require that human programmers differentiate between the similarly shaped symbols for zero and the letter "O".

(Continued on page 45)