AN APPROACH TO BRANCH TESTING

D. P. Yates and M. A. Hennell
Departments of Computer Science
and Statistics and Computational Mathematics
University of Liverpool,
Liverpool L69 3BX
U.K.

ABSTRACT

A method for automatically generating program paths for the purpose of branch testing software is proposed. The method, which utilises graph theoretic techniques, embodies a strategy for optimising on the overall cost of the branch testing exercise by reducing the potential incidence of infeasible paths in the path set that is generated.

1.0 Introduction

The most widely used method for validating a program unit, P, is that of exercising P with a suitable set of test data. As a rule it is not an attainable goal to devise a test data set that will lead to the detection of all sources of error in P. Fortunately, however, there exist strategies for devising data subsets which, by exercising certain program features, will give some degree of confidence in the code. One of the oldest of these strategies is branch testing and, as evidenced by the literature, see for example [1-3], many systems that utilise it have been developed. Although Howden [4] has demonstrated its shortcomings in locating certain types of error, many practitioners agree that the branch testing strategy, especially when used in tandem with other testing strategies, is indeed a profitable one.

Branch testing requires that a set of paths through P be chosen such that every branch of P lies on at least one of the selected paths. A test data set is then derived which, on application to P, drives its execution, in turn, down each path in the chosen path set. Branch testing, therefore, is composed of three steps:

(i) the selection of a path set π,
(ii) the derivation of the corresponding data set D_{π},
and (iii) the execution of P with D_{π}.

Software testing is a high-cost exercise and, consequently, it is important to ensure that any testing strategy employed will achieve its objectives at minimum cost. The two features that characterise the cost of branch testing are:

(A) the number of infeasible paths generated at step (i)

and (B) the level of complexity of the path predicates, derived as a result of step (i), which must be processed in order to perform step (ii).

As used here the term predicate is taken to mean a program statement involving a test with two or more outcomes such that each different outcome defines a different flow of control through P.

The immediate implication of generating infeasible paths is that branch testing will not be a one-pass process and, in essence, will require the iterative execution of steps (i) and (ii) an unpredictable number of times. This situation clearly militates against low cost especially when account is taken of the technological problems involved in performing step (ii). It is not difficult to appreciate that the higher the level of path predicate complexity involved, the greater will be the time and cost overheads of performing step (ii). However, to put this a little more in perspective, it is instructional to note that no wholly automated tool for performing branch testing has ever been developed, and that a principle reason for this is high path predicate complexity. It may be concluded therefore, that in order to keep the cost overheads of branch testing to a minimum, it is appropriate to seek an approach which minimises, a priori, the effects of (A) and (B) above. The purpose of this paper is to propose a strategy for generating the path set π which attempts to go some way towards achieving this aim and to describe a method for implementing this strategy. The method, which is based upon graph theoretic techniques, and the underlying strategy are introduced in section 2 of the paper. In section 3, salient features of the method are discussed and in section 4 the results of applying it to selected code segments are presented.

2. The Path Generation Strategy

As indicated in section 1, the hinge-pin of the current proposals for attempting to minimise the cost of branch testing is a strategy for generating π. The justification for this strategy is embodied in the following observations. If π_1 is a path through a program unit P and π_1 involves q predicates then,

(i) for π_1 to be feasible, all q predicates must be consistent whereas infeasibility of π_1 requires that as few as two predicates be inconsistent,

(ii) the larger the value of q, the greater, on average, is the effort that is likely to be required in order to determine whether or not π_1 is feasible

and (iii) if π_1 is feasible, the larger the value of q, the greater, on average, is the effort that is likely to be required in order to derive a corresponding set of test data.

From these observations, the authors infer that, on average, the number of infeasible paths initially generated in, and the overall effort required by branch testing will be reduced by a path generation strategy which seeks to minimise the number of predicates involved in each path of π. Reductions achieved thus would then be reflected in a saving in the time and cost overheads of the branch testing exercise. Clearly, it would be desirable to supplement this pragmatic argument with one which is logically rigorous, however, to provide a formal theoretical justification would be extremely difficult, if not impossible. Assumptions would have to be made about the interdependence of the program predicates. Two extreme assumptions: independence, and complete inter-dependence, of all predicates, might lead to rigorous arguments, but these would be virtually valueless since it is likely that, in the overwhelming number of code segments, certain groups of predicates will be independent whilst the remainder will not - and this cannot be formally quantified for the 'average' case. This paper, therefore, does not seek to provide any such formal justification.

The succeeding subsection of this paper contains details of the method used to implement the path generation strategy suggested above.

2.1 The Path Generation Method

Any code unit P, be it a complete program or a subdivision of a program such as a module, subroutine or procedure, can be partitioned into groupings of consecutive statements called basic blocks. Each basic block has a unique entry point (its first statement), a unique exit point (its last statement) and no internal branches. A model that is frequently used to represent the structure of, and possible paths through P is the 'control flow graph', $FG_p = (V_p, A_p)$. This is a directed graph in which the vertices, V_p, are the basic blocks of P and the arcs, A_p, represent the possible flows of control between the blocks.

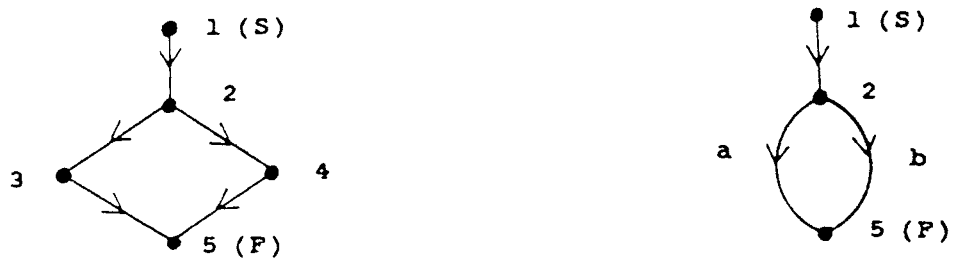
It is assumed that the control flow graphs considered here have a unique source vertex S and a unique sink vertex F. (This corresponds to P, having but a single entry point and a single exit point, and is in line therefore with the dictates of good programming style. Should, however, P have more than one entry and/or exit point, account can be taken of this by introducing a super-source and/or super-sink into FG_p). It is further assumed that $|V_p| = n$, $|A_p| = e$, and that for each $i \epsilon V_p$, there exists a path in FG_p both from S to i and from i to F.

As a consequence of its definition, a basic block can involve at most one predicate, and if, in fact,this is the case for some basic block of P, the corresponding vertex of FG_p will have out-degree in excess of unity. Since, in the majority of code units not all basic blocks will involve a predicate, the control flow graph is not the most appropriate model for generating the required path set. A graph $G_p = (V,A)$ with the necessary characteristics can be derived, however, from the control flow graph in the following manner:

(i) set $V = V_p$ and $A = A_p$

and (ii) for each vertex $j \neq S \epsilon V$ with $d_0(j) = 1$ replace each arc pair: $ij \epsilon A$ and $jk \epsilon A$, by an arc ik, and remove j from V. ($d_0(j)$ is the out-degree of vertex j).

The graph resulting from these transformations is referred to as the DD-graph (Decision to Decision graph) of P. Its vertices, excluding F and possibly S, correspond to those basic blocks that involve a predicate, and its arcs represent distinct paths (DD-paths) in P between these blocks. Since, with the exception of those involving $|V_p|$ and $|A_p|$, the assumptions made above concerning FG_p are invariant under transformations (i) and (ii), it is valid to assume that S/F is the unique source/sink of G_p, and that there exists a path in G_p both from S to i and from i to F for all $i \epsilon V$. It is important to note that frequently G_p, rather than being a graph, will be a multigraph because, in general , there will be more than one distinct path in P between two basic blocks that each involve a predicate. Figure 1, which depicts a control flow graph and the corresponding DD-graph, illustrates such a situation.



A Control Flow Graph. The Corresponding DD-graph.

Figure 1.

Since each branch of P lies on at least one DD-path, a set of S-to-F paths which covers the arcs of G_p will involve each branch of P. Thus, a set of S-to-F paths which covers the arcs of G_p and is such that each path involves a minimum number of vertices of G_p and therefore, a minimum number of arcs, will have all the characteristics of the path set π that is sought. Further, by associating a length $\ell_{ij} = 1$ with each arc $ij \in A$, this set of S-to-F paths can be identified as a set of shortest S-to-F paths through G_p which can be determined as described below.

Denoting by π^*_{ij}, a shortest path in G_p from vertex i to vertex j, and by π^*_{pijq}, a shortest path in G_p through arc ij from vertex p to vertex q, the path set π that is sought is defined by

$$\pi = \{\pi^*_{SijF} : ij \in A\}.$$

Using the Principle of Optimality each path π^*_{SijF} of π can be decomposed as

$$\pi^*_{SijF} = \pi^*_{Si}.ij.\pi^*_{jF} \qquad (2.1)$$

Thus, in order to define π, it is merely necessary to determine the paths π^*_{Si} and π^*_{jF} for all $i,j \in V$, and use these to generate the π^*_{SijF}

An algorithm which is well-suited for determining the π^*_{Si} and π^*_{jF} is that due to Dijkstra. Nonetheless, the special nature of G_p, in that $\ell_{ij} = 1$ for all $ij \in A$, permits a modification of Dijkstra's algorithm which, although still $O(|V|^2)$, is more timewise efficient. This modified algorithm, referred to here as the D^+ algorithm, may be found, for example, in Gondron and Minoux [5], and is essentially equivalent to performing a breadth-first traversal of G_p.

A single application of the D^+ algorithm to G_p using S as the start vertex is sufficient to determine all of the paths π^*_{Si} and the paths π^*_{jF} can be found with almost equal facility by defining the graph $\bar{G}_p = (V,\bar{A})$, derived by reversing the direction of all arcs in G_p, and applying the D^+ algorithm to it using F as the start vertex. Clearly, the required path set could now be generated by selecting, in turn, each arc $k\ell \in A$ and using equation (2.1) to generate $\pi^*_{Sk\ell F}$. This procedure, however, would be inefficient since, for all but the most trivial DD-graphs, and hence the most trivial of code units, certain of the generated paths would be duplicated. This point can be illustrated with reference to the DD-graph of figure 1. Application of the procedure to this graph, for example, results in the generation of the path 1(S)-2-5(F) via arc b, and duplicated generation of the path 1(S)-2-5(F) via arc a (once for arc 12 and once for arc a).

It is desirable, therefore, to ensure that duplicate paths are not generated. This could be achieved by checking that the path about to be generated has not been produced previously, however, a far superior alternative approach is available.

The result of applying the D^+ algorithm to G_p with start vertex S is to generate a shortest distance spanning arborescence of G_p rooted at S. Denoting this arborescence by $T_S = (V, A_S)$, T_S contains all, and only, the paths $\pi^*_{Si}, i \epsilon V$, defined by the D^+ algorithm, and hence, the arc set A_S is composed of only the arcs of G_p that lie on these paths. Using $\bar{A}_S$ to denote the complement of A_S in A (A_S/A) and defining $\pi_p = \{\pi^*_{SijF} : ij \epsilon \bar{A}_s\}$, the following results can be quoted:

Result 1
The path set $\pi^*_p = \pi_p \cup \{\pi^*_{SF}\}$ covers all arcs in G_p.

Result 2
If $ij \epsilon \bar{A}_S$ and $k\ell \epsilon \bar{A}_S$ are two distinct arcs of G_p, then $\pi^*_{SijF} \neq \pi^*_{Sk\ell F}$.
μονοπάτια δεν είναι ίδια (ανεξάρτητα)

Result 3
π^*_p contains $(|A|-|V|+2)$ paths.

In order then to derive a set of paths with the required characteristics, it is only necessary to generate the paths in π^*_p; Result 1 ensuring that all arcs in G_p are covered, and Result 2 that there is no duplication of paths. The main steps in the proposed path generation method, therefore, can be summarised as follows:

(i) Apply the D^+ algorithm to G_p to give $\{\pi^*_{Si}: i \neq S \epsilon V\}$
(ii) Apply the D^+ algorithm to G_p to give $\{\pi^*_{jF}: j \neq F \epsilon V\}$
(iii) Determine $\bar{A}_S$ as A/A_S.

For each arc $k\ell \epsilon \bar{A}_S$ perform step (iv).

(iv) Generate $\pi^*_{Sk\ell F}$ using π^*_{Sk} and $\pi^*_{\ell F}$

(v) Generate π^*_{SF}.

It is an algorithm based upon these steps that has been implemented and used to derive the results reported in section 4.

3. Discussion of the Path Generation Method

A desirable property of any method for generating π is that it should be efficient. That the proposed method satisfies this criterion is suggested by the following result.

Result 4
The proposed method for generating π requires $O(n^2)$ time.

Result 1 ensures that π_p^* provides a cover for the arcs of G_p and Result 4 indicates that the proposed method for generating this path set is potentially efficient. Despite these, however, it is purposeful to enquire whether, for the sake of efficiency, it is possible to remove certain paths from π whilst still maintaining an arc cover. The answer to this question is: 'affirmative', and hence π_p^* will not be a minimal arc cover in all cases. This situation is illustrated by the DD-graph of figure 2.

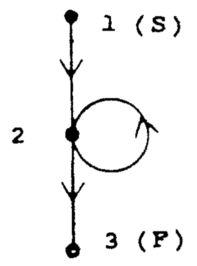


Figure 2.

In this example the minimum number of paths required to cover the arcs of the graph is one: 1(S)-2-2-3(F). The number of paths in π_p^* is two: 1(S)-2-2-3(F) (corresponding to π_{S22F}^*) and 1(S)-2-3(F) (corresponding to π_{SF}^*). Thus the second of these could be removed from π_p^* without destroying the arc cover.

To remove paths from π_p^* in situations such as this would, however, destroy a key property of π_p^* which is generally considered desireable. This property is embodied in the following result.

Result 5

If π_p^* contains no infeasible paths and D is a data set corresponding to π_p^*, the execution of P with D will exercise both the fall-through condition and iterative structure of all loops in P that possess a fall-through condition.

In situations where paths could be removed from π_p^*, their removal, although maintaining an arc cover, would result in the fall-through condition of some or all loops not being exercised. The authors, therefore, contend that the potentially superior test coverage provided by retaining such 'apparently redundant' paths outweighs the questionable increase in efficiency derived from their removal. In fact, the presence of such paths, and therefore, their potential ability to exercise both the fall-through condition and iterative structure of loops should be regarded as a key feature of the path generation method.

A further noteworthy feature of the proposed method is its effectiveness in identifying 'isolated basic blocks': basic blocks within P which, because of P's structure, cannot be accessed on any path through P. The assumption made in section 2.1 that: 'there exists a path in G_p both from S to i and from i to F for all $i \in V$', is equivalent to the assumption that P contains no isolated blocks, and was required merely to facilitate the proof of results 1-5. Should, in practice, this assumption be invalid for some P, the presence of the isolated basic block(s) is straightforwardly recognised by the proposed method during one of the applications of the D^+ algorithm. In effect, the D^+ algorithm recognises and indicates its inability to generate for all vertices i of G_p, the path π^*_{Si} and / or π^*_{iF}. The trapping of isolated basic blocks is thus an intrinsic capability of the path generation method proposed.

Before considering the results which are presented in Section 4, it is important to note that, although the proposed method will always yield a path set π^*_p, this path set may not be unique. Potentially, multiple 'candidate path sets' exist for DD-graphs in which there is a non-unique shortest path from S to some vertex i and/or from some vertex j to F. Such a DD-graph is depicted in figure 3 from which it may be determined that there exist four candidate path sets: {ac, ad, bc}, {ac, ad, bd}, {ac, bc, bd} and {ad, bc, bd}.

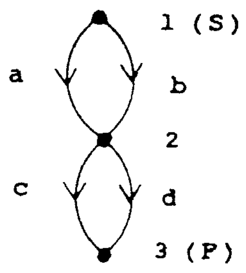


Figure 3

The existence of multiple candidate path sets does not affect the operation of the proposed method. Nonetheless, it is necessary to take them into account when assessing the methods performance since different implementations of the method may generate different path sets which may, in turn, provide different effective arc covers. Consequently, in attempting to determine how well the method performs when applied to a given network, it is necessary to consider the efficacy of <u>all</u> and not just one of the candidate path sets.

4.0 Results

In order to assess its effectiveness, the method described above was applied to the DD-graphs of 18 subroutines taken from the NAG (Numerical Algorithm Group) Mark 7 FORTRAN Library. The path sets thus derived were cross-referenced with the corresponding code to determine wich of the paths were feasible and to obtain the actual arc cover provided as measured by the TER_2 metric [6]. (This is defined to be the ratio of the number of branches covered by the feasible paths to the total number of branches in the code unit.)

To ensure that the results obtained should give a fair and accurate reflection of the methods capabilities, all candidate path sets for each subroutine were generated and the corresponding value(s) of TER_2 derived. For each subroutine i, this gave a set Z_i of values of TER_2, i.e., $Z_i = \{TER_2(ij)\}$, $j = 1,\ldots,K_i$, where K_i is the number of candidate path sets for subroutine i. The values of TER_2 obtained for each of the 18 subroutines are presented in table 1 in the form of Max_i, Min_i and $Mean_i$ where:

$$Max_i = \max_j \{TER_2(ij)\},$$

$$Min_i = \min_j \{TER_2(ij)\},$$

$$Mean_i = [\sum_{i=1}^{K_i} TER_2(ij)]/K_i$$

As can be judged from the table, the branch coverage provided by the proposed method is most satisfactory in the majority of cases. Full branch coverage is guaranteed ($Min_i = 1$) for 7 of the 18 subroutines, and, may be achieved ($Max_i = 1$) for 8 of them. Moreover, for 13 of the subroutines a guaranteed coverage of at least 75% is observed and, in only 3 cases: E02BB; F01CS; F04AQ, does the guaranteed coverage fall below 50%. That the method did not achieve a better coverage in two of these three cases resulted from the existence of nested loops for which the value of the control variables of the innermost loop were dependant upon the number of iterations of the outer loop(s) that were performed. In fact, taking the 18 subroutines as a whole, the majority of generated paths which were found to be infeasible, were found to be such as a direct result of problems associated with rested loops.

It is clear from previous comments that the branch coverage provided by the proposed method may be implementation-dependent. However, a comparison of corresponding values of Min_i and Max_i, and of the values $M_1 = 0.801$ and $M_2 = 0.846$, the mean of Min_i and Max_i respectively, do suggest that different implementations of the method will achieve, on average, a comparable branch coverage for the same code unit; a

i	NAG ROUTINE	No. of branches	$\|V\|$	$\|A\|$	$\|\pi\|$	K_i	Min_i	Max_i	$Mean_i$
1	A02AB	7	3	4	3	4	1.00	1.00	1.00
2	C06DB	16	5	8	5	4	0.75	1.00	0.84
3	D02XH	17	7	11	6	2	1.00	1.00	1.00
4	E01AA	9	5	7	4	1	0.78	0.78	0.78
5	E02BB	20	6	10	6	2	0.35	0.90	0.63
6	F01AF	10	6	9	5	1	0.60	0.60	0.60
7	F01AH	18	8	14	8	1	0.89	0.89	0.89
8	F01AZ	15	7	12	7	1	0.87	0.87	0.87
9	F01BE	10	6	9	5	1	0.80	0.80	0.80
10	F01CL	12	5	8	5	1	1.00	1.00	1.00
11	F01CM	5	4	5	3	1	1.00	1.00	1.00
12	F01CS	11	6	9	5	1	0.46	0.46	0.46
13	F03AL	18	8	12	6	8	1.00	1.00	1.00
14	F03AM	14	6	9	5	1	1.00	1.00	1.00
15	F04AQ	17	8	13	7	1	0.47	0.47	0.47
16	G01AC	21	7	12	7	2	0.71	0.71	0.71
17	G04AD	12	6	9	5	1	0.75	0.75	0.75
18	S17AC	17	5	8	5	2	1.00	1.00	1.00

TABLE 1.

i	NAG ROUTINE	No. of branches	$\|V\|$	$\|A\|$	$\|\pi\|$	K_i	Min_i	Max_i	$Mean_i$
1	A02AB	7	7	9	2	2	1.00	1.00	1.00
2	C06DB	16	13	16	2	4	0.38	1.00	0.72
3	D02XH	17	13	17	5	2	1.00	1.00	1.00
4	E01AA	9	7	9	1	2	0.00	0.00	0.00
5	E02BB	20	18	22	3	12	0.00	1.00	0.53
6	F01AF	10	7	10	1	16	0.00	0.00	0.00
7	F01AH	18	14	20	2	24	0.06	0.06	0.06
8	F01AZ	15	12	17	2	8	0.07	0.07	0.07
9	F01BE	10	7	10	1	18	0.00	0.00	0.00
10	F01CL	12	11	14	2	6	0.08	0.08	0.08
11	F01CM	5	4	5	1	2	0.00	0.00	0.00
12	F01CS	11	8	11	1	4	0.00	0.00	0.00
13	F03AL	18	14	18	2	108	0.50	1.00	0.86
14	F03AM	14	11	14	2	18	0.50	1.00	0.87
15	F04AQ	17	12	17	1	4	0.00	0.00	0.00
16	G01AC	21	18	23	5	12	0.52	0.95	0.79
17	G04AD	12	9	12	1	2	0.00	0.00	0.00
18	S17AC	17	14	17	5	1	1.00	1.00	1.00

TABLE 2.

situation which is highly desirable.

Although the results obtained indicate that the proposed method does provide an effective approach to branch testing, it is difficult to draw any other than tentative conclusions when the results are considered in isolation. In order to provide, therefore, a more substantial basis for assessment, the path generation method proposed by Ntafos and Hakimi [7] was applied to the same 18 subroutines and corresponding results derived. This method seeks to minimise the cost overheads of branch testing by generating a path set containing a minimum number of paths. For any code unit, such a path set can be derived by determining a minimum S-to-F flow in the corresponding control flow graph with a lower bound of unity imposed on all arc-flows; this flow defining the required set of S-to-F paths. Unfortunately, however, the minimum flow is not always unique and, moreover, a minimum flow is not sufficient, in general, to define a unique set of S-to-F paths. Consequently, multiple candidate path sets may occur. The results - direct analogues of those given in table 1 - obtained by applying this method to the 18 subroutines are presented in table 2.

A brief study of table 2 shows that, in most cases, the values of Min_i, Max_i and $Mean_i$ which have been derived differ considerably from those quoted in table 1. The most marked differences are observed in the values of Min_i, the guaranteed branch cover for subroutine i. Full branch cover is guaranteed by Ntafos and Hakimi's method for only 3 of the subroutines, and for 14 of the remaining 15, the guaranteed coverage does not exceed 8%, being in fact, zero on 8 occasions. For not one of the subroutines does the method yield a guaranteed coverage superior to that given by the method proposed in this paper. When the values of Max_i, the best possible branch cover for subroutine i, are considered, Ntafos and Hakimi's method fares somewhat better: a 100% coverage may be achieved for 7 of the subroutines,and in 2 cases the value of Max_i improves on the corresponding value given in table 1. However, the value of Max_i achieved for 7 of the subroutines does not exceed the corresponding value of $Min_i = 0$.

Many of the infeasible paths generated by Ntafos and Hakimi's method resulted from problems associated with nested and sequential loops. In particular, a large percentage of the problems arose because of the methods inability to force iteration of certain loops the required number L of times, this situation being especially acute when L had been defined as a result of calculations performed in an enclosing or preceeding loop.

One of the most striking observations which can be made concerning the results in table 2 is that to each routine for which a zero coverage ($Max_i = 0$) was obtained, there corresponds a path set containing just a single path. Clearly, if a path set contains only one path, and that path is found to be infeasible, a zero path coverage ensues. However, by actively seeking to minimise the number of paths in a path set, Ntafos and Hakimi's method is effectively encouraging the occurence of this situation. The method, therefore, must be criticised for this if for no other reason. In almost direct opposition, the method being proposed produces, for each code unit, a path set containing a maximal independent set of paths and thereby tends to avoid this one path-zero coverage syndrome.

5.0 Summary and Conclusion

In this paper, a pragmatically based strategy for minimising the cost of branch testing has been proposed, a method implementing the strategy has been described, and the results of applying the method to 18 test examples have been presented. The method posesses several desirable characteristics which include: its efficiency - $O(n^2)$; its ability to detect isolated basic blocks, and the capability to exercise both the fall-through condition and iterative structure of loops. The results obtained from the test examples were most encouraging, indicating a full branch coverage ($Min_i = 1$) in 7 cases and mean guaranteed coverage for the 18 subroutines of $M_i = 0.801$. In comparison with the analagous results obtained from the application of Ntafos and Hakimi's method the proposed method gave an equivalent or superior guaranteed coverage in all 18 cases (superior in 15), and the value of M_1 derived, exceeded that of Ntafos and Hakimi's method by 0.518 and a factor of 2.8 better. In conclusion, the authors contend that the evidence presented gives a strong indication of the proposed method's effectiveness as a means for performing branch testing, and of the efficacy of the underlying strategy for reducing the incidence of infeasible paths.

It is certainly true that the aim of branch testing is to cover all branches, and that the proposed method does not guarantee to do this. No method, in fact, which is based purely on the structural characteristics of code segments could make such a guarantee. The proposed method does, however, appear to provide the foundations upon which to build in attempting to achieve $TER_2 = 1$. In pursuit of this goal, it is intended that the method be used as the core of an interactive branch testing tool which extends the philosophy of using shortest S-to-F paths. The functional policy of the tool would be as follows: having applied the method and found path, π^*_{SijF} say, to be infeasible, the tool generates, in turn and as necessary, $\pi^{(k)}_{SijF}$, $k = 2, 3, \ldots,$ - the second, third etc., shortest S-to-F path through ij, until for some t, $\pi^{(t)}_{S_{ij}F}$ is found to be feasible. Currently, work towards implementing such a tool is in progress, and it is hoped to report on this in the near future.

Acknowledgement

The authors express their thanks to NAG Ltd. for permission to access the source code version of their FORTRAN Library and to Dr. M.R.Woodward for the use of his software tool for abstracting basic blocks from FORTAN code.

References

[1] Miller, E.F. et al., "Structurally Based Automatic Program Testing", Proc. EASCON, 1974.

[2] Stucki, L.G., "Automatic Generation of Self- Metric Software", Proc. IEEE Symposium on Computer Software Reliability, IEEE, 1973.

[3] Brown, J.R. et al., "Automated Software Quality Assurance", Program Test Methods, W.C. Hetzel ed., Prentice-Hall, Englewood Cliffs, N.J., 1973.

[4] Howden W.E., "Empirical Studies of Software Validation", Tutorial: Software Testing and Validation Techniques, E. Miller and W.E. Howden eds., IEEE, 1978.

[5] Gondron, M. and Minoux, M., "Graphs and Algorithms", Wiley Interscience, 1984.

[6] Brown J.R., "Practical Applications of Automated Software Tools", TRW Report, TRW-ss-72-05, TRW Systems, One Space Park, Redondo Beach, California, 1972.

[7] Ntafos, S.C. and Hakimi, S.L., "On Path Cover Problems in Digraphs and Applications to Program Testing", IEEE Trans. on Soft. Eng., SE-5, 5, 1979.