

Procedures for Reducing the Size of Coverage-based Test Sets

A. Jefferson Offutt*
4A4, ISSE Department
George Mason University
Fairfax, VA 22030
Phone: 703-993-1654
Fax: 703-993-1638
Email: ofut@isse.gmu.edu

Jie Pan *
Mail Stop C204
PRC, Inc.
12005 Sunrise Valley Drive
Reston, VA 22091
Phone: 703-620-8604
Email: jpan@isse.gmu.edu

Jeffrey M. Voas
Reliable Software Technologies Corp.
Loudon Tech Center
Suite 250
21515 Ridgetop Circle
Sterling, VA 20166 USA
Phone: 703.404.9293
Fax: 703.404.9295
Email: jmvoas@rstcorp.com

Abstract

This paper addresses the problem of reducing the size of test sets for regression testing and test output inspection. Since regression testing requires the execution of some, and in the worst case, all test cases, reducing the number of tests can have a large benefit. Additionally, testers generally have to examine the output of each test case, both during initial and regression testing. Since this is done by hand, reducing the number of outputs that need to be examined can reduce the cost of testing. We observe that for mutation-based test sets, the **order** in which the test cases are executed impacts the size of the test sets. This paper presents several strategies for selecting a smaller number of test cases by reordering the test tests. We illustrate our technique using a proof-of-concept empirically study using mutation testing, achieving approximately a 33% reduction in size, and a corresponding reduction in the cost of regression testing, with a cost of only one extra run of the test case set. We suggest that these results should be extendable to apply to any test strategy that includes a quantifiable measure of test case effectiveness, such as data flow testing and branch testing, and try it with statement coverage with positive results.

1 INTRODUCTION

A *testing criterion* is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*; a test set achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed; partial coverage is defined to be the percent of requirements that are satisfied. *Test requirements* are

*Partially supported by the National Science Foundation under grant CCR-93-11967.

specific things that must be satisfied or covered; e.g., for statement coverage, each statement is a requirement, in mutation, each mutant is a requirement, and in data flow testing, each DU pair is a requirement. Test cases have traditionally been created by hand to satisfy a test criteria, but recent advances are leading to tools that generate test cases automatically. Even with manual test data generation, the test sets that are derived to satisfy the criterion will not necessarily be the smallest set possible. Automatic test generation often results in even larger test sets. The size of the test sets have a direct bearing on the cost of testing, particularly that of regression testing. When tests must be run repeatedly for every change in the program, it is of enormous advantage to have as small a set of test cases as possible. Thus, it is of great practical advantage to reduce the size of test case sets.

In this paper, we present a solution to the problem of reducing the size of test sets generated for some testing criterion [LH90]. Although our experiments use automatically generated test cases, our techniques also apply to manually created tests. We present heuristics to reduce test set sizes based on reordering the test execution sequences. We have evaluated these heuristics empirically with mutation testing [DLS78] and statement coverage [Mye79] and found that they can have an average of about 33% reduction, with a cost of only one extra run of the test case set.

Our conjecture is that this reordering technique can be of benefit when using other criteria as well; in fact, it should apply to any testing criterion that includes a quantifiable measure of test case quality.

Software testing is essentially a two-phase process: (1) generate test cases, and (2) test using those test cases. Given that the first phase can be almost totally automated for mutation testing, it is the second phase that will likely be the more costly since human effort may be required to check the results of a test. A scheme that decreases the number of test cases used in the second phase (while preserving the coverage criterion) could decrease the overall costs of testing. In the remainder of this section we discuss mutation testing, automatic generation of test cases, test case ordering, and present a procedure for evaluating our ideas. In the subsequent sections, we present our test set reduction procedures, results from our empirical evaluation, and a discussion with related work.

1.1 Mutation Testing

Fault-based testing techniques help the tester develop test cases that detect a well-defined class of faults. Mutation testing is a fault-based testing technique introduced by DeMillo et al. [DLS78] and Hamlet [Ham77]. Mutation testing is based on the assumption that a program will be well tested if all so called “simple faults” are detected and removed. The coupling effect [DLS78, Off92] states that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults.

Simple faults, called *mutations*, are introduced into the program by *mutation operators*. Each mutation

created by a mutation operator represents a *mutant program*. A mutant is *killed* by a test case that causes the mutant program to produce incorrect output, dead mutants are not executed against later test cases. *Equivalent mutants* are mutant programs that are functionally equivalent to the original program and therefore cannot be killed by any test case. The goal of mutation is to find test cases that kill all non-equivalent mutants; a test set that does so is said to be *adequate relative* to mutation. The *mutation score* is the ratio of the number of dead mutants to the number of non-equivalent mutants; it measures the adequacy of the test case set.

1.2 Generating Test Cases

One of the most difficult technical tasks in testing software is that of generating the test case values needed to satisfy the testing criteria. Constraint-based test data generation (*CBT*) is a collection of procedures designed to create test data that satisfy mutation [DO91]. CBT generates test data by explicitly targeting each mutation of a program. For each mutation, a system of mathematical constraints is generated that represents the conditions under which that mutation should die. These constraint systems are then satisfied to generate test cases. The number of test cases that are initially generated is bounded by the number of mutants.

Common wisdom says that when the cost to generate the next test case is greater than the cost of applying that test case, then testing should end. But with an automated test case generator, the only cost associated with finding that next test case is computer time. Hence automated test case generation is clearly cost effective when compared to manual test case generation.

Although large numbers of test cases are generated by automated test data generators (such as Godzilla [DO91] and PiSCES [Voa92]), only a fraction are necessary. For mutation testing, an *effective* test case is one that kills at least one mutant; an *ineffective* test case does not kill any mutants. More generally, an *effective* test case is one that contributes to satisfying the testing criterion. Typically, only about 10% of the test cases created by automatic generators are effective. DeMillo and Offutt [DO93] observed that an automatic test data generation capability allows us to view test cases as “throw-away” items rather than expensive, scarce resources. With this view, we can generate test cases, toss them at mutants, and then throw them away if they do not work.

Still, the number of test cases generated to satisfy mutation tends to be higher than for other structural-based techniques. For example, Frankl et al. [FWH94] found that a tester needs more test cases to achieve the same level of mutation coverage as data flow coverage.

1.3 Ordering of Test Sets

Test data generation strategies for mutation order N inputs into a sequence $S = \langle s_1, s_2, \dots, s_N \rangle$. These inputs are fed into the mutants of a program P , in sequence, with the goal being for the inputs to cause as

```

IF (X > 0) THEN
  Statement_1;
ELSE
  Statement_2;
END IF;
IF (Y > 0) THEN
  Statement_3;
ELSE
  Statement_4;
END IF;

```

Test case 1	X = -1, Y = 5	Executes Statement_2 and Statement_3 .
Test case 2	X = 5, Y = 5	Executes Statement_1 and Statement_3 .
Test case 3	X = -1, Y = -1	Executes Statement_2 and Statement_4 .

Figure 1: **Example of the Effect of Ordering on Test Case Selection.**

many mutants as possible to fail. When mutants fail, or are *killed*, by input s_i , the mutants are removed from the set, and input s_{i+1} is fed into the set of remaining mutants. As the tests are executed, the set of mutants is continually shrinking, until either all mutants are killed or the remaining mutants appear to be impossible to kill off.

Call the set of mutants for a program M , and assume that the cardinality of M is K . For input s_1 , M is a complete set, and hence there are K opportunities for s_1 to cause a member of M to fail. If s_1 kills ε mutants, then the most mutants that s_2 has the opportunity to kill is $K - \varepsilon$. If s_2 kills γ mutants, then the most mutants that s_3 has the opportunity to kill is $K - \varepsilon - \gamma$. Thus, the earlier in the sequence that an input occurs, the more likely it is that the input will kill a mutant. Thus there is a degree of “luck” involved, whose effect on the mutation testing process we would like to decrease.

As a related example, consider the program and test cases in Figure 1, where **Statement_1** through **Statement_4** are arbitrary statements. If the test cases are submitted in the order given, then all three test cases are required to cover all four statements. On the other hand, if we execute the test cases in the order $\langle 3, 2, 1 \rangle$, then only test cases 3 and 2 are needed to cover the statements.

We have observed that the first inputs selected during mutation testing kills a disproportionately large number of mutants as compared to later inputs. For instance, the first few inputs may kill off 50% of the mutants, while it may require thousands of later inputs to kill another 40% of the mutants. This suggests that some members of M are “easy targets,” and practically any input could kill them. This is why the first inputs of S appear to have a powerful ability to kill mutants, when in reality they may not.

This suggests that the first members of S may not be particularly good at killing mutants, but rather: (1) were lucky enough to have been selected early, and (2) had the opportunity to kill the mutants that were

easy targets. This paper demonstrates that it is possible and practical to reduce the effect of this luck factor while also reducing the size of the final set of inputs.

2 REDUCTION PROCEDURES

Assume we have a set of test cases T with some reasonably high mutation score for a program, and T has size N . A *minimal* test set is the smallest subset of T that achieves the same mutation score. Since which test cases are effective depends on the order in which the test cases are executed, we need to run all orderings of the test set to find the minimal test set.

This problem can be defined formally as a minimal set cover problem. Each mutant is *covered* by a test case that kills it. Thus, to find the minimal test set that satisfies the mutation criterion, we need to find the smallest set of test cases that covers the mutants. This is an NP-complete problem [GJ79]. Although NP-complete problems are seldom solved completely, good approximate solutions can usually be found using heuristics. We propose three basic orderings for executing test cases, which lead to a total of 12 heuristic reduction procedures. Our procedures run the tests in different orders, with the goal of increasing the “difference” between the original and subsequent orderings as much as possible. The three basic orderings therefore go from the beginning to the end, the end to the beginning, and the middle to both ends:

1. **Forward procedure.**

In the forward procedure, we have a test set T of size N that is ordered in some sequence, and run all the test cases against the mutants in forward order. That is, if $seqT = \langle t_1, t_2, \dots, t_N \rangle$, we run in the order t_1, t_2, \dots, t_N . We then dispose of all the ineffective test cases, leaving a (possibly) smaller set T_f of size N_f .

2. **Reverse procedure.**

In the reverse procedure, we have a test set T of size N in some sequence, and run all the test cases against the mutants in reverse order. That is, if $seqT = \langle t_1, t_2, \dots, t_N \rangle$, we run in the order t_N, t_{N-1}, \dots, t_1 . We then dispose of all the ineffective test cases, leaving a smaller set T_r of size N_r .

3. **Inside_out procedure.**

In the inside_out procedure, we have a sequence $seqT = \langle t_1, t_2, \dots, t_N \rangle$, with size N , and run it from the middle test case to both ends. If N is an odd number, we run in the order $t_{(N+1)/2}, t_{(N+1)/2-1}, t_{(N+1)/2+1}, \dots, t_1, t_N$; if N is an even number, we run in the order $t_{N/2+1}, t_{N/2}, t_{N/2-1}, \dots, t_1, t_N$. We then dispose of all the ineffective test cases, leaving a smaller set T_i of size N_i .

Given the above three basic procedures, we define 12 reduction heuristics that are combinations of these basic procedures. We collectively call these the *ping-pong* heuristics, since they all involve some sort of

bouncing from one end to the other of the sequence of test cases. Our philosophy behind these heuristics is simple – we try to maximize the difference in the ordering of the test sets. Since the first test case will always kill at least one mutant, we can think of the mutation process as applying a very **weak** filter to that test case. The last few effective test cases, however, are executed with only a few mutants still alive, and we can think of the filter as being very **strong**. Thus, we assume that the last few test cases are in some sense “better” than the first few test cases. The ping-pong heuristics we applied in our experiments were forward-reverse, reverse-forward, forward-inside_out, reverse-inside_out, reverse-forward-inside_out, and forward-reverse-inside_out.

3 EMPIRICAL EVALUATION

We chose to evaluate our method first at the unit level, using 10 program units that cover a range of applications. These programs range in size from 10 to 48 executable statements, and have from 183 to 3010 mutants. We used two tools to generate and evaluate our test data. The Mothra mutation system [DGK⁺88, DO91] automates the process of mutation testing by creating and executing mutants, managing test cases, and computing the mutation score. Mutants are created by *mutation operators*, which are rules that describe syntactic changes to elements of the program. We used all twenty-two Mothra mutation operators [KO91] for this study. To generate test data to satisfy mutation, we used Godzilla, an automated constraint-based test case generator that is integrated with Mothra [DO91]. While Godzilla is able to generate test cases that kill most mutants for most programs, it does not kill all non-equivalent mutants, and if a fully mutation-adequate test set is needed, some test cases are usually added by hand. We felt that for this study it was not necessary to have test sets that were fully mutation-adequate, and adding test cases by hand could introduce a bias, therefore, our test sets were not fully mutation-adequate. Most mutation scores were over 90%. To avoid any bias that could be introduced by any particular test set, we generated five independent test sets for each program, for a total of fifty test case sets.

The process that we used in our study is depicted in Figure 2. Each box in the figure represents one test case set. The top box containing T represents the original test set generated by Godzilla. The arcs represent part of our empirical process; for example, the **forward** arc coming out of the top box represents running the forward procedure on T , which results in the set T_f . Each box below the root describes the process used to get to that box, the test set derived, and its relationship with the previous set. The test set derived from T via the forward procedure is called T_f , its size is N_f , it is a subset of T , and its size should be much smaller than N .

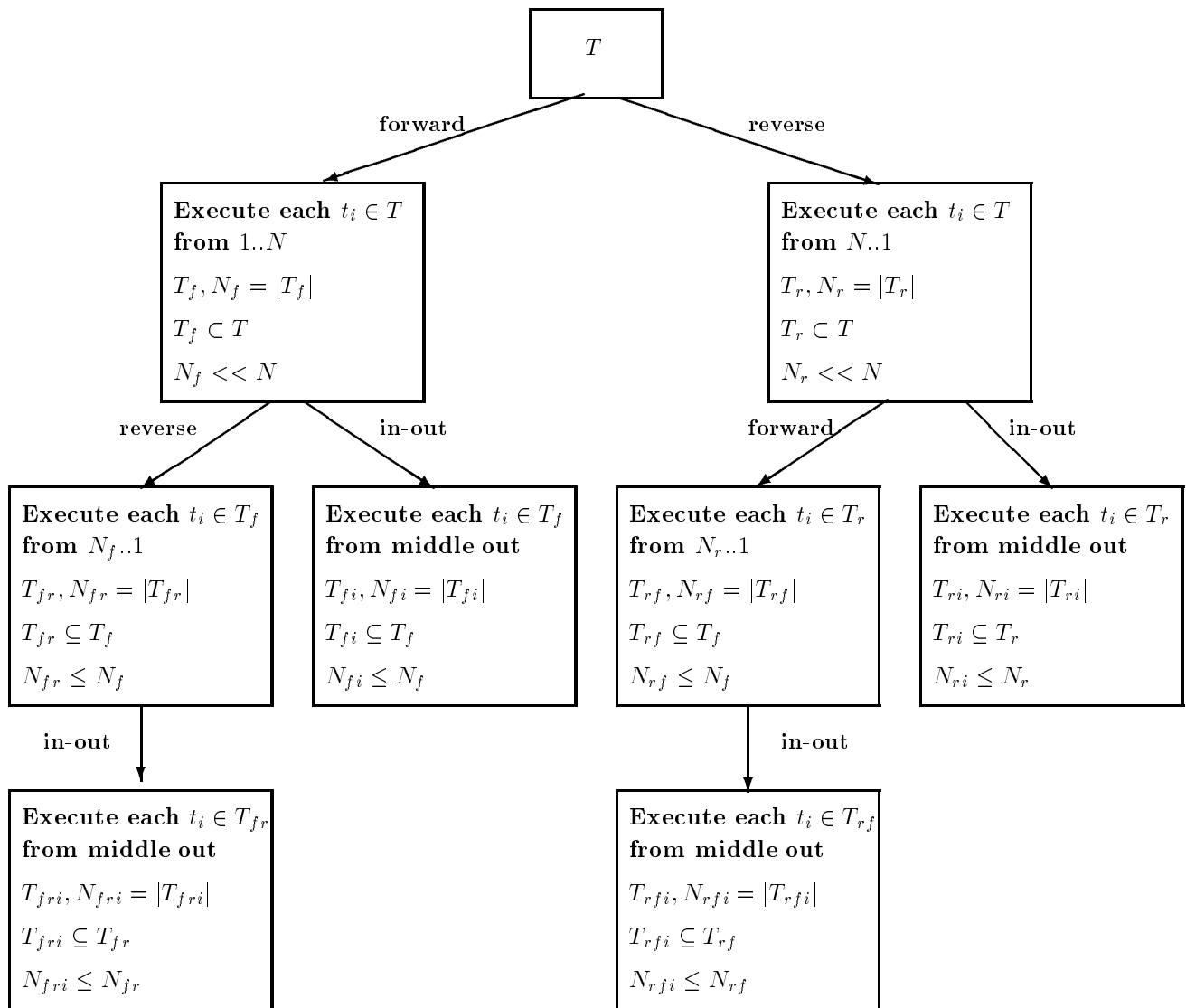


Figure 2: **Heuristics in Empirical Procedure.**

3.1 Empirical Results

Our results are shown in Tables 1 and 2 and Figure 3. The sizes of the test sets for each program are in Table 1. Since we created five sets of test data for each program, these sizes are averaged over the five sets.

We define savings to be *the percentage of test cases eliminated from the initial set of effective test cases*. For example, the savings for the set T_{fr} is $(N_f - N_{fr})/N_f$, and the savings for the set T_{fri} is $(N_f - N_{fri})/N_f$. Table 2 shows the percent savings for each test set for each program. Again, these savings are averaged over the five test sets for each program. Although S_{rf} is larger than S_{fr} , the difference is small. The implication is that we get slightly more savings with T_{rf} because T_f had slightly fewer test cases to start with.

The savings of our six heuristics are summarized in Figure 3. For convenience, we show this in the same format as Figure 2. Since we define our savings in terms of the initial effective set of test cases, the savings

Program	N_f	N_{fr}	N_{fi}	N_{fri}	N_r	N_{rf}	N_{ri}	N_{rfi}
Bub	5.20	3.40	4.20	3.40	6.20	3.80	4.60	3.80
Cal	30.00	16.40	24.20	16.40	31.60	16.40	23.40	16.40
Euclid	3.40	2.80	3.00	2.80	3.80	2.80	3.00	2.80
Find	15.20	10.20	12.40	10.00	18.40	10.60	14.20	10.60
Insert	3.80	2.40	3.00	2.40	4.40	2.80	3.40	2.80
Mid	15.60	12.80	14.40	12.80	15.20	12.80	13.80	12.80
Pat	17.40	10.00	13.40	10.00	17.00	10.00	12.60	10.00
Quad	6.00	5.40	5.80	5.40	6.20	5.60	6.20	5.60
Trityp	36.80	27.80	32.00	27.80	37.80	28.60	33.80	28.60
Warshall	6.60	3.60	4.80	3.60	6.00	2.80	4.40	2.80
Average	14.00	9.48	11.72	9.46	14.66	9.62	11.94	9.62

Table 1: **Test Case Set Sizes**

Program	$Save_{fr}$	$Save_{fi}$	$Save_{fri}$	$Save_{rf}$	$Save_{ri}$	$Save_{rfi}$
Bub	0.35	0.19	0.35	0.39	0.26	0.39
Cal	0.45	0.19	0.45	0.48	0.26	0.48
Euclid	0.18	0.12	0.18	0.26	0.21	0.26
Find	0.33	0.18	0.34	0.42	0.23	0.42
Insert	0.37	0.21	0.37	0.36	0.23	0.36
Mid	0.18	0.08	0.18	0.16	0.09	0.16
Pat	0.43	0.23	0.43	0.41	0.26	0.41
Quad	0.10	0.03	0.10	0.10	0.00	0.10
Trityp	0.24	0.13	0.24	0.24	0.11	0.24
Warshall	0.45	0.27	0.45	0.53	0.27	0.53
Average	0.32	0.16	0.32	0.34	0.19	0.34

Table 2: **Savings of Test Case Set Sizes**

of .32 for T_{fri} means that no additional savings were made as a result of the inside_out procedure. The two sets T_{fr} and T_{rf} clearly resulted in more savings than any of the sets involving the inside_out procedure. The forward-reverse and reverse-forward procedures eliminated an average of one third of the test sets. This leads us to conjecture that the forward-reverse and reverse-forward versions of ping-pong result in test sets that are almost minimal.

3.2 Statement Coverage Results

To test our conjecture that the ping-pong procedure can be used with other coverage-based techniques besides mutation, we tried it with statement coverage. We chose a C program of about 200 lines and 22 basic blocks, and used the Unix coverage tool `tcov` to measure statement coverage. We had a colleague generate five separate sets of test data that covered all statements in the program. This data was generated by hand without knowledge of the intent of our study.

As would be expected, it takes far fewer test cases to satisfy statement coverage than mutation. We took only the effective test cases T_f , and ran them in reverse (getting the T_{fr} set). As Table 3 shows, this procedure reduced the number of test cases by an average of about 50%. Of course, this is only one program, but it gives us hope that our reduction procedure will work for non-mutation testing techniques.

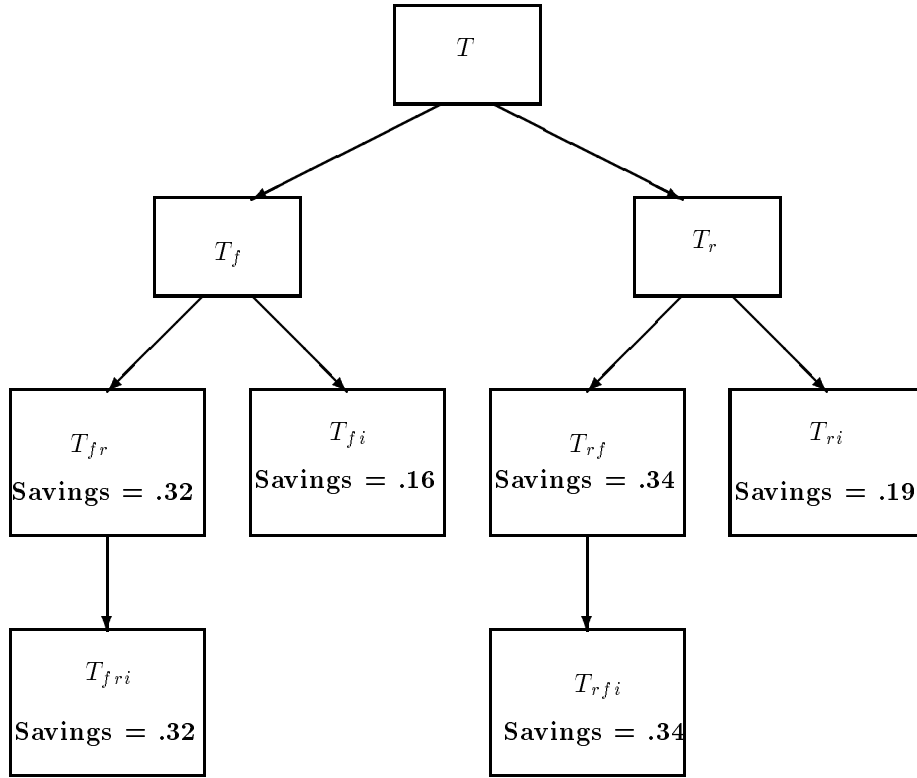


Figure 3: Summary of Savings in Empirical Procedure.

Test Case Set	N_f	N_{fr}	$Save_{fr}$
1	4	2	0.50
2	4	1	0.66
3	3	2	0.33
4	3	1	0.66
5	3	1	0.50
Average	3.0	1.4	0.53

Table 3: Reduction of Statement Coverage Test Sets

3.3 Cost of the Reduction Procedures

There is an extra cost required to get these savings. To reduce the test set, we must run the entire test set again. But if the test set will be used repeatedly during regression testing, the following analysis shows that the cost can be worthwhile.

Assume an initial test set T and a reduced test set T' . The cost of running the test sets can be called C_T and $C_{T'}$, respectively. Since our reduction procedure comes after test data generation, it is independent of the technique used to generate the test data. Thus, we can ignore the cost of test data generation in this analysis. If we assume the cost of running the test sets is proportional to the size of the test sets, and that

the 33% reduction from Section 3.1 is a valid estimate, then we have the following formulae:

$$\begin{aligned} |T'| &\approx .66|T| \\ C_{T'} &\approx .66C_T \end{aligned} \tag{1}$$

If we expect N complete regression test runs, then the expected cost of testing without using reduction procedures will be:

$$C_T + NC_T = (N + 1)C_T \tag{2}$$

Using our reduction procedures will require one extra run of the original test set, thus the cost will be:

$$2C_T + NC_{T'} \tag{3}$$

Substituting from equation 1, we get:

$$2C_T + .66NC_T = (2 + .66N)C_T \tag{4}$$

Clearly, this is less than equation 2 as N gets large, but where is the break-even point? If we factor C_T out of equations 2 and 4, then $(.66N + 2) < (N + 1)$ when N is greater than or equal to 3. So if the 33% reduction in test set size is a valid estimate, this technique can be expected to be cost-effective when 3 or more regression test runs are expected.

4 RELATED WORK

Harrold, Gupta, and Soffa [HGS90] have also studied this problem, and propose a heuristic that contrasts in an interesting way with ours. Their paper presents the problem as that of finding a hitting set, formulated as follows. A test case set TS is created to cover a list of testing requirements r_1, r_2, \dots, r_n that must be tested to provide the desired testing coverage. TS is divided into a list of subsets T_1, T_2, \dots, T_n , where each T_i is associated with r_i , and T_i contains all the test cases t_j that can test requirement r_i . With this description, the problem is that of finding the smallest set of test cases that will satisfy all of the r_i s.

The procedure of Harrold et al. chooses a “representative” set of test cases from the subsets using a standard hitting set heuristic [AHU74]. First, a smallest subset T_i is chosen. From that subset, a test case t_j is chosen such that t_j is in the maximum number of subsets T_1, T_2, \dots . Then, t_j is added to the representative set, and all the requirements that are tested by t_j are removed from consideration. This process continues until all requirements are removed. If n is the number of test sets T_i , and m is the number of test cases t_i , the paper shows that the running time of their algorithm is $O(n * \text{Max}(n, m))$ [HGS90].

A similar procedure is used in the ATAC [HL92] data flow testing tool, which uses an implicit enumeration of the subsets of a test set (based on techniques used in some integer programming software) [Hor94]. This is also similar to the procedure suggested by Lee and He [LH90].

Harrold et al.’s technique is somewhat more expensive than the ping-pong procedure presented here, and uses more information (specifically, the information as to which test cases satisfy each requirement). Their

analysis does not include the time to generate the subsets T_i , which would typically require execution of the program on each test case at least once, and for some testing criteria, once for each testing requirement. On the other hand, it seems likely that their technique would be at least as effective as ours, and probably more effective. Experimental studies are currently being planned to compare the two techniques.

5 CONCLUSIONS

In this paper, we have introduced a procedure for reducing the size of test case sets, and presented empirical evidence for its usefulness. This technique, called ping-pong, is based on applying heuristics to try to find the minimal test set that satisfies a testing criterion. We have presented empirical evidence that ping-pong can reduce mutation-based test sets by over 30%, and statement coverage-based test sets by as much as 50%. Of course, this is a small study, and cannot be deemed conclusive without being replicated under different conditions and assumptions. In Section 3.2, we presented one small study where the procedure also worked for statement coverage.

Reducing our set of test cases can reduce the cost of regression testing when using mutation as a criterion for generating test sets. During maintenance, having the smallest set of test cases that satisfy some testing criteria (to test with after modifications are made to the system) can provide long-term cost savings. This scheme will also reduce the cost of initial testing by reducing the number of test case outputs that have to be evaluated (the oracle function), a factor that was not included in our cost analysis.

The set of possible test sets, and their orderings, created for mutation testing is effectively infinite, and hence the scheme proposed here is just one combination from that set. This scheme is not providing the user with the minimal mutation-based test set, but rather a set that has reduced the impact of the ordering of S , while preserving the mutation score. Although our presentation and experimentation were in terms of mutation testing, it seems likely that ping-pong is generally applicable. It can be used with any testing technique that offers a quantitative measure of test case effectiveness. If the test cases were generated to satisfy one of the branch testing criteria [Mye79], then we can just as easily run the test cases in reverse and find a smaller set that satisfies the criterion. This will also work for statement testing and any of the data flow testing criteria [FW88]. Although our data does not indicate how much savings can be expected, this general applicability means that our procedures can be used within the context of virtually any testing process.

6 FUTURE WORK

Although our procedures will certainly work with other testing criteria, our empirical results from Section 3.1 only apply to mutation testing. In the future, we plan to repeat these experiments with data flow and branch criteria to measure the amount of savings that can be expected when using those criteria. We conjecture

that ping-pong should produce similar savings for these other structural criteria, because once again, there is the same luck factor associated with the ordering of test cases during test case generation.

One implicit assumption in our experiment is that the testing criteria being used is the most valid measure of test case quality. Other measurements could be the relative abilities of the test sets to detect actual faults in the program, or other testing criteria. In the future, we hope to compare the full and reduced sets of test cases using other criteria.

We have plans currently underway to compare the ping-pong procedure with the techniques by Harrold, Gupta, and Soffa [HGS90]. We expect to see a cost versus benefit tradeoff, and hope that an empirical comparison will lead to practical suggestions about when to use which technique. We also would like to compare our procedure within the framework of Rothermel and Harrold [RH94].

7 ACKNOWLEDGEMENTS

We would like to thank Paul Ammann for helpful discussions, and particularly for pointing out that test cases that are executed early in the testing process have a weaker filter applied than test cases executed late in the process.

References

- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company Inc, Reading, MA, 1974.
- [DGK⁺88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.
- [DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [DO93] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.
- [FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
- [FWH94] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. Technical report PUCS-100-94, Department of Computer Science, Polytechnic University, Brooklyn, NY, February 1994. Submitted for publication.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, New York NY, 1979.

- [Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.
- [HGS90] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. In *Conference on Software Maintenance-1990*, pages 302–310, San Diego, CA, Nov 1990.
- [HL92] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.
- [Hor94] J. R. Horgan. Personal communication, March 1994.
- [KO91] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.
- [LH90] J. A. N. Lee and Xedong He. A methodology for test selection. *The Journal of Systems and Software*, 13:177–185, 1990.
- [Mye79] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.
- [Off92] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [RH94] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 201–210, Sorrento, Italy, May 1994. IEEE Computer Society Press.
- [Voa92] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8), August 1992.