Software Quality Journal **4**, 227–242 (1995)

The effort required by LCSAJ testing: an assessment via a new path generation strategy

D.F. YATES[1] and N. MALEVRIS[2]

[1]*Department of Computer Science, University of Liverpool, P.O. Box 147, Liverpool L69 3BX, UK*
[2]*Department of Informatics, Athens University of Economics and Business, 76 Patissiou Street, Athens 10434 Greece*

Received 27 May 1994

In general, LCSAJ testing more thoroughly exercises the control structure of a piece of software than does either statement or branch testing. Despite this, the exceptionally small number of papers which detail experience of using LCSAJ testing clearly indicates that its popularity falls considerably short of that of the other two methods. One factor that has contributed to this situation is the apparent absence of any attempt to assess the effort entailed by LCSAJ testing in practice, thereby precluding any meaningful cost-effectiveness analysis of the method. Such an attempt is reported in this paper.

A significant influence on the effort associated with LCSAJ testing is the number of test paths that must be generated to achieve a specified level of cover, and this is determined by the number of such paths that are found to be infeasible. Effort, therefore, is dependent upon the effectiveness of the path generation method/strategy in avoiding infeasible paths. The attempt to assess the effort entailed by LCSAJ testing which is reported here makes use of a new path selection strategy. This strategy seeks to reduce, *a priori*, the incidence of infeasible paths in the test set that is generated. Details of the strategy are presented, as are the results which were obtained by applying it to a set of subroutines. From the results, a measure of the effort entailed in employing the strategy is deduced. Subsequently, it is argued that features of the strategy enable pertinent comments to be made not only about the effort associated with employing the strategy, but also about the effort entailed by LCSAJ testing in general.

Keywords: LCSAJ, LCSAJ testing, white box testing, infeasable paths, test path selection

1. Introduction

Important tools on the workbench of any software engineer are the 'white box' testing methods, each of which seeks to exercise (cover) all occurrences of a specific element/feature of a piece of software. Such methods include: statement testing, branch testing, the boundary interior method (Howden, 1975), LCSAJ testing (Woodward *et al.*, 1980), mutation testing (DeMillo *et al.*, 1978), and data-flow testing (Fosdick and Osterweil, 1976). The focus of this paper is LCSAJ testing, the aim of which is to achieve $Ter_3 = 1$, where:

$$Ter_3 = \frac{\text{the number of LCSAJs covered}}{\text{the number of LCSAJs in the software}}$$

and an LCSAJ is defined to be a sequence of $n \geq 1$ contiguous basic blocks followed by a

control flow jump or a sequence of n such blocks which together constitute a complete program path.

Of the white box testing methods, the two that are arguably the most straightforward to understand and apply are statement testing and branch testing. Both methods are widely used, and in fact, application of the latter appears to have come to be viewed by many as a *de facto* minimum testing standard. Given that LCSAJ testing more thoroughly exercises the control structure of an item of software than does either statement or branch testing (at the very least in the sense that $Ter_3 = 1$ implies that both $Ter_1 = 1$ and $Ter_2 = 1$, see Woodward *et al.*, 1980), it is not entirely clear why LCSAJ testing is not as, or even more, popular than the other two methods. Hennell (1991) suggests that: "In order to ensure that testing tools and techniques are more widely used ... Managers of software projects need to be educated in the costs and benefits of utilizing modern testing tools and techniques". Moreover, Liggersmeyer (1994) states that: "... practitioners accept systematic (*testing*) techniques only if they are convinced that technical requirements will be achieved with reasonable effort". Unfortunately, the effort entailed by LCSAJ testing has not been quantified, and this fact must therefore be considered a potentially significant element in determining the relative unpopularity of the method. This is not to say that there have been no attempts at quantification, there have, but only very few. The most significant of such attempts are those of Woodward (1984) and Ntafos (1988). Woodward carried out an analysis of an extensive set of FORTRAN subroutines and derived lower bounds on the average (mean and median) number of paths that it would be necessary to test in order to achieve $Ter_3 = 1$. Ntafos, on the other hand, performed a worst-case analysis of the method and showed that for a piece of software containing n basic blocks, it may be necessary to test $O(n^2)$ paths to cover all LCSAJs. These results do provide valuable insight. However, as the two authors themselves point out, they have only theoretical foundation. As a consequence, the results cannot take account of an important practical factor which has the potential to significantly influence the extent of the effort entailed in undertaking not just LCSAJ testing, but any form of white box testing – 'the infeasible path'. (When there exists no data set that will cause the execution of a given path through a piece of software, that path is said to be *infeasible*.) Undesirable effects deriving from the presence of such paths have been noted by many authors, see for example Woodward *et al.* (1980), Hedley and Hennell (1985), DeMillo *et al.* (1987), and Chellappa (1987). Unfortunately, no author has reported an assessment of the true extent of their influence on the effort associated with LCSAJ testing. Although such an assessment can only be made with recourse to the results of practical experience, it is certainly true that, if a given level of LCSAJ cover is to be achieved, the effort entailed by LCSAJ testing increases as the number of infeasible paths in the test path set increases. Consequently, any path selection strategy that elects only feasible paths to the LCSAJ test path set is consistent with minimal expenditure of effort. Unfortunately, the existence of such a strategy is precluded by a result due to Weyuker (1979). Thus, no approach to LCSAJ testing can guarantee minimal effort, and the best that can be hoped for is a good heuristic.

In an attempt to assess the effort entailed by LCSAJ testing, a path selection strategy which seeks to reduce the incidence of infeasible paths in the path set generated, is first proposed in Section 2. To enable the strategy to be interpreted and implemented unambiguously, certain features of the strategy require elaboration. These are considered in Sections 3 and 4. Subsequently, in Section 5, experiments which involved using the strategy to derive LCSAJ test paths for a set of subroutines are described, the corresponding results are reported, and an assessment of the associated effort is presented. Finally, in Section 6, it is argued that the nature of both infeasible paths and the experimental results, together with certain characteristics of the proposed path selection strategy, can justify pertinent comments being made about the effort associated with LCSAJ testing in general.

2. The path generation strategy for LCSAJs

Gabow *et al.* (1976) proved that the problem of avoiding infeasible paths during the process of test path selection is *NP*-complete. Correspondingly, as a heuristic for characterization of infeasible paths, Yates and Hennell (1985) advanced, and argued in support of, the following:

> a program path that involves $q \geq 0$ predicates is more likely to be feasible than one involving $p > q$.

In Malevris *et al.* (1990), this proposition was subjected to formal statistical investigation. As a result, it was possible to conclude with great confidence that path feasibility does tend to decay with an increasing number of predicates, and that the decay is exponential in nature. Consequently, Yates and Malevris (1989) proposed a path selection strategy for use in connection with branch testing which seeks to reduce, *a priori*, the incidence of infeasible paths in the path set generated. This strategy, in the context of testing a code unit C, may be expressed as follows:

(1) Generate Π, a set of program paths, each involving a minimum number of predicates, that will cover the branches of C if none of the paths is infeasible.
(2) Derive a data set corresponding to Π, execute C with it, and determine the branch cover thus achieved.

While *cover* < *maximum* repeat step (3)

(3) Select an uncovered branch, x of C, and successively generate $\pi_k(x), k = 2, 3, \ldots, t$, until $\pi_t(x)$ is found to be feasible, where $\pi_k(x)$ denotes that path through branch x which involves the kth smallest number of predicates. Then, recalculate the value of Ter_2.

The ethos of this strategy derives only from the proposition advanced by Yates and Hennell. In addition, the proposition relates only to the relative number of predicates involved in different program paths. Consequently, an analogous strategy for LCSAJ testing, can be derived from the above merely by replacing all references to 'branch' and 'Ter_2' by 'LCSAJ' and 'Ter_3', respectively. Moreover, since the statistical investigation performed by Malevris *et al.* focused solely upon Yates and Hennell's proposition, the validity of this analogue in respect of its actively seeking to reduce the incidence of infeasible paths in the path set generated for the purpose of LCSAJ testing, is also established. It is this analogue that has been implemented and used to derive the results reported in Section 5.

It might at first appear that the proposed path selection strategy is identical to the one used by Woodward *et al.* (1980), and referred to in Hedley and Hennell (1985), in experiments on the feasibility of program paths. This is not the case. For, although Woodward *et al.* generated paths in an order consistent with increasing path length, as is advocated by the above strategy, their definition of 'path length' is radically different from the one adopted here. The 'length' of a program path is defined by Woodward *et al.* as *the number of LCSAJs it contains*, whereas for present purposes, it is the number of predicates in a path which determines its length. Since the number of predicates that are contained in an LCSAJ is not a constant, but varies between 1 and the total number of predicates in the software under test, the two path generation strategies are not comparable. Consequently, the findings of Woodward *et al.*, namely that relatively few of 'the shortest paths' through a piece of software tend to be feasible, cannot under any circumstances be taken to be applicable to the path generation strategy proposed above.

In order for the proposed strategy to be interpreted and implemented unambiguously, certain of its features require elaboration. These are:

(1) the method adopted in order to derive Π;
(2) the method used in generating the $\pi_k(x)$;
(3) the interpretation of the criterion '*cover* < *maximum*';
(4) the criterion used to determine the order in which uncovered LCSAJs are selected in step (3) of the strategy.

Details of the first two of these are presented in the succeeding section of the paper, and consideration is given to the other two in Section 4.

3. The methods used for generating the required paths

The representation of the structure of a code unit C by its 'control flow graph' (see Paige, 1975) is well known and understood. Its use is appropriate for supporting path generation in respect of many of the white box testing methods. Unfortunately, this is not so for LCSAJ testing, since the graph's structure does not lend itself to the efficient location and identification of LCSAJs. For LCSAJ testing, a more apt representation of C is the *LCSAJ graph*, $G_L(C) = (V, A)$ (see Woodward, 1984), in which each distinct LCSAJ, m, of C is represented by a unique vertex v_m, and $v_r v_s$ is an arc in the graph if and only if the target of the control flow jump which terminates LCSAJ r is the first basic block in LCSAJ s.

Although some LCSAJ graphs possess only one 'source' (a vertex with no incoming arcs) and one 'sink' (a vertex with no outgoing arcs), other have multiple sources and/or sinks (again see Woodward, 1984). Consequently, it is convenient to convert $G_L(C)$, without loss of generality, to a standard form, $G_L^*(C)$, involving a single source and a single sink. To achieve this, an artificial source, S, and artificial sink, F, are introduced into $G_L(C)$ together with arcs from S to each of the original sources and arcs to F from each of the original sinks (as no confusion will arise, and for the sake of simplicity, $G_L^*(C)$ will henceforth be referred to as the control flow graph of C).

Since a path through code unit C can be decomposed uniquely into a sequence of LCSAJs (see Woodward, 1980), that path is represented by a unique path from S to F in $G_L^*(C)$. Consequently, it is the LCSAJ graph that has been adopted in the work reported here in order to support the generation of the paths required by steps (1) and (3) of the path selection strategy.

3.1 *Generating Π for code unit C*

If the length of a path in C is interpreted as being the number of predicates it involves, the problem of generating Π becomes that of generating a set of shortest paths through C which together cover all of C's LCSAJs. If, for every vertex, v_r, of $G_L^*(C)$, all arcs outgoing from v_r are assigned a value equal to the number of predicates contained in that LCSAJ of C which corresponds to v_r, the required path set can be identified with a unique set, $\Pi^{(1)}$, of paths from S to F (S-to-F paths) in $G_L^*(C)$. In particular, $\Pi^{(1)}$ contains, for each vertix v_r of $G_L^*(C)$, the shortest path from S to F which passes through v_r.

By employing the Principle of Optimality (Bellman and Dreyfus, 1962), the shortest path from S to F which passes through vertex v_r can be expressed as the shortest path from S to v_r followed by the shortest path from v_r to F. Now, standard graph theoretic techniques such as Dijkstra's algorithm (see Boffey, 1982, for example) are readily available for determining the shortest path between any two vertices of a graph. Consequently, $\Pi^{(1)}$, and therefore Π, can be derived by appropriately applying Dijkstra's algorithm to $G_L^*(C)$ and employing the Principle of Optimality, as above, for each vertex of $G_L^*(C)$. However, were this to be done, it would be found that,

in general, $\Pi^{(1)}$ contained duplicated paths. To overcome this problem, it is necessary to employ the Principle of Optimality, not for each vertex in $G_L^*(C)$, but only for those vertices which are leaf nodes in the 'shortest distance spanning tree' of $G_L^*(C)$ that is rooted at S. (This spanning tree is formed by aggregating the shortest paths from S to each of the other vertices in $G_L^*(C)$.) Both the question of why the problem of duplication should arise and that of the validity of the stated means for overcoming it, are not issues central to this paper. Consequently, they will not be discussed further, however, a detailed consideration of these matters may be found in Malevris (1989).

3.2 *Generating the* $\pi_k(x)$

In much the same way as a shortest path through C can be identified with a unique shortest S-to-F path in $G_L^*(C)$, the path $\pi_k(b)$ through LCSAJ b of C can be identified as the kth shortest S-to-F path in $G_L^*(C)$ which passes through vertex v_b. Again using the Principle of Optimality, this path can be expressed as the mth shortest path from S to v_b followed by the nth shortest path from v_b to F, where m and n are integers satisfying $1 \leq m \leq k$ and $1 \leq n \leq (k - m + 1)$, respectively.

Several graph theoretic algorithms, such as those due to Dreyfus (1969) and Shier (1976), for determining the jth shortest path, $j = 1, 2, \ldots$, between the two vertices of a graph are available. Consequently, the required paths can be determined by first applying one of these algorithms to $G_L^*(C)$ in order to determine the mth shortest path from S to v_b and the nth shortest path from v_b to F, and then employing the Principle of Optimality, as above, in respect of vertex v_b. The algorithm actually adopted in deriving the results reported in Section 5 was that due to Dreyfus.

4. The criteria associated with step (3) of the strategy

4.1 *Interpreting the predicate 'cover < maximum'*

By definition, full LCSAJ cover is achieved only when $Ter_3 = 1$. However, every basic block involves a predicate, and in general, an LCSAJ will contain several basic blocks. Thus, the possibility of there being no test data that will simultaneously satisfy all predicates associated with a specific LCSAJ, does exist. (An LCSAJ for which this situation obtains cannot be covered, and is referred to as an *infeasible LCSAJ*.) Consequently, in the presence of infeasible LCSAJs, the Ter_3 measure provides less than accurate information since the extent of the LCSAJ cover actually achieved is:

$$\frac{\text{the number of LCSAJs covered}}{\text{the number of feasible LCSAJs in the software}}$$

Unfortunately, in order to evaluate this quantity it is necessary to know the number of LCSAJs that are infeasible, and this information may not be available. None the less, if an LCSAJ is assumed to be feasible until the contrary is proven, Ter_3^*, which will be referred to as the *relative LCSAJ cover*, may be used instead, where:

$$Ter_3^* = \frac{\text{the number of LCSAJs covered}}{\text{the number of LCSAJs thought to be feasible}}$$

In comparison with Ter_3, Ter_3^* will always give at least as accurate, and in most cases more accurate, an indication of the LCSAJ coverage actually achieved since

$$\text{cover actually achieved} \geq Ter_3^* \geq Ter_3$$

Moreover, since $Ter_3^* =$ cover actually achieved $= 1$ when all feasible LCSAJs have been covered, the predicate '*cover* < *maximum*' should ideally be taken to imply $Ter_3^* < 1$.

It would certainly have been desirable to adopt this interpretation of the predicate in deriving the results presented in Section 5. Unfortunately, this was precluded by the necessity of determining by hand the feasibility/infeasibility of each path generated in respect of the strategy (no suitable automated tool was available to the authors). As a consequence, it was necessary to relax the above stated 'ideal interpretation' of the predicate to that of:

$$Ter_3^* < 1 \quad \text{and} \quad K \leq 300$$

Here K is the total number of paths generated in respect of step (3) of the proposed strategy. Thus, at most 300 paths, over and above those in Π, were generated in an attempt to cover all feasible LCSAJs for any one item of software. It can be seen that this relaxation is entirely consistent with the 'ideal interpretation', but merely prevents the attainment of $Ter_3^* = 1$ being fully investigated in some cases.

4.2 *The criterion for selecting uncovered LCSAJs*

All that is required now in order to specify the strategy completely is an elaboration of the criterion that has been adopted in step (3) for selecting an uncovered LCSAJ. In this regard, consider two LCSAJs, the first involving X basic blocks and the second involving $Y \geq X$. Since by definition, every basic block contains one and only one predicate, the two LCSAJs will contain X and Y predicates respectively. If $X > Y$, and both LCSAJs are viewed as program sub-paths, interpretation of the proposition cited in Section 2 suggests that the first LCSAJ is more likely to be feasible than the second. It is therefore implied that there may be advantage in attempting to cover the LCSAJ with X predicates first. On the other hand, if $X = Y$, no such helpful interpretation is available. Correspondingly, the criterion which has been adopted requires selection of the LCSAJ that contains the least number of predicates, but if a tie occurs, one of the tying candidates is selected at random.

5. Experimental regime

In an attempt to assess the effort entailed in applying the strategy to achieve various levels of LCSAJ cover, the strategy was encoded in FORTRAN and applied to 35 subroutines taken from the NAG FORTRAN library. To facilitate a more precise analysis, the strategy was viewed as comprising sub-strategies S1 and S2, corresponding to steps (1) and (3) of the strategy respectively, and the results obtained from the application of the two sub-strategies were recorded independently.

5.1 *Experimental results*

The results derived from the application of sub-strategy S1 to the 35 subroutines are presented in Table 1. The following are reported for each subroutine: the number of constituent LCSAJs; the number of these which were found to be infeasible; the number of LCSAJs that were not covered; the value achieved for both Ter_3 and Ter_3^*; and the branch cover attained as measured by Ter_2. The total and mean value of these quantities are, where relevant, also presented in Table 1.

The application of sub-strategy S1 to each of the 35 subroutines resulted in a total of 324 paths

Table 1. Results derived from the application of sub-strategy S1.

Subroutine	*Number of LCSAJs*	*Number of LCSAJs not covered*	*Number of infeasible LCSAJs*	Ter_3	Ter_3^*	Ter_2
A02ABF	8	0	0	1.00	1.00	1.00
C02ADZ	12	3	0	0.75	0.75	0.86
C02AEZ	5	0	0	1.00	1.00	1.00
C06AAZ	10	2	2	0.80	1.00	1.00
C06ABZ	36	30	1	0.17	0.17	0.35
C06DBF	14	2	0	0.86	0.86	0.88
C06EBT	19	13	1	0.32	0.33	0.39
C06GCF	9	1	1	0.89	1.00	1.00
D02XHF	16	0	0	1.00	1.00	1.00
E01AAF	12	7	1	0.42	0.45	0.71
E01ABF	25	15	0	0.40	0.40	0.50
E02AFF	35	27	3	0.23	0.25	0.28
E02AKZ	23	3	3	0.87	1.00	1.00
E02BBF	19	4	2	0.79	0.88	0.90
E02GBN	26	14	2	0.46	0.50	0.63
E02RBF	24	4	3	0.83	0.95	0.88
F01AFF	3	0	0	1.00	1.00	1.00
F01AHF	19	11	2	0.42	0.47	0.50
F01AZF	17	9	2	0.47	0.53	0.58
F01BEF	12	6	1	0.50	0.55	0.65
F01CLF	13	6	5	0.54	0.88	0.83
F01CMF	7	0	0	1.00	1.00	1.00
F01CSF	18	11	4	0.39	0.50	0.66
F03AGF	32	10	2	0.69	0.73	0.79
F03AMF	16	0	0	1.00	1.00	1.00
F04AGY	9	2	1	0.78	0.88	1.00
F04AQF	21	18	5	0.14	0.19	0.23
F04MAY	19	19	2	0.00	0.00	0.00
F04MCV	16	0	0	1.00	1.00	1.00
G01BCF	46	17	5	0.63	0.71	0.86
G04ADF	18	18	5	0.00	0.00	0.00
G05EAF	41	34	7	0.17	0.21	0.25
S15AEF	18	16	2	0.11	0.13	0.13
S17ACF	14	0	0	1.00	1.00	1.00
S18CCF	20	0	0	1.00	1.00	1.00
Total	652	302	62	–	–	–
Mean	–	–	–	0.62	0.66	0.72

being generated. The subsequent testing of the paths realized full LCSAJ coverage of 9 subroutines and a mean value for Ter_3 of 0.62. Out of a total of 652 LCSAJs, 350 were covered, leaving 302 untested of which 62 (9.51% of the total) ultimately proved to be infeasible. Taking these infeasible paths into account, S1 achieved $Ter_3^* = 1$ for 12 subroutines as well as a mean value for Ter_3^* of 0.66. Of the other 23 subroutines, S1 failed in two cases, namely for F04MAY and G04ADF, to cover any LCSAJ but recorded values of Ter_3^* in the range [0.13, 0.95] for the remainder. In addition, full branch cover was achieved for 13 of the subroutines – all those for which

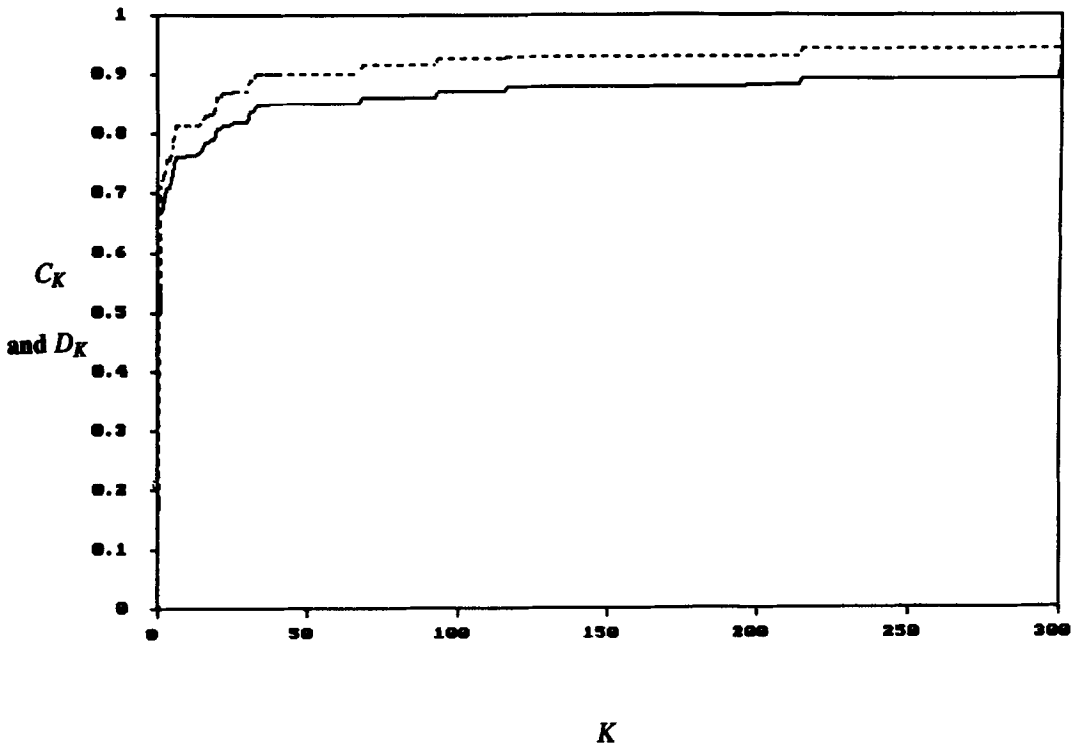

Fig. 1. A plot of C_K (solid line) and D_K (broken line) against K.

$Ter_3^* = 1$ together with F04AGY. No branches were covered at all in subroutines F04MAY and G04ADF, but overall, a mean value of $Ter_2 = 0.71$ was recorded. During testing, logical errors were found in the code of E02GBN, E02RBF, F01CLF and G05EAF, with the result that the maximum achievable branch cover for these subroutines was reduced to 0.94, 0.94, 0.83 and 0.96, respectively.

S2 was then applied to each of the 23 subroutines for which S1 had failed to realize full feasible LCSAJ cover. The results derived from the applications are reported in Fig. 1 which contains a plot (the unbroken line) of C_K against K for K in the range [0, 300]. Here, K is the total number of paths generated in respect of sub-strategy S2 for a given subroutine, and C_K is the mean value of Ter_3^* achieved for all subroutines after K paths have been tested, that is:

$$C_K = \frac{1}{35}\sum_{j=1}^{35} Ter_3^*(j, K)$$

where $Ter_3^*(j, K)$ is the value of Ter_3^* for subroutine j after K paths have been generated for it in respect of sub-strategy S2. (Note that: $C_0 = 0.66$ – the mean value of Ter_3^* resulting from the application of S1 alone; $Ter_3^*(j, K) = 1$ for all values of K in the range [0, 300] if $Ter_3^* = 1$ was achieved for subroutine j by sub-strategy S1; and $Ter_3^*(j, p) = Ter_3^*(j, q)$ for all values of p in the range $[q + 1, 300]$ if $Ter_3^* = 1$ was first achieved for subroutine j by sub-strategy S2 after generating $K = q$ paths.)

It can be seen from the graph that the increase in C_K is quite considerable for smaller values of K; C_K attaining the values 0.76, 0.81, and 0.85 at $K = 10$, 20, and 40, respectively. When K

increases beyond 40, the rate of increase of C_K slows. In fact, no further increase in C_K is observed until $K = 68$. The slowing of the rate of increase in C_K becomes even more marked as K increases through [100, 300], in which range the following values were recorded: $C_{100} = 0.87$, $C_{200} = 0.88$, $C_{300} = 0.89$.

Also reported in Fig. 1 is a plot (the broken line) of D_K against K, where D_K is the mean value of Ter_2 achieved for all 35 subroutines as a result of testing K paths generated by S2. (A more precise definition of D_K is obtained by replacing all references to Ter_3^* in the definition of C_K by Ter_2.) It is readily seen from the graph that as K increases, the increase in D_K very closely resembles that in C_K. In fact, the ratio of D_K to C_K lies between 1.09 (2 dec. pl.) at $K = 0$ and 1.05 (2 dec. pl.) at $K = 300$. It is therefore 'almost constant' in this range (but will necessarily converge to unity as K

Table 2. Results derived from the application of sub-strategy S2.

Subroutine	*Number of LCSAJs*	*Number of feasible LCSAJs not covered*	*Number of infeasible LCSAJs*	Ter_3^*	Ter_2
C02ADZ	12	0	0	1.00	1.00
C06ABZ	36	29	1	0.17	0.35
C06DBF	14	0	0	1.00	1.00
C06EBT	19	0	1	1.00	1.00
E01AAF	12	0	1	1.00	1.00
E01ABF	25	15	0	0.40	0.50
E02AFF	35	24	3	0.25	0.28
E02BBF	19	0	2	1.00	1.00
E02GBN	26	0	2	1.00	0.94
E02RBF	24	0	3	1.00	0.94
F01AHF	19	2	2	0.88	1.00
F01AZF	17	2	2	0.87	1.00
F01BEF	12	0	1	1.00	1.00
F01CLF	13	0	5	1.00	0.83
F01CSF	18	2	4	0.86	1.00
F03AGF	32	8	2	0.73	0.79
F04AGY	9	0	1	1.00	1.00
F04AQF	21	1	5	0.94	1.00
F04MAY	19	4	2	0.77	1.00
G01BCF	46	0	5	1.00	1.00
G04ADF	18	4	5	0.69	1.00
G05EAF	41	13	7	0.62	0.73
S15AEF	18	0	2	1.00	1.00
Total (for the above)	505	104	56	–	–
Mean (for the above)	–	–	–	0.83	0.90
Total (for entire sample)	652	104	62	–	–
Mean (for entire sample)	–	–	–	0.89	0.93

becomes sufficiently large), and hence branch cover can justifiably be said to increase approximately in proportion to LCSAJ cover.

In all, use of sub-strategy S2 resulted in the coverage of an extra 146 LCSAJs, the realization of $Ter_3^* = 1$ for an additional 12 subroutines, and the attainment of maximum branch cover for a further 16 subroutines. Thus, full feasible LCSAJ coverage was ultimately achieved for a total of 24 of the 35 subroutines and maximum branch cover for 30. The value of Ter_3^* and Ter_2 achieved by S2 in respect of each subroutine to which it was applied, is presented in Table 2 together with the corresponding number of LCSAJs which the strategy (both S1 and S2) did not manage to cover. The total and mean value of each of these quantities are, where relevant, also provided.

Sub-strategy S2 failed to cover a total of 104 feasible LCSAJs, and a comparison of Tables 1 and 2 reveals that, in four cases, it did not achieve any increase in the value of Ter_3^*. To investigate the reason(s) for this, the 13 subroutines for which $Ter_3^* = 1$ was not achieved, were examined in detail. Without exception, it was found that each unexecuted LCSAJ lay either within the body of a single loop, or nested within several, and that none of the LCSAJs could be covered unless a sufficiently large number of iterations of the loop(s) were to be performed.

5.2 *The effort required by the strategy*

The nature of the graph in Fig. 1 quite clearly suggests that the level of LCSAJ coverage achieved by the proposed strategy is subject to the law of diminishing returns. This, of course, is what would be anticipated if the proposition of Yates and Hennell is a good heuristic, since, as K increases, so do the lengths of the test paths as measured in terms of the number of predicates they contain. What the form of the graph also indicates is that, in general, it may not be cost-effective to generate more than approximately 40 additional paths in an attempt to increase the level of LCSAJ coverage for a routine beyond that achieved by sub-strategy S1. Cost-effectiveness, however, is likely to be case specific. For example, it may be deemed imperative to attain $Ter_3^* = 1$ for a life-critical application, whereas a lower level of LCSAJ coverage may be acceptable for, say, a given data processing application. In any event, a pre-requisite for the accurate assessment of cost-effectiveness is a quantification of cost or effort, and this does not appear to exist in respect of LCSAJ testing.

As LCSAJ testing predominantly involves the generation and testing of paths to cover a given set of LCSAJs, a suitable normalized measure of effort (and cost) may be taken to be the number of paths per LCSAJ that are generated during the exercise. Moreover, the difference between this value and a given theoretical minimum will provide some measure of the influence on effort of the presence of infeasible paths.

In order to assess the effort entailed in applying the proposed strategy, the number of paths generated in respect of both S1, and S2 for $K \in [1, 300]$, were recorded. An abstraction from this set of values is presented in Table 3 (the values corresponding to $K = 0$ relate to the paths generated in respect of S1), together with the corresponding values of v_K, the number of paths generated per LCSAJ for a given value of K, and the associated value of Ter_3^* achieved.

Had the 35 subroutines contained no infeasible paths, it would have been necessary to generate only the paths required by S1 since these would have realized $Ter_3^* = 1$ in all cases. Thus, given that the 35 subroutines contain 652 LCSAJs in total and that in applying S1 to them, 324 paths were generated, $v_0 = 324/652 = 0.50$ (2 dec. pl.) paths/LCSAJ represents a theoretical base-line value of the mean effort required by LCSAJ testing. Comparing this value with other values of v_K in Table 3, it can be seen, for example, that with the investment of base-line effort, only $Ter_3^* = 0.66$ is

Table 3. Paths generated/LCSAJ for various values of K.

K	*Mean value of* Ter_3: C_K	*Number of paths generated*	*Paths generated per LCSAJ*
0	0.66	324	0.50
10	0.76	519	0.80
20	0.81	689	1.06
30	0.82	855	1.31
40	0.85	1004	1.54
100	0.87	1784	2.74
200	0.88	2996	4.60
300	0.89	4096	6.28

achieved, and that in order to attain $Ter_3^* = 0.81$, approximately twice the base-line effort must be expended. Effort, in fact, increases approximately linearly with Ter_3^* until $Ter_3^* = 0.85$ is achieved, and thereafter, increases at an exponential rate. This is readily discernable from Fig. 2 which is a plot of paths generated per LCSAJ against Ter_3^*. From this graph, it can be seen that the indication

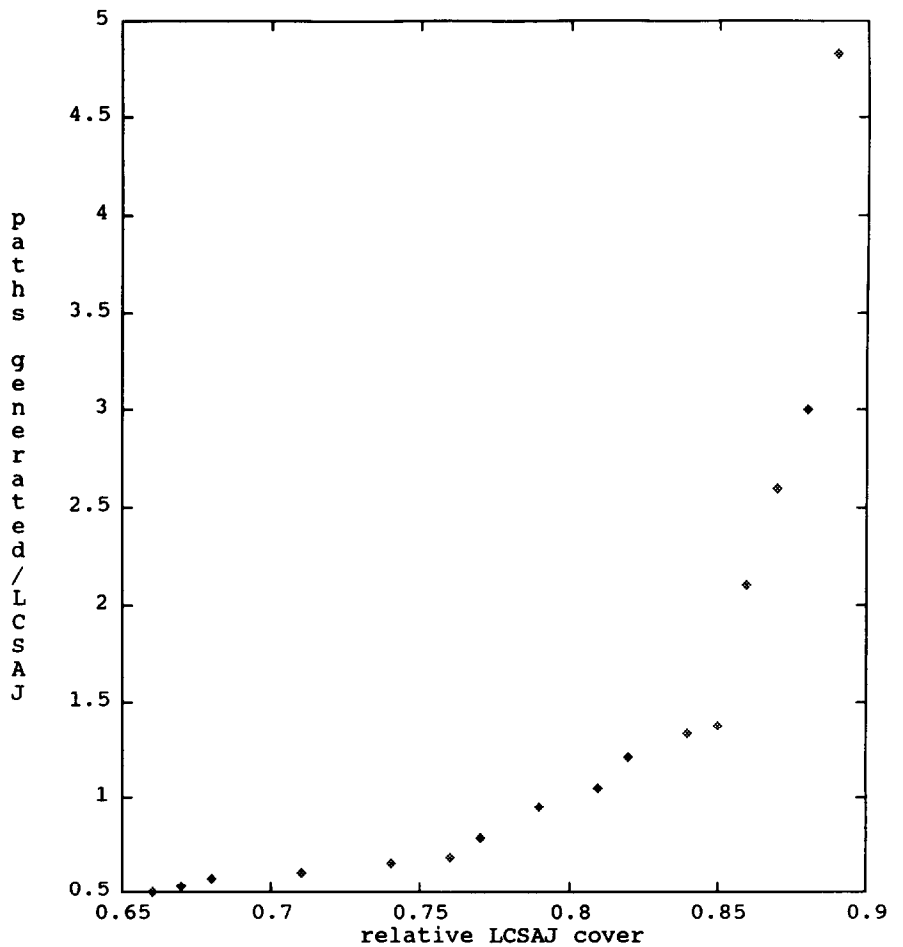

Fig. 2. A plot of paths generated/LCSAJ against Ter_3^*.

given in Fig. 1, namely, that it may not be cost-effective to generate more than about 40 additional paths in order to extend the feasible LCSAJ coverage of a piece of software beyond that attained by sub-strategy S1, is again strongly indicated. None the less, it may justifiably be concluded that the proposed path generation strategy demands only a most acceptable expenditure of effort in achieving levels of coverage up to $Ter_3^* = 0.85$, and that this attests to the strategies effectiveness.

A further perspective on effort and the influence of infeasible paths can be gained if certain of the results from the experiments described above are considered alongside comparable theoretical results obtained by Woodward (1984). In his work, Woodward investigated the number of paths through a piece of software that it would theoretically be necessary to test in order to achieve $Ter_3 = 1$ having once attained $Ter_2 = 1$. Of the sample used, he found that only 16% required more than ten extra paths, that for no piece of software did the number of extra paths exceed 58, and that the mean number of extra paths was 5.6. As a result of the above experiments, it was found that 22.86% of the sample required in excess of ten paths, and that this figure could be greater since for five subroutines (14.29% of the sample) not even $Ter_2 = 1$ had been achieved after generating 300 paths in accordance with sub-strategy S2. It was also found that for 2 subroutines (at least) more than 294 extra paths were required in attempting to achieve (but not actually achieving) $Ter_3^* = 1$, and that the mean number of extra paths was at least 47.49. The differences between these two sets of figures quite clearly illustrates the influence which infeasible paths can have on LCSAJ testing, and especially so when high levels of cover are demanded.

6. Discussion of results

The results described in Section 5.1 were derived having applied the proposed strategy to 35 subroutines containing a total of 652 LCSAJs, and during the experimentation, almost 4100 paths were investigated. The authors' believe that, in comparison with the very few published results which relate to the practical aspects of LCSAJ testing, the study described in this paper is a substantial one. None the less, the vitally important question: "how generally applicable are the results that have been obtained?", must be addressed.

Only two factors were instrumental in determining the results derived by the experiments. These are: the sample of software used, and the path selection strategy adopted. In order then to answer the above question it is sufficient (and necessary) to address two further questions, namely, "how representative of software in general are the 35 subroutines?", and "how typical is the effectiveness of the path selection strategy?".

Consider the first of these. In certain respects, the software sample used in the experiments may be viewed as being non-representative, for example, all 35 subroutines are coded in a common language and derive from a single source – the NAG Library. Despite this, the ultimate aim of the experiments was to shed some light on the extent of the effort necessitated by LCSAJ testing. Consequently, a representative sample should be viewed as one which will lead to a typical/average investment of effort, and since effort is measured in this paper by paths generated per LCSAJ for a given value of Ter_3^*, one which will result in a typical/average value for this quantity. However, these values depend upon the proportion of paths in a typical/average piece of software that are infeasible, and such information is unfortunately unavailable. None the less, consider Fig. 3 which is a histogram depicting the proportion of feasible kth shortest paths through LCSAJs of the sample used in the experiments of Section 5. From this it can be seen that as k increases (non-decreasing predicate involvement), the proportion of infeasible paths tends to increase. Now, if it can be assumed that the proposition cited in Section 2, namely, that path feasibility tends to decrease with increasing path predicate involvement, is indeed valid, then the proportion of

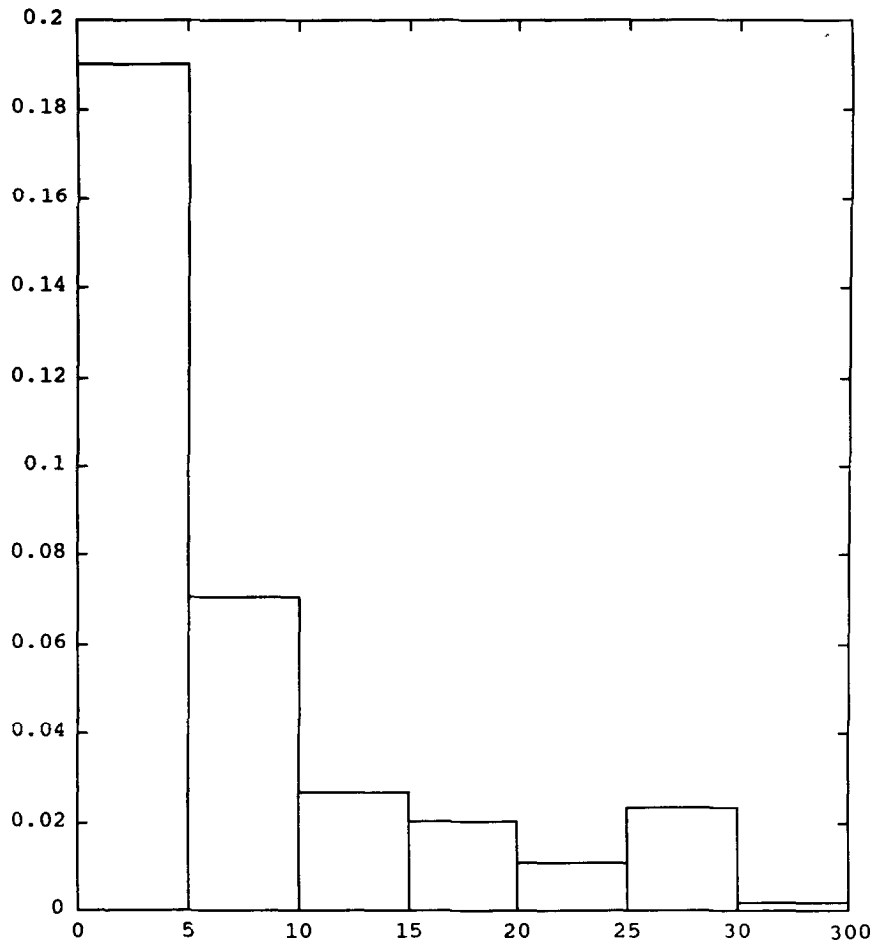

Fig. 3. Proportion of feasible kth shortest paths.

feasible paths involving p predicates in a representative software sample will decrease with increasing p. Under this assumption then, the nature of Fig. 3 lends support to the belief that the sample used in the experiments is representative.

Consider now the second question concerning the effectiveness of the path selection strategy employed in the experiments. This strategy is based only upon the proposition cited above (and in Section 2) which provides, in effect, a heuristic characterization of infeasible paths. (Weyuker, 1979, proved that no other kind of characterization is possible.) Assuming, once again, that the proposition is valid, the characterization it provides must be the best available since the literature contains no other practicable and valid alternative. Given, therefore, that the proposed path selection strategy embodies 'the best' characterization of infeasible paths, it is uniquely situated to take advantage of this in-built knowledge. Consequently, it should be anticipated that, on average, the proposed strategy will generate fewer infeasible paths, and therefore out-perform, all other path selection strategies which employ program structure as a basis for LCSAJ test path generation.

If then the sample used in the experiments is representative, and the proposed path generation strategy, as argued above, will on average out-perform all other structure based selection methods, the extent of the effort (as measured by paths generated per LCSAJ) to achieve the various levels of

LCSAJ coverage for the sample is likely to be the minimum that can be achieved at present. Furthermore, because Fig. 1 indicates that even the proposed strategy will attract an exponentially large amount of effort to achieve values of Ter_3 close to unity, the same will be true for all structure based path selection methods when applied to typical software samples! This situation, moreover, is unlikely to change unless a suitably improved characterization of infeasible paths is found – and that this can be done, is by no means certain. If, on the other hand, the sample is not in fact representative, the extent of the effects upon effort of the infeasible paths in the sample must necessarily be either greater or less than that in a representative sample. If it is greater, then the sample used provides evidence that, at least for some pieces of software, achieving a level of Ter_3^* close to unity will attract an exponentially large expenditure of effort, and if it is less, the same will be true but it will be true for the majority of software. In all, therefore, if the proposition of Section 2 is valid, achievement of $Ter_3^* = 1$ for some items of software will entail an exponentially large investment of effort if path selection takes no account of the detailed logic of the software under test.

7. Summary and concluding remarks

This paper has addressed LCSAJ testing, and in particular, the extent of the effort that, in practice, is entailed by employing the method. To facilitate an assessment of the effort, a path selection strategy which attempts to reduce, *a priori*, the incidence of infeasible paths in the test path set that it generates, has been proposed, and applied to a set of 35 subroutines from the NAG FORTRAN 77 library. The results of these experiments showed that a most acceptable level of LCSAJ coverage: 12 of the 35 subroutines covered completely, and a mean value of 0.66 for Ter_3^*, could be achieved for an expenditure of effort consistent with generating, on average, a quite modest 0.50 paths/LCSAJ. Higher levels of relative LCSAJ coverage in the range (0.66, 0.85) were also attained with effort, as measured in paths generated per LCSAJ, which increased only linearly with Ter_3^*. Achievement of feasible LCSAJ cover in excess of 0.85, however, was found to be subject to the law of diminishing returns. Specifically, the increase in effort was observed to become exponential in nature, and the coverage ultimately attained by the strategy, namely, full coverage of 24 of the 35 subroutines and a mean value for Ter_3^* of 0.89, required the generation of over 4000 paths; this being consistent with 6.28 paths/LCSAJ. (It is noted that the phrase: 'cover ultimately achieved by the strategy', should not be interpreted as implying that the strategy is incapable of attaining a higher mean cover. This is not the case. The limitation, forced by practical considerations, placed on the implementation of the strategy, namely, that no more than 300 paths/subroutine be generated in respect of sub-strategy S2, was responsible for preventing higher coverage levels from being achieved.) In all, the results indicate that a very reasonable level of LCSAJ coverage ($Ter_3^* = 0.85$) can be achieved with quite an acceptable expenditure of effort, and the authors contend that this fact also attests to the effectiveness of the proposed path selection strategy.

A criticism which is frequently levelled at empirical studies is that the samples they use may not be representative. Whence, the results derived from the studies cannot be generalized in any valid way and must, therefore, be viewed as having only limited applicability and value. Such criticism may have validity, but on the reverse of the coin, is the plain fact that, given the current state-of-the-art, a considerable amount of much needed information can only be obtained, perhaps in only approximate form, as a result of empirical investigation. The values quoted above are presented, at the very least, in this spirit. Notwithstanding, given the mild assumption of the validity of the proposition that path feasibility tends to decrease with increasing path predicate involvement, it was argued that there was evidence lending support to the belief that the 35 subroutines do form a

representative sample. It was further argued that because the proposed path selection strategy is based upon, and uniquely situated to take advantage of, the only (and therefore, the 'best') characterization of infeasible paths, the results derived using it might justifiably be viewed as being, in the mean, the best that can be achieved, at present, by any strategy/method which uses any but program logic as a basis for path selection. Consequently, the exponential growth in effort observed during the experiments when seeking levels of feasible LCSAJ coverage that approach unity should be viewed as a characteristic of not only the proposed strategy, but also all others for which detailed program logic does not form part of the foundation for path selection.

Acknowledgements

The authors express their gratitude to NAG Ltd. for permission to access the source code version of their Fortran 77 library, and to the referees for the comments they have made for improvement of the paper.

References

Bellman, R.E. and Dreyfus, S.E. (1962) *Applied Dynamic Programming* (Princeton University Press, Princeton, NJ).

Boffey, T.B. (1982) *Graph Theory in Operations Research* (Macmillan, London).

Chellappa, M. (1987) Nontraversible paths in a program. *IEEE Transactions on Software Engineering*, **SE-13**, 751–756.

DeMillo, R., Lipton, R.J. and Sayward, F.G. (1978) Hints on test data selection: help for the practicing programmer. *IEEE Computer*, **11**(4), 34–41.

DeMillo, R., McCracken, W., Martin, R. and Pasafiume, J. (1987) *Software Testing and Evaluation* (Benjamin Cummings, Mass, USA).

Dreyfus, S.E. (1969) An appraisal of some shortest path algorithms. *Operations Research*, **17**, 395–412.

Fosdick, L.D. and Osterweil, L.J. (1976) Data flow analysis in software reliability. *ACM Computing Surveys*, **8**, 305–330.

Gabow, H.N., Maheshwari, S.N. and Osterweil, L.J. (1976) On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, **SE-2**, 227–231.

Hedley, D. and Hennell, M.A. (1985) The causes and effects of infeasible paths in computer programs, in *Proceedings of the 8th International Conference on Software Engineering*, London, UK, pp. 259–266.

Hennell, M.A. (1991) How to avoid systematic testing. *Journal of Software Testing Verification and Reliability*, **1**(1), 23–30.

Howden, W.E. (1975) Methodology for the generation of program test data. *IEEE Transactions on Computing*, **24**, 554–559.

Liggersmeyer, P. (1994) A method for the selection of suitable unit test techniques. *Proceedings of EUROstar 94*, Brussels, Belgium, pp. 231–240.

Malevris, N. (1988) An effective approach for testing program branches and linear code sequences and jumps, PhD thesis, University of Liverpool, UK.

Malevris, N., Yates, D.F. and Veevers, A. (1990) A predictive metric for the likely feasibility of program paths. *Information and Software Technology*, **32**(2), 115–119.

Ntafos, S.C. (1988) A comparison of some testing strategies. *IEEE Transactions on Software Engineering*, **SE-14**, 868–874.

Paige, M.R. (1975) Program graphs, an algebra and their implications for programming. *IEEE Transactions on Software Engineering*, **SE-1**, 286–291.

Shier, D.R. (1976) Iterative methods for determining the K shortest paths in a network. *Networks*, **6**, 205–229.

Weyuker, E.J. (1979) The applicability of program schema results to programs. *International Journal of Computing and Information Science*, **8**, 387–403.

Woodward, M.R., Hedley, D. and Hennell, M.A. (1980) Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, **SE-6**, 278–286.

Woodward, M.R. (1984) An investigation into program paths and their representation. *Techniques et Science Informatiques*, **3**, 273–286.

Yates, D.F. and Hennell, M.A. (1985). An approach to branch testing, in *Proceedings of the 11th Workshop on Graph Theoretic Techniques in Computer Science*, Wurtzburg, West Germany, pp. 421–433.

Yates, D.F. and Malevris, N. (1989) Reducing the effect of infeasible paths in branch testing, in *Proceedings of the 3rd Symposium on Software Testing, Analysis and Verification (TAV3)*, Key West, Florida, USA, pp. 48–56.