An Empirical Study of the Effect of Time Constraints on the Cost-Benefits of Regression Testing

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

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. Therefore, we conducted an experiment to assess the effects of time constraints on the costs and benefits of prioritization techniques. Our results show that time constraints can indeed play a significant role in determining both the cost-effectiveness of prioritization, and the relative cost-benefit tradeoffs among techniques, with important implications for the use of prioritization in practice.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing & Debugging—*testing tools*

General Terms

Experimentation, Measurement, Verification

Keywords

Regression testing, test case prioritization, empirical studies

1. INTRODUCTION

After modifying software, developers typically regression test it, both to validate changes and to detect whether new faults have been introduced into previously tested source code. Regression testing can be time-consuming and expensive: it is not uncommon for regression test suites to require days or weeks to run to completion, with expensive equipment and engineering costs associated [3, 22, 30]. *Test case prioritization* techniques help with this, 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.

To date, a great deal of research on test case prioritization has been performed, most of it on specific prioritization techniques (e.g., [12, 18, 20, 28]). A common approach for empirically studying these techniques (e.g., [8, 11, 17]) is to first obtain various programs, modified versions, faults, and test suites. Then, one or more prioritization techniques are run on the test suites, and the resulting ordered suites are executed, with measurements taken of their effectiveness at satisfying test 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 [7, 15, 32] that time constraints 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 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, some techniques will perform more or less cost-effectively than others as the time constraints imposed on regression testing increase.

If these conjectures are correct, engineers may be able to better manage their regression testing activities by using prioritization techniques that are most appropriate for their software development processes, given the time constraints those processes impose on regression testing. We have therefore designed and implemented a controlled experiment to examine the effects of time constraints.

The results of our experiment show that the conjectures hold: time constraints matter. In fact, in the cases we examined, when no time constraints applied, prioritization was not even cost-effective. When time constraints applied, on the other hand, prioritization began to yield benefits, and in general, greater constraints resulted in greater benefits. Moreover, in this case, prioritized test suites significantly outperformed unordered or randomly ordered test cases,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-59593-995-1 ...$5.00.

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 analysis of results discusses these effects and several further practical implications.

The rest of the paper is organized as follows. Section 2 discusses background information and related work relevant to prioritization techniques and their assessment. Section 3 presents our study design, Section 4 presents our data and analysis, and Section 5 discusses our results and their further implications. Finally, Section 6 presents conclusions and discusses possible future work.

2. BACKGROUND AND RELATED WORK

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing attempts to validate P'. In practice [22], engineers often reuse all of the test cases in in T to test P'; however, this *retest-all* approach can be expensive [30].

Researchers have studied various methods for improving the cost-effectiveness of regression testing. *Regression test selection techniques* (surveyed in [26]) reduce testing costs by selecting, from T, a subset of test cases to execute on P'. These techniques reduce costs by reducing testing time, but unless they are *safe* [27] they can omit test cases that would otherwise have detected faults. This can raise the costs of software.

Test case prioritization techniques (e.g., [10, 12, 15, 17, 18, 20, 28, 30, 32, 33]) offer an alternative approach to improving regression testing cost-effectiveness. Prioritization can improve cost-effectiveness in two ways. First, prioritization can help engineers reveal faults early in testing, allowing them to begin debugging activities earlier in the testing cycle 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. Second, in the case in which 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.

2.1 Test Case Prioritization Techniques

Various prioritization techniques have been proposed and empirically studied. Most of these techniques depend primarily on code coverage information. Code coverage information is obtained by instrumenting a program such that portions of its source code (e.g, method entries, statements, or basic blocks) that are exercised by test cases can be measured.

Given the results of running tests on instrumented code, a "total-coverage" prioritization technique [28] orders test cases in terms of the total number of code components (e.g., methods, statements, or blocks) that they cover. One way to improve "total-coverage" techniques is to add 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 [28] 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.

Techniques such as these, in which test case orderings are based solely on coverage and calculated by relatively simple algorithms, predominate in the literature and have been the most extensively studied (e.g., [11, 28, 30, 33]). Various other algorithms, however, have also been proposed. Jeffrey and Gupta [12] present an algorithm that prioritizes test cases based on their coverage of statements in relevant slices. Leon and Podgurski [17] present prioritization techniques based on distributions of execution profiles. Li et al. [18] present search-based algorithms. Kim and Porter [15] 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. Malishevsky et al. [19] present a prioritization technique that uses coverage information along with data on test execution times and estimated fault severities. Walcott et al. [32] present a technique that combines information on test execution times with code coverage information, and utilizes a genetic algorithm to obtain test case orderings. Finally, Mirarab and Tahvildari [20] present techniques that use Bayesian networks to prioritize test cases.

Most studies of prioritization techniques to date (e.g., [8, 11, 17]) 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. The potential importance of time constraints has been suggested in prior work [7, 15, 32], and two recent studies [9, 32] have utilized levels of time constraints while investigating prioritization effectiveness. To date, however, no controlled study specifically examining the effects of time constraints on prioritization has been conducted.

To formally examine the effects of time constraints on test case prioritization, in this work we consider two sets of approaches: the conventional coverage-based approaches outlined above (described as total-coverage and additional-block-coverage), and Bayesian-network-based techniques. These heuristics represent the algorithmically simplest and most complex techniques proposed to date, respectively, and thus offer a spectrum of technique costs (and one would expect, benefits) along which to conduct our study.

Bayesian-network-based (BN) techniques [20] attempt to capture the causal relations between variables that affect the outcome of a test case as conditional probabilities. For example, the following causal relations can be stated: 1) changing a software element (e.g., component, method, or function) may *cause* faults to be introduced, 2) low quality may *cause* a software element to be more fault-prone than others in the event of a change, and 3) the presence of a fault in a software element can *cause* the test cases covering that element to reveal the fault. To capture these causal relations as conditional probabilities, BN techniques use information extracted from the software system. For the example just cited, this implies using: 1) change analysis to find the software elements modified between consecutive versions, 2) metric-based prediction models of software quality (e.g., fault-proneness models [2, 31]) to estimate the conditional probability of introducing a fault when a software element changes, and 3) test coverage information to estimate the conditional probability that a test case will reveal a fault.

Once all of these probabilities have been estimated, BN techniques use a Bayesian Network to incorporate the information into a single model. A Bayesian Network is a graph of random variables connected using arcs that indicate conditional dependencies between the corresponding variables [13]. For the causal relations mentioned earlier, a Bayesian Network can be constructed as shown in Figure 1, where *n* represents the number of software elements to be evaluated and *m* represents the number of test cases. For each software element i, a CE_i variable in the first layer represents the event of change, and an FE_i variable in the second layer indicates whether a fault exists in that element. An arc from a CE_i variable to an FE_i variable indicates that FE_i is conditioned on CE_i, a conditional dependence that is estimated using quality metrics (i.e.

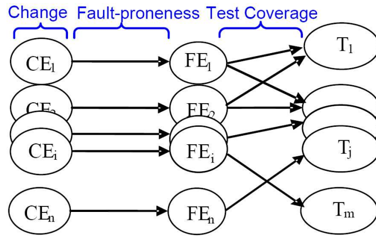

Figure 1: A Bayesian Network Structure for BN Techniques

a fault-proneness model). Finally, in the third layer, T variables represent the probability of revealing a fault for each test case and are conditioned on the FE variables using coverage information.

In a Bayesian Network, values of variables that are known (i.e., observed variables) can be fixed, and probabilistic inference algorithms can be used to find the conditional distribution of the other variables. In the Bayesian Network of Figure 1, the values of CE variables can be fixed (according to change data), the distribution of T variables can be inferred, and the resulting values of T variables can be used to prioritize the test cases. The more probable a test case is to detect a regression fault, the earlier it should appear in the prioritized order.

Just as feedback is used in code-coverage-based prioritization techniques, it can also be used by BN techniques. For this purpose, test cases are added to the prioritized order in iterations. In each iteration, first, the probabilistic inference is performed; then, one or more test cases that have higher probabilities of revealing faults are added to the prioritized order; finally, those test cases are marked as observed variables with no probability of revealing a fault. The iterations are halted when all the test cases have been added to the final order. For details on the algorithm see [21].

2.2 Evaluating Prioritization Techniques

To investigate the cost-effectiveness of prioritization techniques, we utilize the economic model presented in [7]. The model captures costs and benefits of techniques relative to particular regression testing processes, considering techniques in light of their business value to organizations, in terms of the cost of applying the techniques and how much revenue they help organizations obtain. The regression testing process that the model corresponds to is the common process in which system modifications are performed over a period of time (which might range from a day to several weeks or months) and then regression testing is used (overnight, or in succeeding days or weeks) to assess those modifications.

The model involves two equations. The first equation captures costs related to the salaries of the engineers who perform regression testing (to translate time spent by engineers into monetary values), and the second captures revenue gains or losses related to changes in product release time (to translate time-to-release into monetary values). Significantly, 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); however, in this work, to simplify the factors studied, we do not consider such techniques.

The two equations that comprise the model are as follows. Terms and coefficients used in the model are shown in Table 1. Section 3.2.2 provides further descriptions of these terms and coefficients and how their values are collected for our experiment.

Table 1: Terms and Coefficients

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)
$CS(i)$	time to setup for testing activities S_i
$CO_i(i)$	time to identify obsolete tests for S_i
$CO_r(i)$	time to repair obsolete tests for S_i
$CA_{in}(i)$	time to instrument all units in i
$CA_{tr}(i)$	time to collect traces for test cases in S_{i-1}
$CR(i)$	time to execute a technique itself on S_i
$CE(i)$	time to execute test cases on S_i
$CV_d(i)$	time to apply diff tools to the outputs of test cases run on S_i
$CV_i(i)$	(human) time for checking the results of test cases
$CF(i)$	cost associated with a missed fault after the delivery of S_i
$CD(i)$	cost associated with delayed fault detection feedback on S_i
REV	revenue in dollars per unit u
PS	average hourly programmer's salary in dollars per unit u
$ED(i)$	expected time-to-delivery in units u for S_i when testing begins
$a_{in}(i)$	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 the 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 test suite on S_i

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

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

In addition to capturing the costs related to the activities involved in prioritizing and running tests, this model captures two primary costs and benefits of concern when evaluating prioritization techniques. First, $CD(i)$ 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. Second, $CF(i)$ 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.

3. EMPIRICAL STUDY

As our economic model shows, the cost-effectiveness of prioritization involves interactions between several factors, including (1) the cost of applying a prioritization technique, including prerequisite analyses, (2) the costs of executing tests and validating test results, (3) the costs of delayed fault detection feedback, and (4) the costs of missing faults. When no time constraints exist, factor (4) does not apply; when time constraints exist, all factors can be affected. What we wish to study empirically is the ways in which tradeoffs among these factors, and ultimately in the cost-benefits of particular prioritization heuristics, are affected as time constraints vary. This leads to 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 compare under that constraint?

To address these questions, we performed a controlled experiment. The following subsections present, for this experiment, our objects of analysis, variables and measures, experiment setup and design, and threats to validity. Following this presentation, in Section 4 we present our data and analysis, and in Section 5 we discuss practical implications of the results.

3.1 Objects of Analysis

We used five Java programs from the SIR infrastructure [6] 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 [23].

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 object programs that contain faults, so we utilized mutation faults provided with the programs [8]. 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.

3.2 Variables and Measures

3.2.1 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 no testing is omitted; i.e., testing can be run to completion. TCL-25, TCL-50, and TCL-75 represent situations in which time constraints shorten the testing process (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. We further classify the four heuristic techniques into two groups, namely, *non-feedback techniques* and *feedback techniques*. Table 3 summarizes these techniques.

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

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" (*To*) and "random" (*Tr*). *To* 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". *Tr* 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, as described in Section 2.1, we utilize two methodologies: conventional coverage-based (CC) prioritization and Bayesian Network-based (BN) prioritization. As shown in Table 3, for each of these methodologies we consider two techniques: one that incorporates feedback and one that does not. Note that BN techniques use class-level coverage information, whereas CC techniques use block-level information. Further, because BN techniques must be configured via parameters, we utilize results obtained from a prior empirical study [21] to select appropriate parameter values.

3.2.2 Dependent Variable and Measures

Our dependent variable is a relative cost-benefit value produced by applying the economic model presented in Section 2.2, using a further calculation described below (Equation 3). The cost and benefit components are measured in dollars. The cost components include several constituent measures, which we collected as follows. To measure costs that involve human activities we averaged times required by two graduate students to perform the activities.

Cost Components

Cost of test setup (CS). This includes the costs of setting up the system for testing, compiling the version to be tested, and configuring test drivers and scripts.

Cost of identifying obsolete test cases (CO_i). This includes 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.

Cost of repairing obsolete test cases (CO_r). This includes the costs of examining and adjusting test cases and test drivers, and the costs of observing the execution of adjusted tests and drivers.

Cost of supporting analysis (CA). This includes the costs of instrumenting programs (CA_{in}) and collecting traces (CA_{tr}).

Cost of regression testing technique execution (CR).This is the time required to execute a prioritization technique itself.

Cost of test execution (CE). The time required to execute tests.

Cost of test result validation (automatic via differencing) (CV_d). This is the time required to run a differencing tool on test outputs as test cases are executed.

Cost of test result validation (human) (CV_i). This is the time required by engineers to inspect comparisons of test outputs.

Number and cost of missed faults (c and CF). To capture the cost of missed faults, for each application of a test case prioritization technique, we first measure the number c of faults that could have been detected by the entire suite but that were missed by the prioritized suite. Determining the cost of these missed faults, however, is more difficult. Given the many factors that can contribute to, and the long-term nature of, this cost, we cannot obtain this measure directly. Instead, we rely on data provided in [29] to obtain estimates of the cost of missed faults.

Cost of delayed fault detection feedback (CD). To measure the cost of delayed fault detection, for each application of a prioritization technique we measure the rate of fault detection exhibited by the prioritized test suite. Then, following [19], we translate this into the cumulative cost (in time) of waiting for each fault to be exposed while executing test cases under the prioritized order.

Revenue (REV). We cannot measure revenue directly, so we estimate it by utilizing revenue values cited in a survey of software products ranging from \$116,000 to \$596,000 per employee.[1] Because our object programs are relatively small, we use the smallest revenue value cited, and an employee headcount of ten.

Programmer salary (PS). We cannot measure this metric directly so we estimate it. We rely on a figure of \$100 per person-hour, obtained by adjusting an amount cited in [14] by an appropriate cost of living factor.

Expected time-to-delivery (ED). Actual values for ED cannot be obtained for our object programs. Thus, rather than calculate ED, we use Equation 3 to compare techniques (see the following discussion); this causes the value of ED to be cancelled out.

Other coefficients. The values of coefficients $a_{in}(i)$, $a_{tr}(i)$, and $b(i)$ are set to 1 because these coefficients are meant to capture, proportionally, reductions in cost due to the use of incremental analysis techniques, which, as Section 2.2 mentions, we choose not to consider in this experiment.

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 2.2). 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.

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.

To determine the *relative cost-benefit* of prioritization technique *T* with respect to baseline technique *base*, we use the equation:

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

When Equation 3 is applied, positive values indicate that T is beneficial compared to $base$, and negative values indicate otherwise.

Hereafter, for simplicity, we use the term "*cost-benefits*" to refer to these *cost-benefits relative to random orders*.

3.3 Experiment Setup

To perform prioritization, the CC and BN techniques require coverage information and fault data. The BN techniques also require data related to code changes and software quality metrics.

We obtained code change information using `sandmark` [5] as a byte code differencing tool, and we obtained quality metrics from the Chidamber-Kemerer metrics suite [4] using the `ckjm` program.[2]

We obtained coverage information by running test cases on our object programs instrumented using `Sofya` [16]. The resulting information lists which test cases exercised which blocks in the program; 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.

Our economic model 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, for each version of each program we randomly selected several *mutant groups* from the set of that version's mutation faults. Each mutant group contained at most 10 mutants. Then, for each program, we obtained 30 *sequences of mutant groups* by randomly selecting a mutant group for each version of that program.

To collect data we considered each object program in turn, and for each version of that program and each selected mutant group for that version, applied each prioritization technique and collected the appropriate values for cost variables (as indicated in Section 3.2.2). (In the case of the random technique, we did this for 30 different random orders, and averaged the results.) For cost variables potentially affected by time constraints (CE, CV_d, CV_i, CF, c, CD), we collected or calculated their values for each time constraint level. 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 (using Equation 3) for each of the prioritization techniques, at each level of time constraints, on each of the object programs. Each of these calculations required us to calculate the relative cost-benefit of the given technique (following the process

[1] http://www.culpepper.com

[2] http://www.spinellis.gr/sw/ckjm/

described in Section 3.2.2) at the given time constraint level and on the given program, for each of the 30 sequences of mutant groups created for that program. These resulting relative cost-benefit numbers serve as the data for our subsequent analysis.[3]

3.4 Threats to Validity

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

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 [1], the distribution of mutation faults used in our study may not match distributions of faults found in practice. Control for these threats can be achieved only through additional studies with wider populations of programs and faults, different testing 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 [24] present a different set of fault costs. 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, 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. DATA AND ANALYSIS

To provide an overview of the collected data, Figure 2 presents boxplots 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 3.2.2). Higher values indicate greater 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 3.3), 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 2, 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 time constraints (TCL-25, TCL-50, and TCL-75), in all cases but one (*jmeter* on *Tccf*), feedback techniques appear to be more beneficial than the 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 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 5.)

For our statistical analysis, we followed a process well established in prior studies of test case prioritization (e.g., [8, 11, 28])): we used the Kruskal-Wallis non-parametric one-way analysis of variance followed by Bonferroni's test for multiple comparisons [25]. 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.

4.1 Effects of Time Constraints on Techniques

Our hypothesis associated with RQ1 is: *(H1) the cost-benefits between time constraints differ*. To test this hypothesis, we performed the Kruskal-Wallis test (df = 3) for each technique per pro-

[3] Complete data sets can be obtained from the first author.

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

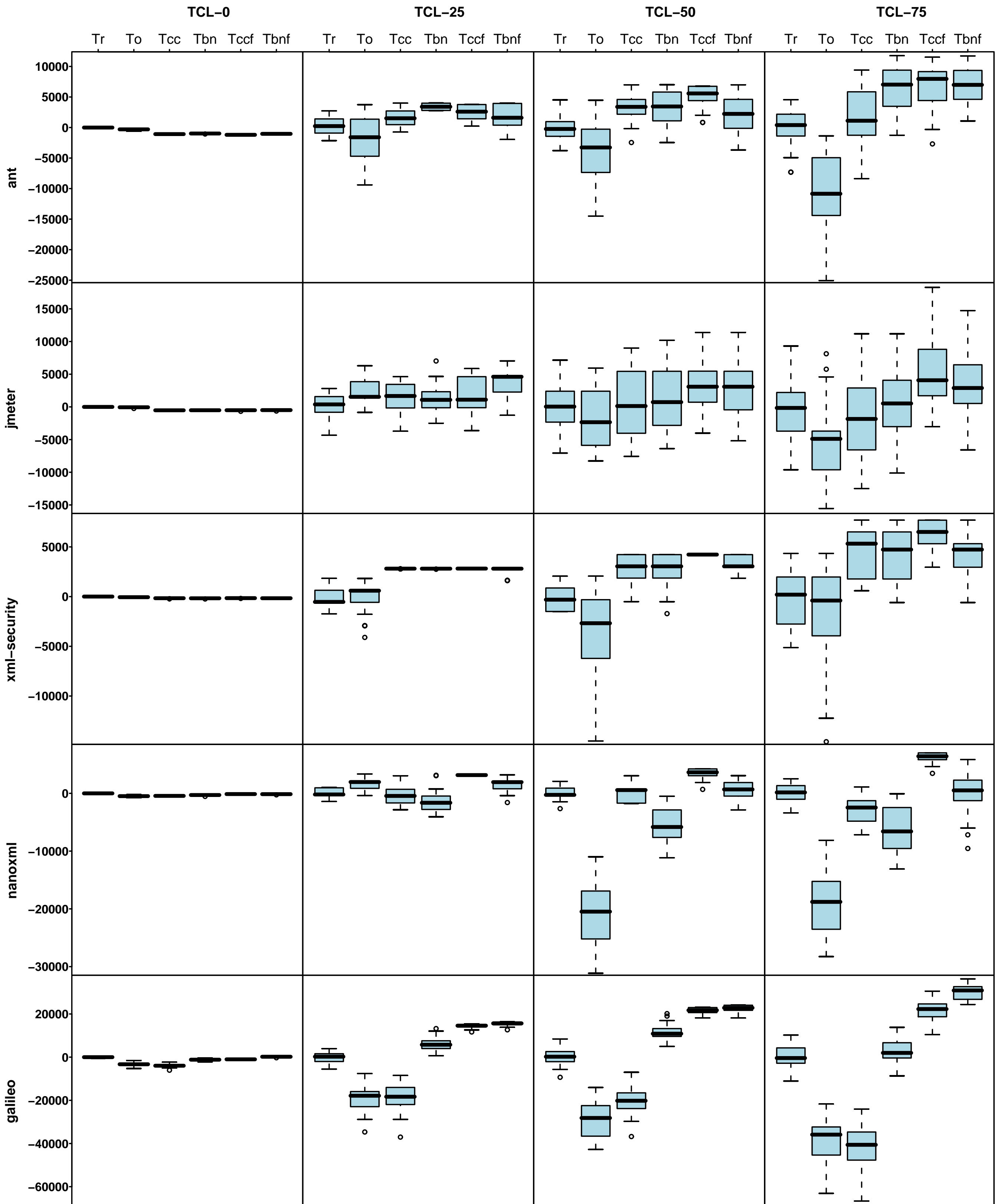

Figure 2: Cost-benefit boxplots, all programs, techniques, and time constraints.

Table 4: Kruskal-Wallis Test Results for RQ1

Program	*To*		*Tcc*		*Tbn*		*Tccf*		*Tbnf*	
	chi.	p-val.	chi.	p-val.	chi.	p-val.	chi.	p-val.	chi.	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

gram, 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.

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 testing. Table 5 presents the results of these tests with a Bonferroni correction [25]. 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 group letters (columns with headers "Grp") to partition the time constraints such that time constraints' results that are not significantly different share the same group letter. For example, the results of *Tccf* for *ant* show that TCL-75 and TCL-50 are not statistically significantly different (they share group letter A), but TCL-25 (group letter B) is statistically significantly different from the other three groups and so is TCL-0 (group letter C).

As the table shows, the results from multiple pair-wise comparisons reveal different trends between time constraints among techniques and object 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 time constraints.

The trends we observe for *To* 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*, 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.

4.2 Effects Among Techniques at Given Levels of Time Constraints

Our hypothesis associated with RQ2 is: *(H2) 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, group 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 object programs. In the case in which no time constraints applied (TCL-0), all control techniques were better than

Table 6: Kruskal-Wallis Test Results for RQ2

Program	*TCL-0*		*TCL-25*		*TCL-50*		*TCL-75*	
	chi.	p-val.	chi.	p-val.	chi.	p-val.	chi.	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

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

heuristics for the 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 outperformed the control techniques, but the results varied across programs. For example, all heuristics were better than both control techniques on *ant* and *xml-security*, 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* outperformed *Tbnf* (though not statistically significantly) at TCL-25 and TCL-50, but this relationship changed at TCL-75. In the case of *jmeter*, differences between heuristics were not statistically significant when no time constraints applied, but when time constraints applied, the ranking between heuristics stayed the same even though the statistical significance of differences between them changed. The other three programs display steady relationships between heuristics with only a couple of exceptions.

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.

5. 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, the reader should keep in mind the threats to validity for this study, and in particular, the fact that implications drawn here may differ under testing scenarios in which the values associated with cost factors differ.

Figure 3 presents lineplots of the mean 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, and together with the formal analysis of Section 4, 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? Bearing this question in mind, we now discuss the major findings of our experiment and the practical implications of the results.

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 3, 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., [11, 15, 17]), 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 measured 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 be misleading, and that more comprehensive economic models can lead to quite different conclusions about the cost-effectiveness of heuristics.

The worst thing you 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 prioritization 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 3 clearly shows this trend: the original orders' lineplots slope downwards, while others more often slope upwards.

Our results might also be interpreted as suggesting that the use of randomized test case orders is preferable to using arbitrary "original" orders. However, recall that our results for the random technique involve averages of multiple runs, and individual random orders may vary widely in performance.

These results do show that when time constraints *might apply*, the worst thing that engineers 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.

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 [11] 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

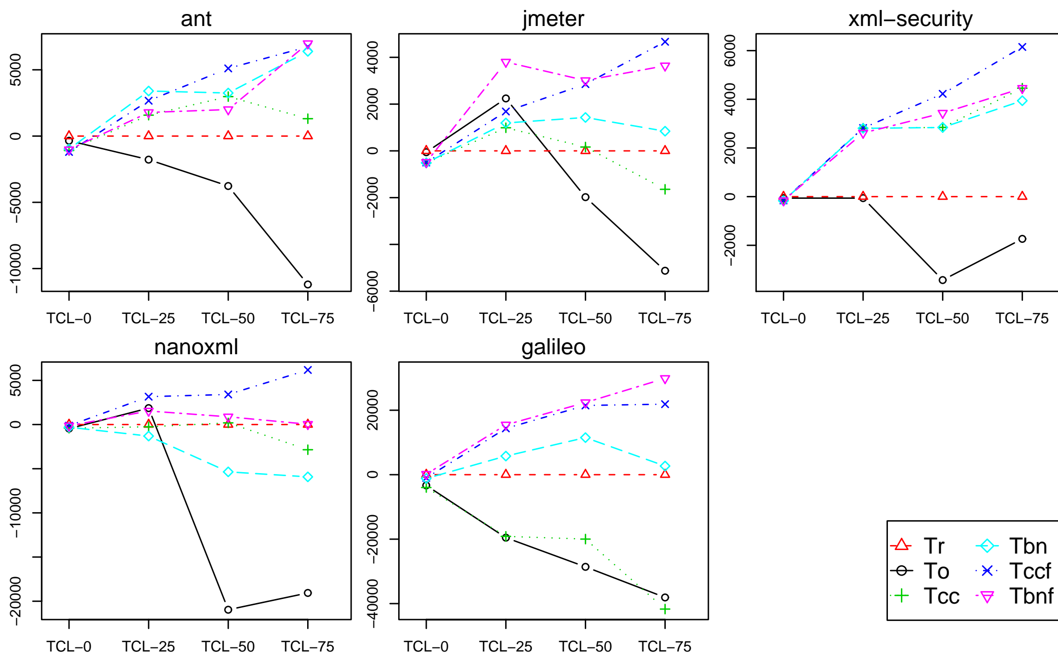

Figure 3: Cost-benefit lineplots, all programs, all techniques, all time constraints.

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.

Practical implications for prioritization.

So far we have discussed our major findings and some surprises seen in the results of our experiment. The next natural questions to ask involve the practical implications of the results. We begin with practical implications for prioritization.

We have already noted that when time constraints may apply, choosing to not prioritize may be problematic. When no time constraints apply, on the other hand, we found that heuristics were often not beneficial. Does this mean that engineers should not use prioritization when they expect not to face time constraints?

The answer to this question must be qualified depending on characteristics of the organizations' regression testing processes. Two additional factors (suggested above) should be considered: (1) the cost of delayed fault detection and (2) technique execution costs. In the first case, greater costs associated with delayed fault detection increase the potential for techniques to be beneficial. In the second case, 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. Note that the first factor is in turn affected by the cost of test execution: more lengthy test execution phases (such as when manual testing is involved) render fault detection delays more costly. On our object programs, test execution time is relatively short compared to technique execution time, and these two factors together render heuristics often non-beneficial.

Our results also indicate that among prioritization techniques, feedback techniques are the best choice when organizations face time constraints in testing. Both of the feedback techniques we studied (*Tccf* and *Tbnf*) were similar in their abilities to produce cost-benefits overall: when we compare them across all time constraint levels and programs, *Tbnf* outperformed *Tccf* 9 times (two statistically significant), and *Tccf* outperformed *Tbnf* 11 times (three statistically significant). On the other hand, *Tccf* consistently led to cost-benefit increases as time constraints increased, whereas *Tbnf* was less predictable in this respect, and this may favor *Tccf*.

Practical implications for testing processes.

Our results also have implications for 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 an incremental testing process such as nightly-build-and-test, the likelihood of being forced to constrain testing activities due to time restrictions is higher than for a batch maintain-and-test process, 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.

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.

6. CONCLUSIONS AND FUTURE WORK

We have presented a study assessing the effects of time constraints 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.

Of course, as with all empirical studies, our results must be interpreted in light of limitations (Section 3.4), and many of these limitations can be addressed only through further studies of additional artifacts. One class of further study that would be particularly germane concerns differences involving types and distributions of faults, and ranges of fault costs. For example, we saw that *Tccf* was more consistently cost-effective than *Tbnf* as time constraints increased and this suggests that *Tccf* might be preferable to *Tbnf*. These results, however, are based on artifacts for which fault distributions were random, and thus may not conform to distributions such as those assumed by fault proneness models. This might have lessened the power of BN techniques, which partially rely on probabilities captured by such models.

A second 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 [19, 32] might also be of interest given their attention to time, but for such studies to capture effects related to test execution times, objects of study including tests having a relatively wide variance in execution times would be required. Note that the technique presented in [32] is also capable of including testing time constraints up front in its calculations, and this might render its performance of interest in time-constrained situations; however, the technique does require that one know in advance how long one will be able to execute tests, a restriction we did not wish to impose in our study.

Our results also led to several further practical implications. The most interesting of these implications for further research, on our view, involve software testing and development 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 2, 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 useful.

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.

Ultimately, given further studies and technique 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.

Acknowledgements

This work was partially funded 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 partially by the Natural Sciences and Engineering Research Council (NSERC) of Canada. We thank the anonymous paper reviewers for providing insightful comments.

7. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Int'l. Conf. Softw. Eng.*, pages 402–411, May 2005.

[2] L. Briand and J. Wüst. Empirical studies of quality models in object-oriented systems. *Adv. Comp.*, 56:98–167, 2002.

[3] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.

[4] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *O.O. Prog. Sys., Langs., and Apps.*, pages 197–211, Nov. 1991.

[5] M. S. Christian Collberg, Ginger Myles. An empirical study of Java bytecode programs. Technical Report TR04-11, Dept. of Computer Science, University of Arizona, 2004.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Int'l. J. Emp. Softw. Eng.*, 10(4):405–435, 2005.

[7] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Found. Softw. Eng.*, pages 141–151, Nov. 2006.

[8] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, 32(9):733–752, 2006.

[9] H. Do and G. Rothermel. Using sensitivity analysis to create simplified economic models for regression testing. In *Int'l. Symp. Softw. Test. Anal.*, pages 51–61, July 2008.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Int'l. Conf. Softw. Eng.*, pages 329–338, May 2001.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, Feb. 2002.

[12] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Int'l. Comp. Softw. Appl. Conf.*, pages 411–420, Sept. 2006.

[13] F. V. Jensen. *Introduction to Bayesian Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[14] C. Jones. *Applied Software Measurement: Assuring productivity and quality*. McGraw-Hill, 1997.

[15] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Int'l. Conf. Softw. Eng.*, pages 119–129, May 2002.

[16] 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, Apr. 2006.

[17] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Int'l. Symp. Softw. Rel. Eng.*, pages 442–453, Nov. 2003.

[18] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.*, 33(4):225–237, Apr. 2007.

[19] A. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Conf. Softw. Maint.*, pages 204–213, Oct. 2002.

[20] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on Bayesian Networks. In *Found. App. Softw. Eng., LNCS 4422-0276*, pages 276–290, Mar. 2007.

[21] S. Mirarab and L. Tahvildari. An empirical study on Bayesian Network-based approach for test case prioritization. In *Int'l. Conf. Softw. Test. Verif. Val.*, pages 278–287, Apr. 2008.

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

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

[24] D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: A case study. In *Eur. S.E. Conf., LNCS 717*, pages 48–67, 1993.

[25] F. L. Ramsey and D. W. Schafer. *The Statistical Sleuth*. Duxbury Press, 1st edition, 1997.

[26] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, Aug. 1996.

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

[28] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Trans. Softw. Eng.*, 27(10):929–948, Oct. 2001.

[29] F. Shull et al. What we have learned about fighting defects. In *Int'l. Softw. Metrics Symp.*, pages 249–258, June 2002.

[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Int'l. Symp. Softw. Test. Anal.*, pages 97–106, July 2002.

[31] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4), Apr. 2005.

[32] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. Roos. Time-aware test suite prioritization. In *Int'l. Symp. Softw. Test. Anal.*, pages 1–12, July 2006.

[33] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Int'l. Symp. Softw. Rel. Engr.*, pages 230–238, Nov. 1997.