Test Case Prioritization: An Empirical Study

Gregg Rothermel

Department of
Computer Science
Oregon State U.
Corvallis, OR
grother@cs.orst.edu

Roland H. Untch

Department of
Computer Science
Middle Tenn. State U.
Murfreesboro, TN
untch@mtsu.edu

Chengyun Chu

Department of
Computer Science
Oregon State U.
Corvallis, OR
chengyun@cs.orst.edu

Mary Jean Harrold

Department of Computer
and Information Science
Ohio State University
Columbus, OH
harrold@cis.ohio-state.edu

Abstract

Test case prioritization techniques schedule test cases for execution in an order that attempts to maximize some objective function. A variety of objective functions are applicable; one such function involves rate of fault detection — *a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during regression testing can provide faster feedback on a system under regression test and let debuggers begin their work earlier than might otherwise be possible. In this paper, we describe several techniques for prioritizing test cases and report our empirical results measuring the effectiveness of these techniques for improving rate of fault detection. The results provide insights into the tradeoffs among various techniques for test case prioritization.*

1. Introduction

Software developers often save the test suites they develop for their software, so that they can reuse those suites later as the software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry [15] and, together with other regression testing activities, can account for as much as one-half of the cost of software maintenance [3, 13]. Running all test cases in an existing test suite, however, can consume an inordinate amount of time. For example, one of our industrial collaborators reports that for one of its products that contains approximately 20,000 lines of code, running the entire test suite requires seven weeks. In such cases, testers may want to order their test cases so that those test cases with the highest priority, according to some criterion, are run first.

Test case prioritization techniques [21] schedule test cases for regression testing in an order that attempts to maximize some objective function. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises subsystems in an order that reflects their historical propensity to fail. When the time required to execute all test cases in a test suite is short, test case prioritization may not be cost-effective — it may be most expedient simply to schedule test cases in any order. When the time required to run all test cases in the test suite is sufficiently long, however, test case prioritization may be beneficial.

In this paper, we describe several techniques for prioritizing test cases and we empirically evaluate their ability to improve *rate of fault detection* — a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during regression testing can provide earlier feedback on a system under regression test and let developers begin debugging and correcting faults earlier than might otherwise be possible.

Our results indicate that test case prioritization can significantly improve the rate of fault detection of test suites. Furthermore, our results highlight tradeoffs between various prioritization techniques. In the next section, we precisely describe the test case prioritization problem and outline several prioritization techniques. Subsequent sections present our experimental design, analysis, and conclusions.

2. Test Case Prioritization

Test case prioritization techniques schedule test cases in an execution order according to some criterion. The purpose of this prioritization is to increase the likelihood that if the test cases are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order.

Test case prioritization can address a wide variety of objectives, including the following:

1. Testers may wish to increase the rate of fault detection – that is, the likelihood of revealing faults earlier in a run of regression tests.

2. Testers may wish to increase the rate of detection of high-risk faults, locating those faults earlier in the testing process.

3. Testers may wish to increase the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process.

4. Testers may wish to increase their coverage of coverable code in the system under test at a faster rate.

5. Testers may wish to increase their confidence in the reliability of the system under test at a faster rate.

In practice, and depending upon the choice of objective, the test case prioritization problem may be intractable: for certain objectives, an efficient solution to the problem would provide an efficient solution to the knapsack problem [8]. Thus, test case prioritization techniques are typically heuristics. A goal of this work is to investigate, for a specific objective function, several such heuristics.

2.1. Prioritization for Rate of Fault Detection

Given a particular objective, various prioritization criteria may be applied to a test suite with the aim of meeting that objective. For example, to attempt to meet the first objective stated above, we may prioritize test cases in terms of the failure rates, measured historically, of the modules they exercise. Alternatively, we may prioritize test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirements specification. In any case, the intent behind the choice of a prioritization criterion is to increase the likelihood that the prioritized test suite can better meet the objective than would an ad hoc or random ordering of test cases.

In this work, we focus on the first objective listed above: increasing the likelihood of revealing faults earlier in the testing process. We describe this objective, informally, as one of improving our test suite's *rate of fault detection*: we describe a measure for this in Section 3.2. The motivation for meeting this objective is clear: an improved rate of fault detection during regression testing can provide faster feedback on the system under test, or early evidence that quality goals have not been met; it can also let debuggers begin their work earlier than might otherwise be possible.

We consider nine different test case prioritization techniques. Table 1 lists these techniques; we next discuss them in an order that facilitates their presentation.

$\mathcal{T}_1$**: No prioritization.** To facilitate our empirical study, one prioritization "technique" that we consider is simply the application of no technique; this lets us consider "untreated" test suites. Note, however, that the success of an untreated

Code	Mnemonic	Description
$\mathcal{T}_1$	unordered	no prioritization (control)
$\mathcal{T}_2$	random	randomized ordering
$\mathcal{T}_3$	optimal	ordered to optimize rate of fault detection
$\mathcal{T}_4$	branch-total	prioritize in order of coverage of branches
$\mathcal{T}_5$	branch-addtl	prioritize in order of coverage of branches not yet covered
$\mathcal{T}_6$	FEP-total	prioritize in order of total probability of exposing faults
$\mathcal{T}_7$	FEP-addtl	prioritize in order of total probability of exposing faults, adjusted to consider effects of previous tests
$\mathcal{T}_8$	stmt-total	prioritize in order of coverage of statements
$\mathcal{T}_9$	stmt-addtl	prioritize in order of coverage of statements not yet covered

Table 1. A catalog of prioritization techniques.

test suite in meeting an objective may depend upon the manner in which it is initially constructed.

$\mathcal{T}_2$**: Random prioritization.** Also to facilitate our empirical study, we apply random prioritization, in which we randomly order the tests in a test suite.

$\mathcal{T}_3$**: Optimal prioritization.** As we shall discuss in Section 3, to measure the effects of prioritization techniques on rate of fault detection, our empirical study utilizes programs that contain known faults. We can determine, for any test suite, which test cases expose which faults, and thus we can determine an optimal ordering of test cases in a test suite for maximizing that suite's rate of fault detection. In practice, of course, this is not a practical technique, as it requires knowledge of which test cases will expose which faults; however, by using it in our study, we gain insight into the success of other practical heuristics.

$\mathcal{T}_4$**: Total branch coverage prioritization.** By instrumenting a program, we can determine, for any test case, the number of decisions (branches) in that program that were exercised by that test case. We can prioritize these test cases according to the total number of branches they cover simply by sorting them in order of total branch coverage achieved. This prioritization can thus be accomplished in time $O(n \, log \, n)$ for programs containing n branches.

$\mathcal{T}_5$**: Additional branch coverage prioritization.** Total branch coverage prioritization schedules test cases in the order of total coverage achieved. However, having executed a test case and covered certain branches, more may be gained in subsequent test cases by covering branches that have not yet been covered. Additional branch coverage prioritization

iteratively selects a test case that yields the greatest branch coverage, then adjusts the coverage information on subsequent test cases to indicate their coverage of branches not yet covered, and then repeats this process, until all branches covered by at least one test case have been covered.

Having scheduled test cases in this fashion, we may be left with additional test cases that cannot add additional branch coverage. We could order these next using any prioritization technique; in this work we order the remaining test cases using total branch coverage prioritization.

Because additional branch coverage prioritization requires recalculation of coverage information for each unprioritized test case following selection of each test case, its cost is $O(n^2)$ for programs containing n branches.

$\mathcal{T}_8$: **Total statement coverage prioritization.** Total statement coverage prioritization is the same as total branch coverage prioritization, except that test coverage is measured in terms of program statements rather than decisions.

$\mathcal{T}_9$: **Additional statement coverage prioritization.** Additional statement coverage prioritization is the same as additional branch coverage prioritization, except that test coverage is measured in terms of program statements rather than decisions. With this technique too, we require a method for prioritizing the remaining test cases after complete coverage has been achieved, and in this work we do this using total statement coverage prioritization.

$\mathcal{T}_6$: **Total fault-exposing-potential (FEP) prioritization.** Statement- and branch-coverage-based prioritization consider only whether a statement or branch has been exercised by a test case. This consideration may mask a fact about test cases and faults: the ability of a fault to be exposed by a test case depends not only on whether the test case reaches (executes) a faulty statement, but also, on the probability that a fault in that statement will cause a failure for that test case [19]. Although any practical determination of this probability must be an approximation, we wished to determine whether the use of such an approximation could yield a prioritization technique superior in terms of rate of fault detection than techniques based on simple code coverage.

To obtain an approximation of the fault-exposing-potential (FEP) of a test case, we use mutation analysis [6, 9]. Given program P and test suite T, for each test case $t \in T$, for each statement s in P, we determine the mutation score $ms(s, t)$ to be the ratio of mutants of s exposed by t to total mutants of s. We then calculate, for each test case t_k in T, an *award value* for t_k, by summing all $ms(s, t_k)$ values. Total fault-exposing-potential prioritization orders the test cases in a test suite in order of these award values.

Such a technique could conceivably be much more expensive to implement than a code-coverage-based technique, however if such a technique shows promise, this might motivate a search for cost-effective methods to approximate fault-exposing potential.

$\mathcal{T}_7$: **Additional fault-exposing-potential (FEP) prioritization.** Analogous to the extensions made to total branch (or statement) coverage prioritization to additional branch (or statement) coverage prioritization, we extend total FEP prioritization to create additional fault-exposing-potential (FEP) prioritization. This lets us account for the fact that additional executions of a statement may be less valuable than initial executions. In additional FEP prioritization, after selecting a test case t, we lower the award values for all other test cases that exercise statements exercised by t.

2.2. Related Work

In [21], Wong, Horgan, London and Agrawal suggest prioritizing test cases according to the criterion of increasing cost per additional coverage. An implied objective of this ranking is to reveal faults earlier in the testing process. The authors restrict their attention, however, to prioritization of test cases for execution on a specific modified version of a program, and to prioritization of only the subset of test cases selected by a safe regression test selection technique (e.g. [2, 4, 18, 20]) from the test suite for the program. The authors do not specify a mechanism for prioritizing the remaining test cases after full coverage has been achieved. The authors describe a case study in which their technique is applied to a program of 5000 lines of code, and evaluated against ten faulty versions of that program, and conclude that the technique was cost-effective in that application.

In this work, we further investigate coverage-based prioritization, but we examine a wide range of prioritization techniques, and we focus on general, rather than modified-version-specific, prioritization.

3. The Experiment

3.1. Research Questions

We are interested in the following research questions.

Q1: Can test case prioritization improve the rate of fault detection of test suites?

Q2: How do the various test case prioritization techniques presented in Section 2 compare to one another in terms of effects on rate of fault detection?

3.2. Efficacy and APFD Measures

To address our research questions, we require measures with which we can assess and compare the effect of using various prioritization techniques.

Analogous to measuring an antibiotic's *potency* and *average activity*, we can seek to measure a prioritized test suite's *efficacy* and *average fault detection*. An antibiotic's potency is a measure of how well the drug can kill bacteria. In much the same way, a test suite's efficacy is a measure of its fault detecting ability. Although in general a test suite's efficacy cannot be measured directly and is often approximated by various coverage or adequacy measures, in our experiment we work with programs that contain known faults and efficacy is measured by how many of these known faults a test suite can detect.

An antibiotic's average activity is a measure of how quickly the bacteria are killed. A *fast acting* antibiotic exhibits earlier antibacterial behavior than a *slow acting* antibiotic. Similarly a *fast detecting* prioritized test suite exhibits earlier fault detection than a *slow acting* prioritized test suite. (It is important to note that changing a test suite's detection rate has no effect on its efficacy.)

As a measure of how rapidly a prioritized test suite detects faults, we use a weighted average of the percentage of faults detected, or *APFD*, over the life of the suite. These values range from 0 to 100; higher APFD numbers mean faster (better) fault detection rates. Thus, although not a direct measure of the rate of fault detection, the APDF does allow us to compare the ability of different prioritization techniques to create *faster detecting* test suites through the ordering of test cases.

To illustrate this measure, consider some example program with 10 faulty versions and a set of 5 test cases, **A** through **E**. Table 2 shows the fault detecting ability of each of the 5 test cases.

Test Case	Fault									
	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

Table 2. Test suite and list of faults exposed.

Suppose we place the test cases in order **A–B–C–D–E** to form a prioritized test suite $\mathcal{X}$. Figure 1 (left) shows the percentage of *undetected* faults versus the fraction of the test suite $\mathcal{X}$ used. After running test case **A**, 2 of the 10 faults were detected; thus 80% of the faults remain undetected after $\frac{1}{5}$ of test suite $\mathcal{X}$ has been used. After running test case **B**, 2 more faults are detected and thus 60% of the faults remain undetected after $\frac{2}{5}$ of the test suite has been used. In Figure 1 (left), the area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults undetected over the corresponding fraction of the test suite. The solid lines connecting the corners of the in-

scribed rectangles interpolate the drop in the percentage of undetected faults. This interpolation is a granularity adjustment when only a small number of test cases comprise a test suite; the larger the test suite the smaller this adjustment.

Figure 1 (right) corresponds to Figure 1 (left) but shows the percentage of *detected* faults versus the fraction of the test suite used. The curve represents the cumulative percentage of faults detected. The shaded area under the curve represents the weighted average of the percentage of faults detected over the life of the test suite. This area is the prioritized test suite's average percentage faults detected measure; the APFD is 50% in this example.

Figure 2 reflects what happens when the order of test cases is changed to **E–D–C–B–A**. Let us call this prioritized test suite $\mathcal{Y}$. Figure 2 (left) clearly depicts that no faults remain undetected after $\frac{3}{5}$ of test suite $\mathcal{Y}$ has been used. This increase in the rate of detection is reflected in Figure 2 (right); the APFD over the entire suite has risen to 64%, indicating $\mathcal{Y}$ is "faster detecting" than $\mathcal{X}$.

Figure 3 shows the effects of using a prioritized test suite $\mathcal{Z}$ whose test case ordering is **C–E–B–A–D**. By inspection, it is clear that this ordering results in the earliest detection of the most faults and illustrates an optimal ordering. From Figure 3 (right) we see that the APFD of test suite $\mathcal{Z}$ is 84%, the best of the three.

Thus to compare the effects of the prioritization techniques on test suites, our experiment investigates changes in APFD.

### 3.3. Subjects and Methods

3.3.1 Programs.

We used seven C programs as subjects (see Table 3). These programs perform a variety of tasks: `tcas` is an aircraft collision avoidance system, `schedule2` and `schedule` are priority schedulers, `tot_info` computes statistics given input data, `print_tokens` and `print_tokens2` are lexical analyzers, and `replace` performs pattern matching and substitution. Each program has a variety of versions, each containing one fault. Each program also has a large universe of inputs (test pool). These programs, versions, and inputs were assembled by researchers at Siemens Corporate Research for a study of the fault-detection capabilities of control-flow and data-flow coverage criteria [11]. We describe the other data in the table in the following paragraphs.

#### 3.3.2 Faulty versions, test cases, and test suites.

The researchers at Siemens sought to study the fault-detecting effectiveness of coverage criteria. Therefore, they created faulty versions of the seven base programs by manually seeding those programs with faults, usually by modi-

4

Test Case Order: A-B-C-D-E
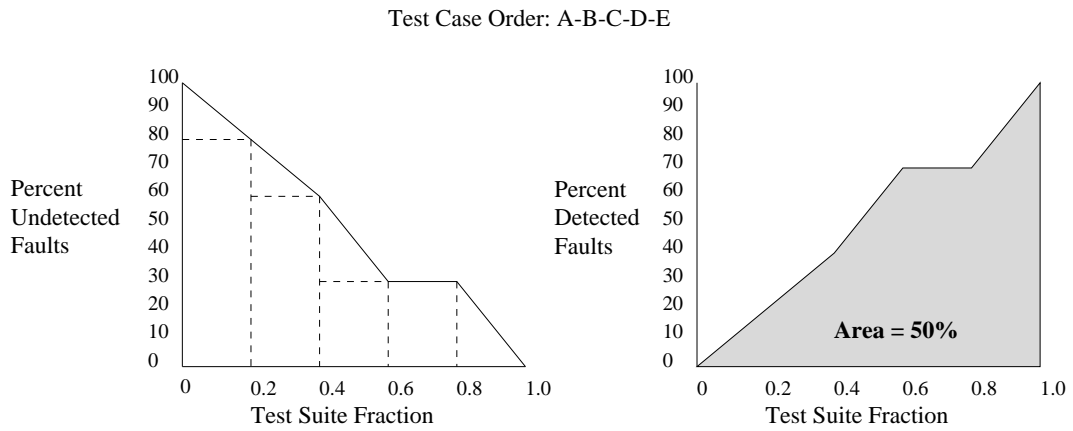
Figure 1. APFD for prioritized test suite $\mathcal{X}$: 50%.

Test Case Order: E-D-C-B-A
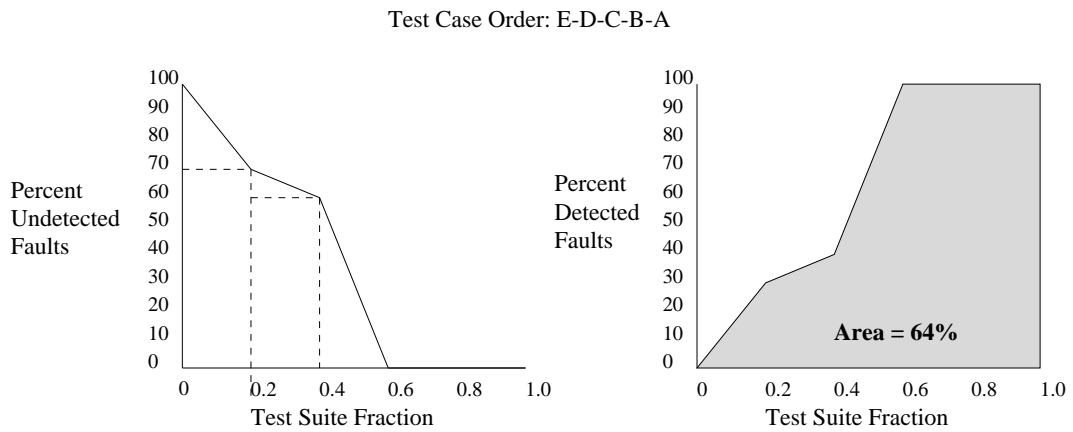
Figure 2. APFD for prioritized test suite $\mathcal{Y}$: 64%.

Test Case Order: C-E-B-A-D
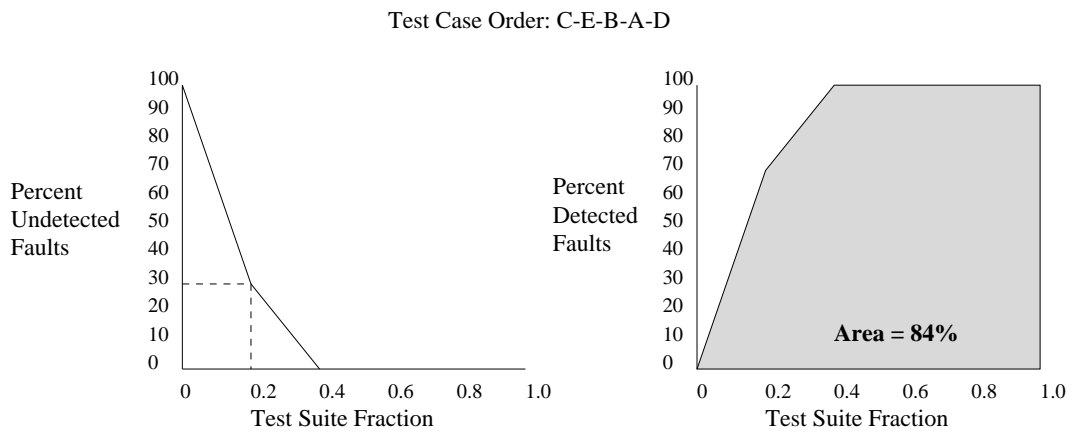
Figure 3. APFD for (optimal) prioritized test suite $\mathcal{Z}$: 84%.

Program	Lines of Code	No. of Versions	Test Pool Size	Test Suite Avg. Size
tcas	138	41	1608	6
schedule2	297	10	2710	8
schedule	299	9	2650	8
tot_info	346	23	1052	7
print_tokens	402	7	4130	16
print_tokens2	483	10	4115	12
replace	516	32	5542	19

Table 3. Experiment subjects.

fying a single line of code in the program. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed the fault seeding, working "mostly without knowledge of each other's work" [11, p. 196].

For each base program, the researchers at Siemens created a large test pool containing possible test cases for the program. To populate these test pools, they first created an initial suite of black-box test cases "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code" [11, p. 194], using the *category partition method* and the Siemens Test Specification Language tool [1, 16]. They then augmented this suite with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control-flow graph was exercised by at least 30 test cases. To obtain meaningful results with the seeded versions of the programs, the researchers retained only faults that were "neither too easy nor too hard to detect" [11, p. 196], which they defined as being detectable by at most 350 and at least 3 test cases in the test pool associated with each program.

To obtain sample test suites for these programs, we used the test pools for the base programs and test-coverage information about the test cases in those pools to generate 1000 branch-coverage-adequate test suites for each program. More precisely, to generate a test suite T for base program P from test pool T_p, we used the C pseudo-random-number generator `rand`, seeded initially with the output of the C `times` system call, to obtain integers that we treated as indexes into T_p (modulo $|T_p|$). We used these indexes to select test cases from T_p; we added each test case t to T only if t added to the cumulative branch coverage of P achieved by the test cases added to T thus far. We continued to add test cases to T until T contained at least one test case that would exercise each executable branch in the base program. Table 3 lists the average sizes of the branch-coverage-adequate test suites generated by this procedure for the subject programs.

3.3.3 Prioritization and analysis tools.

To perform the experiment, we required several tools. Our test coverage and control-flow graph information was provided by the Aristotle program analysis system [10]. We created prioritization tools that implement the techniques outlined in Section 2. To obtain mutation scores for use in the FEP prioritizations we used the Proteum mutation system [5].

3.4. Experiment design

The experiment was run using a 7×9 factorial design with 1000 APFD measures per cell; the two categorical factors were:

- The subject program (7 programs, each with a variety of modified versions).

- The prioritization technique (unordered, random, optimal, branch-total, branch-addtl, FEP-total, FEP-addtl, stmt-total, stmt-addtl).

For each subject program P, the prioritization techniques T_2 through T_9 were applied to each of the 1000 sample test suites, yielding 8000 prioritized test suites. The original test suite (not reordered) was retained as a control; for analysis this was considered "prioritized" by technique T_1. The APFD values of these 63000 prioritized test suites were calculated and used as the statistical data set.

3.5. Results

An initial indication of how each prioritization technique affected a test suite's rate of detection can be determined from Figure 4, which presents boxplots of the APFD values of the 9 categories of prioritized test suites for each program and an all-program total.[1] (Refer to Table 1 for a legend of the techniques.) T_1 is the control group. T_2 is the random prioritization group. T_3 is the optimal prioritization group. Examining the boxplots of T_3 with those of T_1 and T_2, it is readily apparent that optimal prioritization greatly improved the rate of fault detection (i.e., increased APFD values) of the test suites. Examining the boxplots of the other prioritization techniques, T_3 through T_9, it seems that all produce some improvement. However the overlap in APFD values mandates formal statistical analysis.

---

[1] A boxplot is a standard statistical device for representing data sets [12]. In these plots, each data set's distribution is represented by a box and a pair of "whiskers". The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The middle of the three horizontal lines within the box represents the median. The "whiskers" are the vertical lines attached to the box; they extend to the smallest and largest data points that are within the outlier cutoffs. These outlier cutoffs are defined to lie at 1.5 times the width of the inner quartile range (the span of the box) from the upper and lower points in that range.
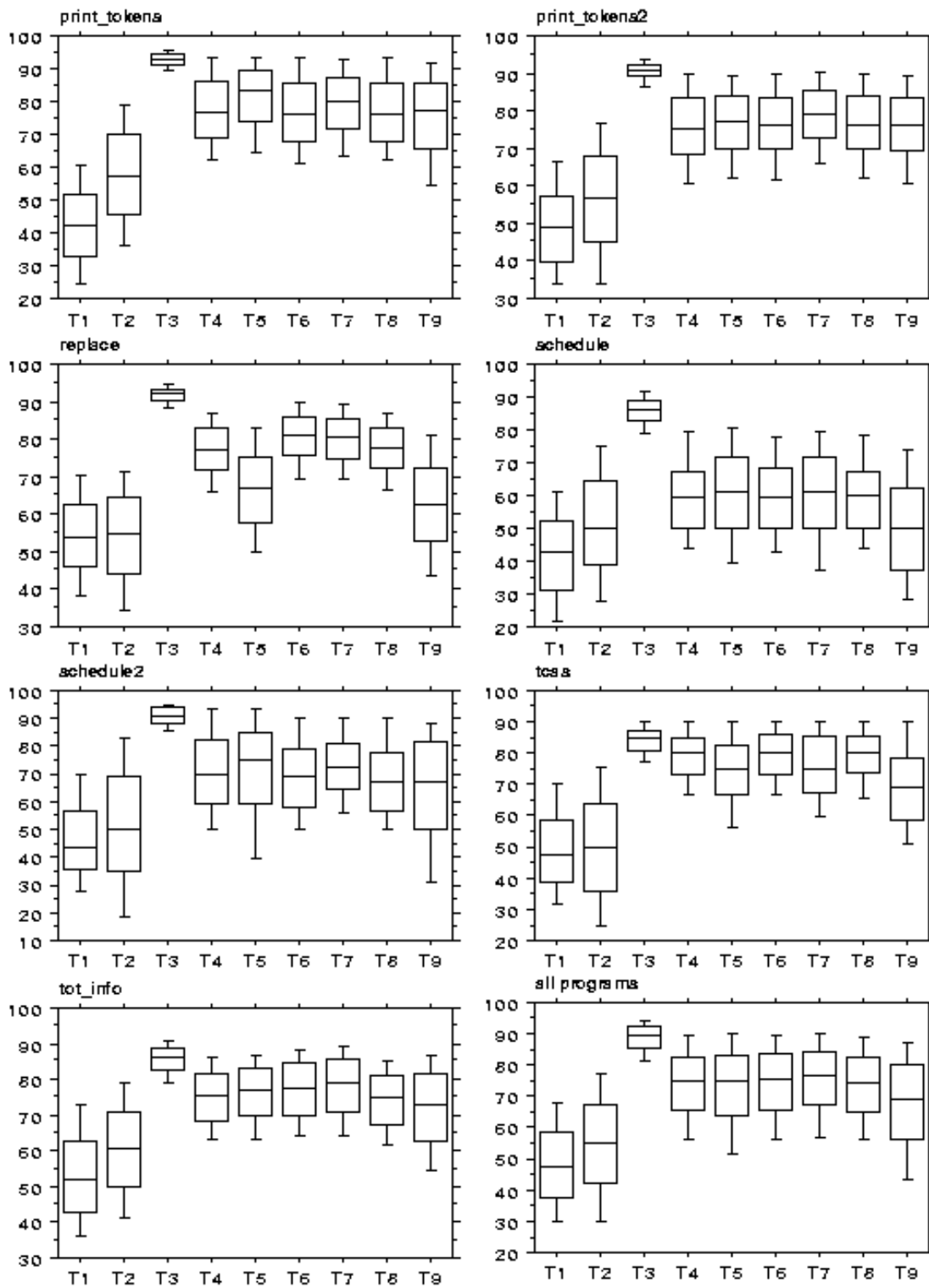
Figure 4. APFD boxplots (vertical axis is APFD score): By program, by technique.
(See Table 1 for a legend of the techniques.)

7

Using the SAS statistical package [7] to perform an ANOVA analysis, we were able to reject the null hypothesis that the APFD means for the various techniques were equal (α=.05), confirming our boxplot observations. However the ANOVA analysis indicated statistically significant cross-factor interactions: programs *have* an effect on APFD values. Thus general statements about technique effects must be qualified.

While rejection of the null hypothesis tells us that some techniques produce statistically different APFD means, to determine which techniques differ from each other requires running a multiple-comparison procedure [17]. Of the commonly used means separation tests, we elected to use the Bonferroni method — for its conservatism and generality.

Using Bonferroni, the minimum statistically significant difference between APFD means was calculated for each program. These are given in Table 4. The techniques are listed within each program sub-table by their APFD mean values, from higher (better) to lower (worse). Grouping letters partition the techniques; techniques that are not significantly different share the same grouping letter.

Examining these sub-tables affirm what the boxplots indicated: that all the heuristic techniques provided some significant improvement in rate of fault detection. Although the relative improvement provided by each technique is dependent on the program, the *All Programs* sub-table does suggest that the FEP-based heuristics (additional FEP prioritization and total FEP prioritization) may perform somewhat better than the others.

4. Discussion

This experiment, like any other, has several limits to its validity, the primary ones being threats to external validity that limit our ability to generalize our results. Our primary concern involves the representativeness of the artifacts utilized. The subject programs, though nontrivial, are small and larger programs may be subject to different cost-benefit tradeoffs. Also, there is exactly one seeded fault in every subject program; in practice, programs have much more complex error patterns. Finally, the test suites we utilized represent only one type of test suite that could appear in practice. These threats can only be addressed by additional studies utilizing a greater range of artifacts.

Keeping this in mind, our data and analysis nevertheless provide insights into the effectiveness of test case prioritization generally and into the relative effectiveness of the prioritization techniques that we examined. We now discuss these insights and their possible implications for the practical application of test case prioritization.

Perhaps of greatest practical significance, our data and analysis indicate that test case prioritization can substantially improve the rate of fault detection of test suites. All

of the heuristics that we examined produced such improvements overall and in only one case, on `schedule`, did any heuristic not outperform the untreated or randomly prioritized test suites.

Overall in our study, additional FEP prioritization outperformed all prioritization techniques based on coverage. Furthermore, total FEP prioritization outperformed all coverage-based techniques other than total branch coverage prioritization. However, these results did vary across individual programs and, where FEP-based techniques did outperform coverage-based techniques, the total gain in APFD was not great. These results run contrary to our initial intuitions and suggest that given their expense, FEP-based prioritization may not be as cost-effective as coverage-based techniques.

Again considering overall results, it is interesting that total branch coverage prioritization outperforms additional branch coverage prioritization and that total statement coverage prioritization outperforms additional statement coverage prioritization. These effects, too, vary across the individual programs. Nevertheless, the worst-case costs of total branch and statement coverage prioritization are much less than the worst-case costs of additional branch and statement coverage prioritization; this suggests that the less expensive total-coverage prioritization schemes may be more cost-effective than additional-coverage schemes.

Another effect worth noting is that generally (on five of the seven programs) randomly prioritized test suites outperformed untreated test suites. We conjecture that this difference is due to the type of test suites and faults used in the study. As described in Section 3.3, our test suites were generated for coverage by greedily selecting test cases from test pools; the order in which test cases were added to suites during this process constitutes their untreated order. We suspect that this process caused test cases added to the "ends" of the test suites to cover (on average) harder to reach statements than test cases added to the "beginnings" of the test suites. The faults embedded in the Siemens programs are relatively hard to detect; a disproportionate number reside in harder-to-reach statements and are detected (on average) by test cases that are added later to the test suites. Random prioritization essentially redistributes test cases that reach and expose these faults throughout the test suites, causing the faults to be detected more quickly.

5. Conclusions and Future Work

In this paper, we have described several techniques for test case prioritization and empirically examined their relative abilities to improve how quickly faults can be detected by those suites. Our results suggest that these techniques can improve the rate of fault detection of test suites and that this result occurs even for the least sophisticated (and hence least expensive) techniques.

print_tokens			
Grouping		Mean	Technique
	A	92.5461	optimal
	B	80.8842	branch-addtl
	C	78.2727	FEP-addtl
D	C	76.8573	branch-total
D	E	76.4770	FEP-total
D	E	76.4647	stmt-total
	E	74.8199	stmt-addtl
	F	57.2829	random
	G	42.6163	untreated
df= 8991 MSE= 155.0369 Critical Value of T= 3.20			
Minimum Significant Difference= 1.7808 (α=.05)			

print_tokens2			
Grouping		Mean	Technique
	A	90.5152	optimal
	B	78.3211	FEP-addtl
	C	76.1678	branch-addtl
	C	75.8848	stmt-total
	C	75.7985	FEP-total
	C	75.5995	stmt-addtl
	C	74.8830	branch-total
	D	55.9729	random
	E	49.3272	untreated
df= 8991 MSE= 124.203 Critical Value of T= 3.20			
Minimum Significant Difference= 1.5939 (α=.05)			

replace			
Grouping		Mean	Technique
	A	91.6901	optimal
	B	80.0171	FEP-total
	B	79.6959	FEP-addtl
	C	77.1355	stmt-total
	C	76.8482	branch-total
	D	66.5639	branch-addtl
	E	62.3795	stmt-addtl
	F	54.4460	untreated
	F	54.0668	random
df= 8991 MSE= 110.782 Critical Value of T= 3.20			
Minimum Significant Difference= 1.5053 (α=.05)			

schedule			
Grouping		Mean	Technique
	A	85.7074	optimal
	B	60.6765	branch-addtl
	B	59.8694	stmt-total
	B	59.8484	FEP-addtl
	B	59.6161	branch-total
	B	59.4430	FEP-total
	C	51.4087	random
	C	50.4418	stmt-addtl
	D	41.9670	untreated
df= 8991 MSE= 222.3662 Critical Value of T= 3.20			
Minimum Significant Difference= 2.1327 (α=.05)			

schedule2			
Grouping		Mean	Technique
	A	90.1794	optimal
	B	72.0518	FEP-addtl
	B	70.6432	branch-total
C	B	70.2513	branch-addtl
C	D	68.0438	FEP-total
	D	67.5409	stmt-total
	E	63.7391	stmt-addtl
	F	51.3077	random
	G	47.0302	untreated
df= 8127 MSE= 280.635 Critical Value of T= 3.20			
Minimum Significant Difference= 2.5199 (α=.05)			

tcas			
Grouping		Mean	Technique
	A	83.8845	optimal
	B	78.9253	stmt-total
	B	78.7998	FEP-total
	B	78.5781	branch-total
	C	75.1880	FEP-addtl
	D	73.3552	branch-addtl
	E	68.5357	stmt-addtl
	F	50.1038	random
	F	49.4311	untreated
df= 8973 MSE= 148.5302 Critical Value of T= 3.20			
Minimum Significant Difference= 1.7447 (α=.05)			

tot_info			
Grouping		Mean	Technique
	A	85.4258	optimal
	B	77.5442	FEP-addtl
C	B	76.8218	FEP-total
C	D	75.8798	branch-addtl
E	D	74.8807	branch-total
E		73.9979	stmt-total
	F	71.4503	stmt-addtl
	G	60.0587	random
	H	53.1124	untreated
df= 8991 MSE= 110.4918 Critical Value of T= 3.20			
Minimum Significant Difference= 1.5033 (α=.05)			

All Programs			
Grouping		Mean	Technique
	A	88.5430	optimal
	B	74.4501	FEP-addtl
	C	73.7049	FEP-total
D	C	73.2205	branch-total
D		72.9030	stmt-total
	E	71.9919	branch-addtl
	F	66.7502	stmt-addtl
	G	54.3575	random
	H	48.2927	untreated
df= 62055 MSE= 162.9666 Critical Value of T= 3.20			
Minimum Significant Difference= 0.6948 (α=.05)			

Table 4. Bonferroni means separation tests.

**Means with the same grouping letter
are not significantly different.**

The results of our study suggest several avenues for future work. First, we are performing additional studies utilizing other programs and types of test suites. To investigate a wider range and distribution of faults, we are also gathering data in which the success of prioritization techniques is measured against the set of mutations of the subject programs. Finally, we are investigating alternative measures of prioritization effectiveness.

Second, because our analysis revealed a sizeable performance gap between prioritization heuristics and optimal prioritization, and our FEP-based techniques did not bridge this gap, we are investigating alternative techniques based on sensitivity analysis [19] that may provide a more suitable approach. Techniques that incorporate static measures of fault-proneness [14] may also be of interest.

Finally, the test case prioritization problem, in general, has many more facets than we have here considered. For example, we have considered only one possible prioritization objective; other objectives, such as those listed in Section 2, are also of interest. Furthermore, the test case prioritization techniques that we have examined can be described as "general prioritization techniques" in the sense that they are applied to a base version of a program, with no knowledge of the location (or probable location) of modifications to the software, in the hopes of producing a test case ordering that will be effective over subsequent (and as yet unknown) versions of the software. Such general techniques could also incorporate information on probabilities of modification. Alternative techniques could utilize knowledge of the location of modifications to prioritize test cases for a particular modified version.

Through the results reported in this paper, and this future work, we hope to provide software practitioners with useful, cost-effective techniques for improving regression testing processes through prioritization of test cases.

6   Acknowledgements

References

[1] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proc. of the 3rd Symp. on Softw. Testing, Analysis, and Verification*, pages 210–218, Dec. 1989.

[2] T. Ball. On the limit of control flow analysis for regression test selection. In *ACM Int'l Symp. on Softw. Testing and Analysis*, pages 134–142, Mar. 1998.

[3] B. Beizer. *Softw. Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[4] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 211–222, May 1994.

[5] M. E. Delamaro and J. C. Maldonado. Proteum–A Tool for the Assessment of Test Adequacy for C Program s. In *Proc. of the Conf. on Performability in Computing Sys. (PCS 96)*, pages 79–95, New Brunswick, NJ, July 25–26 1996.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, Apr. 1978.

[7] R. J. Freund and R. C. Littell. *SAS for Linear Models: a guide to the ANOVA and GLM procedures*. SAS Institute Inc., Cary, NC, 1981.

[8] M. Garey and D. Johnson. *Computers and Intractability*. W.H. Freeman, New York, 1979.

[9] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Trans. Softw. Eng.*, SE-3(4):279–290, July 1977.

[10] M. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar 1997.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[12] R. Johnson. *Elementary Statistics*. Duxbury Press, Belmont, CA, sixth edition, 1992.

[13] H. Leung and L. White. Insights Into Regression Testing. In *Proc. of the Conf. on Softw. Maint.*, pages 60–69, Oct. 1989.

[14] J. Munson and T. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. on Softw. Eng.*, pages 423–433, May 1992.

[15] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. of the ACM*, 41(5):81–86, May 1988.

[16] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Comm. of the ACM*, 31(6), June 1988.

[17] L. Ott. *An Introduction to Statistical Methods and Data Analysis*. PWS-Kent Publishing Company, Boston, MA, third edition, 1988.

[18] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, Apr. 1997.

[19] J. Voas. PIE: A dynamic failure-based technique. *IEEE Trans. on Softw. Eng.*, pages 717–727, Aug. 1992.

[20] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proc. of the 3rd Int'l. Conf. on Rel., Quality & Safety of Softw.-Intensive Sys. (ENCRESS '97)*, May 1997.

[21] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. of the Eighth Intl. Symp. on Softw. Rel. Engr.*, pages 230–238, Nov. 1997.