The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments

Hyunsook Do
North Dakota State U.
hyunsook.do@ndsu.edu

Siavash Mirarab, Ladan Tahvildari
U. of Waterloo
{smirarab, ltahvild}@uwaterloo.ca

Gregg Rothermel
U. of Nebraska - Lincoln
grother@cse.unl.edu

December 6, 2009

Abstract

Regression testing is an expensive process used to validate modified software. Test case prioritization techniques improve the cost-effectiveness of regression testing by ordering test cases such that those that are more important are run earlier in the testing process. Many prioritization techniques have been proposed and evidence shows that they can be beneficial. It has been suggested, however, that the time constraints that can be imposed on regression testing by various software development processes can strongly affect the behavior of prioritization techniques. If this is correct, a better understanding of the effects of time constraints could lead to improved prioritization techniques, and improved maintenance and testing processes. We therefore conducted a series of experiments to assess the effects of time constraints on the costs and benefits of prioritization techniques. Our first experiment manipulates time constraint levels and shows that time constraints do play a significant role in determining both the cost-effectiveness of prioritization and the relative cost-benefit tradeoffs among techniques. Our second experiment replicates the first experiment, controlling for several threats to validity including numbers of faults present, and shows that the results generalize to this wider context. Our third experiment manipulates the numbers of faults present in programs to examine the effects of faultiness levels on prioritization, and shows that faultiness level affects the relative cost-effectiveness of prioritization techniques. Taken together, these results have several implications for test engineers wishing to cost-effectively regression test their software systems. These include suggestions about when and when not to prioritize, what techniques to employ, and how differences in testing processes may relate to prioritization cost-effectiveness.

Keywords: regression testing, test case prioritization, cost-benefits, bayesian networks, empirical studies.

1 Introduction

Software systems that succeed must evolve. Software engineers who enhance and maintain systems, however, run the risk of adversely affecting system functionality. To reduce this risk, engineers rely on *regression testing*: they rerun test cases from existing test suites, and create and run new test cases, to build confidence that changes have the intended effects and no unintended side-effects.

Regression testing is almost universally employed by software organizations [39]. It is important for software quality, but it can also be prohibitively expensive. For example, we are aware of one software development organization that has, for one of its primary products, a regression test suite containing over 30,000 functional test cases that require over 1000 machine hours to execute. Hundreds of hours of engineer time are also needed to oversee this regression testing process, set up test runs, monitor testing results, and maintain testing resources such as test cases, oracles, and automation utilities.

Test case prioritization techniques improve the cost-effectiveness of regression testing by ordering test cases such that those that are more important are run earlier in the testing process. Prioritization can provide earlier feedback to testers and management, and allow engineers to begin debugging earlier. It can also increase the probability that if testing ends prematurely, important test cases have been run.

Many researchers have addressed the test case prioritization problem (Section 8 summarizes related work) but most prior research has focused on the creation of specific prioritization techniques. A common approach for empirically assessing and comparing these techniques (e.g., [12, 18, 31]) is to first obtain several programs, modified versions, faults, and test suites. Then, the prioritization techniques being studied are applied to the test suites, the resulting ordered suites are executed, and measurements are taken of their effectiveness at satisfying testing objectives – typically in terms of the rates at which they detect faults or cover source code.

A limitation of this approach to studying prioritization is that it models the regression testing process as one in which organizations run *all* of their test cases. In practice, however, software development processes often impose *time constraints* on regression testing. For example, under incremental maintain-and-test processes such as nightly-build-and-test, the time required to execute all test cases can exceed the time available, and under batch maintain-and-test processes in which long maintenance phases are followed by long system testing phases, market pressures can force organizations to suspend testing before all test cases have been executed.

It has been conjectured [11, 26, 59] that the imposition of time constraints on regression testing may affect the costs and benefits of test case prioritization techniques. Indeed, as time constraints increase, engineers may need to omit increasingly larger numbers of test cases, and the resulting regression testing efforts may miss increasingly larger sets of faults. Larger sets of missed faults in turn result in increased costs later in the software lifecycle, through lost revenue, decreased customer trust, and higher maintenance costs. We also conjecture that the effects of time constraints may manifest themselves differently across different test case prioritization techniques; that is, the techniques that are most cost-effective under one set of time constraints may differ from those that are most cost-effective under different constraints.

If these conjectures are correct, by studying the effects of time constraints on prioritization, we may be able to help engineers manage their regression testing activities more cost-effectively, by pointing them to the techniques that are most appropriate given their engineering processes. Moreover, we may be able to suggest changes in maintenance and testing processes that will lead to more cost-effective regression testing. We therefore designed and implemented a series of controlled experiments to examine the effects of time constraints.

Our first experiment considers the application of several prioritization techniques to a set of object programs in which faults have been seeded, and in which these seeded faults were applied uniformly to all object programs. The results of our experiment show that both of the foregoing conjectures hold. In fact, for the objects that we examined, when no time constraints applied, prioritization was not cost-effective. When time constraints applied, prioritization began to yield benefits, and greater constraints resulted in greater benefits; moreover, in this case, prioritized test suites significantly outperformed unordered ordered test cases, suggesting that failing to prioritize is the worst choice of all. Finally, time constraints did affect prioritization techniques differently: some techniques were much more stable than others in their response to increased constraints.

Our first experiment allowed us to examine correlations between time constraints and prioritization effectiveness independent of factors related to numbers of faults, by utilizing fixed numbers of faults randomly seeded across locations in our object programs. In practice, however, the numbers of faults in systems may vary with various factors related to the system under test. To assess whether the results of our first experiment generalize, and to investigate issues related to numbers of faults and other threats to validity in the first experiment, we designed and performed two additional experiments.

Our second experiment replicates our first, using numbers of faults chosen to conform with a fault prediction model created by Bell, Ostrand, and Weyuker [3]. The results of this experiment substantially confirm those of the first. Prioritization yields benefits as time constraints increase, prioritized test suites outperform unordered and random test cases, and time constraints affect different prioritization techniques differently. The different fault numbers considered in this experiment, however, did alter results somewhat in relation to specific techniques; in particular, heuristics were found to be significantly more effective than control techniques more often than in the first experiment.

Our third experiment expressly manipulates the numbers of faults present in the systems studied as an independent variable, using three levels of faultiness. Our results continue to demonstrate that time constraints matter, and that when constraints do not exist, prioritization cost-effectiveness is not ensured. However, as the prevalence of faults increases, even with no time constraints, prioritization exhibits benefits more often. Further, when time constraints do exist, the effects of increased faultiness are even stronger.

Based on the foregoing results, we are able to suggest several practical implications for testing practitioners. Among these, we can say that when time constraints do not hold and when numbers of faults are expected to be small, prioritization may not be worthwhile. When time constraints do hold, however, the worst thing one can do is not prioritize. Our results also suggest that a certain class of techniques that employ "feedback" are more beneficial and stable in producing useful results than techniques that do not employ such feedback. Finally, we discuss several ways in which our results relate to prioritization effectiveness across different testing processes.

The rest of this article is organized as follows. Section 2 provides background information on prioritization. Section 3 presents a cost model that we utilize in our experiments for use in assessing prioritization techniques. Sections 4, 5, and 6 present our three experiments, including design, threats to validity, data and analysis, and interpretations of results. Section 7 discusses the results of all three experiments and their practical implications. Section 8 discusses related work. Finally, Section 9 presents conclusions and discusses possible future work.

2 Background: Regression Testing and Test Case Prioritization

Let P be a program that has been modified to create a new version P' and let T be a test suite developed for P. In the transition from P to P', the program could have regressed, that is, previously verified behavior of P could have turned faulty in P'. Regression testing attempts to validate P' in order to investigate whether it has regressed. The existing test suite, T, provides a natural starting point. In practice [39], engineers often reuse all of the test cases in T to test P'. Depending on the situation, this *retest-all* approach can be expensive [55] and since some test cases in T can be irrelevant to changes made in transforming P into P', the cost incurred is not necessarily all worthwhile.

Researchers have proposed various methods for improving the cost-effectiveness of regression testing. *Regression test selection techniques* (surveyed in [46]) reduce testing costs by selecting a subset of test cases from T to execute on P'. These techniques reduce costs by reducing testing time, but unless they are *safe* [47] they can omit test cases that would otherwise have detected faults. This can raise the costs of software.

Test case prioritization techniques (e.g., [48, 60]) offer an alternative approach to improving regression testing cost-effectiveness. These techniques help engineers reveal faults early in testing, allowing them to begin debugging earlier than might otherwise be possible. In this case, entire test suites may still be executed, avoiding the potential drawbacks associated with omitting test cases, and cost savings come from achieving greater parallelization of debugging and testing activities. Alternatively, if testing activities are cut short and test cases must be omitted, prioritization can improve the chances that important test cases will have been executed. In this case, cost savings related to early fault detection (by those test cases that are executed) still apply, and additional benefits accrue from lowering the number of faults that might otherwise be missed through less appropriate runs of partial test suites.

A wide range of prioritization techniques have been proposed and empirically studied (e.g., [22, 36, 48, 59, 60, 61, 62]). Most techniques utilize some form of code coverage information to direct the prioritization effort. Code coverage information is obtained by instrumenting a program such that certain code components (e.g, method entries, statements, branches, conditions, or basic blocks) can be observed to be exercised by test cases.

Given code coverage information for a test suite T, one way to prioritize is to use a “total-coverage” prioritization technique, that orders test cases in terms of the total number of code components that they cover. Total-coverage techniques can be improved by adding *feedback*, using an iterative greedy approach in which each “next” test case is placed in the prioritized order taking into account the effect of test cases already placed in the order. For example, an “additional-block-coverage” technique prioritizes test cases in terms of the numbers of new (not-yet-covered) blocks they cover, by iteratively selecting the test case that covers the most not-yet-covered blocks until all blocks are covered, then repeating this process until all test cases have been prioritized.

More recently, techniques have become more sophisticated in terms of the sources of information they utilize and the manner in which this information is incorporated – Section 8 provides a comprehensive discussion.

3 Measuring the Cost-Benefits of Test Case Prioritization

Most early work on prioritization utilized simple metrics assessing the rate at which faults are detected by test cases (e.g., APFD [48] and APFD$_C$ [34]) to evaluate prioritization technique effectiveness (Section 8 provides a summary). Such metrics relate to the savings in debugging costs that flow from earlier fault detection. These metrics do not suffice to assess time-constrained techniques, however, because such assessment requires that costs related to omitted faults also be measured. Moreover, these savings and costs must be measured in comparable units so that they can both be considered in technique assessment. A comprehensive view of tradeoffs also requires consideration of the costs of applying techniques, and of utilizing them not just on single system releases but across entire system lifetimes, and neither of these are considered by prior metrics.

For this reason, in this work, we rely on an economic model, EVOMO (EVOlution-aware economic MOdel for regression testing), which is currently the only existing economic model capable of capturing the foregoing factors comprehensively. We summarize the model here; for further details we refer the reader to [11, 13].

3.1 A Regression Testing Process Model

Economic models capture costs and benefits of methodologies relative to particular processes, so we begin our discussion by providing a model of the regression testing process on which EVOMO is based. Our model corresponds to the most commonly used approach for regression testing at the system test level [39] – a *batch* process model.

Figure 1 presents a timeline depicting the maintenance, regression testing and fault correction, and post-release phases for a single release of a software system following a batch process model. Time $t1$ represents the time at which maintenance (including all planning, analysis, design, and implementation activities) of the release begins. At time $t2$ the release is code-complete, and regression testing and fault correction begin – these activities may be repeated and may overlap within time interval $(t2 : t3)$, as faults are found and corrected. When this phase ends, at time $t3$, product release can occur – at this time, revenue associated with the release begins to accrue. In an idealized setting, product release occurs on schedule following testing and fault correction, and this is the situation depicted in the figure.

This process model relates to the research questions we wish to investigate as follows. Suppose there are no time constraints limiting testing. In this case, test case prioritization techniques attempt to reduce time interval $(t2 : t3)$

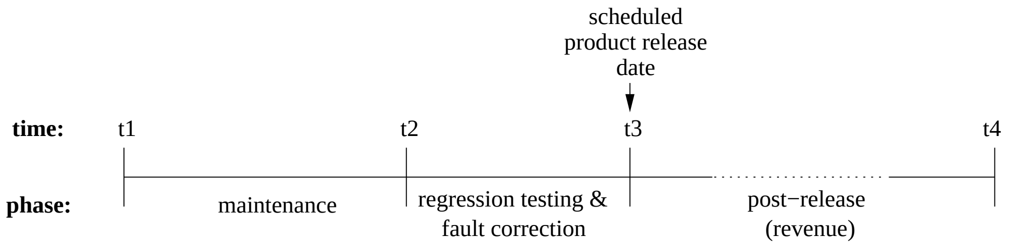
scheduled product release date
time:
t1
t2
t3
t4
phase:
maintenance
regression testing & fault correction
post−release (revenue)

Figure 1: Maintenance and regression testing lifecycle.

by allowing greater portions of the fault correction activities that occur in that period to be performed in parallel with testing, rather than afterward. If this succeeds, the software can be released early and overall revenue can increase. If prioritization is unsuccessful and fault correction activities cause time interval $(t2 : t3)$ to increase, then the release is delayed and revenue can decrease. Next, suppose time constraints force testing to be terminated early. In this case, revenue may increase but with potential for additional costs later due to omitted faults. Here, test case prioritization can decrease such costs by increasing the likelihood that faults are detected prior to the termination of testing.

This batch process model makes several assumptions. For example, organizations also create software for reasons other than to create revenue. Organizations that complete testing early could spend additional time performing other forms of verification until the scheduled release date arrives, and this could lead to increased revenue via reduced fault cost downstream. Revenue may not always increase as time interval $(t3 : t4)$ increases; earlier release could conceivably result in a decrease in revenue. Moreover, revenue itself is not the sole measure of benefit, because market value (e.g., the value of a company's stock) is also important. Also, there are many other regression testing process models that could be utilized in practice; for example, some organizations use incremental testing processes, in which test cases are run each night as maintenance proceeds.

These differences noted, the model does allow us to investigate our research questions, in a manner that is much more comprehensive than that used in research on regression testing to date. We also believe that the EVOMO model can be adjusted to accommodate different regression testing processes, and we discuss this further in Section 9.

3.2 The EVOMO Economic Model

EVOMO (EVOlution-aware economic MOdel) captures the costs and benefits of regression testing methodologies in terms of the cost of applying the methodologies and how much revenue they help organizations obtain.

EVOMO involves two equations: one that captures costs related to the salaries of the engineers who perform regression testing (to translate time spent into monetary values),and one that captures revenue gains or losses related to changes in system release time (to translate time-to-release into monetary values). The model accounts for costs and benefits across entire system lifetimes rather than on snapshots (i.e. single releases) of those systems, through equations that calculate costs and benefits *across entire sequences of system releases.* The model also accounts for the use of incremental analysis techniques (e.g., reliance on previously collected data where possible rather than on complete recomputation of data), an improvement also facilitated by the consideration of system lifetimes.

The two equations that comprise EVOMO are as follows. Terms, coefficients, and potential measures that can be used to capture these are summarized in Table 1.

$$Cost = PS * \sum_{i=2}^{n}(CS(i) + CO_i(i) + CO_r(i) + b(i) * CV_i(i) + c(i) * CF(i)) \tag{1}$$

$$\begin{aligned} Benefit = REV * \sum_{i=2}^{n}(ED(i) - (CS(i) + CO_i(i) + CO_r(i) + a_{in}(i-1) * CA_{in}(i-1) \\ + a_{tr}(i-1) * CA_{tr}(i-1) + CR(i) + b(i) * (CE(i) + CV_i(i) + CV_d(i)) + CD(i))) \end{aligned} \tag{2}$$

Table 1: Terms, Coefficients, and Potential Measures

Term	Description
S	Software system
i	Index denoting a release S_i of *S*
n	The number of releases of the software system
u	Unit of time (e.g., hours or days)
$a_{in}(i)$	Coefficient to capture reductions in costs of instrumentation for S_i due to the use of incremental analysis techniques
$a_{tr}(i)$	Coefficient to capture reductions in costs of trace collection for S_i due to the use of incremental analysis techniques
$b(i)$	Coefficient to capture reductions in costs of executing and validating test cases for S_i due to the use of incremental analysis techniques
$c(i)$	Number of faults that are not detected by a test suite applied to S_i

Term	Description	Potential Measure
$CS(i)$	Time to perform setup activities required to test S_i	The costs of setting up the system for testing, compiling the version to be tested, and configuring test drivers and scripts
$CO_i(i)$	Time to identify tests that are obsolete for S_i	The costs of manual inspection of a version and its test cases, and determination, given modifications made to the system, of the test cases that must be modified for the next version
$CO_r(i)$	Time to repair obsolete tests for S_i	The costs of examining and adjusting test cases and test drivers, and the costs of observing the execution of adjusted tests and drivers
$CA_{in}(i)$	Time to instrument all units in *i*	The costs of instrumenting programs
$CA_{tr}(i)$	Time to collect traces for test cases in S_{i-1}	The costs of collecting execution traces
$CR(i)$	Time to execute a prioritization technique on S_i	The time required to execute a prioritization technique itself
$CE(i)$	Time to execute test cases on S_i	The time required to execute tests
$CV_d(i)$	Time to use tools to check outputs of test cases on S_i	The time required to run a differencing tool on test outputs as test cases are executed
$CV_i(i)$	Human time for inspecting the results of test cases	The time required by engineers to inspect comparisons of test outputs
$CF(i)$	Cost of missed faults after delivery of S_i	To estimate this cost, we rely on data provided in [53]; we used 1.2 hours as the time required to correct faults after delivery
$CD(i)$	Cost of delayed fault detection feedback on S_i	Following [34], we translate the rate of fault detection into the cumulative cost (in time) of waiting for each fault to be exposed while executing test cases under the prioritized order
REV	Revenue in dollars per unit *u*	We estimate this value by utilizing revenue values cited in a survey of software products ranging from \$116,000 to \$596,000 per employee [8]
PS	Average hourly programmer's salary in dollars per unit *u*	We rely on a figure of \$100 per person-hour, obtained by adjusting an amount cited in [24] by an appropriate cost of living factor
$ED(i)$	Expected time-to-delivery for S_i when testing begins	Actual values for ED cannot be obtained for our object programs. Thus, rather than calculate ED, we use the relative cost-benefits to compare techniques; this causes the value of ED to be canceled out

In addition to capturing the costs related to the tasks involved in prioritizing and running tests, this model captures the two primary drivers of costs and benefits that relate to time constraints as outlined above. $CD(i)$ captures costs related to delayed fault detection feedback (and thus, benefits related to reductions in delays); this cost occurs whether time constraints are present or not. $CF(i)$ captures costs related to faults missed in regression testing, this cost occurs when time constraints force testing to end prior to execution of the entire test suite.

4 Experiment 1: The Effects of Time Constraints on Prioritization

Our first experiment addresses the following research questions:

RQ1: Given a specific test case prioritization technique, as time constraints vary, in what ways is the performance of that technique affected?

RQ2: Given a specific time constraint on regression testing, in what ways do the performances of test case prioritization techniques differ under that constraint?

To address these questions we performed a controlled experiment. The following subsections present our objects of analysis, variables and measures, setup and design, threats to validity, data and analysis, and discussion of results.

4.1 Objects of Analysis

We used five Java programs from the SIR infrastructure [9] as objects of analysis: *ant*, *xml-security*, *jmeter*, *nanoxml*, and *galileo*. *Ant* is a Java-based build tool, similar to `make` but extended using Java classes instead of shell-based commands. *Jmeter* is a Java desktop application used to load-test functional behavior and measure performance. *Xml-security* implements security standards for XML. *Nanoxml* is a small XML parser for Java. *Galileo* is a Java bytecode analyzer. Several sequential versions of each of these programs are available. The first three programs are provided with JUnit test suites, and the last two are provided with TSL (Test Specification Language) test suites [40].

Table 2 lists, for each of our objects of analysis, data on its associated "Versions" (the number of versions of the object program), "Classes" (the number of class files in the latest version of that program), "Size (KLOCs)" (the number of lines of code in the latest version of the program), and "Test Cases" (the number of test cases available for the latest version of the program).

Table 2: Experiment Objects and Associated Data

Objects	Versions	Classes	Size (KLOCs)	Test Cases	Mutation Faults
ant	9	627	80.4	877	412
jmeter	6	389	43.4	78	386
xml-security	4	143	16.3	83	246
nanoxml	6	26	7.6	216	204
galileo	16	87	15.2	912	2494

To address our research questions we require faulty versions of our object programs, so we utilized mutation faults created by members of our research group for an earlier study [12] and now available from the SIR repository with the programs. Because our focus is regression testing and detection of regression faults (faults created by code modifications), we considered only mutation faults located in modified methods. The total numbers of mutation faults considered for our object programs, summed across all versions of each program, is shown in the rightmost column of Table 2.

4.2 Variables and Measures

4.2.1 Independent Variables

Given our research questions, our experiments manipulated two independent variables: *time constraints* and *prioritization technique*.

Variable 1: Time Constraints

The time constraints imposed on regression testing by various software development processes directly affect regression testing cost-effectiveness by limiting the amount of testing that can be performed. Thus, to assess the effects of time constraints, our first independent variable controls the amount of regression testing.

For the purpose of this study, we utilize four *time constraint levels*: TCL-0, TCL-25, TCL-50, and TCL-75. TCL-0 represents the situation in which no time constraints apply, and thus, testing can be run to completion. TCL-25, TCL-50, and TCL-75 represent situations in which time constraints reduce the amount of testing that can be done by 25%, 50%, and 75%, respectively.

To implement time constraint levels, for simplification, we assume that all of the test cases for a given object program have equivalent execution times – this assumption is reasonable for our object programs for which test execution time varies only slightly. We then manipulate the number of test cases executed to obtain results for different time constraint levels. For example, in the case of TCL-25, for each version S_i of object program S and for each prioritized test suite T_t for S_i, we halt the execution of the test cases in T_t on S_i as soon as 75% of those test cases have been run (thus omitting 25% of the test cases).

Variable 2: Prioritization Technique

We consider two *control* techniques and four *heuristic* prioritization techniques.

Control techniques are those that are used as experimental controls; these do not involve any "intelligent" algorithms for ordering test cases. We consider two such techniques, "original" and "random". Original ordering utilizes the order in which test cases are executed in the original testing scripts provided with the object programs, and thus, serves as one potential representative of "current practice". Random ordering utilizes random test case orders (in our experiment, averages of runs of multiple random orders) and thus, provides a baseline for technique comparison that abstracts away the possible vagaries of a single control order.

Heuristic techniques attempt to improve the effectiveness of test case orders. As heuristic techniques, we selected four techniques drawn from two overall prioritization methodologies: conventional code-coverage-based prioritization [48] and Bayesian Network-based prioritization [36]. For each of these methodologies we consider two techniques: one that incorporates feedback and one that does not.

Conventional code-coverage-based (CC) prioritization techniques, as discussed in Section 2, rely solely on code coverage information obtained when test cases are run on the prior release P, to order test cases for execution on P'. The techniques we use rely on code coverage measured at the level of basic blocks in control flow graphs built from the Java bytecode of our object programs.

Bayesian Network-based (BN) prioritization techniques use estimates of the conditional probabilities that (1) changes cause faults, (2) code quality renders modules fault-prone, and (3) faults present in code may be revealed by test cases, encodes these in a Bayesian Network, and apply Bayesian Analysis to obtain prioritized test case orders (see [36] for details). Note that BN techniques use code coverage information at the level of classes to obtain certain estimates, and this coarser-grained level of instrumentation can potentially cause their costs to differ from those of CC techniques. Further, BN techniques must be configured via parameters, and we utilize results obtained from a prior empirical study [37] to select parameter values.

The CC and BN methodologies that we study represent two of the earliest and algorithmically simplest techniques proposed to date, and two of the most recent and algorithmically complex techniques proposed to date, respectively. They offer a spectrum of technique costs and potential benefits across which to conduct our study. (Section 8 provides further comments on technique selection relative to other potential approaches.)

Table 3 summarizes the six techniques that we consider, and assigns mnemonics to them (*To*, *Tr*, *Tcc*, *Tbn*, *Tccf*, and *Tbnf*) for use in subsequent discussion.

Table 3: Test Case Prioritization Techniques

Group	Label	Technique	Description
control	To	original	original order
	Tr	random	random order
non-feedback	Tcc	total CC	prioritize on coverage of blocks
	Tbn	total BN	prioritize via Bayesian Network
feedback	Tccf	additional CC	prioritize on coverage of blocks with feedback mechanism
	Tbnf	additional BN	prioritize via Bayesian Network with feedback mechanism

4.2.2 Dependent Variable and Measures

Our dependent variable is a *relative cost-benefit value* produced by applying the economic model presented in Section 3, using a further calculation described below. The cost and benefit components are measured in dollars. The cost components include several constituent measures, which we collected as described in Table 1 in Section 3. To measure costs that involve human activities we averaged times required by two graduate students to perform the activities.

Relative Cost-benefit

We considered two approaches for comparing techniques. The first approach calculates absolute cost-benefit values for each technique, using Equations 1 and 2 (Section 3). A drawback of this approach, however, is that it requires data or estimates pertaining to the ED variable, and it is difficult to find such data or reasonable estimates for our object programs.

The second approach calculates *relative cost-benefit values*, in which the cost-benefits of techniques are determined relative to those of a baseline technique. This approach does not require values for ED; moreover, it normalizes the cost-benefit values calculated for techniques relative to a shared baseline, rendering their comparison independent of particular choices of ED. To determine the *relative cost-benefit* of prioritization technique *T* with respect to baseline technique *base*, we use the following equation:

$$(Benefit_T - Cost_T) - (Benefit_{base} - Cost_{base}) \tag{3}$$

When this equation is applied, positive values indicate that T is beneficial compared to $base$, and negative values indicate otherwise.

We chose the second approach, selecting the random order of test cases as a baseline, and utilizing mean values achieved across 30 different random orders to obtain baseline values. This use of mean values across multiple runs limits the effect of chance on the baseline value. It also produces more reliable comparisons than could be obtained with a single instance of an alternative baseline such as the original test case order, which might exhibit a particular trend that would be propagated to the outcomes of all comparisons.

4.3 Experiment Setup

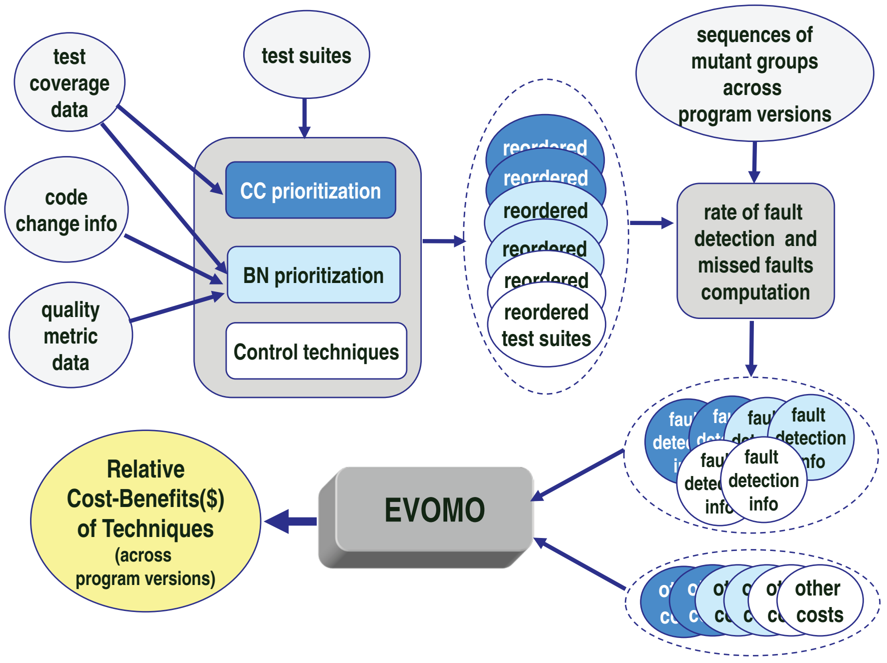

Figure 2: Experiment setup (across sequences of versions for a given object program and a given time constraint level).

Figure 2 helps to illustrate our experiment setup, as applied to sequences of versions per object program P, for a given time constraint level TCL-k. The figure depicts data items in bubbles and major processing elements in rectangles, and edges represent inputs and outputs of data. As shown in the figure, to perform prioritization, the CC and BN techniques require test coverage information. The BN techniques also require data related to code changes and software quality metrics. We obtained coverage information for each object program P by running test cases on each version of P instrumented using `Sofya` [27]. The resulting information lists which test cases exercised which blocks in each version; a previous version's coverage information is then used to prioritize a current version's set of test cases. In the case of BN techniques, block coverage data is abstracted to the class level to determine the percentage of blocks covered in a particular class. We obtained code change information using `sandmark` [7] as a byte code differencing tool, and we obtained software quality metrics (including data on coupling, cohesion, and complexity) from the Chidamber-Kemerer metrics suite [6] using the `ckjm` program.[1] Using this information and the original test suite, prioritization techniques produce reordered test suites for each version of P.

Recall that our economic model (EVOMO) measures the costs and benefits of regression testing techniques across *sequences of program versions*. To obtain the fault data required to investigate our research questions using this model, we required program versions containing multiple faults. To provide these we constructed *mutant groups*. To construct a mutant group for version V of P, we first randomly chose a number n between one and ten. We then randomly selected n mutation faults from those available with version V, and instantiate them in V. Applying this process to each of the versions of P yields a *sequence of mutant groups* for that sequence of versions. We created 30 such sequences of mutant groups for each of our programs P.

To collect required data for P, for each version of P and each selected mutant group for that version, and each prioritization technique, we recorded the appropriate values for cost variables related to applying that technique. In the case of the random technique, we did this for 30 different random orders, and averaged the results. The required data included data on rate of fault detection and faults missed, as well as other cost variables (CE, CV_d, CV_i, CF, c, CD). All machine times were measured on a PC running SuSE Linux 9.1 with 1GB RAM and a 3 GHZ processor.

We used the collected cost variable values to calculate relative cost-benefit values for each of the prioritization techniques on P. Each of these calculations required us to calculate the relative cost-benefit of the given technique (using Equation 3) at the given time constraint level TCL-k for each of the 30 sequences of mutant groups created for P. These resulting relative cost-benefit numbers serve as the data for our subsequent analysis.[2]

4.4 Threats to Validity

This section describes the threats to the validity of our study, and the approaches we used to limit their effects.

External Validity. The Java programs that we study are relatively small (7K - 80K), and their test suites' execution times are relatively short. Complex industrial programs with different characteristics may be subject to different cost-benefit tradeoffs. The testing process and the cost of faults we used are not representative of all processes used or fault costs observed in practice. We examine only four prioritization heuristics, and the prioritization and instrumentation tools that we used in this study are prototypes, and thus may not reflect the performance of more robust industrial tools. Our faults are derived through code mutation, and although there is some evidence that mutation faults can be representative of real faults for purposes of experimental evaluation of the effectiveness of testing techniques [2], the numbers of mutation faults used in our study may not match numbers of faults found in practice. In particular, we utilize random numbers of faults not exceeding 10 for all the object programs irrespective of size. Control for these threats can be achieved only through additional studies with wider populations of programs and faults, different testing

[1] http://www.spinellis.gr/sw/ckjm/
[2] Complete data sets can be obtained from the first author.

processes and prioritization techniques, different fault severities and fault distributions, and improved tools.

Internal Validity. The inferences we have made about the effects of time constraints could have been affected by other factors. One factor involves potential faults in the tools that we used to collect data. To control for this threat, we validated our tools on several simple Java programs. A second factor involves the actual values we used to calculate costs, some of which required estimation. We estimated the costs of test setup, finding obsolete tests, repairing obsolete tests, and validating outputs by measuring the time taken by graduate students to perform these tasks. The values we used for revenue and costs of missing and correcting faults are obtained from surveys found in the literature, but such values can be situation-dependent; for example, Perry and Stieg [42] present a different set of fault costs. A third factor involves our implementations of techniques. In the case of BN techniques, the choice of parameters they utilize can affect their performance. Although these effects have been shown to be insignificant in many cases [37], using simpler and more general configurations can help reduce this threat. Finally, our BN and CC prioritization techniques were implemented by different programmers; however, in our study design we were careful to utilize identical tools for all common tasks (e.g., instrumentation and test execution) related to the prioritization and measurement processes.

Construct Validity. While our economic model is the most comprehensive model created to date for use in assessing regression testing techniques, and the only existing model suitable for assessing our research questions, the dependent measures that we have considered to capture costs and benefits relative to this model are not the only possible measures. Furthermore, other testing costs not captured by the model, such as the costs of initial test case development, initial automation, and test suite maintenance, might play important roles and influence overall costs and benefits in particular testing situations and organizations.

4.5 Data and Analysis

To provide an overview of the collected data, Figure 3 presents boxplots[3] that show cost-benefit results for all techniques, time constraints, and programs. The figure is composed of 20 subfigures. The four columns of subfigures present results for time constraint levels TCL-0, TCL-25, TCL-50, and TCL-75, respectively. The five rows present results for each of the object programs, respectively. To facilitate visual comparisons across constraint levels, cost-benefit scales are fixed per program (across rows). Due to wide differences in cost-benefit scales across different programs, however, we use different scales per program.

Each subfigure contains boxplots for six prioritization techniques (Table 3 presents a legend of the techniques) showing the distribution of cost-benefits in dollars for those techniques, for the given object program and constraint level. The horizontal axis corresponds to techniques, and the vertical axis corresponds to cost-benefits in dollars (recall that these are relative cost-benefits calculated as described in Section 4.2.2). Higher values indicate greater

[3] A boxplot is a standard statistical device for representing data sets [23]. 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. Small circles beyond these cutoffs represent anomalous data values.

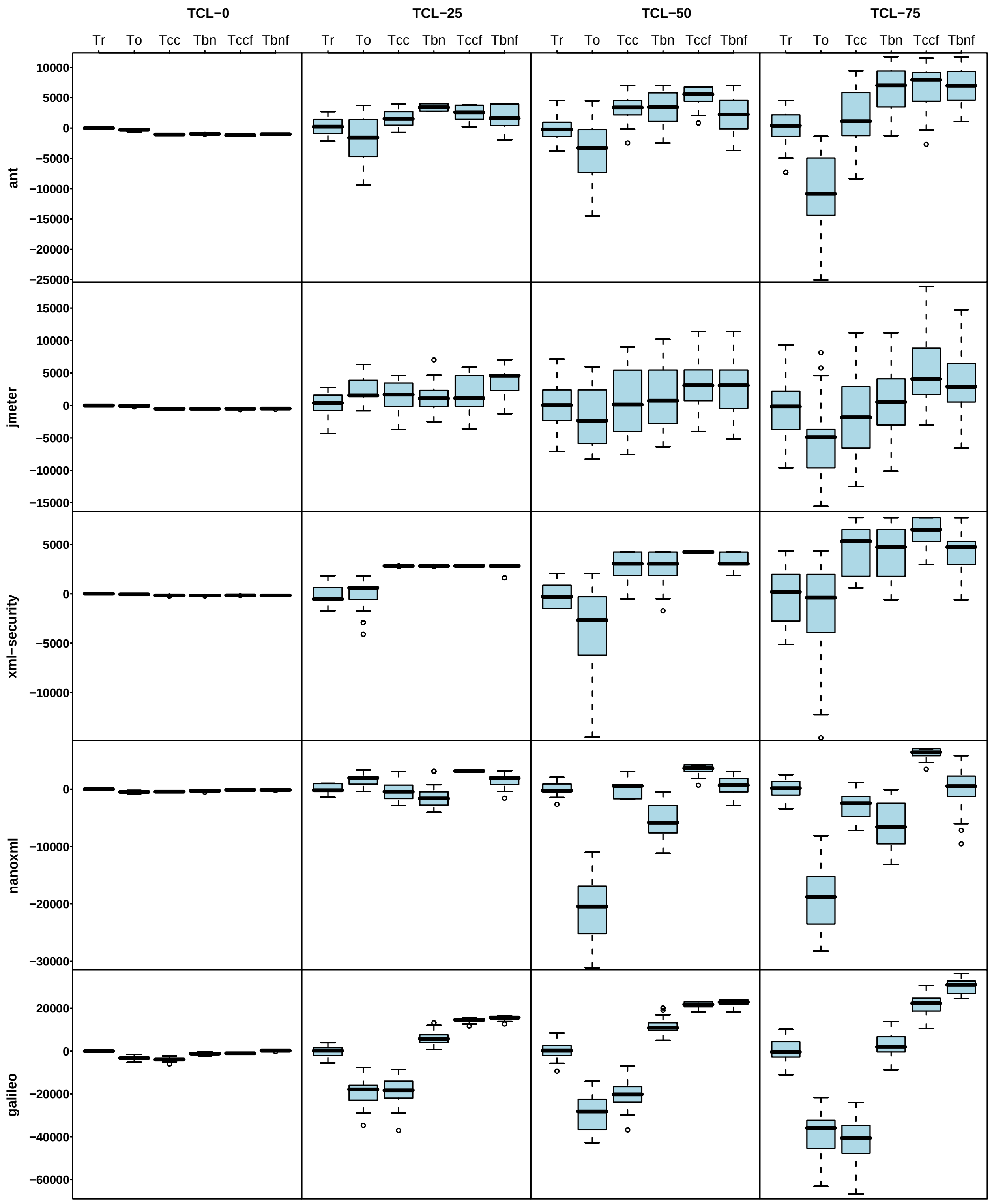

Figure 3: Experiment 1: Cost-benefit boxplots, all programs, techniques, and time constraints.

cost-benefits. The two leftmost boxplots (*Tr* and *To*) present data for the control techniques, and the rest present data for the four heuristics. Because we measured results (for each application of a technique on a given program and constraint level) across 30 sequences of mutant groups (see Section 4.3), the number of data points represented by each boxplot in each subfigure is 30.

We begin with a descriptive analysis of the data in the boxplots, considering the performance of the heuristics in comparison to the control techniques as time constraints vary. Examining the boxplots for each object program in the first column (TCL-0) of Figure 3, we see that none of the heuristics appear to be cost-beneficial compared to the original technique (*To*) for the first two object programs (*ant* and *jmeter*). On the other programs, differences between techniques in the first column are difficult to see, but our subsequent statistical analysis provides more details.

As the time constraint level changes, the relationship between techniques changes. Across all three levels of time constraints (TCL-25, TCL-50, and TCL-75), in all cases but one (*jmeter* on *Tccf* at TCL-25), feedback techniques appear to be more beneficial than control techniques. Cost-benefit gains appear to increase as time constraints increase. In the case of non-feedback techniques, results vary across programs. For example, on *ant* and *xml-security*, control techniques appear to be worse than non-feedback techniques, and as time constraints increase, the cost-benefit gap between control techniques and non-feedback techniques widens. On the other programs, there is no specific common trend visible between non-feedback and control techniques.

Next, to formally address each of our research questions, we wish to compare the effects that occur for given techniques as time constraints change, and then compare the effects that occur between techniques at each given level of time constraints. The following sections provide, for each of our research questions in turn, the statistical analyses and results relevant to that question. (We discuss further implications of the data and results in Section 4.6.)

For our statistical analysis, we followed a process well established in prior studies of test case prioritization (e.g., [12, 18, 48])): we used the Kruskal-Wallis non-parametric one-way analysis of variance followed by Bonferroni's test for multiple comparisons [45]. We used the Kruskal-Wallis test because our data did not meet the assumptions for using ANOVA: our data sets do not have equal variance, and some have severe outliers. For multiple comparisons, we used the Bonferroni method for its conservatism and generality. We used the Splus statistics package[4] to perform the analysis. Because results vary substantially across programs, we analyzed the data for each program separately.

4.5.1 RQ1: Effects of Time Constraints on Techniques

Our hypothesis associated with RQ1 is: *(H1) given a specific technique, the cost-benefits between time constraints differ*. To test this hypothesis, we performed the Kruskal-Wallis test (df = 3) for each technique per program, at a significance level of 0.05, over the four time constraint levels. Table 4 shows the results of this analysis for the four heuristics and *To*. Results for *Tr* are not meaningful in this context, since it is the baseline used in our relative cost-benefit calculation. Considering all techniques and programs, in all cases other than *Tbn* applied to *jmeter* (24 of the 25 cases), the hypothesis is supported.

[4]http://www.insightful.com/products/splus

Table 4: Kruskal-Wallis Test Results for RQ1

Program	To		Tcc		Tbn		$Tccf$		$Tbnf$	
	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.
ant	50	< 0.0001	44	< 0.0001	62	< 0.0001	84	< 0.0001	74	< 0.0001
jmeter	41	< 0.0001	10	0.018	7.4	0.057	34	< 0.0001	36	< 0.0001
xml-security	17	0.0005	66	< 0.0001	46	< 0.0001	104	< 0.0001	87	< 0.0001
nanoxml	100	< 0.0001	28	< 0.0001	59	< 0.0001	99	< 0.0001	27	< 0.0001
galileo	92	< 0.0001	97	< 0.0001	78	< 0.0001	97	< 0.0001	111	< 0.0001

Table 5: Bonferroni Test Results for RQ1: Comparing across Time Constraint Levels per Technique and Program

Technique	ant			$jmeter$			$xml-security$			$nanoxml$			$galileo$		
	TCL	Mean	Grp	TCL	Mean	Grp	TCL	Mean	Grp	TCL	Mean	Grp	TCL	Mean	Grp
To	0	-352	A	25	2243	A	0	-60	A	25	1861	A	0	-3335	A
	25	-1779	A	0	-56	A B	25	-61	A	0	-459	A	25	-19544	B
	50	-3768	A	50	-1978	B C	75	-1735	A B	75	-19064	B	50	-28598	C
	75	-11201	B	75	-5126	C	50	-3427	B	50	-20965	B	75	-38088	D
Tcc	50	2977	A	25	988	A	75	4456	A	50	189	A	0	-3984	A
	25	1578	A	50	164	A	50	2845	B	25	-249	A	25	-19135	B
	75	1310	A	0	-516	A	25	2812	B	0	-416	A	50	-19983	B
	0	-1088	B	75	-1644	A	0	-163	C	75	-2850	B	75	-41699	C
Tbn	75	6391	A	50	1428	A	75	3942	A	0	-304	A	50	11502	A
	25	3399	B	25	1193	A	50	2842	A	25	-1304	A	25	5772	B
	50	3248	B	75	841	A	25	2806	A	50	-5357	B	75	2684	C
	0	-980	C	0	-512	A	0	-171	B	75	-5922	B	0	-1166	D
Tccf	75	6743	A	75	4666	A	75	6150	A	75	6178	A	75	21880	A
	50	5106	A	50	2855	A B	50	4224	B	50	3406	B	50	21476	A
	25	2667	B	25	1684	B C	25	2819	C	25	3157	B	25	14365	B
	0	-1198	C	0	-497	C	0	-155	D	0	-127	C	0	-985	C
Tbnf	75	6983	A	25	3801	A	75	4454	A	50	1535	A	75	29886	A
	50	2002	B	75	3639	A	50	3434	B	75	877	A	50	22379	B
	25	1783	B	50	3010	A	25	2616	B	0	37	A	25	15458	C
	0	-1030	C	0	-487	B	0	-163	C	25	-143	A	0	150	D

We next performed multiple pair-wise comparisons for each technique other than the random technique using Bonferroni tests, which determine the significance in group mean differences in an analysis of variance test. Table 5 presents the results of these tests with a Bonferroni correction [45]. In the table, data is organized per technique (rows) and per program (columns), for each technique and program listing the four time constraint levels in terms of their mean cost-benefit values, from higher (better) to lower (worse). We use grouping letters (columns with headers "Grp") to partition the time constraints such that results that are not significantly different share the same grouping letter. For example, the results of *Tccf* for *jmeter* show that TCL-75 and TCL-50 are not statistically significantly different (sharing grouping letter A), and TCL-50 and TCL-25 are not statistically significantly different (sharing grouping letter B), but TCL-75 and TCL-25 *are* statistically significantly different (sharing no grouping letters).

As the table shows, the results from multiple pair-wise comparisons reveal different trends between time constraints among techniques and programs. In the case of the original technique (*To*), the results show that in all cases but two (*jmeter* and *nanoxml* at TCL-25) negative cost-benefit values were observed, which indicates that in most cases the original technique was worse than the (random) baseline. Overall, cost-benefit values decreased as time constraint levels increased (from TCL-0 to TCL-75); this was particularly evident in the case of *galileo*, on which there were statistically significant differences between all but two pairs of time constraints.

Table 6: Kruskal-Wallis Test Results for RQ2

Program	*TCL-0*		*TCL-25*		*TCL-50*		*TCL-75*	
	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.
ant	162	< 0.0001	92	< 0.0001	87	< 0.0001	109	< 0.0001
jmeter	122	< 0.0001	34	< 0.0001	24	0.0002	45	< 0.0001
xml-security	125	< 0.0001	123	< 0.0001	113	< 0.0001	80	< 0.0001
nanoxml	152	< 0.0001	123	< 0.0001	148	< 0.0001	146	< 0.0001
galileo	160	< 0.0001	164	< 0.0001	165	< 0.0001	163	< 0.0001

The trends change, however, when we consider heuristics. In the case of non-feedback techniques (*Tcc* and *Tbn*), negative cost-benefit values were observed in all cases in which no time constraints applied (TCL-0). When time constraints applied, techniques produced positive cost-benefit values on the three object programs that have JUnit test cases (*ant*, *jmeter*, and *xml-security*) in all but one case (*Tcc* on *jmeter* at TCL-75), with values usually trending upwards as time constraints increase. On the two programs that have TSL test cases, *nanoxml* and *galileo*, however, results and trends were mixed.

In the case of feedback techniques, the positive effects of prioritization, together with upward trends as time constraint levels increase, are more obvious. Cost-benefit values increased as time constraints increased in all cases but two (*jmeter* and *nanoxml* for *Tbnf*), and in these two cases, differences between time constraint levels were not significant. Further, even in cases in which non-feedback techniques were not cost-beneficial (*nanoxml* and *galileo*), feedback techniques produced positive cost-benefit values.

4.5.2 RQ2: Effects Among Techniques at Given Levels of Time Constraints

Our hypothesis associated with RQ2 is: *(H2) given a specific time constraint, the cost-benefits between test case prioritization techniques differ.* To test this hypothesis, we performed the Kruskal-Wallis test (df = 5) for each of the four time constraint levels per program, at a significance level of 0.05, over the six techniques. Table 6 shows the results of this analysis for the four time constraint levels. The hypothesis is supported in all 20 cases.

We next performed multiple pair-wise comparisons for each time constraint level using the Bonferroni tests. Table 7 presents the results of the Bonferroni tests with a Bonferroni correction. In the table, data is organized per time constraint level (rows) and program (columns), for each listing the six techniques in terms of their mean cost-benefit values, from higher (better) to lower (worse). Again, grouping letters indicate statistically significant differences.

As the table shows, the results from multiple pair-wise comparisons show different trends between techniques among time constraint levels and programs. In the case in which no time constraints applied (TCL-0), all control techniques were better than heuristics for the three object programs (*ant*, *jmeter*, and *xml-security*) that have JUnit test cases. Results for *nanoxml* and *galileo*, however, differ, with the random technique significantly superior to heuristics in all but one case (*Tbnf* on *galileo*), and the original technique significantly inferior to all but one heuristic (*Tcc*).

In the cases in which time constraints applied (TCL-25, TCL-50, and TCL-75), however, the relationships between control techniques and heuristics differ. At TCL-25, heuristics often outperformed the control techniques, but the

Table 7: Bonferroni Test Results for RQ2: Comparing across Techniques per Time Constraint Level and Program

TCL	*ant*			*jmeter*			*xml − security*			*nanoxml*			*galileo*		
	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp
0	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tbnf	150	A
	To	-352	B	To	-56	B	To	-60	B	Tccf	-127	B	Tr	0.0	A
	Tbn	-980	C	Tbnf	-487	C	Tccf	-155	C	Tbnf	-143	B	Tccf	-985	B
	Tbnf	-1030	C D	Tccf	-497	C	Tcc	-163	C	Tbn	-304	C	Tbn	-1166	B
	Tcc	-1080	D	Tbn	-512	C	Tbnf	-164	C	Tcc	-416	D	To	-3335	C
	Tccf	-1198	E	Tcc	-516	C	Tbn	-171	C	To	-459	D	Tcc	-3984	D
25	Tbn	3399	A	Tbnf	3801	A	Tccf	2819	A	Tccf	3157	A	Tbnf	15358	A
	Tccf	2667	A	To	2243	A B	Tcc	2812	A	To	1861	B	Tccf	14365	A
	Tbnf	1783	A B	Tccf	1684	A B C	Tbn	2806	A	Tbnf	1535	B	Tbn	5772	B
	Tcc	1578	A B	Tbn	1193	B C	Tbnf	2616	A	Tr	0.0	C	Tr	0.0	C
	Tr	0.0	B C	Tcc	988	B C	Tr	0.0	B	Tcc	-249	C D	Tcc	-19135	D
	To	-1779	C	Tr	0.0	C	To	-61	B	Tbn	-1304	D	To	-19544	D
50	Tccf	5106	A	Tbnf	3010	A	Tccf	4224	A	Tccf	3406	A	Tbnf	22379	A
	Tbn	3248	A B	Tccf	2855	A	Tbnf	3434	A	Tbnf	877	A B	Tccf	21476	A
	Tcc	2977	A B	Tbn	1428	A B	Tcc	2845	A	Tcc	189	B	Tbn	11502	B
	Tbnf	2002	B C	Tcc	164	A B	Tbn	2842	A	Tr	0.0	B	Tr	0.0	C
	Tr	0.0	C	Tr	0.0	A B	Tr	0.0	B	Tbn	-5357	C	Tcc	-19983	D
	To	-3768	D	To	-1978	B	To	-3427	C	To	-20965	D	To	-28598	E
75	Tbnf	6983	A	Tccf	4666	A	Tccf	6150	A	Tccf	6178	A	Tbnf	29886	A
	Tccf	6743	A	Tbnf	3639	A	Tcc	4456	A	Tbnf	37	B	Tccf	21880	B
	Tbn	6391	A	Tbn	841	A B	Tbnf	4454	A	Tr	0.0	B	Tbn	26884	C
	Tcc	1310	B	Tr	0.0	A B C	Tbn	3942	A	Tcc	-2850	B C	Tr	0.0	C
	Tr	0.0	B	Tcc	-1644	B C	Tr	0.0	B	Tbn	-5922	C	To	-38088	D
	To	-11201	C	To	-5126	C	To	-1735	B	To	-19064	D	Tcc	-41699	D

results varied across programs. For example, all heuristics were better than both control techniques on *xml-security*, and all were better than the original technique on *ant*, while on the other three programs feedback techniques were usually but not always better than control techniques, and non-feedback techniques were less consistent.

Further, the relationship between heuristics differed when time constraints applied for *ant* and *jmeter*. In the case of *ant*, *Tbn* exhibited a higher mean cost-benefit value than *Tbnf* at TCL-25 and TCL-50, but this relationship was reversed at TCL-75. In the case of *jmeter*, there were no statistically significant differences between heuristics without time constraints, but when time constraints applied, heuristics maintained stable rankings in terms of mean cost-benefit values, but fell into two or three different strata as time constraints varied in terms of statistically significant differences.

As time constraints increased further (TCL-50 and TCL-75), for *ant* and *xml-security*, heuristics continued to perform better than the control techniques (ranking-wise); in particular, differences between all heuristics and the control techniques for *xml-security* and differences between all heuristics and the original technique for *ant* were statistically significant. The other three programs yielded similar results, with a few exceptions; heuristics were always better (ranking-wise) than the original technique for both time constraint levels, and often better than the random technique.

4.6 Discussion

We now draw on the results of our analyses, together with additional consideration of our data, to derive several implications of these results. Of course, in assessing these implications the reader should keep in mind the threats to validity for this study.

Figure 4 presents lineplots of the mean cost-benefit values observed in our study for each prioritization technique, at each time constraint level, for each program. These lineplots summarize the major trends in our results visually,

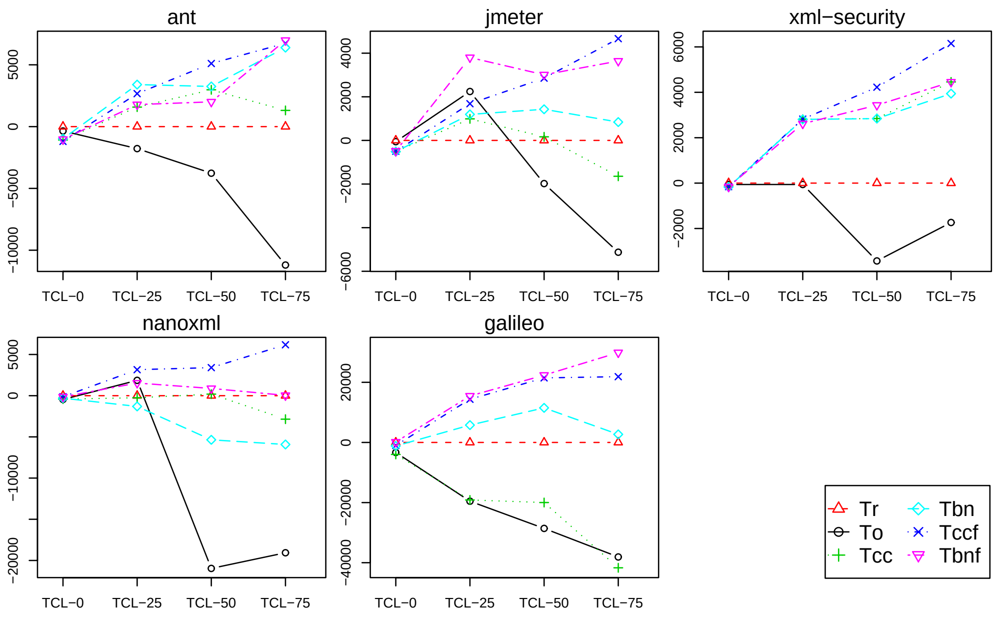

Figure 4: Experiment 1: Cost-benefit lineplots, all programs, all techniques, all time constraints.

and together with the formal analysis of Section 4.5, facilitate our discussion of results. They also provide a way to consider the further question: in what ways do the effects of time constraints vary across different techniques?

4.6.1 Time Constraints Matter

Our analysis of results showed that our hypotheses (H1 and H2) are both supported in a vast majority of cases: for each given prioritization technique, the cost-benefits between time constraints differed; for each given time constraint level L, the cost-benefits between techniques differed. Further, as we can observe from Figure 4, the effects of time constraints on differences between technique cost-benefits increased as time constraint level increased.

In this study we also observed that *when no time constraints applied, we gained little from employing prioritization heuristics*. This result was surprising, as it is quite different from the results of prior empirical studies of prioritization (e.g., [18, 26, 31]), which have concluded that heuristics are more effective than control techniques. We believe that this difference is due to the fact that in prior work, prioritization benefits have been assessed in terms of simple measures of rate of fault detection. The results of this study suggest, in fact, that using simple rate of fault detection measures to assess prioritization techniques may lead to inaccurate observations, and that more comprehensive economic models can lead to quite different conclusions about the cost-effectiveness of heuristics. Such results must be qualified, however, in light of the particular object programs and cost factors studied; we return to this issue in our third experiment.

4.6.2 The Worst Thing One Can Do is Not Prioritize

We observed that the original test case order was almost always worse than the test case order produced by prioritization heuristics. Even at TCL-0, the original test case order was inferior to those produced by heuristics on two of the five programs (*nanoxml* and *galileo*). Moreover, the original order became increasingly worse as time constraint levels increased: considering the 15 observed points of TCL increases (from TCL-0 to TCL-25, TCL-25 to TCL-50, and TCL-50 to TCL-75 on each of the five programs), in 11 of these 15 cases (five statistically significant) the cost-benefits of the original order decreased as TCL increased. Figure 4 clearly shows this trend: the original orders' lineplots slope downwards, while others more often slope upwards. Further, our results show that even the use of a randomized test case order, which can be thought of as the result of using randomization as a simple prioritization strategy, is preferable (at least, in terms of average-case behavior) to using "original" orders.

Thus, these results do show that *when time constraints might apply*, as for example when engineers do not know how long they will be allowed to keep running test cases, *the worst thing that one can do is to not practice some form of test case prioritization*. Furthermore, we expect this implication to hold even more strongly in cases in which fault costs are greater, and we return to this issue in our third experiment.

4.6.3 Time Constraints Affect Techniques Differently

For prioritization heuristics, the cost-benefit values we observed tended to increase as time constraint levels increased, but this trend varied across techniques. The following lists observations for each heuristic relative to the 15 observed points of TCL increases:

- *Tccf* always produced greater cost-benefits as TCL increased (15 increases, 9 of them statistically significant).
- *Tbnf* often produced greater cost-benefits as TCL increased (12 increases, 8 of them statistically significant).
- *Tbn* was less stable in producing cost-benefits as TCL increased (9 increases, 5 of them statistically significant).
- *Tcc* was least stable (8 increases, 3 statistically significant).

Feedback techniques were more effective than their non-feedback counterparts, not only in terms of producing greater cost-benefits, but also in terms of being consistently better as time constraint levels increased. Non-feedback techniques were also guilty of performing quite poorly; in particular, on *galileo* the performance of *Tcc* was as bad as that of the original order. In other words, *feedback techniques are more stable than non-feedback techniques in the presence of variations in time constraints*. Such differences in stability between feedback and non-feedback techniques have been observed previously [18] in relation to non-time-constrained evaluations, and attributed to relationships between test execution patterns and the locations of faults (non-feedback techniques vary more widely when test cases that expose faults execute relatively few functions); a similar pattern appears to hold in this case.

Further inspection of our data and cost factors also suggests the existence of interaction effects between prioritization technique execution time and rate of fault detection. In general, in the absence of time constraints, techniques that had lower execution costs (CR) tended to perform better than those with higher execution costs. As time constraints increased, however, techniques yielding earlier fault-detection became more cost-effective, irrespective of execution cost. Following up on these observations, we determined (to our surprise) that BN techniques tended to have lower costs on average than CC techniques. One plausible explanation for this is that BN techniques use class-level coverage information, whereas CC techniques use block-level information. For example, in the case of *ant*, while the number of instrumentation points for the class-level coverage information is 627, the number of instrumentation points for the block-level coverage information is more than 6000. The use of finer-grained coverage data leads to longer technique execution times, but on the other hand, gives CC techniques (and especially Tccf) an edge in terms of early fault detection.

5 Experiment 2: Reducing Validity Threats Through Replication

The results of Experiment 1 suggest that time constraints can indeed play a significant role in determining both the cost-effectiveness of prioritization techniques, and the relative cost-benefit tradeoffs among techniques. However, as discussed in Section 4.4, the use of fixed numbers of faults applied uniformly to all programs is a threat to external validity for those results. Furthermore, threats to internal validity include choices of values and measures for use in the EVOMO model and choices of parameters for use by BN techniques.

We wished to address these threats to validity, and determine whether our results generalize to cases in which faults occur in more realistic numbers, when model and technique settings are simplified. We thus replicated the first study in a context in which these threats were addressed.

For this experiment, we consider the same research questions as those considered in Experiment 1, and for completeness we repeat these here, but we designate them RQ1′ and RQ2′ in recognition of the different experimental context being conducted.

RQ1′: Given a specific test case prioritization technique, as time constraints vary, in what ways is the performance of that technique affected?

RQ2′: Given a specific time constraint on regression testing, in what ways do the performances of test case prioritization techniques differ under that constraint?

This experiment utilizes the same object programs, variables, and measures as those used in Experiment 1. It also possesses the same threats to validity as Experiment 1, with the exception of those specifically addressed (the effects of fault numbers, values and measures required by EVOMO, and parameters used by BN techniques.) We thus do not repeat discussion of these here. Instead, we describe only the differences between this experiment and the prior one, namely, the simplified EVOMO model, the improved BN techniques, and the alterations made to experiment setup and design. We then present data and analysis and discussion of results.

5.1 The Simplified EVOMO Model: S-EVOMO

While our initial EVOMO model presented in Section 3 has allowed us to assess the cost-benefit tradeoffs for prioritization techniques [10, 11], it does involve several variables that must be estimated, and a simplified model can reduce threats to validity related to these estimates. Model simplification can also render the process of collecting or estimating model data less expensive.

Reference [13] describes our approach to model simplification. Using sensitivity analysis, we identified the four cost component factors that had the smallest influence on the output of the model: a_{in}, *CV*, *CS*, and CA_{in} (Table 1 describes these factors). We then fixed these factors at given values over their range of uncertainty [50]. Through this process, we obtained a simplified version of our original full model, and then we empirically evaluated whether the simplified model possessed the same ability as the original to assess cost-benefit relationships between regression testing techniques. The results showed that our simplified model assessed the relationship between techniques in the same way as the full model.

The simplified EVOMO model (hereafter referred to as S-EVOMO), like the original, involves two equations: one that captures costs related to the salaries of the engineers and one that captures revenue gains or losses related to changes in system release time. The simplified model also continues to account for costs and benefits across entire system lifetimes, and for the use of incremental analysis techniques.

The two equations that comprise S-EVOMO are as follows; terms and coefficients retain the meanings presented originally in Table 1. As just stated, S-EVOMO fixes the four least significant factors, *CS*, *CV*, CA_{in}, and a_{in}; in the equations we represent the value of these fixed factors collectively as constants K_1 and K_2.

$$Cost = PS * (\sum_{i=2}^{n}(CO_i(i) + CO_r(i) + c(i) * CF(i)) + K_1) \quad (4)$$

$$Benefit = REV * (\sum_{i=2}^{n}(ED(i) - (CO_i(i) + CO_r(i) + a_{tr}(i-1) * CA_{tr}(i-1) + CR(i) + b(i) * CE(i) + CD(i))) - K_2) \quad (5)$$

5.2 Simplified BN Techniques

We use a different implementation of the BN technique that employs a simpler set of parameters and information gathering techniques. BN techniques have parameters that can be configured to adjust the technique to a particular environment. Also, their input data can be gathered using different tools and algorithms. In Experiment 1, these configurations were chosen based on an empirical study [37] in which parameters were carefully selected for each object and a complex algorithm for gathering change information was used. Although such complex configurations can in some cases increase the performance of BN techniques, it is not clear how well such improvements generalize in practice. Simpler configurations can reduce the costs of applying the technique and can therefore potentially increase its overall cost-effectiveness. Furthermore, recent studies [35, 37] of BN techniques have suggested that the impact of certain parameters can be statistically insignificant. When a simple implementation can produce results as good as a more complex one, the costs of the complex one can be avoided.

Thus, in this experiment, we use a simpler configuration for BN techniques. The major simplifications are three: (1) the simple Unix DIFF command is used to gather change information, (2) a simple rule of thumb (based on the size of the object) is used to choose a Bayesian inference algorithm, and (3) the same "level of feedback" (a parameter in BN techniques, controlling how often feedback happens) has been used across all programs. More details can be found in [35].

5.3 Experiment Setup

This experiment uses the same setup as Experiment 1 (see Section 4.3), but in addition to the steps detailed for that experiment, we also needed to provide mutant groups considering more realistic numbers of faults.

To do this, we utilized a fault prediction model developed by Bell et al. [3]: the LOC model. The LOC model uses a negative binomial regression model to predict the number of faults in a system based on the number of lines of code in a file [3]. Because our focus is regression testing and detection of regression faults, in applying the model we considered only files that have been changed from the previous version. Using the LOC model, we obtained various ranges of numbers of faults across the different versions of our object programs; these ranges were: *ant* (3-39), *jmeter* (12-26), *xml-security* (5-10), *nanoxml* (1-5), and *galileo* (1-8). Based on these numbers, for each version of each program we randomly selected several *mutant groups* of sizes falling within those ranges from the set of that version's mutation faults. We then gathered prioritization data relative to those mutant groups, following the procedure detailed in Section 4.3.

5.4 Data and Analysis

Figure 5 presents boxplots that show cost-benefit results for all techniques, time constraints, and programs. The figure is structured similar to Figure 3 (see Section 4.5 for details on how to read the figure.)

Examining the boxplots for each object program, we have observed strong similarities with the results from Experiment 1, as follows:

- Under no time constraints, none of the heuristics appear to be cost-beneficial compared to the original technique (*To*) for the first two object programs (*jmeter* and *xml-security*). For *ant* and *galileo*, overall, the heuristics appear to perform slightly better than the original technique.
- As the time constraint level changes, the relationships between techniques change. Across all three time constraints (TCL-25, TCL-50, and TCL-75), in all cases but two (*jmeter* on *Tccf* at TCL-25 and *ant* on *Tbnf* at TCL-50), feedback techniques appear to be more cost-beneficial than the control techniques.
- Cost-benefit gains appear to increase as time constraints increase, in particular for feedback techniques. This trend seems to be consistent in all but two cases. In the case of non-feedback techniques, results vary across programs.

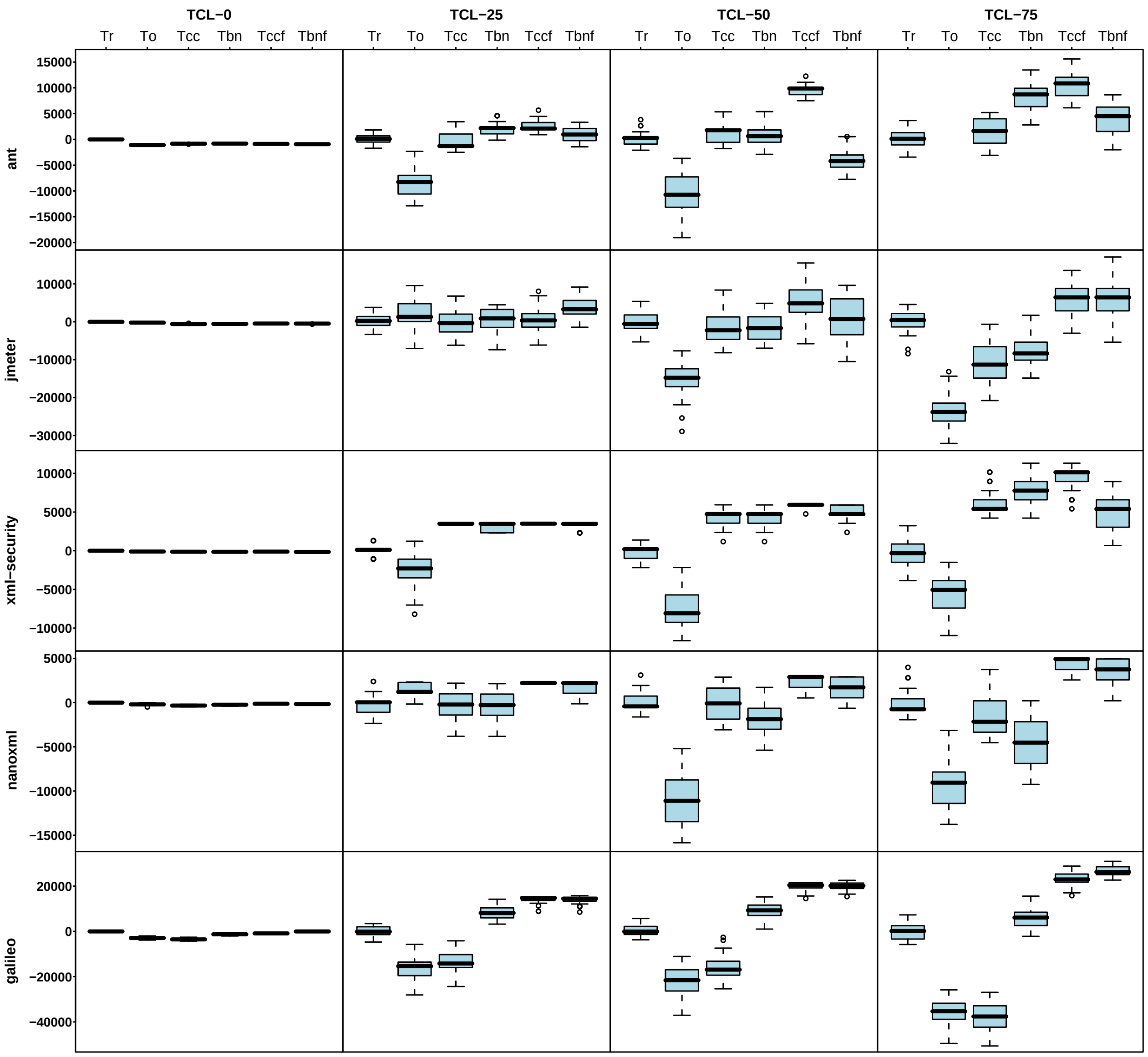

Figure 5: Experiment 2: Cost-benefit boxplots, all programs, techniques, and time constraints.

Table 8: Kruskal-Wallis Test Results for RQ1′

Program	To		Tcc		Tbn		$Tccf$		$Tbnf$	
	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.
ant	102	< 0.0001	28	< 0.0001	100	< 0.0001	93	< 0.0001	92	< 0.0001
jmeter	99	< 0.0001	60	< 0.0001	45	< 0.0001	49	< 0.0001	46	< 0.0001
xml-security	78	< 0.0001	98	< 0.0001	110	< 0.0001	102	< 0.0001	84	< 0.0001
nanoxml	100	< 0.0001	7.8	0.049	98	< 0.0001	55	< 0.0001	72	< 0.0001
galileo	99	< 0.0001	98	< 0.0001	105	< 0.0001	69	< 0.0001	111	< 0.0001

Table 9: Bonferroni Test Results for RQ1′: Comparing across Time Constraint Levels per Technique and Program

Technique	ant			$jmeter$			$xml - security$			$nanoxml$			$galileo$		
	TCL	Mean	Grp	TCL	Mean	Grp	TCL	Mean	Grp	TCL	Mean	Grp	TCL	Mean	Grp
To	0	-1079	A	25	1971	A	0	-106	A	25	1516	A	0	-2857	A
	25	-8161	B	0	-213	A	25	-2449	B	0	-210	B	25	-16083	B
	50	-10639	C	50	-15277	B	75	-5482	C	75	-9238	C	50	-21953	C
	75	-49284	C	75	-23978	C	50	-7215	D	50	-11067	D	75	-35928	D
Tcc	50	1287	A	25	-502	A	75	6234	A	50	-93	A	0	-3477	A
	75	1247	A	0	-570	A	50	4351	B	25	-178	A	25	-13887	B
	25	-467	B	50	-1276	A	25	3492	B	0	-339	A	50	-15812	B
	0	-828	B	75	-11338	B	0	-134	C	75	-1294	A	75	-38135	C
Tbn	75	8570	A	25	234	A	75	7808	A	0	-271	A	50	9124	A
	25	2074	B	0	-546	A	50	4110	B	25	-561	A B	25	7852	A B
	50	647	B C	50	-1489	A	25	3134	C	50	-1880	B	75	6249	B
	0	-810	C	75	-7626	B	0	-141	D	75	-4287	C	0	-1328	C
Tccf	75	10898	A	75	6133	A	75	9465	A	75	4379	A	75	23284	A
	50	9559	B	50	4951	A	50	5888	B	50	2348	B	50	20074	B
	25	2448	C	25	604	B	25	3502	C	25	2208	B	25	13819	C
	0	-884	D	0	-462	B	0	-119	D	0	-143	C	0	-877	D
Tbnf	75	4024	A	75	6053	A	75	5205	A	75	3127	A	75	26370	A
	25	864	B	25	3479	A B	50	4891	A	50	1841	B	50	19932	B
	0	-929	B	50	934	B C	25	3326	B	25	1750	B	25	13765	C
	50	-4234	C	0	-472	C	0	-153	C	0	-159	C	0	-47	D

The following sections provide, for each of our research questions in turn, the statistical analyses and results relevant to that question. (Our analyses employ the same statistical procedures as those used in Experiment 1).

5.4.1 RQ1′: Effects of Time Constraints on Techniques

Our hypothesis associated with RQ1′ is: *(H1′) given a specific technique, the cost-benefits between time constraints differ*. Similar to the results from Experiment 1, the Kruskal-Wallis test (Table 8, df = 3, a significance level of 0.05) shows that the cost-benefits between time constraints differ – and here this difference holds in all 25 cases.

For multiple pair-wise comparisons, as Table 9 shows, the results reveal different trends between time constraints among techniques and object programs, as observed in Experiment 1.

- In most cases the original technique was worse than the (random) baseline (all cases but two – *jmeter* and *nanoxml* at TCL-25). Overall, cost-benefit values decreased as time constraint levels increased (from TCL-0 to TCL-75).
- In the case of feedback techniques (*Tccf* and *Tbnf*), positive effects of prioritization, together with upward trends

Table 10: Kruskal-Wallis Test Results for RQ2′

Program	*TCL-0*		*TCL-25*		*TCL-50*		*TCL-75*	
	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.
ant	154	< 0.0001	127	< 0.0001	150	< 0.0001	152	< 0.0001
jmeter	147	< 0.0001	33	< 0.0001	98	0.0002	150	< 0.0001
xml-security	119	< 0.0001	145	< 0.0001	147	< 0.0001	143	< 0.0001
nanoxml	132	< 0.0001	97	< 0.0001	135	< 0.0001	148	< 0.0001
galileo	165	< 0.0001	162	< 0.0001	164	< 0.0001	163	< 0.0001

Table 11: Bonferroni Test Results for RQ2′: Comparing across Techniques per Time Constraint Level and Program

TCL	ant			$jmeter$			$xml - security$			$nanoxml$			$galileo$		
	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp
0	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A
	Tbn	-810	B	To	-213	B	To	-106	B	Tccf	-143	B	Tbnf	-47	A
	Tcc	-828	B	Tccf	-462	C	Tccf	-119	B C	Tbnf	-159	B C	Tccf	-877	B
	Tccf	-884	C	Tbnf	-472	C	Tcc	-134	C D	To	-210	C	Tbn	-1328	C
	Tbnf	-929	D	Tbn	-546	D	Tbn	-141	D E	Tbn	-271	D	To	-2857	D
	To	-1079	E	Tcc	-570	D	Tbnf	-153	E	Tcc	-339	E	Tcc	-3477	E
25	Tccf	3399	A	Tbnf	3479	A	Tccf	3502	A	Tccf	2208	A	Tccf	13819	A
	Tbn	2667	A B	To	1971	A B	Tcc	3492	A	Tbnf	1750	A	Tbnf	13765	A
	Tbnf	1783	B C	Tccf	604	B	Tbnf	3326	A	To	1516	A	Tbn	7852	B
	Tr	1578	C	Tbn	234	B	Tbn	3134	A	Tr	0.0	B	Tr	0.0	C
	Tcc	0.0	C	Tr	0.0	B	Tr	0.0	B	Tcc	-178	B	Tcc	-13887	D
	To	-1779	D	Tcc	-502	B	To	-2449	C	Tbn	-561	B	To	-16083	D
50	Tccf	9559	A	Tccf	4951	A	Tccf	5888	A	Tccf	2348	A	Tccf	20074	A
	Tcc	1287	B	Tbnf	934	B	Tbnf	4891	A B	Tbnf	1841	A	Tbnf	19932	A
	Tbn	647	B	Tr	0.0	B	Tcc	4351	B	Tr	0.0	B	Tbn	9124	B
	Tr	0.0	B	Tcc	-1276	B	Tbn	4110	B	Tcc	-93	B	Tr	0.0	C
	Tbnf	-4234	C	Tbn	-1489	B	Tr	0.0	C	Tbn	-1880	C	Tcc	-15812	D
	To	-10639	D	To	-15277	C	To	-7215	D	To	-11067	D	To	-21953	E
75	Tccf	10898	A	Tccf	6133	A	Tccf	9465	A	Tccf	4379	A	Tbnf	26370	A
	Tbn	8570	A	Tbnf	6053	A	Tbn	7808	A B	Tbnf	3127	A	Tccf	23284	A
	Tbnf	4024	B	Tr	0.0	B	Tcc	6234	B C	Tr	0.0	B	Tbn	6249	B
	Tcc	1247	C	Tbn	-7626	C	Tbnf	5205	C	Tcc	-1294	B	Tr	0.0	C
	Tr	0.0	C	Tcc	-11338	C	Tr	0.0	D	Tbn	-4287	C	To	-35928	D
	To	-49284	D	To	-23978	D	To	-5482	E	To	-9238	D	Tcc	-38135	D

as time constraint levels increase, are observed. All cases but two (*ant* and *jmeter* for *Tbnf*) showed that cost-benefit values increased as time constraints increased.

- In the case of non-feedback techniques (*Tcc* and *Tbn*), the trend was mixed. When no time constraints applied, negative cost-benefit values were observed in all cases. When time constraints applied, techniques produced positive cost-benefit values on four cases (*ant*, *xml-security*, and *galileo* on *Tbn*, and *xml-security* on *Tcc*).

5.4.2 RQ2′: Effects Among Techniques at Given Levels of Time Constraints

Our hypothesis associated with RQ2′ is: *(H2′) given a specific time constraint, the cost-benefits between test case prioritization techniques differ.* Similar to the results from Experiment 1, the Kruskal-Wallis test (Table 10, df = 5, for significance level of 0.05) shows that the cost-benefits between test case prioritization techniques differ in all 20 cases.

For multiple pair-wise comparisons, as Table 11 shows, the results reveal different trends between techniques among time constraint levels and object programs, as we observed in Experiment 1.

- When no time constraints applied (TCL-0), the random technique was better than heuristics in all cases. In the case of the original technique, the results varied across programs: for *jmeter* and *xml-security*, all heuristics were inferior to the original technique; for other programs, the results were mixed.
- When time constraints applied (TCL-25, TCL-50, and TCL-75), the overall trends were mixed. In the case of *xml-security*, across all time constraint levels, all heuristics are significantly better than both control techniques. In the case of *galileo*, across all time constraint levels, all heuristics but *Tcc* are significantly better than both control techniques. In the case of *ant*, the heuristics were not always better than the control techniques. In the case of *nanoxml*, feedback techniques were better than the random technique and non-feedback techniques were worse than the random techniques.

5.5 Discussion

As just outlined, Experiments 1 and 2 yield primarily consistent results, through which we are able to reduce threats to validity related to Experiment 1. Thus, we discuss the results of our analyses in light of our earlier discussion of the results of Experiment 1.

5.5.1 Time Constraints Matter

We confirm the findings of Experiment 1 that time constraints affect the cost-benefits between techniques, and that when no time constraints applied, heuristics provided little benefit. Whether we consider fixed numbers of faults or fault numbers obtained through the LOC model, we reach the same conclusion, and the use of the simplified model and BN techniques did not affect this.

5.5.2 The Worst Thing One Can Do is Not Prioritize

This conclusion from Experiment 1 also holds: the original test case order was almost always worse than the test case order produced by prioritization heuristics, and as time constraint levels increased, the original order became worse.

5.5.3 Time Constraints Affect Techniques Differently

The following lists observations for each heuristic relative to the 15 observed points of TCL increases:

- *Tccf* always produced greater cost-benefits as TCL increased (15 increases, 14 statistically significant).
- *Tbnf* often produced greater cost-benefits as TCL increased (13 increases, 10 statistically significant).
- *Tbn* was less stable in producing cost-benefits as TCL increased (8 increases, 8 statistically significant).
- *Tcc* was least stable (8 increases, 4 statistically significant).

While the trends in this result compared to those observed in Experiment 1 remained the same, we observed that in this case, results are stronger. There are more cases here in which the cost-benefits of prioritization heuristics increased as TCL increased, and more cases in which cost-benefit increases yielded by heuristics are statistically significant, than in the first experiment. In particular, *Tccf* achieved statistically significant gains with respect to other heuristics in almost all cases (14 out 15), while there were only nine such cases in Experiment 1. The more realistic numbers of faults, and the simplified cost model and BN techniques, appear to augment the ability of heuristics to provide benefit as time constraints increase.

Through this experiment, then, we have gained several things with respect to prior findings. By considering different and more realistic numbers of faults, we are able to address an external threat to validity for Experiment 1. By using S-EVOMO and simplified BN, we were able to reduce internal threats to validity found in Experiment 1. Overall, by confirming the consistency of our results across the two experiments, we increase our confidence in the accuracy of those results.

6 Experiment 3

One interesting finding of our first experiment was that in the absence of time-constraints, heuristics were usually not beneficial; that is, they produced negative benefits compared to random orderings of test cases. We noted that this finding should be interpreted with the threats to validity of the experiment in mind. One important threat to validity concerned the numbers of faults present in the system under test. Experiment 2 showed, however, that when we utilized a more realistic "expected" number of faults this finding still held.

These results run counter to the fact that prior studies of prioritization have found heuristics effective in the absence of time constraints. We speculate that the use of a more comprehensive cost model, and the consequent factoring in of costs related to technique execution, is responsible for this difference. If this is true, then differences in the numbers of faults present in programs (and consequently, in costs related to early detection and omission of faults), may cause these results to vary, by counterbalancing the costs related to technique execution.

To examine this issue and its implications further, we designed and performed a controlled experiment considering the following research question:

RQ3: Given a specific faultiness level, in what ways do the performances of test case prioritization techniques differ under that level?

In this experiment, we again use the five Java systems described in Section 4 (Table 2), together with their versions, tests, and faults, as objects of analysis. We also use the same dependent variable as Experiment 2 which is based on the S-EVOMO model described in 5.1. Our threats to validity remain the same as those for Experiment 2. We thus do not repeat discussion of these here. Instead, we describe only the differences in this experiment, which are restricted to independent variables and experiment setup.

6.1 Independent Variables

Our experiment manipulated two independent variables: prioritization technique and faultiness level.

Variable 1: Prioritization Technique

We again use the prioritization techniques described in Section 4.2.1, with the adjustments to the BN techniques described in Section 5.2.

Variable 2: Faultiness Level

To investigate the impact of the numbers of faults present in a system on the cost-effectiveness of prioritization techniques, we utilize a variable, "faultiness level", that manipulates numbers of faults placed in systems. We consider three different faultiness levels yielding different numbers of faults (mutants) randomly chosen for inclusion in each version of each object program under test. The first level, FL1, involves cases in which mutant groups contain between 1 and 5 faults, the second level, FL2, involves cases in which mutant groups contain between 6 and 10 faults, and the third level, FL3, involves cases in which mutant groups contain between 11 and 15 faults. As in Experiment 1, mutants are randomly selected from each version's pool of mutation faults.

6.2 Experiment Setup

This experiment used the same setup as Experiment 1 (see Section 4.3), but in addition, we repeated the mutant grouping procedure used there for each of the three different faultiness levels considered.

6.3 Data and Analysis

To provide an overview of the collected data, we present boxplots in Figure 6, showing the relative cost-benefit results for different faultiness levels. The three columns of the graph present results for faultiness level 1 (FL1), faultiness level 2 (FL2), and faultiness level 3 (FL3), respectively.

We begin with descriptive analysis of the data in the boxplots, considering the performance of the heuristics in comparison to the control technique at each faultiness level. Examining the boxplots for each object program in the first column (FL1) of Figure 6, we see that none of the heuristics appear to outperform the control techniques (*To* and *Tr*) for the first three object programs, which have JUnit test suites. On the last two programs, which have TSL test suites, the techniques using feedback appear to be slightly better than the original technique (*To*) but no better than the random technique (*Tr*).

As faultiness level changes, however, this trend changes. For *jmeter*, the control techniques still appear to outperform the heuristics as faultiness level increases, but the gap between control techniques and heuristics becomes smaller. For *ant* and *xml-security* a similar pattern can be observed but at FL3, heuristics begin to outperform some of the control techniques (original in the case of *ant* and random in the case of *xml-security*). On the last two programs,

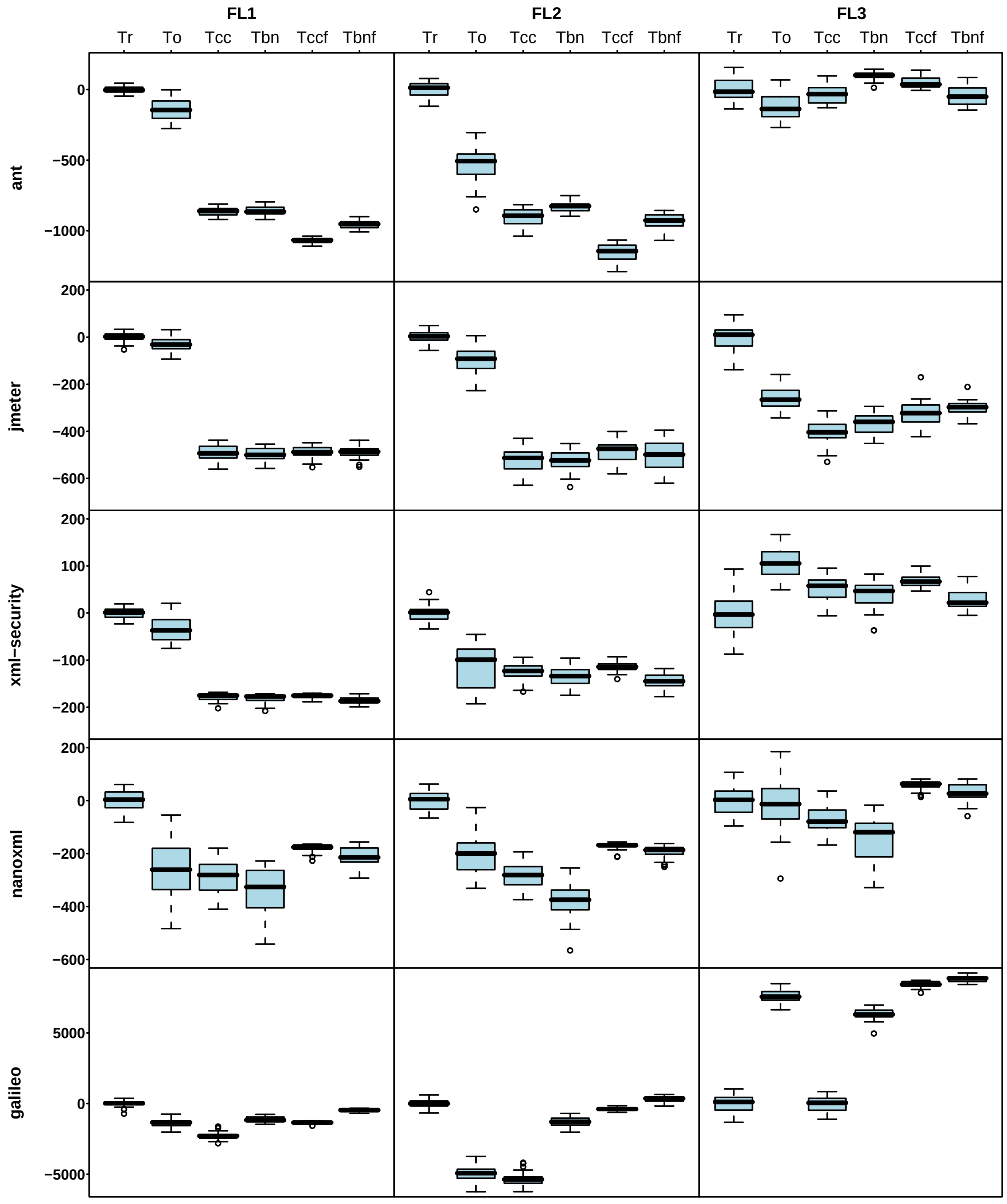

Figure 6: Experiment 3: Relative cost-benefit boxplots, all programs, all techniques, different faultiness levels.

Table 12: Kruskal-Wallis Test Results for RQ3

Program	$FL1$		$FL2$		$FL3$	
	χ^2	p-val.	χ^2	p-val.	χ^2	p-val.
ant	168	< 0.0001	164	< 0.0001	99	< 0.0001
jmeter	122	< 0.0001	127	< 0.0001	136	< 0.0001
xml	132	< 0.0001	101	< 0.0001	96	< 0.0001
nanoxml	127	< 0.0001	144	< 0.0001	106	< 0.0001
galileo	157	< 0.0001	164	< 0.0001	166	< 0.0001

Table 13: Bonferroni Test Results for RQ3, Comparing Across Faultiness Levels per Program

FL	ant			$jmeter$			$xml - security$			$nanoxml$			$galileo$		
	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp	Tech.	Mean	Grp
	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A
	To	-139	B	To	-31	B	To	-32	B	Tccf	-179	B	Tbnf	-481	B
FL1	Tbn	-861	C	Tccf	-488	C	Tccf	-176	C	Tbnf	-208	C	Tbn	-1129	C
	Tcc	-864	C	Tbnf	-489	C	Tcc	-178	C	To	-267	C D	Tccf	-1335	D
	Tbnf	-956	D	Tcc	-492	C	Tbn	-181	C	Tcc	-288	D E	To	-1385	D
	Tccf	-1069	E	Tbn	-498	C	Tbnf	-185	C	Tbn	-342	E	Tcc	-2287	E
	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tr	0.0	A	Tbnf	315	A
	To	-528	B	To	-101	B	To	-110	B	Tccf	-173	B	Tr	0.0	A
FL2	Tbn	-826	C	Tccf	-488	C	Tccf	-113	B	Tbnf	-195	B	Tccf	-379	B
	Tcc	-904	D	Tbnf	-498	C	Tcc	-124	B C	To	-201	B	Tbn	-1303	C
	Tbnf	-935	D	Tcc	-520	C	Tbn	-132	C	Tcc	-284	C	To	-4925	D
	Tccf	-1155	E	Tbn	-523	C	Tbnf	-145	C	Tbn	-374	D	Tcc	-5354	E
	Tbn	98	A	Tr	0.0	A	To	105	A	Tccf	56	A	Tbnf	8852	A
	Tccf	53	A B	To	-260	B	Tccf	68	B	Tbnf	30	A B	Tccf	8431	B
FL3	Tr	0.0	B C	Tbnf	-300	B C	Tcc	51	B C	Tr	0.0	A B	To	7630	C
	Tcc	-30	C	Tccf	-323	C	Tbn	40	C	To	-24	B C	Tbn	6335	D
	Tbnf	-49	C	Tbn	-371	D	Tbnf	30	C	Tcc	-71	C	Tr	0.0	E
	To	-128	D	Tcc	-403	D	Tr	0.0	D	Tbn	-151	D	Tcc	-7	E

which have TSL test suites, the gap between control techniques and heuristics becomes even narrower, and at FL3, feedback techniques appear to outperform both control techniques.

The following sections provide the statistical analyses and results relevant to our research question. For statistical analysis, for reasons similar to those used in Experiments 1 and 2, we used a Kruskal-Wallis non-parametric one-way analysis of variance followed by Bonferroni's test for multiple comparisons.

Our hypothesis associated with RQ3 is: *(H3) given a specific faultiness level, the cost-benefits between test case prioritization techniques differ.* Table 12 presents the results of the Kruskal-Wallis tests (df = 5, significance level 0.05), and shows that faultiness levels have significant effects in all cases.

Table 13 presents the results of the Bonferroni tests using a Bonferroni correction. As the table shows, the results reveal different trends between techniques among faultiness levels and object programs.

- At faultiness levels FL1 and FL2, in all cases but one, heuristics failed to outperform random orderings (the single exception occurring for *Tbnf* on *galileo* at FL2).

- At faultiness level FL3, however, several techniques outperformed random orderings. Moreover, the ranking between techniques changes as faultiness level moves from FL2 to FL3 in more than half of the cases, with the performance of heuristics improving. In particular, techniques using feedback performed better than the control techniques in several cases: *Tccf* on *ant*, *nanoxml*, and *galileo*, and *Tbnf* on *nanoxml* and *galileo*.

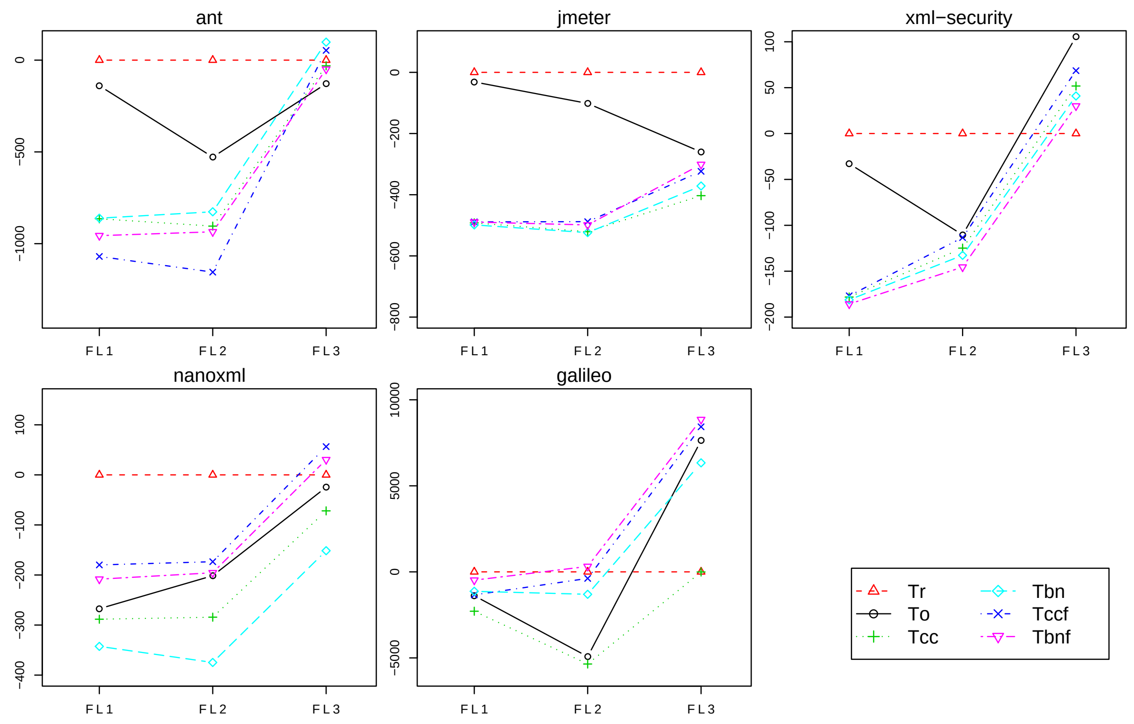

Figure 7: Cost-benefit line-plots, all programs, all techniques, all faultiness level, no time-constraints

6.4 Discussion

Figure 7 presents line-plots of the mean benefit values for each prioritization technique, at each faultiness level, for each program. These line-plots, together with the formal analysis of Section 6.3, facilitate our discussion of results.

6.4.1 Understanding the Effects of Faultiness Levels

As the graphs illustrate, considering all four heuristics across all five programs, as faultiness levels move from FL1 to FL2, technique benefits improve in 12 of 20 cases. As faultiness levels move from FL2 to FL3, however, technique benefits improve in all 20 cases. While at the lower levels techniques do not produce positive benefits, the trend is generally upward as faultiness level increases, and benefits begin to accrue at the higher faultiness level (in 12 of 20 cases at FL3). The most consistent patterns are exhibited by feedback techniques, with *Tccf* increasing in eight of ten cases and *Tbnf* increasing in nine of ten cases (across both faultiness level increments).

These upward trends can be explained relative to our cost model. When no time constraints exist, heuristics have negative benefits not because the test order they produce is not as good as that of random ordering but because the benefits they produce through *early fault detection* do not compensate for the cost of running the techniques. When fewer faults are present, heuristics have fewer opportunities to make a difference in rate of fault detection, even when they do an effective job of ordering. When greater numbers of faults are present, if techniques indeed order test cases better than random orderings, they have more opportunities to produce benefits by detecting faults faster. In other

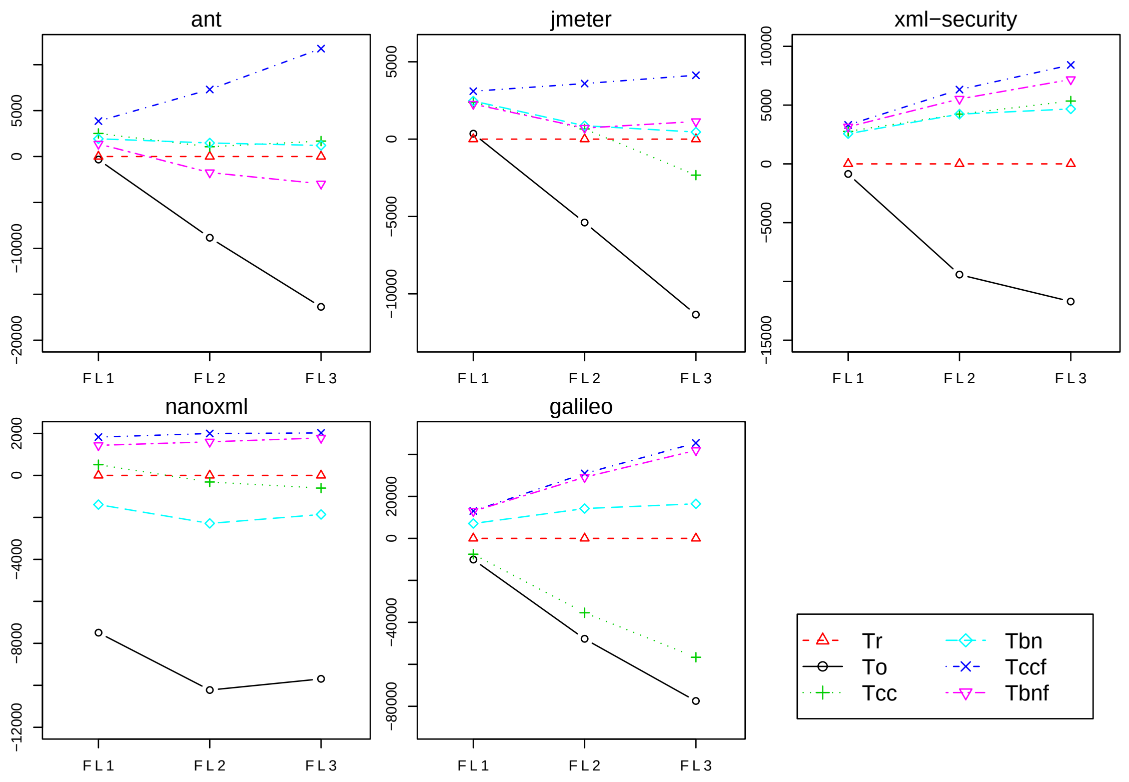

Figure 8: Cost-benefit lineplots, all programs, all techniques, all faultiness-levels, 50% time constraints.

words, if enough faults exist, the benefit due to *early fault detection* alone can indeed justify the costs imposed by heuristic techniques, and this is important because in the absence of time constraints, early fault detection is the only source of benefit for these techniques. Clearly, this result will have implications for practice, and we discuss these in Section 7.

These results also underscore two important points relevant to the further study of prioritization techniques. First, the cost-effectiveness of heuristics depends to a large extent on the characteristics of the object (in this case, characteristics related to the prevalence of faults), and thus, specifying these characteristics in studies is important. Second, the results of earlier studies using simple rate of fault detection metrics, while likely over-optimistic about technique cost-effectiveness in practice when levels of faultiness are low, are likely to be more accurate at higher faultiness levels. Thus the simpler metrics can plausibly be used to provide initial data on trends between techniques. However, ultimately, more comprehensive models will provide a clearer picture of tradeoffs.

6.4.2 Faultiness Levels Under Time Constraints

While we did not include time constraints as an independent variable in this experiment, for the sake of comparison we did gather data on technique performance, under the three faultiness levels, at TCL-50 (where 50% of the test cases in the prioritized order are executed). Figure 8 depicts the line-plots for this case.

In this case, considering all four heuristics across all five programs, as faultiness levels move from FL1 to FL2, technique benefits improve in 11 of 20 cases, and as faultiness levels move from FL2 to FL3 technique benefits improve in just 13 of 20 cases. The most consistent patterns continue to be exhibited by feedback techniques, with *Tccf* increasing in all 10 cases and *Tbnf* increasing in 7 of 10 cases across both faultiness level increments. *Tcc*, in contrast, exhibits a *reduction* in benefit in 8 of 10 cases. Overall, then, the upward trends observed when no time constraints are present are somewhat lessened in the presence of time constraints.

Despite these trends, as observed in prior studies, most techniques provide benefit in most cases, at all three faultiness levels. The addition of fault omission costs to the benefits gained by increasing rate-of-fault-detection produces this result, and is also likely responsible for the differences in performance trends across faultiness levels.

7 Practical Implications

So far we have discussed our major findings and some surprises seen in the results of our experiments, and some of the implications of these results. The results do lead to many additional observations including practical implications for test case prioritization and testing processes, and we discuss these now.

7.1 Prioritization and Context Factors

We have already noted that when time constraints may apply, choosing to not prioritize may be problematic. When no time constraints apply, the benefits of prioritization tended to become evident only with increases in faultiness levels. One might wonder, then, whether consideration of just these two context factors — time constraints and faultiness levels — would be sufficient to help practitioners determine whether or not to use prioritization. Historical data on fault prevalence could help organizations estimate probable fault levels, and time constraints can conceivably be estimated as well; thus, it may be possible to provide predictors of cost-benefits relative to these factors.

Further qualitative analysis of our results suggests, however, that there may be additional context factors to consider, namely: (1) the cost of delayed fault detection, and (2) prioritization technique execution cost. In the case of factor (1), greater costs associated with delayed fault detection increase the potential for techniques to be beneficial even in the absence of time constraints. This could occur, for example, in cases where test suites require particularly long times to execute to completion (such as when manual testing is involved) or on safety-critical systems requiring reuse of entire comprehensive test suites. In the case of factor (2), if technique execution costs are low relative to the costs of other activities in the testing process, the use of heuristics has less potential to negatively impact overall cost-benefits. On our object programs, test execution time is relatively short compared to technique execution time, so the two factors together render heuristics often non-beneficial. But this is not the case on all programs.

Another potential context factor when time constraints exist involves the characteristics of those constraints. In this work, we have focused on the case in which time constraints are unknown or not easily predicted in advance. When time constraints are known or can be predicted, different considerations can apply to assessing cost-effectiveness. For example, prioritization techniques need not consider every test case, they need only prioritize test cases until enough have been chosen to fill the available time. Note that in such cases, techniques that consider the times taken by individual test cases (e.g., [17, 41, 59]) may also be required, and these generally rely on predictions of test runtime based on prior runs, which can prove incorrect following code modifications, so this could be a source of imprecision.

7.2 Regression Testing Processes

Our results also have implications for regression testing processes. For one thing, an organization's choice of prioritization technique could reasonably be influenced by the testing processes they use. For example, for incremental testing processes in which tests are run more often, the likelihood of being forced to constrain testing activities due to time restrictions may be higher than for batch maintain-and-test processes, given sufficiently long-running regression test suites. Process implications may also extend to the types of testing being performed: if execution of test cases (or checking of results) is largely manual, this increases the likelihood that an organization will face time constraints. In both of these cases, process considerations favor prioritization.

On the other hand, under some incremental testing processes (e.g., nightly-build-and-test) prioritization becomes unnecessary from a rate-of-fault-detection standpoint. This occurs because in such cases, fault correction does not commence until the testing phase has ended. Here, prioritization might still be beneficial given its ability to cause fewer faults to be omitted, but regression test selection techniques could conceivably do just as well at filling the known testing time slot, as test order within the time slot in this case is unimportant. However, this does not preclude using prioritization on the system testing phase that often precedes final product release following many cycles of incremental development and testing.

The relationship between testing processes and the cost of the analysis needed to support prioritization is also important, because in practice, the costs that are practically significant for a prioritization technique in a given context can vary with the regression testing process used. For example, in a typical batch maintain-and-test process, analysis costs can be distributed across the maintenance and testing phases, while in a more incremental testing process a greater proportion of analysis costs may be relegated to the testing phase. In the latter case, analysis costs may actually lead to increased time constraints, and this in turn may cause fewer test cases to be executed, and larger numbers of faults to be missed. In such cases, choosing techniques for which analysis costs during the testing phase can be minimized may be important.

As we observed in Experiments 2 and 3, faultiness levels can also affect the choice of prioritization techniques. For instance, when programs contain large numbers of faults, our results showed that heuristics could become cost-effective even when no time constraints applied. The LOC model that we used in Experiment 2 primarily utilizes the size of the program and the changes between two consecutive versions to calculate the number of faults. Under this model, as program size increases, and as the number of changes between versions increases, the number of faults the program could contain increases. This suggests that in general, it is more reasonable to employ heuristics within a batch maintain-and-test process, and more particularly in cases where larger portions of the system have changed, than in incremental processes. Possibly, later lifecycle releases where less code churn occurs will benefit less from prioritization.

8 Related Work

We discuss two areas of related work: work on test case prioritization, and work on cost models.

8.1 Prior Work on Test Case Prioritization

A wide range of prioritization techniques have been proposed and studied. As mentioned in Section 2, initially, most techniques depended on code coverage information to drive the prioritization [14, 15, 16, 18, 25, 48, 49, 55, 60]. Restricting the foregoing techniques to consider coverage of changed components has also been explored [48, 55].

More recently, several prioritization techniques that go beyond the use of code coverage information have also been proposed. Leon and Podgurski [31] present prioritization techniques based on distributions of execution profiles. Jeffrey and Gupta [22] present an algorithm that prioritizes test cases based on their coverage of statements in relevant slices. Li et al. [33] present search-based prioritization algorithms. Korel et al. [28, 29] propose prioritization techniques based on coverage of system models. Yoo et al. [61] study the use of expert knowledge for prioritization by pair-wise comparison of test cases and propose clustering test cases into similar groups to facilitate the process. Hou et al. [21] study prioritization of test cases when testing web services software and Sampath et al. [51] study test suite prioritization strategies for web applications. Sherriff et al. [52] utilize change history to gather change impact information and prioritize test cases accordingly. Qu et al. [44] consider prioritization in the context of configurable systems, presenting algorithms for prioritization of configurations. Malishevsky et al. [17] present a prioritization technique that uses coverage information along with data on test execution times and estimated fault severities, and Park et al. [41] propose a technique for estimating these times and severities using historical information. Finally, Mirarab and Tahvildari [36] present the techniques that use Bayesian networks to prioritize test cases, and that are also studied here along with the simplest coverage-based techniques.

Only a few papers have considered issues related to the presence of time constraints during prioritization. Kim and Porter [26] present a technique for "history-based test prioritization" which, while not ordering individual test cases, does prioritize the subsets of test cases selected across a succession of releases. Still, the technique is better classified as a regression test selection technique that utilizes history information from prior releases.

More relevant to this article is work by Walcott et al. [59], who present a technique that combines information on test execution times with code coverage information, and utilizes a genetic algorithm to obtain test case orderings. Zhang et al. [62] use similar input data and utilize integer linear programming for ordering test cases. Alspaugh et al. [1] also study the application of several knapsack solvers to prioritization. In each of these cases, the research considers testing process contexts in which test suite execution is foreshortened by time constraints, and attempt to accommodate this through their techniques. Individual test case times are also considered (as in several papers described just above). The primary difference between these approaches and those we consider here, however, is that *they assume that the time constraints faced in prioritization are known beforehand*, and they factor these in to their algorithms. Our goal in this article was to study the case in which the time constraints are *not* known and hence, we did not choose these techniques for use in our experiments.

Where prior studies of prioritization are concerned, the vast majority reported in the literature (in the papers cited above) have focused on the effects of prioritization on rate of fault detection or rate of code coverage under the scenario in which all test cases are executed. In these scenarios, the cost-effectiveness of prioritization lies in detecting faults earlier or attaining coverage more quickly, and studies focus on measures of these metrics in their assessments. Under these scenarios, however, the possible costs of missing faults due to foreshortened testing are not captured.

Two recent studies [13, 59] *have* utilized time constraints while investigating prioritization effectiveness, and these form part of the motivation for this work. In this work, however, we focus specifically on designing experiments that manipulate time constraints as an independent variable, and this lets us draw well-founded conclusions about the effects of constraints overall and on particular techniques.

8.2 Prior Work on Economic Models

Most early work on prioritization utilized simple rate-of-fault-detection metrics (e.g., APFD [48] or its derivatives) to evaluate prioritization effectiveness. Such metrics, however, do not suffice to assess time-constrained techniques, because assessment of such techniques requires costs of omitted faults to be measured, as well as savings in rate of fault detection. For this reason, in this work, we rely on more comprehensive economic models that capture both costs and benefits of prioritization, including factors related to rate of fault detection, omission of faults, and cost of applying techniques. While subsequent sections of this article provide details on our cost model, the related work is discussed in this section.

Initial models of regression testing were relatively simple. Leung and White [32] present a model that considers some of the cost factors (e.g., testing time, technique execution time) that affect the cost of regression testing. Harrold et al. [20] present a coverage-based predictive model of regression test selection effectiveness, but this predictive model focuses only on reducing numbers of test cases. Malishevsky et al. [34] present cost models for regression test selection and test case prioritization that incorporate benefits related to the omission of faults and to the rate of fault detection. Do et al. [15] extend Malishevsky's model, for test case prioritization, to incorporate additional cost factors, including analysis and execution time.

There are some works on economic models for testing (as distinct from regression testing). Muller et al. [38] present an economic model for the return on investment of Test-Driven Development (TDD) compared to conventional development, provide a cost-benefit analysis, and identify a break-even point at which TDD becomes beneficial over conventional development. Wagner [57] proposes an analytical model of the economics of defect detection techniques that incorporates various cost factors and revenues, considering uncertainty and sensitivity analysis to identify the most relevant factors for model simplification. Wagner [58] also applies global sensitivity analysis (which investigates how output uncertainty can be apportioned to input factors' uncertainty) to the COCOMO model to investigate which input factors are most important.

While economic models in the software testing and regression testing areas are not well established, in other software engineering areas models have been considered much more extensively. These include the models of Ostrand et al. [56] and Freimut et al. [19] which have been already discussed. Kusumoto et al. [30] also propose a model for assessing the cost-effectiveness of software inspection processes. There have also been many models created to address software process and development costs and benefits; Boehm et al. [5]'s COCOMO-II model, mentioned earlier, is probably the most well-known. Recent research in value-based software engineering has also sought to model various engineering activities. Return On Investment (ROI) models, which calculate the benefit return of a given investment [43], provide one such approach, supporting systematic, software business value analysis [4, 38, 54].

9 Conclusions and Future Work

We have presented a series of controlled experiments assessing the effects of time constraints and faultiness levels on the costs and benefits of test case prioritization techniques. Our results show that time constraints can indeed play a significant role in determining both the cost-effectiveness of prioritization techniques, and the relative cost-benefit tradeoffs among techniques. The results also show that when a software product contains a large number of faults, employing heuristics could be beneficial even when no time constraints apply. This indicates that the benefits gained from early fault detection are high enough to compensate for the costs incurred by applying heuristics.

Of course, as with all empirical studies, our results must be interpreted in light of threats to validity and many of these can be addressed only through further studies of additional artifacts.

One class of further study involves other types of prioritization techniques. We chose to study the two techniques that are the simplest and most complex presented to date, as these presented a range of potential technique costs and presumably benefits. Techniques that incorporate test execution time into their prioritization [34, 59] might also be of interest given their attention to time, for the case in which time constraints are known beforehand.

Our results also led us to suggest several further practical implications. The most interesting of these implications for further research, on our view, involve differences in software maintenance and testing processes, and regression testing techniques.

For example, our results suggest that regression testing within constrained software development processes might be improved by manipulating test prioritization technique costs. If an organization knows that they cannot execute all of their regression tests, then potentially, they can lower analysis costs by prioritizing fewer tests, an approach that we could call "partial prioritization". Partial prioritization approaches will require data on test execution times, and will need to estimate expected execution times following modifications with sufficient precision.

A second example of further work related to processes and techniques involves the use of incremental supporting analyses. As mentioned in Section 3, our cost model facilitates the measurement of costs and benefits related to the use of incremental program analysis techniques (e.g., for instrumentation and probe placement) to support prioritization, but we did not explore these in this work. The results of this study suggest that these approaches might vary in cost-effectiveness across different development and testing processes (e.g., batch versus incremental). Techniques for better leveraging incremental analysis techniques within various forms of time-constrained processes could be worth exploring.

Finally, efforts such as those just described can be directed at other common testing processes, such as test-first methodologies, for which time constraints may play an even more important role. To truly consider some of the questions that arise regarding differences in testing processes, however, we need to adapt the economic model used here, which focuses on batch processes, to those processes. However, we believe that such an adaptation can be achieved. For example, one way to adapt the process model depicted in Figure 1 to depict an incremental model is to partition the maintenance phase into a sequence of maintain/test pairs. The regression testing phase then represents the system testing that typically precedes an ultimate system release. The economic model can then be adjusted to capture costs and benefits relevant to this process.

Ultimately, given further studies, techniques and model development, we expect this research to help test engineers better manage their regression testing efforts by enabling them to select testing processes and prioritization techniques that are most appropriate for their organizational and process contexts.

Acknowledgments

This work was supported in part by the National Science Foundation under Awards CNS-0454203, CCR-0080898, and CCR-0347518 to the University of Nebraska - Lincoln. This work was also funded in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada-Strategic Project Grants (SPG) and Ontario Ministry of Research and Innovation (MRI)-Early Researcher Award (ERA) to the University of Waterloo. Bouchaib Falah helped collect the number of faults based on the LOC model. We thank the anonymous reviewers of the conference version of this article, and of earlier versions of this article, for providing insightful comments.

References

[1] S. Alspaugh, K.R. Walcott, M. Belanich, G.M. Kapfhammer, and M.L. Soffa. Efficient time-aware prioritization with knapsack solvers. In *Proceedings of the ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 17–31, November 2007.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, pages 402–411, May 2005.

[3] R. Bell, T. Ostrand, and E. Weyuker. Looking for bugs in all the right places. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 61–72, July 2006.

[4] B. Boehm. Value-based software engineering. *ACM SIGSOFT Software Engingeering Notes*, 28(2):4, 2003.

[5] B. Boehm, C. Abts, A.W. Brown, S. Chulani, E. Horowitz B.K. Clark, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II.* Prentice-Hall, 2000.

[6] S.R. Chidamber and C.F. Kemerer. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notes*, 26(11):197–211, November 1991.

[7] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. Technical Report TR04-11, Department of Computer Science, University of Arizona, 2004.

[8] Culpepper and Inc. Associates. Culpepper Compensation and Benefit Surveys. http://www.culpepper.com.

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

[10] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 71–82, November 2008.

[11] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 141–151, November 2006.

[12] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[13] H. Do and G. Rothermel. Using sensitivity analysis to create simplified economic models for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 51–61, July 2008.

[14] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 113–124, November 2004.

[15] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering: An International Journal*, 11(1):33–70, 2006.

[16] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112, August 2000.

[17] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of Internationaol Conference on Software Engineering*, pages 329–338, May 2001.

[18] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engingeering*, 28(2):159–182, 2002.

[19] B. Freimut, L.C. Briand, and F. Vollei. Determining inspection cost-effectiveness by combining project data and expert opinion. *IEEE Transactions on Software Engingeering*, 31(12):1074–1092, 2005.

[20] M.J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engingeering*, 27(3):248–263, 2001.

[21] S. Hou, L. Zhang, T. Xie, and J. Sun. Quota-constrained test case prioritization for regression testing of service-centric systems. In *Proceedings of the International Conference on Software Maintenance*, pages 257–266, September 2008.

[22] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Proceedings of the Annual International Computer Software and Applications Conference*, pages 411–420, September 2006.

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

[24] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 1997.

[25] J. Jones and M.J. Harrold. Test suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engingeering*, 29(3):193–209, 2003.

[26] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of Internationaol Conference on Software Engineering*, pages 119–129, May 2002.

[27] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A Flexible Framework for Development of Dynamic Program Analysis for Java Software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska-Lincoln, April 2006.

[28] B. Korel, G. Koutsogiannakis, and L. Tahat. Application of system models in regression test suite prioritization. In *Proceedings of the International Conference on Software Maintenance*, pages 247–256, September 2008.

[29] B. Korel, L. Tahat, and M. Harman. Test prioritization using system models. In *Proceedings of the International Conference on Software Maintenance*, pages 559–568, September 2005.

[30] S. Kusumoto, K. Matsumoto, T. Kikuno, and K. Torii. A new metric for cost-effectiveness of software reviews. *IEICE Transactions on Information Systems*, E75-D(5):674–680, 1992.

[31] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the International Symposium on Software Reliability Engineering* , pages 442–453, November 2003.

[32] H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In *Proceedings of the International Conference on Software Maintenance*, pages 201–208, October 1991.

[33] Z. Li, M. Harman, and R.M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engingeering*, 33(4):225–237, 2007.

[34] A. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance*, pages 204–213, October 2002.

[35] S. Mirarab. A Bayesian Framework for Software Regression Testing. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, August 2008.

[36] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on Bayesian Networks. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering , LNCS 4422-0276*, pages 276–290, March 2007.

[37] S. Mirarab and L. Tahvildari. An empirical study on Bayesian Network-based approach for test case prioritization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 278–287, April 2008.

[38] M.M. Muller and F. Padberg. About the return on investment of test-driven development. In *Proceedings of the International Workshop on Economics-Driven Software Engineering Research*, pages 2631–2636, May 2003.

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

[40] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–688, 1988.

[41] H. Park, J. Ryu, and J. Baik. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *Proceedings of the International Conference on Secure System Integration and Reliability Improvement*, pages 39–46, July 2008.

[42] D.E. Perry and C.S. Stieg. Software faults in evolving a large, real-time system: A case study. In *Proceedings of the European Software Engineering Conference, LNCS 717*, pages 48–67, September 1993.

[43] J.J. Phillips. *Return on Investment in Training and Performance Improvement Programs*. Gulf Publishing Company, 1997.

[44] X. Qu, M.B. Cohen, and K.M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Proceedings of the International Conference on Software Maintenance*, pages 255–264, October 2007.

[45] F.L. Ramsey and D.W. Schafer. *The Statistical Sleuth*. Duxbury Press, 1997.

[46] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engingeering*, 22(8):529–551, 1996.

[47] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[48] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold. Test case prioritization. *IEEE Transactions on Software Engingeering*, 27(10):929–948, 2001.

[49] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, August 1999.

[50] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. *Sensitivity Analysis in Practice*. John Wiley, 2004.

[51] S. Sampath, R. Bryce, G. Viswanath, V. Kandimalla, and A. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 141–150, April 2008.

[52] M. Sherriff, M. Lake, and L. Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 81–90, November 2007.

[53] F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Proceedings of the International Software Metrics Symposium*, pages 249–258, June 2002.

[54] R. Solingen. Measuring the ROI of software process improvement. *IEEE Software*, 21(3):32–38, 2004.

[55] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *ACM SIGSOFT Software Engineering Notes*, 27(4):97–106, 2002.

[56] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engingeering*, 31(4):340–355, 2005.

[57] S. Wagner. A model and sensitivity analysis of the quality economic of defect-detection techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–84, July 2006.

[58] S. Wagner. An approach to global sensitivity analysis: FAST on COCOMO. In *International Symposium on Empirical Software Engineering and Measurement*, pages 440–442, September 2007.

[59] A. Walcott, M.L. Soffa, G.M. Kapfhammer, and R.S. Roos. Time-aware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–12, July 2006.

[60] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 264–274, November 1997.

[61] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 201–212, July 2009.

[62] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. Time-aware test-case prioritization using integer linear programming. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 213–224, July 2009.