2018

A cost-effective software testing strategy employing online feedback information

Zhiquan Zhou
University of Wollongong, zhiquan@uow.edu.au

Arnaldo Sinaga
Del Institute of Technology, Indonesia, ams939@uowmail.edu.au

Willy Susilo
University of Wollongong, wsusilo@uow.edu.au

Lei Zhao
Beijing University of Aeronautics and Astronautics

Kai-Yuan Cai
Seihang University, kycai@buaa.edu.cn

A cost-effective software testing strategy employing online feedback information

Abstract

An online partitioning strategy is presented, in which test cases are selected based on feedback information collected during the testing process. The strategy differs from con- ventional approaches because the partitioning is performed online rather than off-line and because the partitioning is not based on program code or specifications. It can, therefore, be implemented in the absence of the source code or specification of the program under test. The cost-effectiveness of the proposed strategy has been empirically investigated with a set of subject programs, namely, SPACE, SED, GREP, and the Siemens Suite of Programs. The results demonstrate that the proposed strategy constantly achieves large savings in terms of the total number of test case executions needed to detect all faults.

Keywords

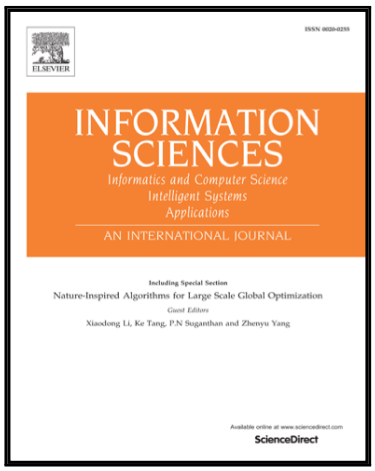
employing, information, online, cost-effective, software, testing, strategy, feedback

Disciplines

Engineering | Science and Technology Studies

Publication Details

Zhou, Z., Sinaga, A., Susilo, W., Zhao, L. & Cai, K. (2018). A cost-effective software testing strategy employing online feedback information. Information Sciences, 422 318-335.



A cost-effective software testing strategy employing online feedback information

Zhi Quan Zhou, Arnaldo Sinaga, Willy Susilo, Lei Zhao, Kai-Yuan Cai

PII: S0020-0255(17)30920-9
DOI: 10.1016/j.ins.2017.08.088
Reference: INS 13093

To appear in: *Information Sciences*

Received date: 26 September 2009
Accepted date: 28 November 2015

Please cite this article as: Zhi Quan Zhou, Arnaldo Sinaga, Willy Susilo, Lei Zhao, Kai-Yuan Cai, A cost-effective software testing strategy employing online feedback information, *Information Sciences* (2017), doi: 10.1016/j.ins.2017.08.088

A cost-effective software testing strategy employing online feedback information ⋆

Zhi Quan Zhou [a,*], Arnaldo Sinaga [a,**], Willy Susilo [a], Lei Zhao [b], Kai-Yuan Cai [b]

[a] *School of Computer Science and Software Engineering*
University of Wollongong
Wollongong, NSW 2522, Australia
Email address: {zhiquan, ams939, wsusilo}@uow.edu.au

[b] *Department of Automatic Control*
Beijing University of Aeronautics and Astronautics
Beijing 100191, China
Email address: zhaolei@asee.buaa.edu.cn, kycai@buaa.edu.cn

Abstract

An online partitioning strategy is presented, in which test cases are selected based on feedback information collected during the testing process. The strategy differs from conventional approaches because the partitioning is performed online rather than off-line and because the partitioning is not based on program code or specifications. It can, therefore, be implemented in the absence of the source code or specification of the program under test. The cost-effectiveness of the proposed strategy has been empirically investigated with a set of both large and small subject programs, namely, SPACE, SED, GREP, and the Siemens Suite of Programs. The results demonstrate that the proposed strategy constantly achieves large savings in terms of the total number of test case executions needed to detect all faults.

Key words: software testing, random testing, partition testing, dynamic partitioning, online feedback, early fault detection

⋆ A preliminary version of this paper was presented at the International Conference on Quality Software (QSIC'09) [36].

* All correspondence should be addressed to Dr. Zhi Quan Zhou.

**Arnaldo Sinaga is also with Del Polytechnic of Informatics, North Sumatra, Indonesia.

1 Introduction

Testing is a very expensive part of software production and maintenance. It is estimated that, in general, testing consumes between 30 to 50 percent of the software development time [27]. Random testing (RT) [23] is a basic testing method, and often serves as a benchmark against which the effectiveness of other testing techniques are measured. In RT, test cases are selected randomly from the input domain. It can, therefore, avoid the overhead incurred in code- or specification-based partitioning strategies. RT is often employed to test real-world products and incorporated into test case generation tools [1,4,17,18,24,34].

Malaiya introduced an *antirandom testing* technique [26] to enable quicker detection of the first failure. In other words, antirandom testing attempts to reduce the number of test cases needed to be run before the first failure can be revealed. Compared with RT, which requires each test case to be selected randomly regardless of the previously executed test cases, antirandom testing only selects the first test case randomly, and each subsequent test case is selected by choosing the one with the maximum total distance to all the previously executed test cases. Antirandom testing requires that the total number of test cases be known in advance. Its randomness is also limited since only the first test case is chosen randomly and the sequence of subsequent test cases is deterministic.

In order to detect the first failure quicker without the limitations of antirandom testing, an *adaptive random testing* (ART) strategy has been proposed [2,8–10,13,25]. ART is based on the intuition that when failure-causing inputs are clustered, selecting an input close to previously executed non-failure-causing test cases will be less likely to detect a failure. ART, therefore, proposes to have test cases evenly spread over the entire input domain. It differs from antirandom testing in that it preserves the randomness since all test cases in ART are randomly selected and ART does not require the predetermination of the total number of test cases to be run.

A related technique is known as *adaptive testing*, developed by Cai et al. [6]. Both the objective and the approach of adaptive testing is different from ART. ART was developed as an enhancement to RT with the objective of using fewer test cases to detect the first failure. Adaptive testing, on the other hand, adjusts the selections of test actions online following the idea of adaptive control, to achieve an optimization objective, such as minimizing the total cost of detecting and removing multiple faults. Adaptive testing involves off-line partitioning of the input domain. It has been shown that partitioning strategies can have a significant impact on testing effectiveness [6]. While various partitioning techniques have been proposed [12,14,29], most of these techniques partition the input domain off-line and they incur non-trivial overheads.

In order to find a way of achieving effective partitioning without heavy overheads,

Cai et al. conducted a pilot study [7] following the idea of *software cybernetics*, which explores the interplay between software and control [5]. Cai et al. [7] adopted a new testing paradigm. First, a very large test suite is given. Second, test cases are selectively executed through dynamically partitioning the test suite online. Their strategy is based on the intuition that a test case that has detected a failure previously should have a higher capability of detecting a failure again during the evolution of the software under test. In their approach, there are two partitions: partition 0 and partition 1. Initially, all test cases are stored in partition 1, and partition 0 is empty. A test case t is randomly selected from partition 1 to test the program. If no failure is detected, then t is considered not powerful and moved to partition 0. Otherwise, the program under test may be modified to remove a fault. In practice, such an attempt to debug the program does not always correctly remove the fault, and may sometimes introduce new faults [22]. In any case, the modified program needs to be retested on t. If a failure is detected again, then the above modify-and-retest process will be repeated until no more failure can be detected. Then t will be returned to partition 1, from which the next test case will be selected randomly. The testing process will stop when partition 1 becomes empty or when the given stopping criterion is met. Cai et al. experimentally compared this dynamic partitioning strategy with two other random testing strategies and found that the dynamic partitioning strategy outperformed the other two [7].

We note that the above dynamic partitioning algorithm is quite simple, and that all the proposed test case selection strategies used sampling with replacement [7]. In real-world software testing, it is more practical to use sampling without replacement to save cost. This paper follows the direction shown in Cai et al. [7] to further investigate the usefulness of online feedback information.

The motivation of this research is the fact that in real-world software development, the software will undergo many changes/versions. The test pool is very large and consists of both already-applied and not-yet-applied test cases. Every time a change is made, the software needs to be retested. Considering the total cost of testing across multiple versions, how can we select test cases in the most cost-effective manner so as to use the minimum number of test case executions to detect and remove the maximum number of defects? This situation is more complex than conventional regression testing and we want a practical method that is easy to implement without the need for sophisticated tool support (such as those for change tracking and analysis) or the need for any kind of test case coverage information or assumptions of the availability of the source code of the program under test. In short, we want a method that only depends on information that can be readily collected from test case executions (for example, pass/fail information) – this will enable the technique to be accepted by real-world practitioners.

The contributions of this research are summarized as follows: (i) A family of dynamic partitioning algorithms are proposed to selectively execute test cases from a given large test suite for evolving software. The dynamic partitioning strategy is

based on online feedback collected during test case executions without the need to refer to any kind of coverage information, change analysis, or human judgement. (ii) Empirical study results demonstrate that the proposed strategy is more cost-effective than the strategy without using feedback information. (iii) Empirical study results show that feedback information on early fault detection is very useful for further improving the cost-effectiveness of testing. In particular, one such algorithm DP1SE constantly outperforms all the others and, hence, is the best among all 10 algorithms studied in this paper. The results of this research further justify the emergence of the area of software cybernetics.

The rest of this paper is organized as follows: Section 2 proposes a family of 7 test case selection algorithms, where the first is pure random testing and serves as a benchmark for measuring the cost-effectiveness of the other 6 algorithms that employ online feedback information, which is pass / fail information collected during test case executions. To investigate the cost-effectiveness of the proposed algorithms, Section 3 presents the design and results of experiments with 3 large subject programs, namely, the SPACE, SED and GREP programs. In Section 4 , we consider more feedback information (namely, information as to how early a fault can be detected) and develop 3 more algorithms. Empirical study results of these 3 algorithms are also presented. Section 5 conducts an additional series of experiments with a set of smaller subject programs, namely, the Siemens Suite of Programs, to enhance the external validity of our study. Section 6 presents further discussion and concludes the paper.

2 Selecting test cases through online partitioning

2.1 The research question

Let $T = \{t_1, t_2, \ldots, t_n\}$ be a finite test set with n distinct test cases, where n can be very large. Test cases in T have been selected from the input domain using certain methods. For example, T could be constructed using specification-based or code-based coverage criteria, using a random sampling strategy, ad hoc approaches, or a mixture of these approaches. It is to be noted, however, that we are not concerned with the methods used to construct T. Now the program needs to be tested by selectively executing test cases in T. In real-world large-scale development, the program normally contains multiple faults in its initial version, and will undergo many rounds of testing, debugging, and retesting. Suppose a failure is detected after a certain number of test cases in T have been run. An attempt is then made to remove the fault, resulting in a new version of the program. Now the question is: how should this new version (and subsequent ones) be tested in a cost-effective way using the test cases in T? Ideally, a regression test selection technique should be applied to re-run some or all of the previously executed test cases [33] and if no

failure is revealed, then we can continue with the unused test cases in T. It must be noted, however, that real-world software testing is always carried out with limited resources [21,30]. An important contributor to the cost of testing is the total number of test case executions (if the same test case is executed twice, the count of test case executions is 2, even though it is the same test case). Suppose we can only afford to conduct m test case executions in total for the entire testing process, then how should we cost-effectively select test cases for each version of the program under test? That is, how should we select test cases, whose executions total m for all versions, so that as many faults as possible can be detected? Or, if the stopping criterion is to stop testing after r faults have been removed, then how can we conduct as few test executions as possible to achieve this goal? For practical purposes, we also require that the test case selection strategy be easy to adopt without heavy overheads or the need for sophisticated supporting tools such as those for coverage or change analysis. Note that the context and objectives stated above are very different from those of regression test selection or prioritization techniques [19,28,32,33].

In the following subsections, a family of 7 test case selection algorithms will be proposed to investigate the above question.

2.2 Pure random testing (PRT)

Figure 1 shows a *pure random testing* (PRT) algorithm. Code for measuring the cost-effectiveness of the algorithm is also embedded. This algorithm will serve as a benchmark in our empirical study. In Figure 1 , words enclosed between "/*" and "*/" are comments. Line 1 of the algorithm initializes two counters to store the up-to-date number of failures detected and the number of test case executions conducted. Our measurement uses these two numbers to indicate the cost-effectiveness of testing. A cost-effective testing algorithm must use a small number of test cases to detect a large number of failures.

Fig. 1. The Pure Random Testing (PRT) Algorithm

The algorithm divides test cases into two disjoint sets, namely, Set_0 and Set_1. Set_0 is initialized to an empty set in lines 2, and Set_1 is initialized to contain all the test cases in line 3. During the testing process, Set_0 stores test cases that have already been run for the current version of the program under test, whereas Set_1 stores test cases that have not been run for the current version. Line 4 controls when the algorithm should terminate. When either of the following two conditions becomes true, the algorithm will terminate: (i) Set_1 becomes empty, which means that the current version of the program under test has passed all the given test cases. It is beyond the scope of this paper to discuss how to continue the testing process by generating new test cases from outside the given test suite, which is assumed to be

very large. (ii) The given test stopping criterion is satisfied. Such a criterion can be, for instance, a prescribed number of faults have been detected, or a prescribed number of test cases have been run.

Whenever a test case is needed, an element of Set_1 will be selected randomly and executed, as shown in lines 5 and 6. Then line 7 immediately updates the counter to record the number of test case executions conducted so far. Line 8 checks whether a failure is detected. If no failure is detected, the control goes to line 9, which deletes the current test case from Set_1 and puts it into Set_0. Set_0 stores test cases already executed for the current version. Members of Set_0 will never be selected to test the program. This strategy of sampling without replacement is used in all the algorithms proposed in this paper. If a failure is detected, then the control goes to line 10, which increases the counter to record the number of failures detected so far. The values of this counter and the other counter (see line 7) serve as the up-to-date cost-effectiveness indicators and are printed in line 11. Since a failure is detected, an attempt can be made to remove the fault from the program under test, as indicated in line 12. In practice, such an attempt to modify programs may not necessarily remove the genuine fault, and may sometimes introduce new faults. Hence, the new version of the program needs to be tested on all the test cases in both Set_0 and Set_1. Therefore, in line 13, all the test cases in Set_0 are moved back to Set_1 so that they can be selected again.

Before the algorithm terminates, the overall cost-effectiveness is printed in line 15.

2.3 Regression-random testing (RRT)

Pure random testing treats the test case that has just detected a failure in the same way as all the other test cases. This subsection proposes an algorithm that gives a higher priority to the last failure-causing input: once a failure is detected, the failure-causing input will be applied repeatedly until no more failure can be detected. This algorithm is based on the thought that a test case that has just detected a failure might be able to detect another failure in the next test. The algorithm is also based on the common practice that after the program under test is modified with an attempt to remove a fault, the modified version is often first retested using the last failure-causing input to ensure that the modification has been made correctly. This algorithm is called *regression-random testing* (RRT), where "regression" refers to the first phase that re-runs the last failure-causing input, and "random" refers to the strategy that, when the last failure-causing input cannot detect more failures, the selection becomes random again. As we do not assume the availability of supporting tools for regression testing such as those for change analysis, no further regression test selection techniques will be incorporated. The algorithm is shown in Figure 2 .

Fig. 2. The Regression-Random Testing (RRT) Algorithm

Lines 1 to 3 of Figure 2 are the same as those of the PRT algorithm. Line 4 initializes an empty set $currentSet$. This set always contains no more than one element, namely, the test case currently being executed. Line 5 randomly selects a test case from the test suite and puts it into $currentSet$. The two termination conditions checked in line 6 are similar to those of the PRT algorithm except that it is $currentSet$ instead of Set_1 that is checked. The logic of lines 7 to 10 are similar to that of the PRT algorithm. Line 11 randomly selects a new test case from Set_1 (if Set_1 is empty, then no test case will be selected) and moves this test case to $currentSet$. The logic of lines 12 to 15 is similar to that of the PRT algorithm. Note, however, that in this branch (lines 12 to 15) no test case is selected from Set_1 and, therefore, the present element in $currentSet$ will be used again next time.

2.4 Testing through dynamic partitioning with fixed membership (DPFM)

RRT reapplies the last failure-causing input repeatedly until no more failure can be detected. Then any previous failure-causing input will be treated as an ordinary test case and will no longer be given priority in future test case selection.

Intuitively, test cases that have detected failures in the past are likely to be powerful in detecting a failure again during the evolution of the software. Based on this intuition, several test case selection algorithms will be proposed in this and the next subsections. The first one is called the *dynamic partitioning with fixed membership* (DPFM) algorithm, as shown in Figure 3 .

Fig. 3. The Dynamic Partitioning with Fixed Membership (DPFM) Algorithm

The DPFM algorithm partitions the given test suite online into 4 disjoint sets, namely, $fair$, $good$, $poor$, and $currentSet$. The set $currentSet$ is used in the same way as explained in the RRT algorithm: it stores the test case currently being executed and, hence, always contains no more than one element. For the other 3 sets ($fair$, $good$, and $poor$), each is further divided into two parts, namely, the used part, which stores test cases that have already been applied for the current version of the program P, and the unused part, which stores test cases not yet applied for the current version of P. Initially, all test cases are stored in $fair\_unused$ (that is, the unused part of the $fair$ set), as indicated in line 2 of the algorithm. All the other sets are initialized to be empty in line 3.

Initially, a test case t_i is randomly selected from the $fair$ set. If no failure is detected, t_i will be considered not powerful and moved to the $poor$ set. Otherwise t_i will be repeatedly applied to test P, in the same way as the RRT algorithm, until no more failure can be detected, and then t_i will be moved to the $good$ set — this is because t_i detected a failure in the past and, therefore, is considered a powerful test case. Once t_i is moved to either the $poor$ or the $good$ set, the membership of t_i will never be changed (the algorithm is hence named "Fixed Membership"). Every

time a new test case is needed, the algorithm will first look at the unused part of the $good$ set. If it is empty, then the $fair$ set. If this is also empty, then the $poor$ set.

The above process is elaborated below. Lines 4 and 5 initialize 2 variables, where $failureDetected$ is a flag indicating whether a failure has been detected, and $currentSetName$ records from which the element of $currentSet$ has been selected. It can only take 3 values, namely, $good$, $fair$, or $poor$. Line 6 randomly selects a test case from the given test suite, and moves it to $currentSet$. The program P is then tested in line 8 using the selected test case.

If a failure is detected, the control goes to line 14. The logic from line 14 to line 17 is very similar to that of the RRT algorithm. Line 18 sets the flag to $true$ to indicate that a failure has been detected. Note that $currentSet$ has not been updated and, therefore, the same test case will be used again to test the program P in the next iteration.

If no failure is detected, the true branch (line 11) will be taken and the sub-algorithm *removeFromCurrentSet* will be called. This sub-algorithm moves the element in $currentSet$ to one of the other sets as follows: If the element was selected from the $good$ or $poor$ set, it will be returned to the same set because of the use of fixed membership. If the element was selected from the $fair$ set, it will be moved to the $good$ set if it detected a failure last time. Otherwise it will be moved to the $poor$ set. After this sub-algorithm returns, $currentSet$ will become empty. Then line 12 calls another sub-algorithm *moveToCurrentSet*, which selects a new test case (with the $good$ set having the highest priority, followed by the $fair$ set, and the $poor$ set has the lowest priority) and moves it to $currentSet$. Note that if the unused parts of all 3 sets are empty, then $currentSet$ will remain empty, which will cause the loop to terminate.

2.5 Testing through dynamic partitioning with one-step varying membership (DP1S)

The membership of test cases in the DPFM algorithm is fixed in the sense that once a test case is moved to the $good$ or $poor$ set, it will always remain there. In other words, when a test case selected from the $good$ set no longer detects a failure, it is still returned to the $good$ set, and test cases selected from the $poor$ set will always be returned to the $poor$ set regardless of whether they can detect a failure or not. This strategy does not reflect the fact that in the real world a good test case may become poor, and a poor test case may become good. This subsection proposes an algorithm that adjusts the membership of test cases more dynamically in real time. The algorithm is shown in Figure 4 .

Fig. 4. The Dynamic Partitioning with One-Step Varying Membership (DP1S) Algorithm

This algorithm differs from DPFM only in the sub-algorithm *removeFromCur-*

rentSet. Let t_i be the test case in $currentSet$. When t_i no longer detects a failure, it will be removed from $currentSet$ as follows. If t_i was selected from the $good$ set, it will be returned to the $good$ set only if it detected a failure in the last test. Otherwise it will be downgraded by one step to the $fair$ set (see lines 3 and 4). If t_i was selected from the $fair$ set, it will be either upgraded or downgraded by one step depending on whether a failure was detected in the last test (see lines 7 and 8). If t_i was selected from the $poor$ set, it will be returned to the $poor$ set if no failure was detected. Otherwise it will be upgraded by one step to the $fair$ set (see lines 10 and 11). The algorithm is therefore named *dynamic partitioning with one-step varying membership* (DP1S), where "one-step varying membership" refers to the strategy that upgrades or downgrades by one step every time the membership is to be adjusted.

2.6 Testing through dynamic partitioning with two-step varying membership (DP2S)

In the DP1S algorithm, a test case is upgraded or downgraded by one step each time. It is also possible to upgrade or downgrade by two steps each time because when a "poor" test case detects a failure, it might become a "good" (rather than "fair") test case in the future and, in the same way, when a "good" test case does not detect a failure, it might become a "poor" (rather than "fair") test case in the future. The algorithm implementing this strategy is named *dynamic partitioning with two-step varying membership* (DP2S), which can be obtained by changing only two words in the DP1S algorithm shown in Figure 4 : in lines 3, change "fair" to "poor"; in line 11, change "fair" to "good".

2.7 Testing through dynamic partitioning with no upgrade (DPNU)

In DP1S and DP2S, suppose a test case t detected a failure last time but did not detect a failure this time, then t is upgraded to a partition with a higher priority unless it is already in the $good$ set. A simpler treatment is that instead of upgrading t to a different partition, we can just return t to where it was selected from, either the $fair$ set or the $poor$ set. This strategy never uses the $good$ set and never upgrades a test case. The algorithm is therefore named *dynamic partitioning with no upgrade* (DPNU), as shown in Figure 5 .

Fig. 5. The Dynamic Partitioning with No Upgrade (DPNU) Algorithm

It should be noted that the algorithms proposed in this paper and the one proposed by Cai et al. [7] are not directly comparable as the former uses sampling without replacement whereas the latter uses sampling with replacement. Nevertheless, among all the algorithms proposed in this paper, DPNU is closest to Cai et al.'s algorithm [7].

2.8 Testing through dynamic partitioning that returns to poor (DPRP)

In DPNU, a test case t is returned to the set from which it was selected, either the $fair$ set or the $poor$ set. An even simpler strategy is to always return t to the $poor$ set when it does not detect a failure regardless of where it was selected from. This algorithm is therefore named *dynamic partitioning that returns to poor* (DPRP). This algorithm differs from the DPFM algorithm only in that DPRP's sub-algorithm *removeFromCurrentSet* contains only one statement as follows:
`move the test case in currentSet to poor_used.`

3 Experiments with SPACE, SED and GREP programs

A series of experiments have been conducted to investigate the cost-effectiveness of the proposed algorithms. In the experiments, these algorithms were applied to test 3 real-world programs, namely, the SPACE, SED, and GREP programs, downloaded from the Software-artifact Infrastructure Repository (SIR, http://sir.unl.edu) [15]. Each package of subject programs provides both the original and associated faulty versions as well as a pool of test cases. For each subject program, we combined the faults into one version to create a program that contains multiple faults. The original version was used as the test oracle. Every time a test case is executed, the output of the faulty program is compared with that of the original version, and any discrepancy indicates a failure. [1] During a test execution, if any fault is encountered, its label will be recorded in a log file. When a failure is detected, the log file will be checked, and the first fault encountered in the execution path will be removed to simulate the debugging process. To "remove a fault", we delete the corresponding faulty statement(s) and restore the corresponding original statement(s). While a fault thus removed might not be the genuine cause for the failure, this process properly simulates the debugging process because when the debugger manually traces the failed execution, the first fault in the execution path will have a higher chance to be identified and corrected first. The testing will stop when all the seeded faults have been removed. The total number of test case executions will indicate the overall cost-effectiveness: the smaller, the better. For each algorithm and each subject program, the experiment has been repeated 1,000 times and the mean, median and standard deviation data have been calculated.

[1] In software testing research, the original version of the program is often used as the specification / oracle to verify the outputs of the faulty versions. In real-world software testing, such an original version is, of course, not available; instead, the tester can serve as the ultimate test oracle. We use the original version rather than a human tester in the experiments to enable quick verification of large amounts of outputs.

3.1 Results of experiments with SPACE

SPACE is an interpreter for an array definition language. It consists of 6,199 lines of executable C code and 136 functions. It is a subject program that has often been used in the study of testing effectiveness [31]. The faulty version used in our experiment involves 33 faults which, according to the Software-artifact Infrastructure Repository, were real faults discovered during the development of the program. The 13,551 test cases included in the package have been used in our experiments. Experimental results are summarized in Table 1 , where R_{mean}, R_{med} and R_{SD} refers to the ranking of mean, ranking of median, and ranking of standard deviation, respectively.

Table 1
Results of Experiments with SPACE (33 Faults, 13,551 Test Cases, 1,000 Trials)

Algorithm	Mean	Median	Std Deviation	R_{mean}	R_{med}	R_{SD}
PRT	2360	2077	1231	7	7	6
RRT	1707	1360	1237	4	4	7
DPFM	1779	1495	1132	5	5	4
DP1S	1586	1356	1000	1	3	1
DP2S	1881	1535	1230	6	6	5
DPNU	1616	1310	1127	2	1	3
DPRP	1659	1344	1106	3	2	2

To detect and remove all 33 faults included in SPACE, the pure random testing (PRT) algorithm conducted a total of 2,360 test case executions on average. As a comparison, the RRT algorithm used only 1,707 test cases. In other words, it achieved a saving of 27.7% on average. (In this paper, the word "saving" refers to the comparison with the Pure Random Testing algorithm.) The average cost-effectiveness of the DPFM algorithm lies between PRT and RRT, with a saving of 24.6%. The DP1S algorithm achieved the highest average saving among all the algorithms, namely, 32.8%. The stability of DP1S was also the best as it yielded the lowest standard deviation. In contrast, the DP2S algorithm achieved the smallest average saving (20.3%) among all 6 non-PRT algorithms. The DPNU and DPRP algorithms, albeit simple, achieved good average savings of 31.5% and 29.7%, respectively, and their median values were also the best among all of the algorithms.

In summary, all 6 algorithms that employ online feedback information outperformed PRT in cost-effectiveness, and DP1S had the best mean value and best stability.

Table 1 shows the final numbers of test case executions used to detect and remove all 33 faults. Figure 6 further depicts the average results for detecting each

failure. As there are 33 faults included in the program and each fault-removal activity removes one and only one fault, a total of 33 failures will have been detected when the last fault has been removed. Figure 6 shows that until the detection of about the 20th failure, the numbers of test case executions have been quite small for all algorithms (ranging from 40 to 53). The numbers increase significantly in the later stages of testing, from failure 28 or so. This is reasonable because, when there are many faults in the program, the failure rate is high and, hence, it is easy to detect a failure even with Pure Random Testing. When more and more faults are removed from the program, the failure rate gets smaller (in other words, it becomes more expensive to detect a failure), and the algorithms employing online feedback information start to achieve considerable savings.

Fig. 6. Average Numbers of Test Case Executions for Detecting Each Failure (for the SPACE Program)

3.2 Results of experiments with SED

SED is a Unix/Linux utility and performs text transformations on an input stream (http://www.gnu.org/software/sed). The program used in our experiments consists of 14,427 lines of C code and 255 functions. The downloaded package includes 7 version pairs of the SED program (from version 1 to version 7), where each pair consists of an original version and its corresponding faulty version that contains multiple faults. According to the SIR Web site, the faulty versions include both real and seeded faults. The latest version (namely, version 7) was used in our experiments. To create a faulty version 7 that contains many faults, we took a total of 11 faults from the faulty versions 5, 6 and 7. We further created 7 more faults manually with the aim of making them as realistic as possible. Each of these 7 faults was obtained by making a slight change to the operators or operands of a single statement in the original version 7. The final faulty version, therefore, includes a total of 18 faults. Our test suite consists of 12,238 test cases, of which 415 were provided directly by the downloaded package, and the other 11,823 were generated using random sampling based on input data available in the SIR package and on the GNU Web site (such as sample inputs in the SED manual).

Table 2 summarizes the experimental results. To detect and remove all 18 faults, the PRT algorithm used 4,844 test cases on average. The RRT, DPFM, DP1S, DP2S, DPNU and DPRP algorithms achieved a saving of 16.6%, 23.7%, 26.2%, 28.1%, 25.9% and 25.5%, respectively. The DP2S algorithm achieved the best performance in terms of both mean and median values. The DP1S algorithm ranks second in mean value and has a smaller standard deviation than DP2S.

Figure 7 shows the average results for detecting each failure. It can be seen that the number of test case executions needed to detect a failure in the later stage of testing

Table 2
Results of Experiments with SED (18 Faults, 12,238 Test Cases, 1,000 Trials)

Algorithm	Mean	Median	Std Deviation	R_{mean}	R_{med}	R_{SD}
PRT	4844	4412	2442	7	7	7
RRT	4038	3629	2359	6	6	6
DPFM	3697	3405	2009	5	5	1
DP1S	3576	3229	2029	2	4	2
DP2S	3485	3138	2056	1	1	3
DPNU	3589	3182	2099	3	2	5
DPRP	3609	3228	2070	4	3	4

increases dramatically, and the algorithms employing online feedback information bring considerable savings in this stage.

Fig. 7. Average Numbers of Test Case Executions for Detecting Each Failure (for the SED Program)

3.3 Results of experiments with GREP

GREP is another Unix/Linux utility that searches input files for lines containing a match to a specified pattern (http://www.gnu.org/software/grep). GREP has 5 versions seeded with faults and a suite of about 470 test cases. We used version 3 in our experiments. The program contains about 10,068 lines of C code and 146 functions. To create a faulty version that includes many faults, we combined 22 faults collected from faulty versions 1, 2 and 3. We also expanded the test suite by using a random sampling approach similar to that for the SED program. The final test suite contains 10,065 test cases. The results of experiments are shown in Table 3 .

Table 3 shows that, to detect and remove all 22 faults, PRT conducted 1,150 test case executions on average. The RRT, DPFM, DP1S, DP2S, DPNU and DPRP algorithms achieved a saving of 3.5%, 19.2%, 21.6%, 18.7%, 12.8% and 15.0%, respectively. The performance of DP1S was the best in terms of both the mean value and the standard deviation. The average results for detecting each failure are shown in Figure 8 .

Fig. 8. Average Numbers of Test Case Executions for Detecting Each Failure (for the GREP Program)

Table 3
Results of Experiments with GREP (22 Faults, 10,065 Test Cases, 1,000 Trials)

Algorithm	Mean	Median	Std Deviation	R_{mean}	R_{med}	R_{SD}
PRT	1150	893	860	7	7	7
RRT	1110	857	825	6	6	6
DPFM	929	683	726	2	1	3
DP1S	902	687	685	1	2	1
DP2S	935	687	714	3	2	2
DPNU	1003	740	805	5	5	5
DPRP	978	721	760	4	4	4

3.4 Summary

Among the 7 algorithms, PRT serves as a benchmark, and the other 6 algorithms employ online feedback information to dynamically partition the given test suite. Experimental results with SPACE, SED and GREP programs show that all 6 of these algorithms outperformed PRT in all 3 statistics (mean, median and standard deviation), except that for the SPACE program the standard deviation of RRT (1,237) was slightly higher than that of PRT (1,231).

Among all 7 algorithms investigated, DP1S gave the best performance: its mean value and standard deviation ranked first with SPACE and GREP and ranked second with SED. Table 4 summarizes the experimental results over the 3 subject programs.

Table 4
Summary of Experimental Results with SPACE, SED and GREP Programs

Algorithm	Total	Saving	Mean of R_{mean}	Mean of R_{SD}
PRT	8354	0.0%	7.00	6.67
RRT	6855	17.9%	5.33	6.33
DPFM	6405	23.3%	4.00	2.67
DP1S	6064	27.4%	1.33	1.33
DP2S	6301	24.6%	3.33	3.33
DPNU	6208	25.7%	3.33	4.33
DPRP	6246	25.2%	3.67	3.33

Table 4 shows that, to detect and remove all of the faults included in all 3 subject programs, PRT used a total of 8,354 ($= 2,360 + 4,844 + 1,150$) test cases, and had an average ranking of 7.00 ($= (7+7+7)/3$) in mean value and 6.67 ($= (6+7+7)/3$)

in standard deviation. In comparison, DP1S used the smallest total number of test cases (namely, $1,586 + 3,576 + 902 = 6,064$), hence achieving an overall saving of 27.4% (the best among the 7 algorithms). The average ranking of DP1S in both mean value and standard deviation was $(1 + 2 + 1)/3 = 1.33$, which was also the best among the 7 algorithms.

The finding that DP1S gave the best performance among the 7 algorithms agrees with our prediction because, compared with the other algorithms, DP1S utilizes more feedback information with finer granularity. PRT does not employ any feedback information. RRT employs limited feedback information, which is only about the last (one) failure-causing test case. DPFM does not utilize up-to-date performance information of test cases once their membership is fixed. DP2S does utilize the up-to-date performance information but not in a gradual way because test cases are moved directly between the $good$ and $poor$ sets. This strategy is sometimes very good (such as for the SED program, where $R_{mean} = 1$) and sometimes very poor (such as for the SPACE program, where $R_{mean} = 6$). The DPNU and DPRP algorithms do not make use of the $good$ set and, hence, have coarser granularity than DP1S.

In the next section we will see that the incorporation of more feedback information into the test case selection process will further improve the cost-effectiveness of the algorithms.

4 Employing information on early fault detection

The algorithms discussed so far have employed only pass/fail information of previous tests. More types of feedback information can be employed for test case selection. In this section we look at the information as to how early a fault can be detected (referred to as EFD). We consider EFD a kind of simple indicator of defect severity, and this kind of information is easy to get from a black-box testing perspective. Our proposal complements Xu et al.'s suggestion that in a test case selection process, a higher priority should be given to test cases that are likely to find "high-severity defects in the most often used functions" [32]. While Xu et al. offered this suggestion from the perspective of the importance of test cases, we look at the fault-detection capabilities of test cases without assuming the availability of user operational profiles such as the frequency of function usage.

Our strategy is based on the idea that a test case that detects a failure earlier in its execution should have a higher fault-detection capability than those that detect failures at a later stage of their executions. Consider, for instance, the following simplified example: we have a program P supposedly implementing a non-numerical function f whose output is a string of characters. Let t_1 and t_2 be 2 test cases. Suppose $f(t_1)$ ="abcdefg" and $f(t_2)$ ="hijkl". Suppose the outputs of P for t_1 and

t_2 are "axcdefg" and "hijky", respectively. Obviously, both these test cases can detect a failure: t_1 detects the failure when the second output character "x" is printed; whereas t_2 detects the failure at the fifth output character "y". In this situation, we say t_1 detects a failure earlier than t_2. The EFDs of t_1 and t_2 are 2 and 5, respectively. Note that EFD applies only to failure-causing test cases.

Figure 9 shows the DPFME algorithm, which is the DPFM algorithm with EFD incorporated in its test case selection process. There are 2 highlighted statements, which show the difference between this algorithm and the original DPFM algorithm. In statement 16 of the main algorithm, the variable *earlinessInfo* records the up-to-date EFD of the current test case after a failure has been detected. Note that *earlinessInfo* is only kept for test cases selected from or to be sent to the $good$ set. This is because the other two sets may contain test cases that have never detected a failure and, hence, *earlinessInfo* is not applicable. In statement 10 of the main algorithm, if no failure is detected, and if the current test case is selected from the $good$ set, then the value of its *earlinessInfo* remains the same as before. In statement 2 of the sub-algorithm *moveToCurrentSet*, when a test case is being selected from the $good$ set, the original DPFM algorithm will make a random selection; whereas the DPFME algorithm will select a test case whose *earlinessInfo* has the smallest (that is, earliest) value among all the elements in the $good$ set. If more than one test case in the $good$ set meets this requirement, then it randomly chooses one from them.

Fig. 9. The DPFME Algorithm: DPFM Incorporating EFD

Using a similar treatment, we designed two further algorithms named DP1SE and DP2SE, which are the counterparts of DP1S and DP2S, respectively, that employ EFD. Note, however, that this strategy does not apply to the PRT, RRT, DPNU and DPRP algorithms as they do not involve the $good$ set. Results of experiments with the SPACE program are summarized in Table 5 .

Table 5
Results of Experiments with SPACE (33 Faults, 13,551 Test Cases, 1,000 Trials)

Algorithm	Mean	Median	Std Deviation
DPFME	1517	1310	875
DP1SE	1492	1294	867
DP2SE	1761	1380	1266

The results shown in Table 5 and Table 1 are compared as follows. First, comparing DPFM and DPFME, we can see that the mean value has improved (decreased) by 14.7% (from 1,779 for DPFM to 1,517 for DPFME), the median has improved from 1,495 to 1,310 and the standard deviation has improved from 1,132 to 875. Similarly, DP1SE outperformed DP1S in all 3 of the statistics (mean, median and standard deviation). DP2SE also outperformed DP2S in mean and median values but had a slightly higher standard deviation (1,230 for DP2S and 1,266 for DP2SE).

Among all 10 algorithms investigated in this paper (that is, PRT, RRT, DPFM, DP1S, DP2S, DPNU, DPRP, DPFME, DP1SE and DP2SE), DP1SE was the best: it outperformed all the other algorithms in all 3 statistics (mean, median and standard deviation) with an average saving of 36.8% ($= 1 - 1,492/2,360$).

Next, from the SED results shown in Table 6 and Table 2 we can see that the DPFME, DP1SE and DP2SE algorithms outperformed their respective counterparts in all 3 statistics, except DPFME and DPFM had the same standard deviation (2,009). Among all 10 of the algorithms, DP1SE once again outperformed all the other algorithms in all 3 statistics, with an average saving of 36.4% ($= 1 - 3,079/4,844$).

Table 6
Results of Experiments with SED (18 Faults, 12,238 Test Cases, 1,000 Trials)

Algorithm	Mean	Median	Std Deviation
DPFME	3420	3048	2009
DP1SE	3079	2756	1767
DP2SE	3417	3086	2004

Finally, the GREP results listed in Table 7 and Table 3 show that DPFME, DP1SE and DP2SE outperformed their respective counterparts in all 3 statistics except DPFM had a lower median (683) than DPFME (711). Among all 10 algorithms, DP1SE was still the best in mean value and standard deviation, and its median (653) was only slightly higher than that of DP2SE (652). The average saving of DP1SE was 27.8% ($= 1 - 830/1,150$).

Table 7
Results of Experiments with GREP (22 Faults, 10,065 Test Cases, 1,000 Trials)

Algorithm	Mean	Median	Std Deviation
DPFME	919	711	690
DP1SE	830	653	603
DP2SE	874	652	646

In summary, all 3 algorithms employing EFD constantly outperformed their respective counterparts in mean values. In most cases, these 3 algorithms also outperformed their respective counterparts in median and standard deviation.

Among all 10 algorithms investigated in this paper, DP1SE outperformed all the other 9 algorithms in all 3 statistics and for all 3 subject programs, except for GREP where median(DP1SE) = 653, whereas median(DP2SE) = 652, but this difference is negligible.

5 Additional experiments with the Siemens Suite of Programs

Empirical studies in the previous sections show that (i) all 9 algorithms employing feedback information outperformed PRT; (ii) DP1SE was the best among all 10 algorithms proposed in this paper; (iii) the 3 algorithms employing EFD outperformed their respective counterparts that do not use EFD and (iv) among all 7 algorithms that do not require EFD, DP1S was the best. The experiments were conducted with 3 large subject programs, namely, the SPACE, SED and GREP programs.

In this section we will conduct a further series of experiments with smaller programs to enhance the external validity of our study. We use the Siemens Suite of Programs [20] as the subject programs. This set of subjects (including the original versions, faulty versions and test cases) has been used often in the research community to study the effectiveness of testing and debugging techniques [3,11]. The suite includes 7 programs as listed in Table 8 . For each of the 7 subject programs, we combined all of the faults into one faulty version to conduct the experiments. There are situations where two faults are mutually exclusive (for example, fault #1 is to modify a statement whereas fault #2 is to delete the same statement or to modify it in a different way) or situations where a fault cannot be detected by any of the test cases in the given test suite. In these situations, the mutually exclusive faults and non-detectable faults were removed from our experiments. Furthermore, the tcas program only prints a single integer as its output and, therefore, our original method of collecting EFD (by means of string comparison) is not applicable. Nevertheless, we still successfully collected EFD for each failure-causing test case of tcas by measuring the execution time that led to the failure. Obviously, the quicker, the better.

Table 8
The Siemens Suite of Programs

Program	Lines of executable code	Faulty versions	Test cases	Description
print_tokens	472	7	4130	lexical analyzer
print_tokens2	399	10	4115	lexical analyzer
replace	512	32	5542	pattern replacement
schedule	292	9	2650	priority scheduler
schedule2	301	10	2710	priority scheduler
tcas	135	41	1608	altitude separation
tot_info	346	23	1052	information measure

Table 9 summarizes the mean values (out of 1,000 trials) of the total number of test cases required by each of the 10 algorithms to detect and remove all of the faults for each Siemens program. We have the following findings: (i) All 9 algorithms that employ feedback information outperformed the PRT algorithm for each and every Siemens program. (ii) DP1SE is the best among all 10 algorithms for each and every Siemens program with an overall saving of 38.4%. (iii) The mean values

of DPFME, DP1SE and DP2SE are lower than (or sometimes equal to) those of DPFM, DP1S and DP2SE, respectively, for each and every Siemens program. (iv) Among all 7 algorithms that do not require EFD, DP1S is the best for each and every Siemens program except for print_tokens where mean(DP1S) = 166 whereas mean(DPNU) = 165. The above observations become more evident in Table 10 , which summarizes the R_{mean} values (the rankings of mean values). These observations are consistent with the experimental results with larger subject programs reported in the previous sections.

Table 9
Summary of Mean Values (1,000 trials for each of the 7 Siemens programs)

Program	PRT	RRT	DPFM	DP1S	DP2S	DPNU	DPRP	DPFME	DP1SE	DP2SE
print_tokens	190	170	168	166	169	165	168	165	165	168
print_tokens2	194	155	151	146	152	147	147	150	142	152
replace	290	261	258	249	254	256	250	247	235	242
schedule	221	65	65	63	64	63	63	61	59	61
schedule2	147	103	103	98	103	105	103	101	93	100
tcas	703	457	433	417	421	424	432	422	398	411
tot_info	72	29	30	29	29	29	29	29	27	29
subtotal	1817	1240	1208	1168	1192	1189	1192	1175	1119	1163
overall saving	0.0%	31.8%	33.5%	35.7%	34.4%	34.6%	34.4%	35.3%	38.4%	36.0%

Table 10
Summary of R_{mean} (Siemens programs)

Program	PRT	RRT	DPFM	DP1S	DP2S	DPNU	DPRP	DPFME	DP1SE	DP2SE
print_tokens	10	9	5	4	8	1	5	1	1	5
print_tokens2	10	9	6	2	7	3	3	5	1	7
replace	10	9	8	4	6	7	5	3	1	2
schedule	10	8	8	4	7	4	4	2	1	2
schedule2	10	5	5	2	5	9	5	4	1	3
tcas	10	9	8	3	4	6	7	5	1	2
tot_info	10	2	9	2	2	2	2	2	1	2
average	10.00	7.29	7.00	3.00	5.57	4.57	4.43	3.14	1.00	3.29

Tables 11 and 12 summarize the standard deviations and their rankings (R_{SD}), respectively. Out of the 7 Siemens programs, PRT ranked tenth (worst) 5 times with an average ranking of 9.57 (the worst among all 10 algorithms); DP1SE ranked first (best) 5 times, with an average ranking of 1.86 (the best among all 10 algorithms). In terms of average ranking (the last row of Table 12), DPFME, DP1SE and DP2SE outperformed DPFM, DP1S and DP2S, respectively, and DP1S was the best among all 7 algorithms that do not use EFD. These observations are consistent with the findings reported in the previous sections.

Table 11
Summary of Standard Deviations (1,000 trials for each of the 7 Siemens programs)

Program	PRT	RRT	DPFM	DP1S	DP2S	DPNU	DPRP	DPFME	DP1SE	DP2SE
print_tokens	141	130	119	121	127	128	133	131	116	135
print_tokens2	127	128	115	114	122	118	106	110	102	115
replace	207	224	199	185	204	211	201	177	169	190
schedule	138	77	83	73	79	73	74	74	74	70
schedule2	87	78	77	73	75	74	75	78	69	70
tcas	385	351	309	311	309	300	311	299	295	295
tot_info	38	22	23	20	22	24	24	20	22	21

Table 12
Summary of R_{SD} (Siemens programs)

Program	PRT	RRT	DPFM	DP1S	DP2S	DPNU	DPRP	DPFME	DP1SE	DP2SE
print_tokens	10	6	2	3	4	5	8	7	1	9
print_tokens2	9	10	5	4	8	7	2	3	1	5
replace	8	10	5	3	7	9	6	2	1	4
schedule	10	7	9	2	8	2	4	4	4	1
schedule2	10	8	7	3	5	4	5	8	1	2
tcas	10	9	5	7	5	4	7	3	1	1
tot_info	10	4	7	1	4	8	8	1	4	3
average	9.57	7.71	5.71	3.29	5.86	5.57	5.71	4.00	1.86	3.57

6 Discussions and conclusion

6.1 The cost model

The cost model used in this research is the total number of test case executions needed to detect and remove all of the faults included in the program under test. This model treats all test cases equally (in terms of test cost) and all faults equally (in terms of debugging cost), although in practice, the costs may not be uniform [16]. Future research may consider varying test costs (for example, different setup, execution and result verification costs for different test cases, where the "costs" may involve time and other resources) and varying debugging costs (such as the cost needed to locate and correct a fault).

6.2 Internal and external validity

To address internal validity and avoid potential errors, all our test drivers have been carefully checked and tested, and all the subject programs have also been carefully inspected manually. All the experimental results, including both final results and intermediate results, have been carefully verified.

Like most of the other empirical studies in software engineering, there are threats to external validity. Our experiments were conducted with a limited number of subject programs that include a limited number of faults. The number of test suites was also limited. Therefore, we cannot claim that the effectiveness of our testing strategy can be generalized to other software. Nevertheless, the results of our experiments with both the large programs and the small programs are highly consistent and this provides us with confidence in the effectiveness of our testing strategy. A question of practical importance is:

To what degree are our subject programs, faults and test suites a representative sample of the real world?

The above question is answered as follows. First, the subject packages (including the original versions, faulty versions and test cases) used in our experiments are also used in many other empirical studies in the literature to compare the effectiveness of different testing and debugging techniques. For instance, Baah et al. [3] used the same set of subjects, namely, SPACE, SED, GREP, and the Siemens Suite of Programs. Regarding the source where the subjects were obtained, all our packages were downloaded from SIR, which is a repository of subjects that provide a common ground for software testing and debugging experiments and have a large number of users and publications (the SIR usage information can be found at http://sir.unl.edu/portal/usage.php).

Next, we will briefly explain how the test suites and mutants were created, and readers are referred to the SIR Web site http://sir.unl.edu/content/bios/flex.php and the original papers [20,31] for more details.

The Siemens Suite of Programs were initially assembled by researchers at Siemens Co. to investigate the fault-detection capabilities of control-flow and data-flow coverage criteria [20]. These programs perform a variety of tasks including: aircraft collision avoidance, priority scheduling, statistics computation, lexical analysis, and pattern matching and substitution. Ten researchers at Siemens manually created faulty versions for each of the 7 base programs, aiming at introducing faults as realistic as possible. Each faulty version contains a single fault, usually by modifying a single statement. For each of the 7 base programs, the Siemens researchers created an initial test pool which was then populated by adding black-box test cases created using the category-partition method and the Siemens Test Specification Language (TSL) tool. Each test pool was further augmented with manually created white-box test cases that intensively exercise every executable statement, edge, and definition-use pair of the base program. In order to produce meaningful results with the faulty versions, the Siemens researchers only retained those faulty versions that were detectable by at most 350 and at least 3 test cases in the test pool.

SPACE was developed by the European Space Agency and the faults were real faults discovered during the development of the software. Vokolos and Frankl cre-

ated an initial pool of 10,000 randomly generated test cases for SPACE [31]. This pool was then expanded by including additional white-box test cases to intensively exercise the statements and edges in the program or in its control flow graph.

GREP is a Unix/Linux utility. The creators of SIR obtained several of its previously released versions from the GNU Web site. Because comprehensive test suites were not available, they used the category-partition method and a TSL tool to construct a suite of black-box test cases that exercise each parameter, special effect and erroneous condition of the base programs. They then added additional test cases to increase code coverage at the statement level. The researchers wished to evaluate testing techniques for the detection of regression faults and, therefore, used the following procedure to create mutants: They recruited graduate and undergraduate students in computer science with at least two years of programming experience in C. The students were instructed to create faults which were as realistic as possible. The faults involved code deletion, insertion and modification. Then the researchers executed their test cases to check which faults could be detected by which test cases. They excluded those faults that were not detected by any test cases. They also excluded those faults that were detected by more than 25% of the test cases. We augmented the GREP test suite by adding randomly sampled test cases based on input data available in the SIR package and in the GNU Web site, such as sample inputs in the online manual.

SED is another Unix/Linux utility. The SIR download page for the SED package shows that its "Test Types" are "tsl, other", which means that the test suite was constructed using an approach similar to that of GREP. The SIR download page also shows that the "Fault Types" of SED are "real, seeded", which means that the faulty versions of SED include both real and seeded faults. The seeded faults were created in a similar way to those of the GREP program. [2] For our experiments, we wished to have more faults and more test cases. Therefore, we manually seeded 7 more faults, each of which was created by slightly changing the operator(s) or operand(s) of a single statement. These 7 faults have a similar nature to the other faults included in the SED packages. We also augmented the SED test suite using a random sampling approach similar to that for the GREP program.

In summary, we experimented with both large and small programs from the real world, with both real and seeded faults, and with systematically generated test suites (including black-box, white-box, and randomly sampled test cases). It is reasonable to believe that these subject programs, together with the faults and test suites, are a representative sample of the real world. Having said that, additional studies with more types of programs, faults and test suites are needed to increase

[2] SIR did not provide a dedicated page for the SED package; instead, SED users were referred to the GREP page under the heading "C Object Biographies" (http://sir.unl.edu/content/bios/flex.php) that "provides information about each of the C objects we currently make available, or hope to soon make available".

confidence in the generalization of the results.

6.3 Concluding remarks

We have proposed and investigated a dynamic partitioning strategy that improves the cost-effectiveness of large-scale software testing processes, where the given test suite is large and the software under test may undergo many versions. Because in our algorithms different partitions have different priorities for test case selection, this research also relates to test case prioritization. A fundamental difference, however, is that the research problem we investigate is not the test case permutation problem defined for test case prioritization [28].

Most conventional test case selection strategies partition test suites off-line. In comparison, our dynamic partitioning strategy employs online feedback information collected during test case executions from a black-box testing perspective. This strategy is easy for practitioners to adopt as it does not require coverage information or change analysis.

To implement the dynamic partitioning strategy, we developed a total of 10 algorithms, where algorithm #1 is pure random testing, algorithms #2 to #7 employ only pass / fail information, and algorithms #8 to #10 employ both pass / fail and EFD. These two types of information can be readily collected online during test case executions.

Empirical study results demonstrate that the cost-effectiveness of testing can be improved considerably by simply employing the online pass / fail information, and DP1S is the best among algorithms #1 to #7 in terms of cost-effectiveness and standard deviation, with an overall saving of 27.4% for the 3 large programs and 35.7% for the Siemens programs. The utilization of more feedback information, namely, EFD, further improved the cost-effectiveness of testing, and DP1SE is the best (most cost-effective and most stable) among all 10 algorithms studied in this paper, with an overall saving of 35.3% for the 3 large programs and 38.4% for the Siemens programs. This magnitude of savings can effectively reduce the cost of real-world large-scale software development, maintenance, and evolution, where the total number of test cases can be very high [35]. DP1S and DP1SE showed such performance because they utilize more feedback information with finer granularity, as analyzed in Section 3.4. This finding suggests that the cost-effectiveness of testing could be further improved by considering more types of feedback information and involving a more comprehensive hierarchy of partitions beyond the 3 sets of $fair$, $good$ and $poor$. The results of this research further justify the emergence of the area of software cybernetics.

Acknowledgments

This project was supported in part by a linkage grant of the Australian Research Council (Project ID: LP100200208) and an international links grant of the University of Wollongong. Cai was supported in part by the National Natural Science Foundation of China (Grant Number 61272164) and the Beijing Natural Science Foundation (Grant Number 4112033).

We would like to thank Ke Cai for his discussions on the design of some of the algorithms presented in Section 2 and helps with their implementation. We would also like to thank Phillip McKerrow for reading our manuscript and providing valuable comments.

We wish to gratefully acknowledge the help of Dr. Madeleine Strong Cincotta in the final language editing of this paper.

References

[1] A. Adir, A. Goryachev, L. Greenberg, T. Salman, G. Shurek, A new test-generation methodology for system-level verification of production processes, in: Proceedings of the 8th Haifa Verification Conference, Lecture Notes in Computer Science 7857, Springer-Verlag, 2013.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (2013) 1978–2001.

[3] G. K. Baah, A. Podgurski, M. J. Harrold, The probabilistic program dependence graph and its application to fault diagnosis, IEEE Transactions on Software Engineering 36 (4) (2010) 528–545.

[4] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), 2008.

[5] K.-Y. Cai, T. Y. Chen, T. H. Tse, Towards research on software cybernetics, in: Proceedings of 7th IEEE International Symposium on High Assurance Systems Engineering, IEEE Computer Society Press, 2002.

[6] K.-Y. Cai, B. Gu, H. Hu, Y.-C. Li, Adaptive software testing with fixed-memory feedback, Journal of Systems and Software 80 (2007) 1328–1348.

[7] K.-Y. Cai, T. Jing, C.-G. Bai, Partition testing with dynamic partitioning, in: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), IEEE Computer Society Press, 2005.

[8] T. Y. Chen, F.-C. Kuo, R. G. Merkel, T. H. Tse, Adaptive Random Testing: The ART of test case diversity, Journal of Systems and Software 83 (1) (2010) 60–66.

[9] T. Y. Chen, F.-C. Kuo, Z. Q. Zhou, On favourable conditions for adaptive random testing, International Journal of Software Engineering and Knowledge Engineering 17 (6) (2007) 805–825.

[10] T. Y. Chen, H. Leung, I. K. Mak, Adaptive random testing, in: Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004), Lecture Notes in Computer Science 3321, Springer-Verlag, 2004.

[11] T. Y. Chen, T. H. Tse, Z. Q. Zhou, Semi-proving: An integrated method for program proving, testing, and debugging, IEEE Transactions on Software Engineering 37 (1) (2011) 109–125.

[12] T. Y. Chen, Y. T. Yu, The universal safeness of test allocation strategies for partition testing, Information Sciences 129 (2000) 105–118.

[13] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, ARTOO: Adaptive random testing for object-oriented software, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM Press, 2008.

[14] Z. Ding, K. Zhang, J. Hu, A rigorous approach towards test case generation, Information Sciences 178 (2008) 4057–4079.

[15] H. Do, S. G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, Empirical Software Engineering: An International Journal 10 (4) (2005) 405–435.

[16] S. Elbaum, A. Malishevsky, G. Rothermel, Incorporating varying test costs and fault severities into test case prioritization, in: Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), IEEE Computer Society Press, 2001.

[17] J. E. Forrester, B. P. Miller, An empirical study of the robustness of Windows NT applications using random testing, in: Proceedings of the 4th USENIX Windows Systems Symposium, 2000.

[18] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), ACM Press, 2005.

[19] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, G. Rothermel, An empirical study of regression test selection techniques, ACM Transactions on Software Engineering and Methodology 10 (2) (2001) 184–208.

[20] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria, in: Proceedings of the 16th International Conference on Software Engineering (ICSE'94), IEEE Computer Society Press, 1994.

[21] J.-M. Kim, A. Porter, A history-based test prioritization technique for regression testing in resource constrained environments, in: Proceedings of the 24th International Conference on Software Engineering (ICSE'02), ACM Press, 2002.

[22] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S. D. Fleming, How programmers debug, revisited: an information foraging theory perspective, IEEE Transactions on Software Engineering 39 (2) (2013) 197–215.

[23] P. S. Loo, W. K. Tsai, Random testing revisited, Information and Software Technology 30 (7) (1988) 402–417.

[24] A. MacHiry, R. Tahiliani, M. Naik, Dynodroid: An input generation system for Android apps, in: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE'13), ACM Press, 2013.

[25] I. K. Mak, On the effectiveness of random testing, Master's thesis, The University of Melbourne, Melbourne, Australia (1997).

[26] Y. K. Malaiya, Antirandom testing: Getting the most out of black-box testing, in: Proceedings of the 6th International Symposium on Software Reliability Engineering, 1995.

[27] L. Padgham, Z. Zhang, J. Thangarajah, T. Miller, Model-based test oracle generation for automated unit testing of agent systems, IEEE Transactions on Software Engineering 39 (9) (2013) 1230–1244.

[28] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (10) (2001) 929–948.

[29] A. F. Tappenden, J. Miller, Automated cookie collection testing, ACM Transactions on Software Engineering and Methodology 23 (1) (2014) 3:1–3:40.

[30] S. Ukimoto, T. Dohi, H. Okamura, Software testing-resource allocation with operational profile, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12), 2012.

[31] F. I. Vokolos, P. G. Frankl, Empirical evaluation of the textual differencing regression testing technique, in: Proceedings of the International Conference on Software Maintenance, 1998.

[32] Z. Xu, K. Gao, T. M. Khoshgoftaar, N. Seliya, System regression test planning with a fuzzy expert system, Information Sciences 259 (2014) 532–543.

[33] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification and Reliability 22 (2012) 67–120.

[34] T. Yoshikawa, K. Shimura, T. Ozawa, Random program generator for java JIT compiler test system, in: Proceedings of the 3rd International Conference on Quality Software (QSIC'03), IEEE Computer Society Press, 2003.

[35] Z. Q. Zhou, A. Sinaga, W. Susilo, On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites, in: Proceedings of the 45th Hawaii International Conference on System Sciences (HICSS-45), IEEE Computer Society Press, 2012.

[36] Z. Q. Zhou, A. Sinaga, L. Zhao, W. Susilo, K.-Y. Cai, Improving software testing cost-effectiveness through dynamic partitioning, in: Proceedings of the 9th International Conference on Quality Software (QSIC'09), IEEE Computer Society Press, 2009.

Purpose: This is a Pure Random Testing algorithm, which tests a given program P using a given huge test suite $\{t_1, t_2, \ldots, t_n\}$ (n is very large). P can be modified for fault-removal upon the detection of a failure during the testing process. This algorithm also includes code to monitor the cost-effectiveness of itself in terms of the number of failures detected and the number of test cases executed.
Precondition: A testing stopping criterion has been given.

```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize Set0 to empty. /* to store used test cases */
3.   Initialize Set1 to {t1, t2, …, tn}. /* to store unused test cases */
4.   While (Set1 is not empty and the given testing stopping criterion is not met)
5.      Randomly select a test case ti from Set1.
6.      Test P using ti.
7.      Increase nbOfAllTests by 1. /* record the number of executed test cases */
8.      If (no failure is detected)
        Then
9.          Move ti from Set1 to Set0 /* so ti will not be chosen again */
        Else
10.         Increase nbOfDetectedFailures by 1.
11.         Print nbOfDetectedFailures and nbOfAllTests. /* current cost-effectiveness */
12.         An attempt can be made to remove the fault from P.
13.         Move all elements in Set0 to Set1. /* Hence, Set0 will become empty. */
        EndIf
14.  EndWhile
15.  Print nbOfDetectedFailures and nbOfAllTests. /* overall cost-effectiveness */
     End of Algorithm
```

Purpose: This is a Regression-Random Testing algorithm, which tests a given program P using a given huge test suite $\{t_1, t_2, \ldots, t_n\}$ (n is very large). P can be modified for fault-removal upon the detection of a failure during the testing process. This algorithm also includes code to monitor the cost-effectiveness of itself in terms of the number of failures detected and the number of test cases executed.
Precondition: A testing stopping criterion has been given.

```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize Set0 to empty.
3.   Initialize Set1 to {t1, t2, …, tn}.
4.   Initialize currentSet to empty. /* currentSet always contains no more than one element */
5.   Randomly select a test case ti from Set1, and move ti from Set1 to currentSet.
6.   While (currentSet is not empty and the given testing stopping criterion is not met)
7.      Test P using the test case in currentSet.
8.      Increase nbOfAllTests by 1.
9.      If (no failure is detected)
        Then
10.         Move the test case in currentSet to Set0.
11.         If Set1 is not empty, then randomly select a test case from Set1, and move
            it to currentSet.
        Else
12.         Increase nbOfDetectedFailures by 1.
13.         Print nbOfDetectedFailures and nbOfAllTests.
14.         An attempt can be made to remove the fault from P.
15.         Move all elements in Set0 to Set1. /* The element in currentSet remains there. */
        EndIf
16.  EndWhile
17.  Print nbOfDetectedFailures and nbOfAllTests.
     End of Algorithm
```

Purpose: This is an algorithm for testing through Dynamic Partitioning with Fixed Membership. It tests a given program P using a given huge test suite $\{t_1, t_2, \ldots, t_n\}$ (n is very large). P can be modified for fault-removal upon the detection of a failure during the testing process. This algorithm also includes code to monitor the cost-effectiveness of itself in terms of the number of failures detected and the number of test cases executed.
Precondition: A testing stopping criterion has been given.

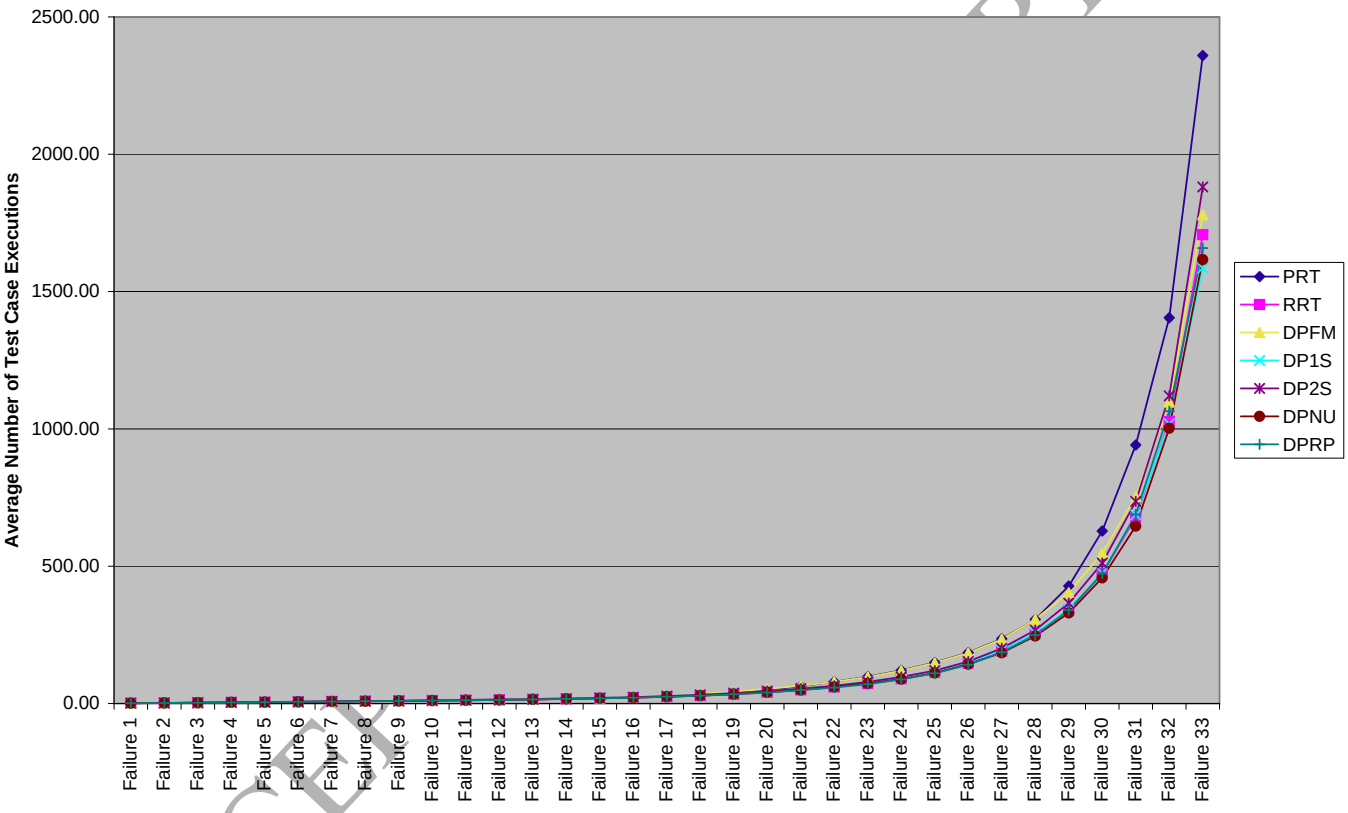
```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize fair_unused to {t1, t2, …, tn}.
3.   Initialize good_unused, good_used, fair_used, poor_unused, poor_used,
     and currentSet to empty.
4.   Initialize failureDetected to false.
5.   Initialize currentSetName to "fair".
6.   Randomly select a test case ti from fair_unused, and move ti to currentSet.
7.   While (currentSet is not empty and the given testing stopping criterion is not met)
8.      Test P using the test case in currentSet.
9.      Increase nbOfAllTests by 1.
10.     If (no failure is detected)
11.     Then Call sub-algorithm removeFromCurrentSet.
12.          Call sub-algorithm moveToCurrentSet.
13.          Set failureDetected to false.
14.     Else Increase nbOfDetectedFailures by 1.
15.          Print nbOfDetectedFailures and nbOfAllTests.
16.          An attempt can be made to remove the fault from P.
17.          Move all elements in good_used to good_unused, in fair_used to fair_unused, and
             in poor_used to poor_unused.
18.          Set failureDetected to true.
        EndIf
19.  EndWhile
20.  Print nbOfDetectedFailures and nbOfAllTests.
     End of Algorithm

     Begin Sub-Algorithm removeFromCurrentSet
1.   If (currentSetName is "good")
2.   Then Move the test case in currentSet to good_used.
3.   Else If (currentSetName is "fair")
          Then
4.             If (failureDetected is false)
5.             Then Move the test case in currentSet to poor_used.
6.             Else Move the test case in currentSet to good_used.
               EndIf
          Else /* currentSetName is "poor" */
7.             Move the test case in currentSet to poor_used.
          EndIf
     EndIf
     End of Sub-Algorithm removeFromCurrentSet

     Begin Sub-Algorithm moveToCurrentSet
1.   If (good_unused is not empty)
2.   Then Randomly select a test case ti from good_unused.
3.        Move ti to currentSet.
4.        Set currentSetName to "good".
5.   Else If (fair_unused is not empty)
6.        Then Randomly select a test case ti from fair_unused.
7.             Move ti to currentSet.
8.             Set currentSetName to "fair".
9.        Else If (poor_unused is not empty)
               Then
10.              Randomly select a test case ti from poor_unused.
11.              Move ti to currentSet.
12.              Set currentSetName to "poor".
               EndIf
          EndIf
     EndIf
     End of Sub-Algorithm moveToCurrentSet
```

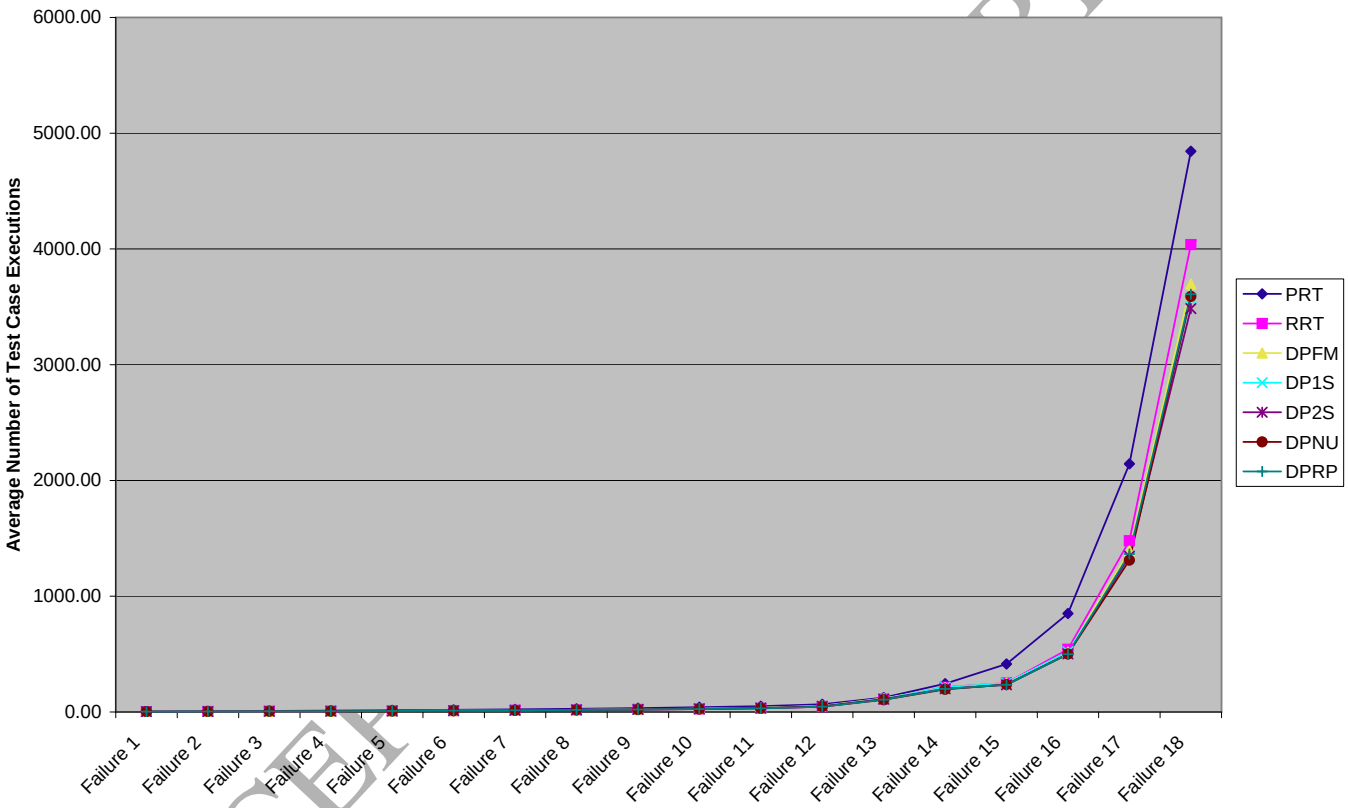
Note: This is an algorithm for testing through Dynamic Partitioning with One-Step Varying Membership. This algorithm differs from the Dynamic Partitioning with Fixed Membership (DPFM) algorithm only in the following sub-algorithm.

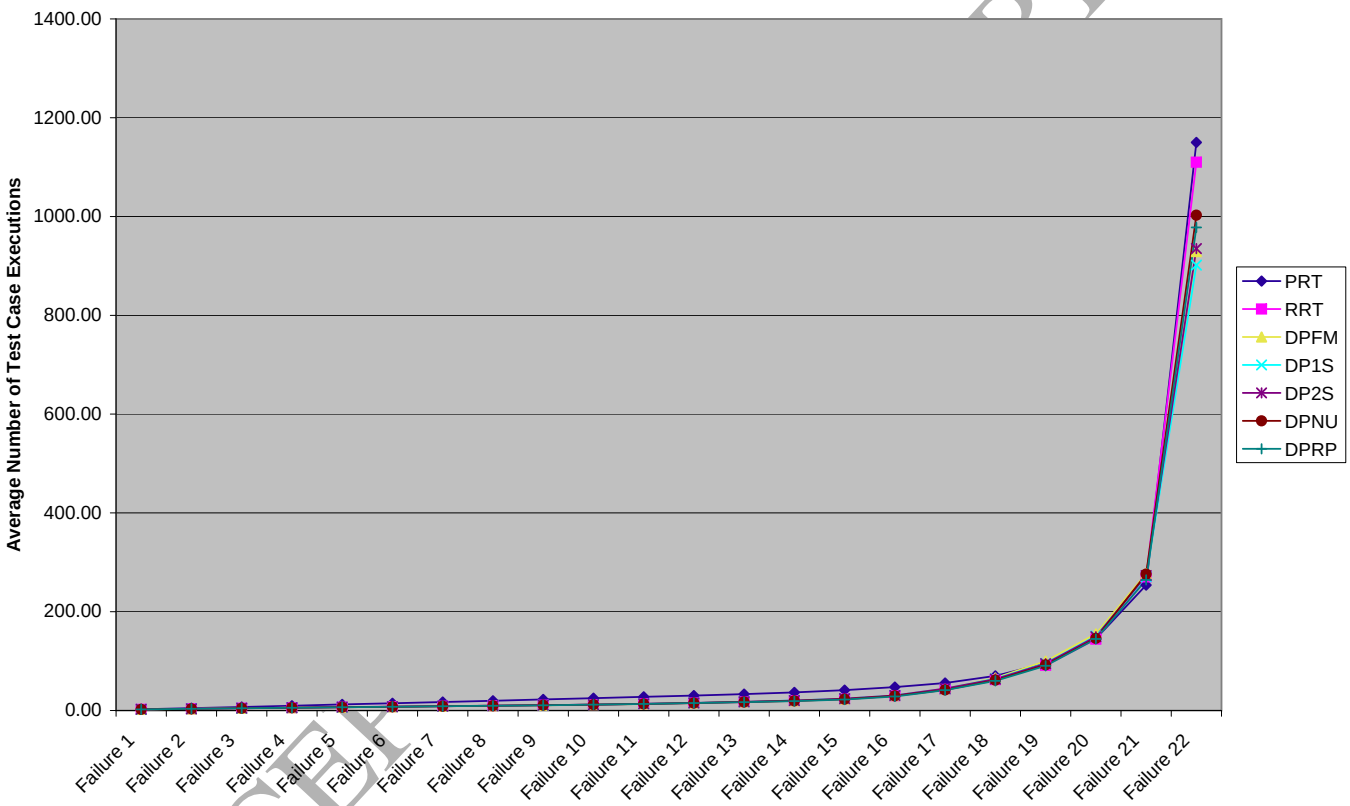
```
     Begin Sub-Algorithm removeFromCurrentSet
1.   If (currentSetName is "good")
2.   Then If (failureDetected is false)
3.        Then Move the test case in currentSet to fair_used.
4.        Else Move the test case in currentSet to good_used.
          EndIf
5.   Else If (currentSetName is "fair")
          Then
6.            If (failureDetected is false)
7.            Then Move the test case in currentSet to poor_used.
8.            Else Move the test case in currentSet to good_used.
              EndIf
          Else /* currentSetName is "poor" */
9.            If (failureDetected is false)
10.           Then Move the test case in currentSet to poor_used.
11.           Else Move the test case in currentSet to fair_used.
              EndIf
          EndIf
     EndIf
     End of Sub-Algorithm removeFromCurrentSet
```

Note: This is an algorithm for testing through Dynamic Partitioning with No Upgrade. This algorithm differs from the Dynamic Partitioning with Fixed Membership (DPFM) algorithm only in the following sub-algorithm. In this algorithm, when a test case detected a failure last time, but could not detect a failure this time, it will be returned to where it was selected from, either the "fair" or the "poor" set.

```
     Begin Sub-Algorithm removeFromCurrentSet
1.   If (currentSetName is "fair")
     Then
2.       If (failureDetected is true)
3.       Then move the test case in currentSet to fair_used.
4.       Else move the test case in currentSet to poor_used.
         EndIf
     Else /* currentSetName must be "poor" as the "good" set is never used. */
5.       move the test case in currentSet to poor_used.
     EndIf
     End of Sub-Algorithm removeFromCurrentSet
```







```
     Begin Algorithm
1.   Initialize nbOfAllTests and nbOfDetectedFailures to 0.
2.   Initialize fair_unused to {t1, t2, …, tn}.
3.   Initialize good_unused, good_used, fair_used, poor_unused, poor_used,
     and currentSet to empty.
4.   Initialize failureDetected to false.
5.   Initialize currentSetName to "fair".
6.   Randomly select a test case ti from fair_unused, and move ti to currentSet.
7.   While (currentSet is not empty and the given testing stopping criterion is not met)
8.      Test P using the test case in currentSet.
9.      Increase nbOfAllTests by 1.
10.     If (no failure is detected)
11.     Then Call sub-algorithm removeFromCurrentSet.
12.          Call sub-algorithm moveToCurrentSet.
13.          Set failureDetected to false.
14.     Else Increase nbOfDetectedFailures by 1.
15.          Print nbOfDetectedFailures and nbOfAllTests
16.          Update earlinessInfo of the current test case.
17.          An attempt can be made to remove the fault from P.
18.          Move all elements in good_used to good_unused, in fair_used to fair_unused, and
             in poor_used to poor_unused.
19.          Set failureDetected to true.
        EndIf
20.  EndWhile
21.  Print nbOfDetectedFailures and nbOfAllTests.
     End of Algorithm

     Begin Sub-Algorithm removeFromCurrentSet
1.   If (currentSetName is "good")
2.   Then Move the test case in currentSet to good_used.
3.   Else If (currentSetName is "fair")
          Then
4.             If (failureDetected is false)
5.             Then Move the test case in currentSet to poor_used.
6.             Else Move the test case in currentSet to good_used.
               EndIf
          Else /* currentSetName is "poor" */
7.             Move the test case in currentSet to poor_used.
          EndIf
     EndIf
     End of Sub-Algorithm removeFromCurrentSet

     Begin Sub-Algorithm moveToCurrentSet
1.   If (good_unused is not empty)
2.   Then Select a test case ti, whose earlinessInfo is minimum, from good_unused.
3.        Move ti to currentSet.
4.        Set currentSetName to "good".
5.   Else If (fair_unused is not empty)
6.        Then Randomly select a test case ti from fair_unused.
7.             Move ti to currentSet.
8.             Set currentSetName to "fair".
9.        Else If (poor_unused is not empty)
               Then
10.                 Randomly select a test case ti from poor_unused.
11.                 Move ti to currentSet.
12.                 Set currentSetName to "poor".
               EndIf
          EndIf
     EndIf
     End of Sub-Algorithm moveToCurrentSet
```