THE CAUSES AND EFFECTS OF INFEASIBLE PATHS IN COMPUTER PROGRAMS

D.Hedley and M.A.Hennell

Liverpool Data Research Associates Ltd.,

Merseyside Innovation Centre,

131 Mount Pleasant,

Liverpool L3 5TF, U.K.

and

Department of Computational Mathematics,

University of Liverpool,

Liverpool L69 3BX, U.K.

ABSTRACT

An analysis is presented of infeasible paths found in the NAG library of numerical algorithms. The construction of program paths designed to maximise structural testing measures is shown to be impossible without taking infeasibilities into account. Methods for writing programs which do not contain infeasible paths are also discussed.

PREFACE

A major problem throughout the testing process is that of test data generation. The problem is to find values of input data so that the program will behave in a particular fashion. For example, given a part of a program that has not yet been tested, how can test data be obtained that will cause it to be executed?

In many cases, a knowledge of which code is to be executed will enable a tester to generate test data immediately from a knowledge of the requirements or specification. This paper discusses the problem which occurs when this test data generation is not obvious.

Resolution of this problem involves finding a path from the start of the program which can reach the desired part of the program, and then determining whether test data exists that will cause the program to execute that path.

The path will contain a set of predicates that must take particular values for the path to be executed. From these predicates it is possible to obtain a set of inequalities representing the conditions that must be met by the program input data to follow the particular path.

A path is feasible if it can be exercised by some set of input data. Otherwise it is called infeasible.

The great majority of program paths are found to be infeasible, thus making test data generation a very difficult task. An analysis is presented which describes the causes of infeasible paths which were found in a subset of the NAG Mark 7 Fortran library.

INTRODUCTION

Figure 1 contains a program that reads three integers and evaluates the type of triangle they produce, if any.

Before discussing this program further some definitions shall be introduced.

```
1        READ (5,100) J, K, L
2        WRITE (6,100) J, K, L
3  100   FORMAT (3I5)
4        IF (J+K.GT.L .AND. K+L.GT.J .AND. L+J.GT.K)
5      *                GO TO 1
6        WRITE (6,101)
7  101   (15H NOT A TRIANGLE)
8        STOP
9    1   MATCH = 0
10       IF (J.EQ.K)
11     *                MATCH = MATCH + 1
12       IF (K.EQ.L)
13     *                MATCH = MATCH + 1
14       IF (L.EQ.J)
15     *                MATCH = MATCH + 1
16       IF (MATCH-1) 2, 3, 4
17   2   WRITE (6,102)
18 102   FORMAT (17H SCALENE TRIANGLE)
19       STOP
20   3   WRITE (6,103)
21 103   FORMAT (19H ISOSCELES TRIANGLE)
22       STOP
23   4   WRITE (6,104)
24 104   FORMAT (21H EQUILATERAL TRIANGLE)
25       STOP
26       END
```

Figure 1

	start point	end point	target point
1	1	4	6
2	1	5	9
3	6	8	–
4	9	10	12
5	9	12	14
6	9	14	16
7	9	16	20
8	9	16	23
9	9	19	–
10	12	12	14
11	12	14	16
12	12	16	20
13	12	16	23
14	12	19	–
15	14	14	16
16	14	16	20
17	14	16	23
18	14	19	–
19	16	16	20
20	16	16	23
21	16	19	–
22	20	22	–
23	23	25	–

Figure 2

An **LCSAJ** (which stands for linear code sequence and jump) is defined as a linear sequence of executable code commencing either from the start of a program or from a point to which control flow may jump. It is terminated either by a specific control flow jump or by the end of the program. It may contain predicates which must be satisfied in order to execute the linear code sequence and terminating jump.

A **path** in a program or subroutine is defined as a unique sequence of executable statements in a program [10], i.e. any possible route from one point to another in that program.

An **infeasible path** is a path that can never be executed because contradictory predicates are required to be satisfied to execute the path.

The concept of an LCSAJ is a difficult one to comprehend and is perhaps best demonstrated by example. LCSAJs are language independent, although most languages need some degree of reformatting to allow LCSAJs to be expressed purely in terms of line numbers. In Fortran, the only construct requiring reformatting is the logical IF statement which must be split across two lines as in figure 1.

This program has 23 LCSAJs which are listed in figure 2.

LCSAJ start points are one of:

1. the start of the program (line 1),
2. lines containig labels which may be jumped to from other than the previous line (lines 9, 20 and 23), or
3. other lines to which control flow may jump from other than the previous line (lines 6, 12, 14 and 16). For example, if the logical IF statement on line 4 is false, control will jump from line 4 to line 6.

LCSAJs are much used in construction of program paths. A sequence of LCSAJs forms a program path if the target point of one is the same as the start point of its successor. For example, from figure 2, a path can be formed from LCSAJs 10, 16 and 22. A complete path through the program can be formed from LCSAJs 1 and 3.

Generating paths in terms of LCSAJs has been shown to be more efficient than the more usual methods of using basic blocks [12]. Using figures from [7], the average LCSAJ is about four times longer than the average basic block. The average number of successors to an LCSAJ is four, compared with two for a basic block. Hence, the cost of a depth – first search for paths of length n LCSAJs (or 4n blocks) is proportional to 4^n ($=2^{2n}$) when searching in terms of LCSAJs, compared with 2^{4n} searching in terms of basic blocks.

As can be seen from figure 2, several LCSAJs may share the same start point, end point or target point. Six LCSAJs start from line 9, each being terminated by a different jump. Although two of these linear code sequences end on the same line, they are followed by jumps to different points in the program. The longest linear code sequence is that from line 9 to line 19. This contains four IF statements, each of which allows control flow to pass on to the next line. However, this LCSAJ can be seen to be infeasible, as the outcome of the IF statement on line 16 is predetermined by the outcome of the previous three IF statements. The required outcome is impossible. There are, in fact, three infeasible LCSAJs (numbers 6, 7 and 9) and six others which, although apparently feasible, are infeasible in any combination when used to build complete paths through the program (numbers 12, 13, 14, 17, 18 and 20).

Returning to the problem of test data generation, if a particular section of code or branch or LCSAJ has not been executed, the (shorter) paths to it can be produced easily and automatically, together with all the predicates and statements in which controlling variables have their values changed. Usually, however, the complexity of this information is such that there is only limited success in finding data to satisfy all the predicates in the path by inspection. In fact, it has been shown that this data is unlikely to exist. A pilot study using shortest paths in terms of LCSAJs is described in [13]. Most paths were found to be infeasible, i.e. no test data exists such that the flow of control in the program would be taken down that path.

If the shortest paths are infeasible there is a problem called the "path explosion problem". Looking at the shortest complete paths in a routine or the shortest paths to a particular point inside the routine, the number of paths has an exponential relationship to their length [13].

In considering an extreme case involving a routine which sorted a vector of numbers, apart from the trivial cases of no numbers and one number to be sorted, the authors of [13] found that the next 5,000 shortest paths were all infeasible. In general, for most routines, finding 20 of the shortest 1,000 paths to be feasible was considered good. One of the main problems of structural testing to a high level is to remove such infeasible paths from consideration. It can be seen that it is not easy to obtain data to perform a comprehensive structural test on a program without mechanical help.

At the basic level many LCSAJs are themselves infeasible in that no data can cause the execution of every statement in the LCSAJ along with the final jump. To show that an LCSAJ is feasible, its predicates must be listed, represented in terms of its input variables and a solution must then be found to satisfy them.

We feel intuitively that LCSAJs should be feasible. If a section of program code contains sequences of conditional statements such that program control flow seems to be able to pass onto the succeeding line at all these conditionals, then one would hope that this would actually be possible. If this is not possible, then there exist one or more infeasible LCSAJs in the code. These are often due to poor coding practices (programs can always be rewritten to remove the infeasibilities [3]). They have also been found to exist due to "patching" errors, and creating new errors in the process [6]. Many LCSAJs are found to be infeasible due to the number of times a DO loop is executed. This is governed by the DO loop indices. Where a program contains more than one DO loop, their relative execution counts are often interlinked.

RESULTS

The results described here are obtained from a system which generates test data for programs written in ANSI standard Fortran.

This system is part of the "Fortran Testbed" suite of programs which was designed and written by the authors and their colleagues at Liverpool. Paths are input to the test data generation system and are symbolically executed in order that the predicates to be satisfied are expressed solely in terms of input variables to the path. Test data generation is then performed on these predicates. The process reveals many infeasible paths automatically – in fact, over 90% of the infeasible paths discussed below were found in this way. The others were a subset of those paths for which data was not found, and required some manual checking.

The routines used to produce these results are taken from the NAG library. This is a library of mainly numerical software which is used in all British universities and at many other sites in Britain and abroad [4].

All NAG code conforms to a portable subset of ANSI standard Fortran and is generally acknowledged to be of high quality.

A selction of 88 routines were examined covering most sections of the library. These routines contained 4,790 LCSAJs, an average of 56 per routine.

INFEASIBLE LCSAJS

619 infeasible LCSAJs were found – 12.5% of the total.

Why do infeasible LCSAJs occur in programs? There are several basic main reasons. The 619 infeasible LCSAJs have been analysed and an attempt made to divide them into these categories. It should be noted, however, that many of the actual infeasibilities in each category are much more complicated than the straightforward examples given below for illustrative purposes.

a) 326 (52.7%) are due solely to the number of times that loops must be executed. 209 of these (33.8% of the total) involve loops in combinations, e.g.

```
11        DO 20 J=1,N
12        DO 40 K=1,N
          . . .
16     40 CONTINUE
17     20 CONTINUE
```

where an infeasible LCSAJ would start at line 11, pass through the "K" loop once, and jump back from line 17 to line 11.

As a further example,

```
11          DO 20 J=1,N
12          DO 40 K=1,N
            . . .
16       40 CONTINUE
17       20 CONTINUE
```

also contains infeasible LCSAJs. Upon first execution of the "J" loop, the "K" loop must only be executed once. On subsequent executions of the "J" loop, the "K" loop must be executed more than once.

Thus an LCSAJ starting before line 11 cannot terminate with the jump from line 16 to line 12.

The other 117 (18.9%) infeasible LCSAJs involve single loops with constant (or calculated) bounds. For example,

```
      DO 20 J=1,10
      . . .
   20 CONTINUE
```

or

```
      N = 2
      DO 40 J=1,N
      . . .
   40 CONTINUE
```

where the loops must be executed a fixed number of times.

It is unlikely that anyone would take exception to these constructs appearing in programs. It may be argued, however, that the infeasible LCSAJs generated by the simpler of these cases, especially that of a loop with constant bounds, should be excluded from the list of LCSAJs and should never pass through this system.

b) 79 (12.8%) are due to loops which contain, or are associated with, conditions on the "loop variables", e.g.

```
      DO 20 J=1,N
      IF (J.EQ.1) . . .
   20 CONTINUE
```

where the first execution of the loop cannot be followed by the condition J.EQ.1 being false, and further executions cannot be followed by the condition being true.

As a further example, consider

```
      DO 20 J=1,N
      DO 40 K=1,N
      IF (K.LT.J) . . .
   40 CONTINUE
   20 CONTINUE
```

where the first execution of the "J" loop cannot be followed by the condition K.LT.J being true.

These cases could be rewritten to remove the infeasibilities but not in a way that would greatly improve the readibility and understanding of the program.

Again, there are good reasons for programming these constructs, and it is possible that some of these infeasible LCSAJs could also be weeded out at an early stage.

c) 143 (23.1%) are due to testing a value immediately after it has been set.

17 of these occur in conjunction with loops, e.g.

```
      M = 0
      DO 40 J=2,N
      IF (M.EQ.2) . . .
      IF (. . .) M = M + 1
   40 CONTINUE
```

This code is more confusing but difficult to write in an improved manner.

The other 126 in this category can be rewritten more easily. For example,

```
      K = 0
      IF (. . .) K = 4
      IF (K.NE.0) GO TO 220
```

can be rewritten as

```
      K = 0
      IF (.NOT.(. . .)) GO TO 10
      K = 4
      GO TO 220
   10 CONTINUE
```

which contains no infeasible paths and will also be faster at execution time.

Often various error exits are trapped in the form

```
      ERROR = 0
      IF (. . .) ERROR = 1
      IF (. . .) ERROR = 2
      IF (. . .) ERROR = 3
      IF (ERROR.GT.0) GO TO 300
```

where the variable ERROR is a parameter to the error handling routine called at label 300. Having satisfied one of the first three logical IFs, it is not then possible for the fourth logical IF to be false.

The six routines tested in the GO2 chapter of the NAG library were the only routines to contain code of this form, producing a total of 52 infeasible LCSAJs. This form of coding would appear to exist purely as a matter of personal taste by the authors or coordiantor of this chapter.

An alternative form

```
ERROR = 1
IF (. . .) GO TO 300
ERROR = 2
IF (. . .) GO TO 300
ERROR = 3
IF (. . .) GO TO 300
ERROR = 0
```

which produces no infeasible paths, is more commomnly used by other contributors. It does still, however, contain data flow anomalies.

Perhaps the form

```
IF (. . .) CALL ERREXT (1)
IF (. . .) CALL ERREXT (2)
IF (. . .) CALL ERREXT (3)
```

would be the best representation.

As a final example in this section, consider the first few lines of a sorting routine:

```
    K = 0
    IF (II) 20, 20, 40
 20 K = 1
 40 IF (JJ) 600, 600, 60
 60 IF (K) 620, 80, 620
 80 K = 4
    J = II - JJ
    IF (J) 100, 640, 620
100 M = 1
    . . .
```

As it is possible to go from every line to its successor, an LCSAJ exists from the start to the end of this section of code and continues down the program.

However, control flow is prevented from executing the code sequence by the assignment K = 1 and the subsequent test on the value of K.

This code is very confusing. It is very difficult to read and understand and must be considered an example of bad programming. It can be rewritten to remove all infeasible LCSAJs whils still retaining the arithmetic IF style, giving a clearer form:

```
    IF (II) 20, 20, 60
 20 K = 1
    IF (JJ) 600, 600, 620
 60 K = 0
    IF (JJ) 600, 600, 80
 80 K = 4
    J = II - JJ
    IF (J) 100, 640, 620
100 M = 1
    . . .
```

It is, of course, possible to further improve the legibility of the code by turning some arithmetic IFs into logical IFs:

```
    IF (II.GT.0) GO TO 60
    K = 1
    IF (JJ.LE.0) GO TO 600
    GO TO 620
 60 K = 0
    IF (JJ.LE.0) GO TO 600
    K = 4
    J = II - JJ
    IF (J) 100, 640, 620
100 M = 1
    . . .
```

Readers may decide for themselves whether this is a better program than the original.

d) 57 (9.2%) are due to consecutive conditions, where the path taken in the first condition controls the path through the second, e.g.

```
IF (MAX.GT.Q) . . .
IF (MAX.LE.Q) . . .
```

These are essentially due to a deficiency in the version of the Fortran language used in these particular programs, the "IF ... THEN ... ELSE" construct not being available. The most recent standard for Fortran includes this construct.

One case of this type was found where two disjoint conditions in the logical IF statements, if true, caused the same assignment statement to take place:

```
IF (JNS.EQ.10) RANMIN = 1.5*RANMIN
IF (JNS.EQ.11) RANMIN = 1.5*RANMIN
IF (JNS.GE.12) RANMIN = 2.0*RANMIN
```

This could obviously be improved.

Further examples in this section are due to repeated conditions in sequence, e.g.

```
IF (MAX.GT.Q) A = B
IF (MAX.GT.Q) C = D
```

Again, the basic cause of these is a deficiency in the version of the Fortran language used, but they could be written in a better fashion as e.g.

```
      IF (MAX.LE.Q) GO TO 20
      A = B
      C = D
   20 CONTINUE
```

This rewriting removes the infeasible paths and only requires one condition to be evaluated at runtime instead of two.

As a further example, consider the following seven lines of code taken from a routine in chapter D04 of the NAG library:

```
      IF (NEGER.GT.2) EREST = -EREST
      IF (NEGER.GT.2) GO TO 340
      IF (TERM.LT.RANMIN) EREST = -EREST
      IF (-TERM.GE.RANMIN) EREST = -EREST
      IF (EREST.LT.0) NEGER = NEGER + 1
      IF (EREST.GE.0) NEGER = 0
  340 CONTINUE
```

This contains a total of 34 paths, of which only 9 are feasible. Would not

```
      IF (NEGER.LE.2) GO TO 330
      EREST = -EREST
      GO TO 340
  330 ABSTRM = ABS(TERM)
      IF (ABSTRM.LE.-RANMIN .OR.
    * ABSTRM.LT.RANMIN) EREST = -EREST
      NEGER = NEGER + 1
      IF (EREST.GT.0) NEGER = 0
  340 CONTINUE
```

be a better representation? This contains only five paths, all of which are feasible.

The original code is particularly confusing as the conditions TERM.LT.RANMIN and -TERM.GE.RANMIN can both be true. If so, the sign of EREST is reversed twice, and EREST retains its original value!

e) the final 14 LCSAJs (2.3%) were too complicated to decide which, if any, of these categories they fall into.

Most of the routines looked at had some infeasible LCSAJs. Only 30 out of 88 (34.1%) had no infeasible LCSAJs. The others were distributed:

% of infeasible LCSAJs	No. of routines
less than 10	28
10 - 20	15
20 - 30	11
30 - 40	3
40 - 50	0
50 - 60	1

POINTS ARISING FROM INFEASIBILITY OF LCSAJS

Consider paths of greater length than 1 LCSAJ. From this analysis, 4,351 out of 4,970 LCSAJs (87.5%) are actually feasible. So the possibility of a path of length 1 LCSAJ being feasible is roughly 0.875. The chance of a path of n LCSAJs containing only these feasible LCSAJs is 0.875^n. For a path of length 10 LCSAJs, this value is 0.264. For a path of length 50 LCSAJs, this value is $1.29*10^{-3}$, and for a path of length 100 LCSAJs, this value is $1.67*10^{-6}$. In other words, if a program path containing 100 control flow jumps was constructed by taking an arbitrary route at each decision point, there are less than two chances in a million that the path will be feasible. In the other 999,998 cases, it will not be possible to execute the path.

These figures demonstrate the problem caused by infeasible LCSAJs in the construction of program paths. It must also be pointed out that these figures are over-generous in that they do not take account of infeasible combinations of LCSAJs. For example, a path of length 2 LCSAJs, both of which are feasible on their own, may itself be infeasible. That is because some assignments in the first LCSAJ may make certain conditions in the second LCSAJ infeasible. Many LCSAJs have also been found which are, in themselves, feasible, but which are infeasible in any combination with others, as in the triangle program discussed earlier.

Surprisingly little mention of the problem of infeasible paths appears in the literature. Various systems exist, and have been described, e.g. [9,11] which produce the "best" path available through a routine to execute every branch, say. This is an interesting theoretical problem but in practice the solution would seem to be of little value, as the path produced would almost certainly be infeasible. [5,10] have produced algorithms for generating paths taking into account infeasible pairs of conditions. A practical implementation of this is described in [8], but was later abandoned as being too hard to use.

Figures obtained for the number of infeasible paths in practice [13] bear out these problems. In a sample of 6 programs from the NAG Algol 68 library, the authors of [13] found one for which every path was feasible (the only non-trivial program they have discovered with this characteristic), but for the other five, an average of only 18 of the 1,000 shortest paths were feasible.

The figures show that the construction of program paths is a very hazardous business, and that maximising structural testing measures by the execution of predetermined paths is not as easy as theoretical papers would have us believe.

From the results described in this paper, it would appear that a good path generation system would be able to ignore paths containing many of the infeasibilities either by detecting certain infeasible patterns in the original code or by incorporating an analysis of the path constraints similar to that described here.

A POSSIBLE SOLUTION

It is possible to write software in such a way that all paths are feasible. A survey by Brown and Nelson of TRW [3] of this style of programming, called "Functional Programming", shows that it has potential for improving the reliability, maintainability and performance of programs.

The functional programming method involves partitioning the input domain of a program into a number of distinct subsets associated with program logical structures and with functional requirements. Functional programming produces programs having simpler structure, with fewer paths and fewer executable statements. A functional program is thus easier to read and understand. The authors claim that the simplicity of the structure results in a more efficient program containing fewer errors.

Functional programming was originally motivated by testing concerns. The use of testing tools to thoroughly test the structure of a program made it clear that much effort was wasted in trying to execute infeasible paths. Functional programming helps in selecting input values for program test cases and in defining a set of test cases to maximise all the TER measures of structural testing [2,13].

As an example of functional programming, consider again the triangle program of figure 1. Apparently a straightforward program, it contains 23 LCSAJs and 25 paths from start to end. Of these, however, only 14 LCSAJs and 6 paths are feasible.

The program is muddled and over complex. It contains code which would be used if two pairs of triangle sides were equal in length, but the third pair was of different length. When line 16 is reached, there appears to be a choice of branches, but by this stage of execution the value of the arithmetic IF has been determined solely by the route that execution has taken up to that point. There is no choice of branches – the route to the end of the program has already been determined.

This program could be rewritten as a functional program as in figure 3. This program contains four complete paths, all of which are feasible. It is shorter, easier to understand, and easier to test.

A more efficient functional program could be written to eliminate the need to evaluate any equality twice, either by using logical variables or by altering the structure of the program slightly.

```
      READ (5,100) J, K, L
      WRITE (6,100) J, K, L
  100 FORMAT (3I5)
      IF (J+K.GT.L .AND. K+L.GT.J .AND. L+J.GT.K)
     *                  GO TO 1
      WRITE (6,101)
  101 FORMAT (15H NOT A TRIANGLE)
      STOP
    1 IF (J.EQ.K .AND. K.EQ.L) GO TO 3
      IF (J.EQ.K .OR. K.EQ.L .OR. J.EQ.L) GO TO 2
      WRITE (6,102)
  102 FORMAT (17H SCALENE TRIANGLE)
      STOP
    2 WRITE (6,103)
  103 FORMAT (19H ISOSCELES TRIANGLE)
      STOP
    3 WRITE (6,104)
  104 FORMAT (21H EQUILATERAL TRIANGLE)
      STOP
      END
```

Figure 3

Brown and Nelson found their experiences with functional programming to be positive and encouraging, and compare the advantages to be gained from the technique to those gained from structured programming in the 1960s. They claim that an investment in functional programming principles would bring truly reliable and highly maintainable software within a short time.

Perhaps future generations will be encouraged to program in this way.

For the present, however, testing is beset by problems of logical infeasibilities which even structured programming principles seem to do little to help.

Other than the results of small investigations by the authors [6,13], there appear to be no references in the literature describing why program paths are infeasible. Once the answer to this question is known, the building of a system to find infeasibilities automatically should be an easier problem, since many types of infeasibility could be explicitly stated and looked for.

SUMMARY

The experimentally determined infeasibilities of LCSAJs (and hence program paths) reported in this paper arise from three main sources:

1. The number of times loops are executed. Loops are recognised as a major source of problems in structural testing, although our analysis shows that many loop bounds can be analysed quite easily to remove infeasible paths.

2. Inadequate language constructs. It cannot be a surprise to the world at large that the control structures of the 1966 version of Fortran lead to programmers using tortuous constructs of their own devising.

Nevertheless, readers with a tendency to dismiss these conclusions as of limited value are cautioned that many similar problems arise in the control structures of well-structured languages – in fact, six of the seven programs analysed in [13] are written in Algol 68. This suggests that all today's major languages suffer from these defects.

3. Poor coding style. In almost all cases when infeasibiities are discovered, it is possible to rewrite the code in a clearer style which executes more efficiently. It is interesting to note that a structured programming style does not necessarily minimise these problems – indeed, it may well increase them. The only clearly superior programming style is that of Functional Programming of Brown and Nelson [3]. This style is related to the "programming without assignment" style [1]. However, the development of programs with tree-like control structures requires careful control of the predicates.

ACKNOWLEDGEMENT/APOLOGY

The authors wish to acknowledge the Science Research Council and Liverpool Data Research Associates for providing grants enabling this work to be carried out.

We also wish to acknowledge/apologise to NAG. It is our sincere hope that readers will not construe this paper as in any way being a criticism of the NAG library. Our experiments with NAG routines have convinced us that the care and work which have gone into them are apparent by the high quality and reliability of the library.

REFERENCES.

1. J.Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ACM, Vol. 21, No. 8, pp. 613–641, August 1978.

2. J.Brown, "Practical Applications of Automated Software Tools", TRW Report TRW-SS-72-05, presented at WESCON, 1972.

3. J.R.Brown and E.C.Nelson, "Functional Programming Final Test Report", TRW Defense and Space Systems Group for Rome Air Development Center, July 1977.

4. B.Ford and D.K.Sayers, "Developing a Single Numerical Algorithms Library for Different Machine Ranges", ACM Transactions on Mathematical Software, Vol. 2, No. 2, pp. 115–131, June 1976.

5. H.N.Gabow, S.N.Maheshwari and L.J.Osterweil, "On Two Problems in the Generation of Program Test Paths", IEEE Transactions on Software Engineering, Vol. 2, No. 3, pp. 227–231, September 1976.

6. D.Hedley, M.A.Hennell and M.R.Woodward, "On the Generation of Test Data for Program Verification", Conference on Production and Assessment of Numerical Software, eds. M.A.Hennell and L.M.Delves, Academic Press, pp. 127–135, 1980.

7. M.A.Hennell and J.A.Prudom, "A Static Analysis of the NAG Library", IEEE Transactions on Software Engineering, Vol. 6, No. 4, pp. 323–333, July 1980.

8. R.H.Hoffman, "The Impossible Pairs Detection Capability (IMPAIR) of the Automated Test Data Generator (ATDG)", TRW Defense and Space Systems Group for National Aeronautics and Space Administration, January 1977.

9. J.C.Huang, "An Approach to Program Testing", ACM Computing Surveys, Vol. 7, No. 3, pp. 113–128, September 1975.

10. K.W.Krause, R.W.Smith and M.A.Goodwin, "Optimal Software Test Planning Through Automated Network Analysis", Proceedings of IEEE Computer Software Reliability Symposium, New York, pp. 18–22, April 1973.

11. E.F.Miller and R.A.Melton, "Automated Generation of Testcase Datasets", Proceedings of International Conference on Reliable Software, Los Angeles, pp. 51–58, June 1975.

12. M.R.Woodward, M.A.Hennell and D.Hedley, "The Analysis of Control Flow Structure in Computer Programs", Proceedings of CP77 Combinatorial Programming Conference, Liverpool, September 1977, ed. T.B.Boffey, pp. 190–202, 1977.

13. M.R.Woodward, D.Hedley and M.A.Hennell, "Experience with Path Analysis and Testing of Programs", IEEE Transactions on Software Engineering, Vol. 6, No. 3, pp. 278–286, May 1980.