AN ABSTRACT OF THE THESIS OF

Joseph Ronald Ruthruff for the degree of Honors Baccalaureate of Science in Computer Science presented on May 28, 2002. Title: Cost-Cognizant Test Case Prioritization Techniques for Regression Testing: An Empirical Study.

Abstract approved: _____

Gregg Rothermel

Regression testing is a common and necessary task carried out by software practitioners to validate the quality of evolving software systems. Unfortunately, regression testing is often an expensive, time-consuming process, particularly when applied to large software systems. Consequently, practitioners may wish to prioritize the test cases in their regression test suites to execute more important test cases earlier in the regression testing process. One common goal of this test case prioritization is to increase a test suite's rate of fault detection, a measure of how quickly faults are detected by a test suite. Recent research has begun to incorporate the practical issues of varying test costs, test criticalities and fault severities into test case prioritization techniques and metrics measuring the rate of fault detection of these techniques. However, more empirical data is needed to sufficiently examine both the metrics and techniques developed by this research. To address this need, we conducted a case study to further examine the effectiveness of various prioritization techniques that incorporate varying test costs and criticalities. Software practitioners can use the results of this study to help enhance the effectiveness of their regression testing procedures and, in doing so, improve the quality of their software.

Cost-Cognizant Test Case Prioritization for Regression Testing: An Empirical Study

by

Joseph Ronald Ruthruff

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science (Honors Scholar)

Presented May 28, 2002
Commencement June 2002

Honors Baccalaureate of Science in Computer Science project of Joseph Ronald Ruthruff presented on May 28, 2002.

APPROVED:

Dr. Gregg Rothermel, Mentor, representing Computer Science

Dr. Jon Herlocker, Committee Member, representing Computer Science

---

Alexey Malishevsky, Committee Member, representing Computer Science

---

Dr. Jon Hendricks, Dean of the University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, University Honors College. My signature below authorizes release of my project to any reader upon request.

Joseph Ronald Ruthruff, Author

ACKNOWLEDGMENTS

This thesis, along with my interest in scientific research, would never have been possible without Dr. Gregg Rothermel. I am extremely grateful for the opportunity he gave me almost two years ago to serve as an undergraduate research assistant. He has exhibited endless patience with me when my course work began to consume the time I could put into his research. Furthermore, he has provided both helpful feedback during the revision of my thesis, as well as invaluable input into my career plans. If life leads me to a faculty position in academia, then I will certainly be striving to reach the bar he has set as a researcher, teacher and mentor.

I am also grateful to Dr. Jon Herlocker, both for agreeing to serve on my committee and for his capable instruction as an educator. The two courses I have taken from Dr. Herlocker have easily proven to be the most valuable in my four years at Oregon State University, and I credit this to his dedication and commitment to teaching. Without the constructive changes he made to these courses, and his exceptional teaching ability, I would not have experienced the success in OSU's Computer Science program that I have.

My thanks go to Alexey Malishevsky, who has served not only as a valuable committee member but as a capable leader during my two years of work under Dr. Rothermel. I am grateful for his willingness to provide a helping hand when necessary and for his diligence in completing the necessary tasks of our work. Alexey is also responsible for implementing a variety of the automated tools used in this project, including the test case prioritization tools, the $APFD_c$ computational tools and some of the tools used to measure the cost-cognizant metrics of our test subject.

This thesis is distilled from a portion of the research being conducted by Dr. Rothermel. Consequently, I have a number of people to thank for their contributions

to the work that has led to this project. Dr. Sebastian Elbaum at the University of Nebraska-Lincoln gathered the fault index, DIFF and binary DIFF data used in this case study. Brian Davia was responsible for creating the `Empire` test suite. Xuemei Qiu spent countless hours contributing to the organization of the `Empire` test subject and was responsible for computing the fault matrices and gathering the timing data I used in this case study. Amit Goel implemented a variety of tools to calculate the test case criticalities and fault severities of a test subject.

Thanks to Dr. Margaret Burnett, who was kind enough to take me aside during my sophomore year and discuss with me both graduate school and the opportunities available in research. When instructing a class, many professors view their students simply as individuals who must be taught so that they might continue to pursue their research at an academic institution. Dr. Burnett was the first professor who actively took an interest in and cared about what I was going to do with what she taught me. She encouraged me to think about taking my education one step further, and were it not for her inspiration, I might not have ever considered research as something to do with my life.

Finally, I am indebted to my parents, Ron and Kathy Ruthruff, who have served as my mentors, role models, counselors, parents and friends for my entire life. They are unquestionably the two strongest and greatest individuals I know, even if I don't always tell them so.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

COST-COGNIZANT TEST CASE PRIORITIZATION TECHNIQUES FOR REGRESSION TESTING: AN EMPIRICAL STUDY

Chapter 1

INTRODUCTION

The software industry employs a large amount of resources in testing software. In many cases, this testing is ad-hoc, expensive and time-consuming. A common consequence is unreliable software that is difficult to maintain, arduous to modify and comes at an unnecessarily high cost.

One widespread testing strategy employed by software engineers is to group the individual test cases for a particular software system into a set of test cases called a *test suite*. This test suite, or a portion of the suite, is run during the development of the software for the purpose of detecting faults within the system. A fault is an observed deviation from the specified functionality of a software system [5].

Often times, a developer will make changes to a software system after previously testing, and possibly releasing, that system. At this time, the developer is faced with the possibility that his or her changes will unfavorably affect the system, causing it to behave erroneously in cases where it formerly did not. We call such an occurrence a regression fault. Regression testing is a process software engineers carry out for, among other things, the purpose of detecting regression faults [10]. Regression testing is a widespread process within the software industry because (1) during the development process, as changes are made, several versions of a software system may be internally released among developers and (2) after the development process, many

upgrades or patches to a system may be released to consumers. Upgrades to a system add new features that were not included in previous releases of the system, while patches correct faults in the software system that were either not discovered or not resolved during the system's development.

There are many ways to implement regression testing. A developer could simply rerun all the test cases that were run before the modification of the previous version of the software system; this approach is often referred to as the *retest-all* approach [20]. However, this may involve unnecessary work when changes affect only a portion of a system. Furthermore, some test suites are large and can take days or weeks to fully execute. For example, Elbaum et al. report that one industrial test suite of a system of about 20,000 lines of source code requires a full seven weeks to run [9].

At times, therefore, it is desirable to (1) run only a subset of the test suite, which can be chosen by using regression test selection techniques [7, 20, 29]; or (2) permanently reduce the number of test cases by eliminating redundant test cases from the test suite, which can be accomplished by using test suite reduction techniques [6, 15, 25]; or (3), prioritize the test cases within the test suite in a particular order to maximize some goal. *Test case prioritization techniques* [8, 9, 10, 14, 30, 31, 33] are often used to implement this latter strategy by ensuring that those test cases that best accomplish this goal are executed early in the regression testing process. Potential goals of prioritization techniques can range from detecting faults as early as possible during the testing process (maximizing *rate of fault detection*) to exercising certain subsets of the system based on their frequency of use or previous history of failure, or achieving code coverage as quickly as possible [31]. It is these prioritization techniques that attempt to maximize rate of fault detection that we focus on in this thesis.

The most commonly cited goal of test case prioritization is to increase a test suite's rate of fault detection. "Fault detection", however, is a relative term because certain faults in a software system can have a greater severity than other faults. For example, a fault that blocks system execution may (all other things being equal) be considered much more severe than a simple cosmetic fault in the system's appearance to the user. Using a different metric, a fault in software functionality that a user makes use of multiple times each day might be considered more severe than a fault in functionality that a user exercises once a month. Furthermore, some test cases have greater costs than other test cases. A test case that requires one hour to fully execute may (again, all other things being equal) be considered much more expensive to run than a test case that requires one minute to execute. Finally, some test cases may be intrinsically more critical to execute than other test cases. A test that exercises a software system operation commonly used by software users may be considered more critical than a test that exercises an operation that users rarely utilize.

To capture such factors, while providing a way to compare test orders for relative cost-effectiveness, a metric is needed. Elbaum et al. [9] present such a "cost-cognizant" metric, $APFD_c$, which can be used to measure rate of fault detection by numerically rewarding (prioritized) test case orders, taking into account variance in fault costs, test costs and test criticalities. This cost-cognizant metric, however, requires that prioritization techniques be adapted to consider the cost and criticality (determined by the performance goal the technique is attempting to meet) of test cases when prioritizing a test suite. Some cost-cognizant prioritization techniques are suggested in [9].

Although variance in the cost of tests and the severity of faults is often encountered in software development, the research literature has not thoroughly examined cost-cognizant test case prioritization techniques. While [9] introduced the $APFD_c$ metric

and some cost-cognizant techniques, more empirical data is needed to sufficiently examine both the metric, and the techniques. Furthermore, there has been little formal research regarding the use of operational profiles, which measure the frequency of use of each operation within a software system, as one approach to assigning test criticalities to the test cases of a test suite, and fault severities to the faults of a software system.

We therefore conducted a case study to investigate the use of cost-cognizant prioritization techniques. Using a software system with several releases, containing various regression faults, and a regression test suite for this system, we gathered an operational profile for the system, calculated the test costs and criticalities of each test case within the suite and measured the severity of each known fault. We then prioritized the test cases of the system's test suite using a variety of prioritization techniques and measured the $APFD_c$ of each prioritized suite. Through this design, we observed that a random ordering of test cases performed comparably, in terms of cost-cognizant rate of fault detection, to one category of cost-cognizant prioritization techniques, while a separate category of cost-cognizant techniques consistently performed better than a randomized test suite. Furthermore, we noted that all prioritized test suites were more effective at revealing faults that are difficult to detect than a randomly ordered test suite.

The rest of this thesis is organized as follows. Chapter 2 provides further background regarding the problem of test case prioritization, the $APFD_c$ metric and test case prioritization techniques. Chapter 3 describes the software system used as a case study object, outlines the design of the study, and then presents the results. Chapter 4 discusses further implications of the results of the study. Chapter 5 draws final conclusions and discusses possible future work.

Chapter 2

BACKGROUND

Before describing the software system with which this case study was performed, we provide details regarding the test case prioritization problem, the $APFD_c$ metric and the prioritization techniques used in this case study.

2.1 The Test Case Prioritization Problem

Rothermel et al. [31] formally define the *test case prioritization problem*:

> *Given*: T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real numbers.
>
> *Problem*: Find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')\ [f(T') \geq f(T'')]$.

To elaborate on this definition, PT represents the set of all possible prioritizations (orders) of the test suite T, and f is a function that takes a test suite and generates an *award value* based on that test suite's ability to meet some performance goal, such as rate of fault detection. The definition assumes that high award values are more desirable than lower award values.

As stated earlier, test case prioritization techniques are usually developed to maximize a test suite's ability to meet some performance goal. However, there exist

various performance goals a software quality engineer may wish to meet when prioritizing a test suite for regression testing. Some of these possible goals are qualitatively described by Rothermel et al. [31]:

- Software engineers may wish to increase the probability of revealing the faults in a software system as early in the regression testing process as possible. This is referred to as a test suite's rate of fault detection, and, in practice, is a common goal of prioritization, as detecting faults is typically considered the primary purpose of testing a software system.

- Software engineers may wish to cover as great a percentage of a software system's (coverable) source code as possible in the shortest amount of time. This is referred to as a test suite's rate of code coverage. Increasing the rate of code coverage of a software system is a common goal in practice because software engineers may be required by regulating agencies to use testing to cover a certain percentage of a software system's source code. Consequently, it may be desirable to accomplish this task as quickly as possible.

- Software engineers may wish to use testing to increase their confidence in the reliability of the software system at as fast a rate as possible. This goal might be accomplished, for example, by using testing to exercise those features of a system that have tended to fail in the past early in the regression testing process.

- Software engineers may wish to detect high-risk faults, i.e. those faults that a software user is most likely to encounter, or that are the most safety-critical, as early in the testing process as possible so that these faults can be quickly resolved.

- Software engineers may wish to execute modified source code, or specific sections of modified code, early in the regression testing process, therefore attempting to detect faults related to these code changes as quickly as possible.

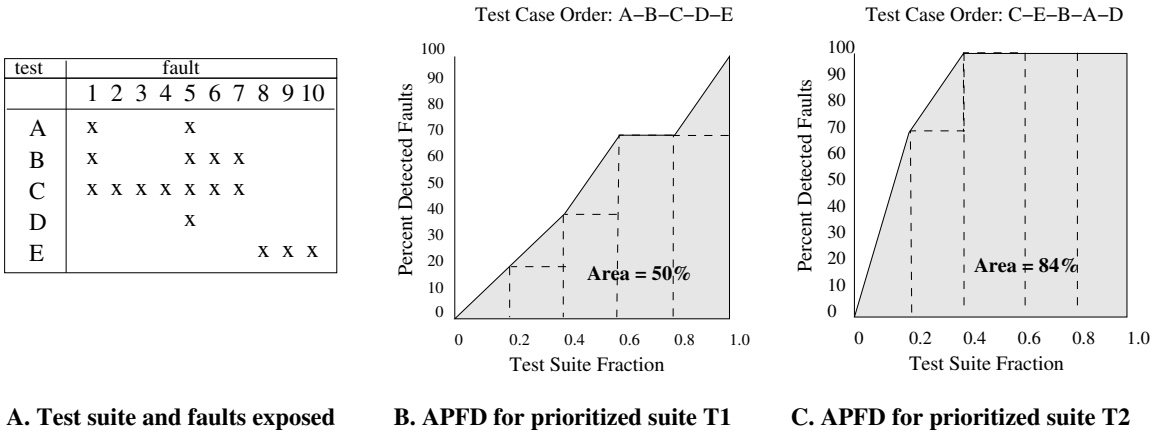
In order to effectively judge a prioritization technique's success in meeting any performance goal, however, it is necessary to provide a quantitative means of measuring that technique's effectiveness. In the definition of the test case prioritization problem provided above, this quantification takes the form of a function *f*. In this thesis, that quantification is the $APFD_c$ metric.

2.2 Measuring Rate of Fault Detection

As mentioned earlier, $APFD_c$ is a cost-cognizant metric for quantitatively measuring a prioritization technique's rate of fault detection, given some software system with a regression test suite. However, the foundation for $APFD_c$ was laid by a preliminary metric, $APFD$, which Rothermel et al. introduce in [31]. Therefore, to ensure an understanding of $APFD_c$, we first review $APFD$.

2.2.1 The $APFD$ metric

$APFD$ is a weighted *average of the percentage of faults detected* over a run of a test suite, using a value ranging from 0 to 100 to measure the rate of fault detection. Higher $APFD$ values correspond to higher rates of fault detection. Clearly, in practice, the faults in a software system are not known at the beginning of the regression testing process, and in fact, regression testing will often fail to detect all the faults in a software system. However, when conducting case studies where test suites are executed in an attempt to detect known faults, the $APFD$ metric provides a quantitative estimate of a test suite's rate of fault detection. We can then use this metric



A. Test suite and faults exposed **B. APFD for prioritized suite T1** **C. APFD for prioritized suite T2**

FIGURE 2.1: Example illustrating the *APFD* metric.

to compare the effectiveness of various prioritization techniques in yielding test suites with a high rate of fault detection.

The *APFD* metric is calculated by computing the *percent-of-faults-detected* per *unit-of-test-suite-executed*. To illustrate the calculation of their *APFD* metric for some test suite, Rothermel et al. provide examples in [31]. These examples focus on a fictional software system, presented in Figure 2.1.A, with 10 known faults and a regression test suite containing five test cases.

Example 1: Suppose the test suite is prioritized such that the test cases are ordered **A**–**B**–**C**–**D**–**E** to form the prioritized test suite TS_1. When this test suite is executed, test case **A** will expose two faults, thereby exposing 20% of the faults in the system after 0.2 of test suite TS_1 has executed. Two additional faults will be exposed by test case **B**, thereby exposing 40% of the faults after 0.4 of TS_1 has executed. Three additional faults are exposed by test case **C**, followed by no additional faults by test case **D**, and one additional fault by test case **E**. Figure 2.1.B graphically illustrates the *percent-of-faults-detected* per *unit-of-test-suite-executed*. In

this graph, the dashed rectangles indicate the percentage of faults detected at each interval of test suite completion (signified by the completion of a test case). A solid line connects the upper-left-hand corners of each rectangle to form an interpolated curve of the percentage of faults detected during the lifecycle of the test suite. Calculating the area under the curve yields the weighted average percentage of the faults detected ($APFD$) by the test suite. In this case, the $APFD$ of the prioritized test suite **A**–**B**–**C**–**D**–**E** is 50%.

Example 2: Now consider the test suite ordering **C**–**E**–**B**–**A**–**D**, identified as the prioritized test suite TS_2. By plotting the curve of the *percent-of-faults-detected* per *unit-of-test-suite-executed* for TS_2 and calculating the area under this curve, we find that the $APFD$ of TS_2 is 84%, indicating that TS_2 has a much higher rate of fault detection than TS_1. Furthermore, a close examination of Figure 2.1.A indicates TS_2 yields an optimal ordering of the test cases, as this prioritized test suite results in the earliest detection of the most faults. Figure 2.1.C depicts the $APFD$ for TS_2.

Elbaum et al. provide a formulaic presentation of the $APFD$ metric in [10]. Let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T' of T that reveals fault i. The $APFD$ for the test suite T' is given by the equation:

$$APFD = 1 - \frac{TF_1+TF_2+...+TF_m}{nm} + \frac{1}{2n} \tag{2.1}$$

2.2.2 The $APFD_c$ metric

The $APFD$ metric presented above may appear to provide a reasonable quantitative function f for assessing solutions to the test case prioritization problem. However, the metric assumes that (1) all test cases have equal costs and (2) all faults are equally severe. In practice, software engineers are often unable to make these assumptions.

Some test cases may take days to complete, while others may only take hours. Additionally, a fault in one area of the system may be considered fatally severe, while a fault in a separate area of the system may be considered only cosmetically severe. Under these conditions, the $APFD$ metric may mislead software engineers concerning the effectiveness of a given prioritized test suite, or concerning a comparison between two separate prioritized test suites. Elbaum et al. present the following examples in [9] to illustrate this possibility.

Example 1: Consider the software system and corresponding test cases introduced in Figure 2.1.A, the test case ordering **A**–**B**–**C**–**D**–**E**, depicted by Figure 2.1.B and identified as TS_1, and a new test case ordering **B**–**A**–**C**–**D**–**E**, identified as prioritized test suite TS_1'. TS_1 and TS_1' each have an $APFD$ of 50%; although the two prioritized test suites reveal different faults at different times, the total number of faults revealed after each test case is identical. However, suppose that test case **B** requires two hours to execute, while test case **A** requires only a single hour, thus making test case **B** twice as costly. Under this scenario, TS_1 is preferable to TS_1' because the former ordering detects more faults—specifically, the first four faults—in a shorter amount of time than the latter ordering. However, the $APFD$ metric does not distinguish between the two test case orderings.

Example 2: Suppose the five test cases of the same software system have equal costs, but that fault 1 has a severity of $2k$ while faults 2-10 have a severity of $1k$. In considering the test case orderings TS_1 and TS_1' from Example 1, we observe that TS_1 detects one very severe fault and one less severe fault after the first test case. However, TS_1' only detects two less severe faults after the first test case. The $APFD$ metric again fails to distinguish between the two test case orderings, even though our intuition tells us TS_1 is preferable to TS_1'.

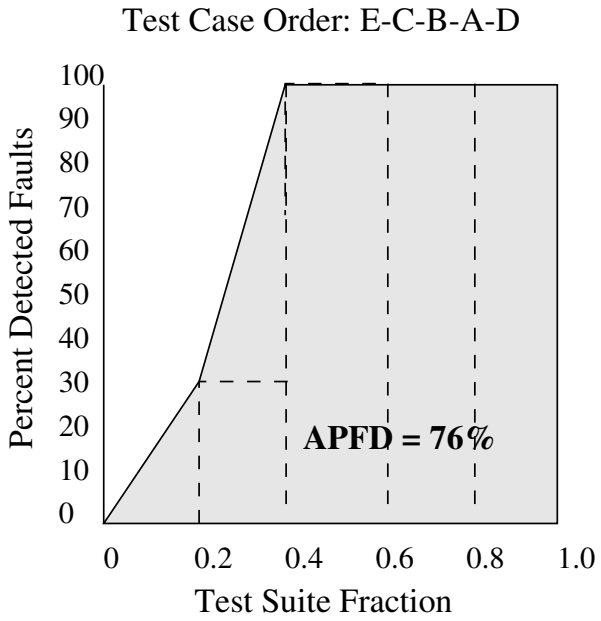


FIGURE 2.2: *APFD* for Example 3.

Example 3: As a third example, consider the "optimal" ordering **C**–**E**–**B**–**A**–**D** of the same software system, identified as TS_2, that was presented earlier, and a new ordering **E**–**C**–**B**–**A**–**D**, identified as TS_2'. Figure 2.2 shows the standard *APFD* of TS_2' to be 76%, making TS_2, with an APFD of 84%, appear to be a better prioritization of the test suite. However, suppose that in this software system, while the ten faults each have equal severity, the test cases **A**, **B**, **D** and **E** take one hour to execute while test case **C** takes ten hours to execute. Under this scenario, TS_2 takes ten hours to reveal seven faults, resulting in a weighted average of a little over one fault per hour. However, because TS_2' reveals three faults in the first hour, our intuition tells us that TS_2' is the better test case ordering. In this case, the standard *APFD* metric appears to favor the wrong prioritized test suite.

Example 4: Our final example presents a scenario where both the test costs and fault severities of the software system differ, as is often encountered in practice. Recall

Example 1, which considered the test case orderings TS_1 and TS_1', and where test case **B** required twice the time to execute as did the other four test cases. In this situation, our intuition told us the test case ordering of **A**–**B**–**C**–**D**–**E** in TS_1 was the preferable prioritized test suite—in terms of rate of fault detection. However, if test case **B** reveals faults that are more severe than those faults revealed by test case **A**, we may consider TS_1' to be the desirable prioritization, despite the extra time required to execute test case **B**. For simplicity, let us assume the units of test costs and fault severity are equivalent. Let test case **B** have a test cost of 2, while the other four test cases have a test cost of 1. If the faults revealed by test case **B** (faults 6 and 7) have a severity greater than 2, then we would consider TS_1' to be the desirable prioritization, because it detects a greater number of fault severity units per test cost unit spent. However, the standard $APFD$ metric would consider the two test case orderings to be equivalent in terms of rate of fault detection.

These four examples raise the possibility that the standard $APFD$ metric, which ignores varying test costs and fault severities of a software system, may be an inadequate function f when addressing the test case prioritization problem. This possibility must be considered because varying test costs and fault severities are often issues that may need to be taken into consideration during the regression testing of a software system. Consequently, Elbaum et al. propose in [9] that a rate of fault detection metric *reward test case orders proportionally to their rate of "units-of-fault-severity-detected-per-unit-test-cost".* They present the cost-cognizant metric $APFD_c$, an adaption of the standard $APFD$ metric, to address this issue.

The cost-cognizant $APFD_c$ metric requires two modifications to the standard $APFD$ metric; both of these modification can be expressed in terms of the graphical $APFD$ plots, such as Figures 2.1 and 2.2. First, the horizontal axes of these graphs are now measured in terms of "Percentage of Total Test Cost Incurred", as opposed

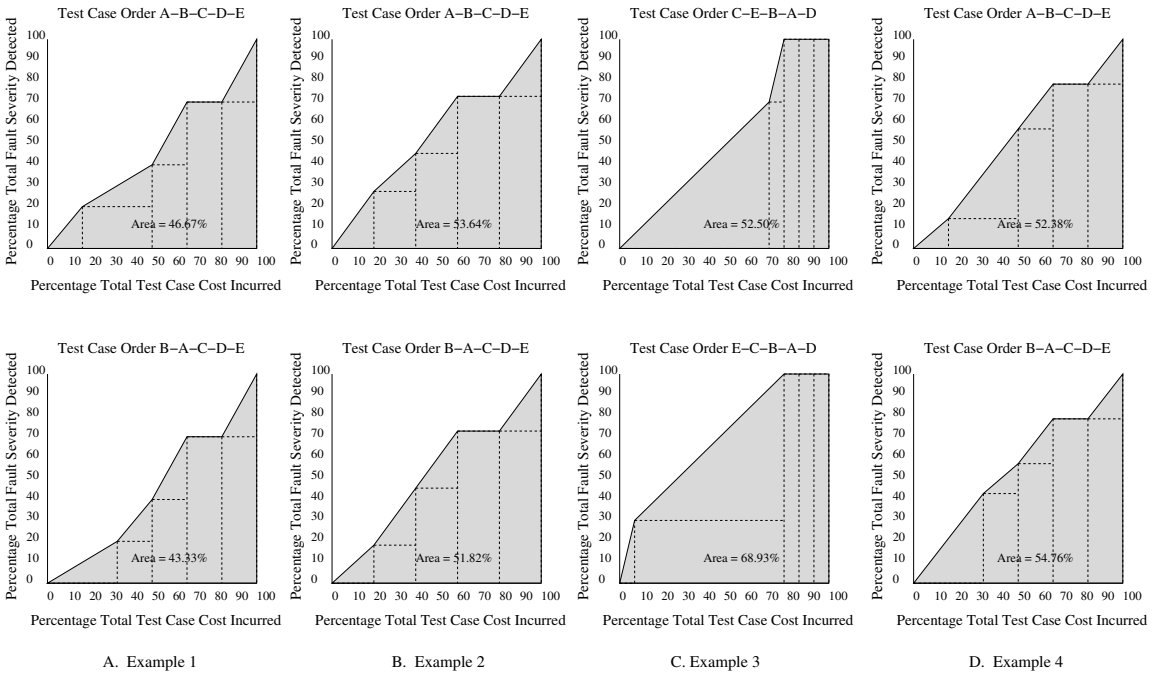


FIGURE 2.3: Examples illustrating the $APFD_c$ metric.

to the percentage of the test suite that has completed. This means that a test case with a high test cost will assume a greater horizontal portion of the graph than a test case with a low test cost. Second, the vertical axes of these graphs are now measured in terms of "Percentage of Total Fault Severity Detected", rather than in terms of the percentage of detected faults. Again, this results in the detection of a severe fault assuming a larger vertical portion of the curve than the detection of a less severe fault.

Figure 2.3 graphically presents the $APFD_c$ values of the four examples presented earlier in this subsection. The first column of plots, Figure 2.3.A, represents Example 1, the second column, Figure 2.3.B, represents Example 2, and so on. In each example, the $APFD_c$ metric favors the test case that our intuition supported.

Finally, [9] presents a quantitative description of the $APFD_c$ metric. Let t be a test suite containing n test cases with costs $t_1, t_2, ..., t_n$. Let F be a set of m faults revealed by T, and let $f_1, f_2, ..., f_m$ be the severities of those faults. Let TF_i be the first case in an ordering T' of T that reveals fault i. The (cost-cognizant) weighted average percentage of faults detected during the execution of T' is yielded by the equation:

$$APFD_c = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{2.2}$$

Test Costs and Fault Severities

In Example 4 of the previous subsection, we described a scenario where the software system in question contained both varying test costs and fault severities. For simplicity, we assumed that the units-of-measurement for both these attributes were equivalent. In practice, however, there are various methods of measuring test costs and fault severities [22, 26, 32]. For example, software engineers might measure test cost based on the number of seconds, minutes or hours the test case takes to execute, the amount of CPU time it requires, the amount of setup time required for the test case, and so on. Likewise, to measure fault severity, software engineers might consider the likelihood a user will encounter the fault, the amount of liability the company would incur (particularly applicable in safety-critical industries) should the fault adversely affect software users, the time required to locate and resolve the fault, and so on.

$APFD_c$ supports any (quantitative) criterion used to assign test costs and fault severities. However, if, for example, test costs are assigned on a five-level exponential scale and fault severities are assigned on a five-level linear scale, then clearly test costs will have a much greater impact on test case ordering than fault severities because,

as the level of each metric increases, test costs will tend to dominate fault severities. It is therefore the responsibility of the software engineer to ensure that the metrics used to measure test costs and fault severities are appropriate to the software system. It is also notable that if the test costs and criticalities of each test case in the test suite are identical, and the fault severities of each fault in the software system are identical, $APFD_c$ reverts to $APFD$. In Section 3.2.1, we discuss the metrics used to measure test costs and fault severities, as well as test criticalities, in this case study.

2.3 Test Case Prioritization Techniques

While a broad range of test case prioritization techniques can be developed to satisfy particular performance goals, the most common performance goal to be met is to increase the rate of fault detection of a regression test suite. Consequently, various test case prioritization techniques have been proposed to meet this goal. Furthermore, empirical studies of these techniques have shown that even primitive, inexpensive prioritization techniques can result in a higher rate of fault detection than a random or impromptu ordering of test cases [10, 31].

In this work, we focus on nine specific test case prioritization techniques, labeled T1-T9 and listed in Table 2.1. The first technique is used for the purpose of evaluating the rate of fault detection of the last eight experimental techniques. Each of these techniques is described in detail below:

T1: Random test case ordering (random). The first prioritization technique randomly orders the test cases in the test suite. This technique serves as an experimental control.

T2: Total function coverage prioritization (fn-total). By inserting instrumentation code into the original source code of the software system, we can trace the

Code	*Mnemonic*	*Description*
$T1$	random	randomized ordering
$T2$	fn-total	prioritize in order of total functions covered
$T3$	fn-addtl	prioritize in order of coverage of functions not yet covered
$T4$	fi-fn-total	prioritize in order of sum of fault indices of covered functions
$T5$	fi-fn-addtl	prioritize in order of sum of fault indices of functions not yet covered
$T6$	diff-fn-total	prioritize in order of total syntactic differences in functions between versions
$T7$	diff-fn-addtl	prioritize in order of total syntactic differences in functions not yet covered
$T8$	bdiff-fn-total	prioritize in order of total modified functions
$T9$	bdiff-fn-addtl	prioritize in order of coverage of modified functions not yet covered

TABLE 2.1: A listing of prioritization techniques.

functions each test case executes. When ignoring test costs and criticalities and fault severities, fn-total prioritizes the test cases in a test suite by using this trace data to sort the test cases in decreasing order of the number of functions each executes. Consequently, those test cases that execute the most functions are placed at the beginning of the test suite. If multiple test cases execute the same number of functions, those test cases are arranged in the order in which they originally appeared.

To make a cost-cognizant adjustment to this technique, however, we must account for the fact that the cost and criticality of each test case in a test suite may vary. Therefore, instead of choosing the next test case for the prioritized test suite based

on the number of functions executed by that test case, for each test case j, given the number of functions i executed by that test case, we compute an *award value* $\frac{criticality_j}{test\ cost_j} \times i$, where $criticality_j$ is the estimated criticality of running test case j and $test\ cost_j$ is the cost of running test case j. After computing the award value of each test case in the test suite, the test cases are sorted in decreasing order of these award values, resulting in a prioritized test suite. Using this methodology, the test cases with the greatest ratio of fault-severity-detected per unit-test-cost are rewarded.

Note that if $criticality_j$ and $test\ cost_j$ are equal for each test case j in a test suite, the cost-cognizant technique will revert to a non-cost-cognizant technique.

T3: Additional function coverage prioritization (fn-addtl). Similar to total function coverage prioritization, additional function coverage prioritization requires trace data reporting the functions executed by each test case. However, whereas total function coverage prioritization sorts test cases strictly on the number of functions they execute, additional function coverage prioritization detects those functions that are not yet executed by any test case of the developing prioritized test suite. The next test case is then greedily chosen based on its history of executing the unexecuted functions of the software system. That is, the next test case chosen is the test case that adds the most additional function coverage to the system. When no more coverage can be added, each function's coverage is reset and the remaining test cases are then recursively added to the test suite.

As with total function prioritization, to adapt additional function prioritization into a cost-cognizant technique, for each test case j, given i, the number of functions executed by the test case that have not yet been covered by the tests prioritized so far, we compute the award value $\frac{criticality_j}{test\ cost_j} \times i$, where $criticality_j$ is the estimated criticality of running test case j and $test\ cost_j$ is the cost of running test case j.

T4: Total fault index (FI) prioritization at the function level (fi-fn-total). Certain measurable attributes of a function can be associated with the probability this function contains a fault, also known as its fault proneness [1, 3, 4, 19, 21]. With respect to regression testing, we are interested in the probability that certain modifications to a software system will cause regression faults. Using this information, we can calculate the fault proneness of each function and prioritize test cases based on the likeliness that the functions each test case executes contain a fault.

Previous studies [12, 24] have shown a *fault index* to be an effective metric for measuring the fault proneness of a function. To calculate the fault index of a function, a number of steps must be fulfilled [10]. First, a measurable set of function attributes must be derived, and then acquired from each function within the software system. Many factors, such as the programming language of the system, play a vital role in determining the attributes that impact the fault proneness of a function.

For this study, we used fault index data calculated by Elbaum for our test subject. As the software system used in our case study is written in the C programming language, Elbaum used the framework provided in [11] to accomplish this task. After measuring the occurrences of these attributes in the system, the second step was to standardize the metrics using the results from the baseline version. This step was performed for purposes of comparison with later versions of the system on which we were intending to perform regression testing. Third, principal components analysis [18] was performed to reduce the set of standardized metrics from the system, obtained in the second step, into a smaller set of domain values. This was performed for the purpose of simplifying the dimensionality of the problem and removing colinearity of the metrics. Finally, this smaller set of domain values corresponding to individual functions in the system were weighted by their variance and applied to a linear function to obtain a fault index for that function.

Given fault indices for a base version P and a subsequent version P' of P, the next task is to generate *regression fault indices* for the purpose of regression testing P'. To complete this objective, we require the additional information of a function-by-function comparison between the indices P' against those corresponding indices of the base version P. Completing this comparison yields the *regression fault proneness* of each function in P' in the form of a new regression fault index for that function, determined by analyzing the complexity of the changes to that function.

The total fault index coverage prioritization technique is similar to total function coverage prioritization. The sum of the regression fault indices is computed for each function i that is executed by each test case j, and these test cases are sorted in decreasing order by these sums. Consequently, the first test case of the prioritized test suite is that test case with the greatest sum of regression fault indices.

To make a cost-cognizant adjustment to this technique so that it handles scenarios where the costs and criticalities of the test cases in a test suite vary, the sum of the regression fault indices is not computed directly. Instead, for each test case j, the award value $\frac{criticality_j}{test\ cost_j} \times \sum_{k=1}^{i} fi_k$ is computed, where i is the total number of functions executed by test case j and fi_k is the regression fault index of a function k executed by test case j. As before, the test cases are sorted in decreasing order of these award values to form a prioritized test suite. This procedure still rewards those test cases with a high fault-severity-detected per unit-test-cost. However, using total fault index prioritization, the fault-proneness of each function executed by a test case is taken into consideration as well.

T5: Additional fault index (FI) prioritization at the function level (fi-fn-addtl). The additional fault index prioritization technique is similar to that of the additional function coverage prioritization technique. During execution, the algorithm maintains information listing those functions that are executed by at least one test

case in the current test suite. When adding a new test case j to the test suite, for each test case, the algorithm computes the sum of the regression fault indices, which have already been computed by the cost-cognizant method outlined in the total fault index prioritization technique description, for each function i executed by the test case, except those functions that have already been covered by at least one test case of the current test suite. The test case with the highest sum of these regression fault indices is then the next test case added to the suite. Furthermore, just as with the additional function coverage prioritization technique, when no more coverage can be added, each function's coverage is reset and the remaining test cases are then recursively added to the test suite.

As with total fault index prioritization, to adapt additional fault index prioritization into a cost-cognizant technique, for each test case j, we compute the award value $\frac{criticality_j}{test\ cost_j} \times \sum_{k=1}^{i} fi_k$, where i is the total number of functions executed by test case j that have not yet been executed by a test case in the current test suite and fi_k is the regression fault index of an uncovered function k executed by test case j.

T6: Total DIFF prioritization at the function level (diff-fn-total). Because the computation of regression fault indices for a software system, particularly a large system, can be complicated, and because the tools used to compute fault indices are not widely accessible, a software engineer may prefer to use a simpler alternative, such as total DIFF prioritization. "Diff" tools are widely accessible to software engineers, and can be used to compute the syntactic differences between two source code files, or in our case, two versions P and P' of a software system. These "diff" tools can also output the number of lines changed in a particular section of a system, such as a function.

Total DIFF prioritization for a version P' of P uses a "diff" tool to compute the syntactic differences between each function i in the software system. The number of

source code lines that have been inserted, removed, or modified measures the degree of change between these functions. Total DIFF prioritization then behaves very similar to total fault index prioritization, except that to compute the award value of test case j, instead of summing the regression fault indices of each function k executed by test case j, total DIFF prioritization sums the number of lines inserted, removed or modified from each function k executed by test case j.

To apply a cost-cognizant adjustment to this technique, the award value of each test case j is $\frac{criticality_j}{testcost_j} \times n_k$ where n_k represents the degree of change in function k across two versions P, P'.

T7: Additional DIFF prioritization at the function level (diff-fn-addtl). This technique incorporates feedback concerning the current coverage achieved by the developing test suite, in a manner similar to additional fault index prioritization, into the total DIFF prioritization technique, while continuing to factor in the cost-cognizant methods of criticalities and costs.

T8: Total binary-DIFF prioritization at the function level (bdiff-fn-total). Instead of tracking the number of syntactic changes in each function across a version P' of P, an even simpler (and possibly more effective) prioritization technique records whether or not each function across these two versions has been modified. Because the technique records a binary value regarding a function's modification, this technique is identified as total binary-DIFF prioritization. While we use conventional "diff" tools to determine if a function has been modified across two software versions, in practice, software practitioners could use other means, such as RCS logs or timestamps, to judge whether this level of difference occurs at a low cost. Otherwise, this technique is the same as total DIFF prioritization, including the adaptions to make the technique cost-cognizant, except that the degree of change in a function k across two version P, P' is merely the binary value regarding that function's modification.

T9: Additional binary-DIFF prioritization at the function level (bdiff-fn-addtl. Analogous to additional DIFF prioritization, additional binary-DIFF prioritization incorporates feedback regarding current coverage achieved into the total binary-DIFF prioritization technique.

2.4 Related Work

As stated in Chapter 1, despite its applicability in practice, to date, the research literature has not adequately addressed test case prioritization. However, in the following paragraphs, we overview relevant related work.

Chapter 1 discussed the presentation of the $APFD_c$ metric by Elbaum et al. in [9] and described its relevance to this work. Furthermore, earlier in this chapter, we discussed the work concerning the test case prioritization problem by Rothermel et al. in [31].[1] Finally, as described in Section 2.3, many of the prioritization techniques utilized in this case study (though not with adaptations to handle varying costs and severities) are introduced in [10, 31]. Consequently, we do not discuss these works further here.

Wong et al. discuss test case prioritization with the goal of "increasing cost per additional coverage" in [33]. One potential goal of this prioritization, which is not explicitly stated by the authors, is to increase the rate of fault detection of the test suite. The authors also limit their investigation of test case prioritization to a subset of test cases selected by a regression test selection technique; they do not offer a procedure for prioritizing the test cases remaining after the desired coverage is reached. Our study specifically addresses the prioritization of entire test suites

[1] [31] expands upon the work of an earlier conference paper, [30].

during the regression testing process. Furthermore, rather than focusing on obtaining coverage of a software system, we concentrate on detecting faults as early in the testing process as possible.

Avritzer and Weyuker provide an approach to generating ordered test cases when test suites have not been created in [2], provided operational profile data is available. Again, although the authors do not explicitly use the term "prioritization", to cover a larger proportion of the software system states likely to be reached by users, their techniques essentially generate test cases in a prioritized order to increase the likelihood that the faults likely to be encountered by users are detected early in the testing process. However, the techniques presented by the authors can only be applied to software that can be modeled by Markov chains, and are applicable only when a test suite has not already been created. The techniques utilized in this study are intended for use on a preexisting test suite, and the software system does not need to be modeled by Markov chains.

Chapter 3

CASE STUDY

Previous work [9] has introduced both new metrics and techniques for performing cost-cognizant test case prioritization. However, while [9] laid the groundwork for cost-cognizant test case prioritization, more empirical data is necessary to sufficiently examine the techniques introduced in that work. Furthermore, [9] only investigates three cost-cognizant prioritization techniques; we would like to gather empirical data concerning the effectiveness of additional cost-cognizant techniques for regression testing.

Consequently, we designed and performed a case study to investigate the incorporation of the cost-cognizant metrics test cost, test criticality and fault severity into regression testing. Our goal was to investigate the following research questions:

> **RQ1**: Can the use of cost-cognizant prioritization techniques improve the rate of fault detection of a regression test suite, in comparison to a random ordering of test cases?
>
> **RQ2**: Can the use of cost-cognizant prioritization techniques improve a test suite's ability to reveal "hard to detect faults" early in the regression testing process?

3.1 Object of Study

To conduct this case study, we used eleven versions of `Empire`,[1] a non-trivial, open-source client-server software system programmed in the C programming language. `Empire` is an infrastructure that, in previous work [28], was partially assembled by the members of the Galileo Research Group at Oregon State University (by Alexey Malishevsky, Xuemei Qiu, Brian Davia and Amit Goel).

`Empire` is a strategic game of world domination between multiple human players. Each player begins the game as the "ruler" of a country. Throughout the game, the players acquire resources, develop technology, build armies and engage in diplomacy with other players in an effort to conquer the world. Either defeating or acquiring a concession from all opposing players—meaning those players not participating in a strategic alliance—achieves victory. In addition to the players, each game of `Empire` has one or more "deities" who install, activate and maintain the game on a host server. A deity has the power to perform almost any imaginable action, such as changing a mountain into a grass plain or bestowing riches upon a nation, and is responsible for using this power in a manner to ensure the game they run is both enjoyable and fair. As might be expected, certain powers are reserved only for a deity; a player, for example, cannot change a mountain into a grass plain.

The `Empire` software system is divided into a server and a client, which are named `emp-server` and `emp-client`, respectively. After the initialization of `emp-server` on the host server by the deity, the game progressed with the players and deities using `emp-client` to connect and issue commands, which correspond to their desired actions, to `emp-server`, which processes each command issued by an `emp-client`

---

[1] http://www.empire.cx

and returns the appropriate output. We have chosen to conduct our case study on the `emp-server` portion of the `Empire` software system.

There are various reasons justifying our selection of `emp-server` as a test subject. First, `emp-server` meets our basic requirements for test subjects. Each version of `emp-server` is a large system (approximately 63,000 to 64,000 lines of non-blank, uncommented source code) that is written in the C programming language,[2] used in the real world, and has many versions available. Second, because `Empire` is distributed under the GNU General Public License Agreement, it is available at no cost. Third, `Empire` has good documentation describing the operations available to a user. Fourth, as we shall describe in Section 3.2.1, collecting an operational profile for this system is a feasible task. Finally, previous studies [28] have created test suites for `emp-server`, meaning we can apply our cost-cognizant prioritization techniques to an existing test suite in the interest of gathering empirical data concerning the incorporation of cost-cognizant metrics into regression testing.

As mentioned earlier, we used eleven versions of the `Empire` software system. Table 3.1 presents the number of functions and non-blank, uncommented lines of source code for each version, as well as the number of functions changed for each version from the previous version.[3]

---

[2] Section 3.2.2 describes why we require our test subjects to be written in the C programming language.

[3] The number of functions changed is the sum of the number of functions that were modified in or added to each version with respect to the previous version, or the number of functions in the previous version that were removed from the current version.

Version	*Functions*	*Changed Functions*	*Lines of Code*
4.2.0	1,188	—	63,014
4.2.1	1,188	51	63,014
4.2.2	1,197	245	63,658
4.2.3	1,196	157	63,937
4.2.4	1,197	9	63,988
4.2.5	1,197	101	64,063
4.2.6	1,197	32	64,108
4.2.7	1,197	156	64,439
4.2.8	1,189	52	64,381
4.2.9	1,189	12	64,396
4.2.10	1,191	16	64,457

TABLE 3.1: The `emp-server` test subject.

3.2 Design

Our research questions narrow the goal of this study to evaluating the performance of various cost-cognizant test case prioritization techniques, as measured by the $APFD_c$ metric. Consequently, to answer our research questions, for each version, we need to calculate the $APFD_c$ of the prioritized and random orderings of our test suite.

The following subsections outline the steps undertaken in this case study to facilitate the application of our prioritization techniques, and the subsequent calculation of the $APFD_c$ for various prioritized test suites for Empire.[4] First, we had to complete

[4] A graphical representation of the steps required to calculate the $APFD_c$ of a prioritized test

a number of tasks to prepare our test subject in accordance with the specifications of our case study design. Concurrently, we had to utilize existing tools to prioritize the test cases of our test suite and implement additional tools to both measure various attributes of our test subject and complete our $APFD_c$ calculations. Finally, we had to determine the manner in which to calculate our $APFD_c$ values to properly address our two research questions.

3.2.1 Test Subject Preparation

We took the following steps to prepare `emp-server` for our case study:

Obtaining a Test Suite

Various test suites for `emp-server` were created for use in previous research [28]. Of these test suites, we chose to conduct our case study using a specification-based test suite created for that earlier research, containing 1985 test cases. This test suite, created by the category partition method [27], is of a very *fine granularity*, meaning the individual test cases of the test suite each test individual units of functionality independently. We chose this test suite because it represents a test suite that might be created by testers in practice, if they were doing functional testing.

Inserting Faults into `Empire`

To measure the rate of fault detection of our prioritized test suites, we calculate the $APFD_c$ of a test suite based on how quickly, based on units of test cost, it detects units of fault severity. Consequently, to determine whether a test case detected a unit of fault severity, we must design our case study such that a test suite is attempting

---

suite appears in Appendix A.

to detect known faults. The $APFD_c$ of a test suite is then a measure of how quickly each prioritized test suite detects these known fault severity units. Clearly, in practice, the faults of a software system are not known during the regression testing process. However, by designing a case study where we are attempting to detect known faults in our test subject, we are able to empirically evaluate the cost-cognizant prioritization techniques addressed in our research question.

While `Empire` comes with extensive documentation concerning the commands available to users, it does not have adequate documentation describing the known faults in each version of the software system. Furthermore, for reasons that will be described in Section 3.2.1, we wish to have the ability to toggle faults on and off, meaning, for each faulty section of code, we desire an alternative section that is not faulty. We could attempt to locate faults distributed with each `Empire` version, but that would require us to implement a resolution to each fault. This is not a desirable strategy because, as we do not have the intricate knowledge of the system possessed by the developers, we risk unknowingly creating additional faults in the system.

Consequently, a strategy of inserting *seeded faults* into the software system was selected. A *seeded fault* is a fault that is intentionally introduced into the software system, usually for the sole purpose of measuring the effectiveness of a test suite.

To introduce seeded faults into the software system, several graduate students with at least two years of C programming experience, and no knowledge of the study, were asked to introduce faults into the software system. These students used their programming knowledge to insert faults in sections of code that a software engineer might accidently introduce a fault in practice. Furthermore, because the focus of this and previous work is the detection regression faults, the areas of code in which a student may insert a fault were limited to those that were modified from a previous version of the test subject. Each student, after identifying a section of code in which

to introduce a fault, made a copy of that section of code and surrounded each section by preprocessor statements which, depending on the definition of certain variables, toggles on either the faulty or the non-faulty code. This process was completed for versions v1–v10.

Following the precedent set by previous studies [9, 10, 31], we did not wish to base our case study upon faults that were either impossible or too easy to detect, as the former are not the target of test case prioritization and the latter, we assume, are likely to be detected by developers prior to the system testing phase we are concerned with. Therefore, all faults that were detected either by more than 20% of the test cases in our test suite, or by no test cases at all, were excluded. As illustrated by Table 3.2, which lists the percentage of test cases that reveal the faults in each version of our test subject, this criterion process yielded ten seeded faults per `Empire` version.

Obtaining an Operational Profile

Software systems consist of various *operations* that users utilize to complete the task for which the system was designed. For example, some of the operations of a word-processing software system may be text insertion, text deletion, and saving, creating and opening a document. These operations exist so that a software user can efficiently and successfully create documents using the system. Likewise, `Empire` consists of a number of operations so that the players and deities can participate in an `Empire` game. Examples of some `Empire` operations include `fire`, `announce`, `move` and `update`. These operations come in the form of commands the players and deities issue to `emp-server`. Using the documentation provided with the system, we were able to identify 182 operations in our base `Empire` version.

One strategy that may be utilized by software engineers when testing a software system is gathering an operational profile for that system. An *operational profile*

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
Fault 1	5.8	8.0	7.4	0.5	6.1	0.1	0.6	0.9	7.4	1.2
Fault 2	0.1	6.8	6.7	6.7	0.3	6.8	7.5	1.1	8.7	1.2
Fault 3	2.5	0.9	7.1	6.7	6.7	0.1	0.2	0.4	0.3	0.2
Fault 4	0.1	6.8	6.9	7.2	0.5	6.8	7.4	7.2	0.7	1.2
Fault 5	6.6	6.7	7.3	0.5	6.0	6.7	7.3	7.2	0.8	1.2
Fault 6	0.1	6.7	6.7	7.6	0.1	6.8	7.7	7.2	7.4	0.3
Fault 7	0.1	0.2	0.1	0.1	0.2	0.1	7.3	7.2	8.5	0.2
Fault 8	0.7	6.7	7.7	1.3	6.0	7.2	7.2	7.2	7.9	1.2
Fault 9	6.3	6.7	0.1	0.1	6.1	0.2	0.2	7.2	7.4	1.2
Fault 10	5.8	6.9	6.6	0.2	0.6	6.8	7.3	7.6	0.4	1.4

TABLE 3.2: Percentage of test cases that detect the faults in each version.

quantitatively lists, for each operation, the frequency with which each operation is utilized by a user. This frequency is usually relative to all other operations. Using our preceding example, in a word-processing program, the "text insertion" operation may be utilized 85% of the time, the "text deletion" operation 10% of the time, the "save document" operation 2% of the time, the "create document" operation 1% of the time, the "open document" operation 1% of the time and the "print document" operation 1% of the time. By indicating each operation's frequency of use, operational profiles enable software engineers to make a variety of decisions regarding the testing of a software system. For example, software engineers may wish to focus their testing on those operations that users utilize at least 5% of the time. Additionally, software engineers may decide that a fault in an operation a user frequently utilizes is more

severe than a fault in an operation a user rarely utilizes, regardless of that fault's impact on the execution of the system.

To collect an operational profile for our case study, we ran an `Empire` game with three experienced players, all of whom responded to a game announcement posted on the `Empire` newsgroup. We modified the source code of `emp-server` to record the commands issued by each player.[5] At the completion of our game, we calculated the number of times each command was issued by the players. By dividing this number by the total number of commands issued in the game, we were able create an operational profile estimating each command's frequency of use. To explain later metrics, for each operation i, we define this frequency as the *importance of operation i*.

The manner in which an operational profile is obtained and utilized is highly dependent on the software system corresponding to that profile [23]. The `Empire` software system can host a wide variety of games. For example, some `Empire` may have a small number of players, such as our three player `Empire` game, while other games may have well over 40 players. Many `Empire` games may run for months while others run for weeks or even days. Some `Empire` games see the players obtaining nuclear technology, resulting in the use of a wide variety of nuclear-related operations, while other games have technology frozen to that of the Dark Ages. The point of this discussion is that `Empire` is a software system where the operational profile gathered is highly dependent on the type of game being played, and there are numerous types of games possible. Our operational profile is merely one example out of many that could be gathered for this system.

---

[5] The players of our `Empire` game were not informed that their usage was being monitored, as that might have altered their behavior; however, all data was collected in such a manner that the anonymity of the commands issued by each player was preserved.

Creating an Operation Matrix

An *operation matrix* documents, for each test case in a test suite, the operations of the software system executed by that test case. For `Empire`, because of the fine granularity of the test cases, each test case executes only one `Empire` operation. We use the operation matrix to calculate attributes about the test cases of the test suite, such as test case criticality.

Calculating Test Case Criticality

Test case criticality is a measure of the importance of a test case. This metric is derived without the knowledge of what faults each test case might expose.

While there are many methods for deriving test case criticality, in our case study, this metric is derived from the operational profile we obtained and the operation matrix we created. In our case study, we measure the criticality of each test case as the importance of the operation, as defined earlier in this section, being executed by the test case, which we determine from our operational profile. However, in practice, the test cases of a test suite may execute a number of operations. In this scenario, the criticality of each test case is the sum of the importance of each operation executed by the test case. We can determine the operations executed by a test case by referring to the operation matrix.

Calculating Test Case Cost

As its name implies, *test case cost* is an indicator of the expense incurred by running a test case. Practitioners can use a variety of criteria to measure test case cost. For example, a software engineer may choose to measure test case cost based on the time required to execute a test case, the time required to determine, after each test case has completed execution, whether that test case revealed any faults in the

software system, or based on the amount of resources required by a test case during its execution.

In our case study, we chose to measure each test case's cost by summing the amount of time required, in seconds, to execute that test case and the amount of time required to validate that test case's output by comparing it against expected output (from a previous run). To gather this timing data, Qiu completed two runs of our test suite. During each run, she used standard UNIX system calls to record the time at the beginning and end of each test case's execution, and at the beginning and end of the process to determine whether each test case revealed any faults in the software system. After completing these two runs of our test suite, based on the timing results described above, we calculated the cost of each test case run and averaged these two results. This average is then the metric we use to represent the cost of each test case in our suite.

All `emp-server` timing-related data was gathered on a Sun Microsystems Ultra 10 with 128 MB of memory. While the timing data was being collected, our executing test suites were the only active user processes on the machines. Furthermore, to reduce the network traffic created by these processes, all writing was done into machine-local storage.

Creating a Fault Matrix

Recall from earlier in this section that we used ten faults seeded into versions v1–v10 of our test subject. To calculate the $APFD_c$ of each prioritized test suite for each of these versions, we must know whether each test case of our suite reveals any of the ten faults we seeded in each version.

To address this need, for each version v1–v10 of our test subject, Qiu gathered a *fault matrix*, which documents the faults (if any) each test case reveals in that version.

The fault matrix is used both to calculate the severity of each seeded fault, and to assist in the $APFD_c$ calculations of each prioritized test suite.

To gather a fault matrix for each version, Qiu executed numerous runs of our test suite, where the test cases were unordered. Recall from Section 3.2.1 that the ten seeded faults of each version were surrounded by preprocessor statements, allowing us to toggle each seeded fault on and off. In the first run of our test suite, all seeded faults were turned off, and the output of each test case was recorded in an output file. Qiu then executed a sequence of ten test suite runs. In the first, she turned seeded fault 1 on, and left the remainder of the faults off. By recording and comparing the output of each test case during this test suite run with the output of the very first test suite run, Qiu was able to determine whether that test case revealed fault 1. Repeating this process for the other nine seeded faults yielded the necessary information to complete a fault matrix for this version of our test subject.

Throughout this study, however, Malishevsky and Qiu noticed that due to various nondeterministic issues in running `Empire`, such as a bad transmission from the `emp-client` to the `emp-server`, a command will occasionally (approximately once per thousand test executions) fail for no apparent reason. Consequently, Qiu executed an additional ten runs of our test suite, repeating the process outlined above, to gather a second fault matrix. By performing a logical `and` operation between these two fault matrixes, she kept only those faults that were revealed by both set of test suite runs, thereby substantially reducing the possibility of erroneously recording a revealed fault due to a nondeterministic issue.

Calculating Fault Severity

As mentioned earlier, there are various methods for calculating the severity of a fault. In our case study, we chose to base a fault's severity upon the likelihood that a

user might encounter that fault during his or her use of our test subject, which we determine from our operational profile.

To calculate the severity of each fault in a version of our test subject, we begin with the operation matrix and the fault matrix for that version. By using the fault matrix, which maps faults to various test cases, and the operation matrix, which maps each test case to an operation, we are able to associate various operations with each fault. Next, using the operational profile, we sum the importance of each operation associated with a fault. This sum is then the severity we assign to each seeded fault in a version of our test subject.

3.2.2 Prioritization Tools

We required several tools to complete the case study. To insert instrumentation code into `emp-server` and generate test coverage and control-flow information, Malishevsky, Qiu and Davia used the `Aristotle` program analysis system [16].[6] The prioritization techniques outlined in Section 2.3 were implemented by Malishevsky. Goel and Malishevsky created a variety of automated tools to compute the test costs and criticalities of the test cases of our test suite, and to compute the fault severities of the faults in `emp-server`. The fault indices were created by Elbaum using information gathered from three categories of tools [11, 13]: source code measurement tools for generating complexity metrics, a fault index generator, and a comparator for evaluating each version against the baseline version. The syntactic differences between the versions of our software system were computed by Elbaum using a modified ver-

---

[6] `Aristotle` can only analyze software systems created using the C programming language, which is why we require this prerequisite of a test subject for our case study.

sion of the UNIX `diff` tool. Finally, Malishevsky used a combination of C programs and UNIX scripts, created for previous prioritization experiments, to implement our prioritization techniques and to calculate the $APFD_c$ of each prioritized suite.

3.2.3 Methodology

After preparing our test subject for the case study and creating the necessary additional tools required to obtain the cost-cognizant data and fault information for the subject, our next step was to create prioritized suites using the cost-cognizant techniques described in Section 2.3. For the randomized prioritization technique, however, to control for the influence of variance, we created 20 suites per version.

Having obtained these suites, our remaining task becomes one of calculating $APFD_c$ values. We therefore used the fault information for each test case to calculate various $APFD_c$ values of each prioritized test suite for versions v1–v10 in a manner to properly address our two research questions, as follows.

RQ1: Random Test Case Orderings versus Cost-Cognizant Prioritizations

One method of evaluating our cost-cognizant prioritization techniques is to compare these techniques to a randomized test case ordering. To perform this comparison, we measured the $APFD_c$ of the prioritized test suites for versions v1–v10. For the randomized prioritizations, we averaged the $APFD_c$ values obtained on the 20 test suites mentioned earlier. We can analyze these results to compare the performance of each prioritization technique, in terms of cost-cognizant rate of fault detection, against the average $APFD_c$ obtained for the random technique.

RQ2: "Hard to Detect Faults": Random Orderings versus Cost-Cognizant Prioritizations

To compare the performance of our cost-cognizant prioritization techniques when detecting "hard to detect faults", we measured the $APFD_c$ of the same test case orderings created to address **RQ1**, except that we considered only those faults that were detected by a certain percentage of the test cases in our test suite. We divided the faults of each version of our test subject into three categories: (1) those faults detected by only 1% of all test cases, (2) those faults detected by 0.5% of all test cases, and (3) those faults detected by 0.25% of all test cases. Using this information, we can compare the $APFD_c$ values for cost-cognizant prioritization to the average $APFD_c$ for the random test case orderings, with respect to the "hard to detect faults" of a software system.

Table 3.3 lists the number of faults that fall into each of these three categories for versions v1–v10.

3.3 Threats to Validity

This section discusses potential threats to the validity of our case study, and where possible, how we attempted to limit the impact of these threats on the validity of our results. The threats we discuss include: (1) threats to internal validity (could other factors, affecting the dependent variables of our study, be responsible for our results), (2) threats to construct validity (are the dependent variables of our study appropriate), (3) threats to external validity (to what extent could the results yielded by this case study be generalized) and (4) threats to conclusional validity (what are the limitations of the conclusions drawn from this experiment, and how could a stronger study be designed).

	All	$\leq 1\%$	$\leq 0.5\%$	$\leq 0.25\%$
v1	10	5	4	4
v2	10	2	1	1
v3	10	2	2	2
v4	10	5	5	3
v5	10	5	4	3
v6	10	4	4	4
v7	10	3	2	2
v8	10	2	1	0
v9	10	4	2	0
v10	10	3	3	3

TABLE 3.3: Number of faults per "difficulty to detect" category per version.

3.3.1 Threats to Internal Validity

One threat to internal validity lies in the number of steps and tools required in our design to calculate the $APFD_c$ of each prioritized test suite. Some of these steps, such as fault seeding, involved humans. Furthermore, many of the tools utilized were created specifically for this study. Each of these factors increases the risk of variability in our results. We implemented a variety of procedures to control and minimize these concerns. For example, the programmers that seeded faults into the versions of our test subject performed their assignment in a similar manner. Furthermore, this task was completed by multiple programmers, and each programmer had no knowledge of the faults inserted into the test subject by another programmer. In the case of our automated tools, each tool was tested on a small sample of programs with

accompanying test suites. These tools were then refined as they were tested on larger subjects.

Conducting this study with only a single test suite may also affect the internal validity of our results. Unfortunately, while multiple test suites would have been ideal, the test suite utilized in this study required months to create, rendering the production of further test suites infeasible given the time frame allotted to this study.

3.3.2 Threats to Construct Validity

In this study, we focus on a prioritized test suite's rate of fault detection. However, in practice, software engineers may wish to maximize different performance goals when prioritizing their test suites. Furthermore, the quantitative function we used to measure rate of fault detection, $APFD_c$, has two limitations. First, $APFD_c$ does not reward a test case for detecting a fault that has been previously detected. Such test cases may prove useful to software engineers in practice, however. For example, during the debugging process, a test case that reveals a fault that has already been exposed may help software engineers localize the fault, a process that is often difficult in practice [17]. In our study, the fine granularity of the test cases in our test suite helps mitigate this factor because each test case exercises a single operation. However, the $APFD_c$ metric itself is still limited because a test suite with a coarser granularity may be quite susceptible to this issue. Second, $APFD_c$ does not consider the analysis time required to prioritize the test cases of a test suite. Previous work [31], however, suggests that analysis time is small in comparison to test suite execution time, or that analysis can be performed before the critical regression testing process.

3.3.3 Threats to External Validity

Test case representativeness is an issue facing our study for two reasons. First, if the experimental techniques employed in our case study do not mimic a regression testing process practiced by industry, our results may not generalize. Second, the random ordering of test cases, used as a control in this experiment, may not mimic the baseline technique that may be employed in industry. Although it would sacrifice some internal validity, a solution to this issue may be replicating this case study using industrial systems with actual faults. A better solution may be to gather additional empirical evidence through future studies using a greater range and number of software systems.

The fault seeding process, which helped control the threats to internal validity, increases the threat to external validity because faults and fault patterns may differ from those introduced into our test subject by our fault seeders.

3.3.4 Threats to Conclusional Validity

Although we performed regression testing on ten versions of our test subject, in practice, software systems often have more than ten versions tested over their lifetimes, and possibly more than ten versions released to the public.[7] Consequently, the use of more versions would enhance the power of our case study. Unfortunately, as illustrated by Table 2, the `emp-server` versions used in our study ranged from 4.2.0 to 4.2.10, which are the most recent versions of the system available. To adequately accommodate the operations available in previous versions of this system, we would

---

[7] Chapter 1 discussed why regression testing is performed during both the development cycle and as upgrades and patches to the system are released to the users of a publicly released software system.

have been required to create an additional test suite, which, as stated earlier, would take a great deal of effort. Furthermore, each version of `emp-server` required over 80 hours to prepare. These factors limited our ability to include earlier versions of the test subject in the study.

Using statement level prioritization techniques[8] [10, 31] would have increased the strength of this study by providing additional empirical evidence concerning the effectiveness of cost-cognizant statement level techniques [9]. Unfortunately, statement level coverage was not available for `emp-server`.

Finally, as can be deduced from Table 3.2, there are a limited number of faults, for each version v1–v10, that fall under the fault exposure categories of 1%, 0.5%, and 0.25%. In fact, the versions v8 and v9 of our test subject do not have any faults that fall into the third fault exposure category. These factors may limit the power of the conclusions regarding **RQ2**.

3.4 Results

Tables 3.4, 3.5, 3.6 and 3.7 are used to represent the results of this case study, with further discussion of these results in Chapter 4. The results pertaining to each research question are discussed in the following subsections.

---

[8] These are similar to function level techniques, except their coverage operates at the level of individual source code statements rather than entire functions.

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	Mean
random	97.3	99.4	99.1	98.1	94.2	98.7	99.0	99.2	99.1	95.8	98.0
fn-total	96.7	97.0	97.2	96.4	96.6	96.7	96.7	96.8	96.7	96.5	96.7
fn-addtl	99.6	99.8	99.8	99.5	97.7	99.3	99.6	99.8	99.6	99.3	99.4
fi-fn-total	97.6	98.5	97.1	95.9	95.7	99.3	97.1	96.7	95.7	95.4	96.9
fi-fn-addtl	97.6	99.8	99.9	98.8	96.1	99.3	99.8	99.5	99.2	96.6	98.7
diff-fn-total	97.6	98.5	97.1	95.9	95.7	99.3	97.1	96.7	95.7	95.4	96.9
diff-fn-addtl	97.6	99.8	99.9	98.8	96.1	99.3	99.8	99.5	99.2	96.6	98.7
bdiff-fn-total	97.6	98.2	97.3	96.1	95.9	97.8	97.1	96.7	95.7	95.9	96.8
bdiff-fn-addtl	97.6	99.8	99.9	98.8	96.1	99.3	99.7	99.9	99.2	96.6	98.7

TABLE 3.4: $APFD_c$ values for cost-cognizant prioritization techniques.

3.4.1 RQ1: Random Test Case Orderings versus Cost-Cognizant Prioritizations

Table 3.4 displays the $APFD_c$ values of the random test case orderings and the cost-cognizant prioritized test case orderings for v1–v10.

As can be seen, the $APFD_c$ values of this table indicate that test suites prioritized by the cost-cognizant implementations of additional function coverage, additional fault index coverage, additional DIFF coverage and additional binary DIFF coverage—hereafter referred to as the additional coverage prioritization techniques—always outperform a random test case ordering. However, the cost-cognizant implementations of total function coverage, total fault index coverage, total DIFF coverage and total binary DIFF coverage—hereafter referred to as the total coverage prioritization techniques—do not always outperform a random test case ordering. Of particular

note, all prioritized test suites from the total coverage techniques failed to outperform the randomized test suites for versions v2, v3, v4, v7, v8 and v9, and at least one total coverage technique failed to outperform the randomized test suites in v1, v6 and v10. However, there were cases, such as v5, in which every test suite prioritized from a total coverage technique outperformed, based on $APFD_c$ values, the average of the random orderings of the test suite.

3.4.2 RQ2: "Hard to Detect Faults": Random Orderings versus Cost-Cognizant Prioritizations

Table 3.4 depicted the results of our cost-cognizant prioritization techniques when considering all faults, where all $APFD_c$ values exceeded 90. Furthermore, the observed differences between $APFD_c$ values were not large. Tables 3.5, 3.6 and 3.7[9] build on these results by illustrating both the effectiveness of our cost-cognizant prioritization techniques in revealing faults that are increasing difficult to detect as well as the effectiveness of the random prioritization technique in this task.

Table 3.2 illustrated that the majority of the faults in the versions of our test subjects were detected by between 6%–8% of the test cases in our test suite, except for v10, where the majority of the faults were detected by approximately 1.2% of all test cases. These averages of 6%–8% are quite a bit higher than the 1% cutoff, above which we begin to exclude faults. As Tables 3.5, 3.6 and 3.7 show, as the faults grew increasingly more difficult to detect, the difference in the performance of the random test case orderings and the prioritized test case orderings for each version became more pronounced. Moreover, every prioritization technique generally began

---

[9] Some cells of these tables contain '- - -'. This indicates that there were no faults meeting the criteria of these categories—refer to Table 3.3—meaning the $APFD_c$ values are not displayed.

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	Mean
random	80.8	88.0	40.2	66.7	80.0	73.0	89.5	89.8	88.9	75.2	77.2
fn-total	92.5	79.6	76.8	80.1	95.1	82.0	86.7	89.7	84.4	83.4	85.0
fn-addtl	97.5	90.5	84.7	90.7	91.9	83.5	93.7	98.1	95.2	97.8	92.4
fi-fn-total	86.6	83.5	76.9	80.2	94.8	82.3	87.9	90.0	84.5	64.2	83.1
fi-fn-addtl	86.6	99.7	97.8	82.2	97.0	80.0	98.1	99.7	87.5	64.2	89.3
diff-fn-total	86.6	84.3	77.0	80.2	94.8	82.4	87.1	90.0	84.5	64.2	83.1
diff-fn-addtl	86.6	97.8	97.8	82.2	97.0	80.0	98.1	99.7	87.5	64.2	89.1
bdiff-fn-total	86.6	83.4	77.8	80.2	95.4	83.7	88.4	90.0	84.5	64.2	83.4
bdiff-fn-addtl	86.6	97.6	97.7	82.2	97.0	79.9	97.8	99.7	87.5	64.2	89.0

TABLE 3.5: $APFD_c$ values when considering faults detected by $\leq 1\%$ of all test cases.

to perform better than its corresponding random test case ordering as the faults of each version grew increasingly more difficult to detect.

There are two characteristics of these results that deserve discussion. First, as the faults in a version became increasingly harder to detect, the performance of a technique, as measured by cost-cognizant rate of fault detection, began to decrease. There were a few anomalies in this behavior, however, such as the performance of the total fault index prioritization technique in v4 and v5, where the $APFD_c$ values increased slightly when the criterion for fault detection dropped from 0.5% to 0.25%. These "anomalies", however, can be explained. Clearly, the exclusion of additional faults from our $APFD_c$ calculations is not going to increase the rate at which any particular fault is detected in the testing process. On the other hand, suppose a fault with a moderate severity is detected rather late in the regression testing process. If

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	Mean
random	48.0	74.3	40.2	66.7	70.5	73.0	81.9	88.8	85.1	75.2	70.4
fn-total	80.7	76.6	76.8	80.1	89.2	82.0	84.1	92.6	84.4	83.4	83.0
fn-addtl	91.5	79.0	84.7	90.7	94.9	83.5	87.3	98.3	94.8	97.8	90.3
fi-fn-total	61.0	75.2	76.9	80.2	88.6	82.3	83.0	92.7	84.5	64.2	78.9
fi-fn-addtl	61.0	95.4	97.8	82.2	94.7	80.0	96.5	99.8	82.1	64.2	85.4
diff-fn-total	61.0	75.2	77.0	80.2	88.7	82.4	82.5	92.7	84.5	64.2	78.8
diff-fn-addtl	61.0	95.5	97.8	82.2	94.7	80.0	96.4	99.8	82.1	64.2	85.4
bdiff-fn-total	61.0	77.3	77.8	80.2	90.2	83.7	83.7	92.7	84.5	64.2	79.5
bdiff-fn-addtl	61.0	95.4	97.7	82.2	94.7	79.9	96.4	99.8	82.1	64.2	85.3

TABLE 3.6: $APFD_c$ values when considering faults detected by $\leq 0.5\%$ of all test cases.

this fault is excluded from the $APFD_c$ calculations when using a stricter "difficulty of detection" criterion, such as 0.25% instead of 0.5%, a test suite may detect 100% of the fault severity in the software system at an earlier stage in the testing process. This occurrence could result in a higher $APFD_c$ when considering faults that are more difficult to detect.

The second characteristic of these results that warrants discussion is the performance of the techniques when compared across these levels of "difficulty of fault detection". Once again, our results indicate that additional coverage prioritization techniques may perform better in revealing the faults in a software system, as those faults grow increasingly difficult to detect, than total coverage prioritization techniques. While many techniques performed the same in version v1, in versions v2–v9, the $APFD_c$ values of our four additional coverage techniques ranged from the high

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	Mean
random	48.0	74.3	40.2	55.8	67.2	73.0	81.9	- - -	- - -	75.2	64.5
fn-total	80.7	76.6	76.8	81.4	90.5	82.0	84.1	- - -	- - -	83.4	81.9
fn-addtl	91.5	79.0	84.7	92.8	94.4	83.5	87.3	- - -	- - -	97.8	88.9
fi-fn-total	61.0	75.2	76.9	81.3	89.7	82.3	83.0	- - -	- - -	64.2	76.7
fi-fn-addtl	61.0	95.4	97.8	79.7	93.4	80.0	96.5	- - -	- - -	64.2	83.5
diff-fn-total	61.0	75.2	77.0	81.3	89.9	82.4	82.5	- - -	- - -	64.2	76.7
diff-fn-addtl	61.0	95.5	97.8	79.7	93.4	80.0	96.4	- - -	- - -	64.2	83.5
bdiff-fn-total	61.0	77.3	77.8	81.3	91.8	83.7	83.7	- - -	- - -	64.2	77.6
bdiff-fn-addtl	61.0	95.4	97.7	79.7	93.4	79.9	96.4	- - -	- - -	64.2	83.5

TABLE 3.7: $APFD_c$ values when considering faults detected by $\leq 0.25\%$ of all test cases.

70s to the high 90s, even when considering faults detected by less than 0.25% of all test cases, while the values of the total coverage techniques ranged from the mid 70s to the low 90s. Of these additional coverage techniques, additional function prioritization was particularly effective.

Chapter 4

DISCUSSION

As discussed in Section 3.3, this study has several threats to its validity. Nevertheless, the data provides several insights into the effectiveness of cost-cognizant test case prioritization.

Perhaps the most surprising result of our study is the performance of the random prioritization technique when all the seeded faults in the versions of our test subject are considered in the $APFD_c$ calculations. More often than not, our results indicate that a cost-cognizant prioritization technique will outperform a random test case ordering in this situation. However, in terms of the $APFD_c$ metric, the difference in rate of fault detection of these two strategies is often very small. Moreover, in many cases, the random prioritization technique actually outperformed a cost-cognizant technique when we considered all the faults in our test subject.

One possible reason for the effectiveness of the randomized prioritization technique is the size of our test suite and the number of faults considered in the test subject. As discussed earlier, the size test suite used in our case study contained 1985 test cases. However, each version of our test subject contained only ten seeded faults. Such a large discrepancy between test suite size and number of system faults may result in a randomly prioritized test suite performing comparably to a cost-cognizant test case prioritization technique if the faults are relatively easy to detect. While Table 3.2 shows that some faults in each version are very hard to reveal—detected by as few as 0.1% of all test cases in some instances—with the exception of v10, the

majority of the faults in our test subject are detected by approximately 6% to 8% of all test cases (6.5% is equivalent to approximately 129 test cases). This means that an average randomized test case ordering actually has a respectable chance of detecting the faults in our test subject early in the testing process.

The performance of the randomized prioritization technique seems contrary to the results of previous work by other researchers. These studies [8, 10, 31] have provided strong evidence that any test case prioritization technique will provide a noticeable increase in rate of fault detection over a randomized test case ordering. However, the studies cited above were constructed in such a manner that the fault severities in the subject software systems were considered uniform. In our study, the severity of a fault was estimated based on the likelihood, using an operational profile, that a user might encounter this fault.

In an initial paper [9] studying the incorporation of cost-cognizant metrics into regression testing, Elbaum et al. found that three cost-cognizant prioritization techniques performed much better than random prioritization, as measured by the $APFD_c$ metric.[1] In their study, however, the severity of the faults in the test subject were assigned based on their projected impact on program execution. While [9] only investigates three cost-cognizant techniques, and the fault severity assignments differs from those of this study, the difference in results between this study and the studies cited above warrants further investigation.

However, when the $APFD_c$ calculations only considered "hard to detect" faults, the majority of the time, prioritized test suites performed better than the randomly

---

[1] The three prioritization techniques used in that study were additional statement coverage (not used in this study), additional function coverage and additional fault index coverage at the function level.

ordered test suites. These results suggest that the cost-cognizant prioritization of a test suite is more likely to improve rate of detection for faults that are difficult to detect (but not necessarily the most severe faults of a system). Practically speaking, therefore, since a software system is likely to contain faults whose ease of detection vary, it might be worthwhile for software engineers to consider test case prioritization techniques in their regression testing; normal faults might be detected at approximately the same rate, but concealed faults might be detected faster. Further research investigating this issue may be helpful.

Going back to earlier discussion concerning possible reasons for the performance of the randomized prioritization technique when all faults were considered, we now see where cost-cognizant test case prioritization techniques have an advantage over a random test case ordering. Tables 3.5, 3.6 and 3.7 demonstrate that when the faults become increasingly difficult to detect, the chances that a random test case ordering will detect the faults early in the testing process grow slimmer. Test case prioritization techniques, however, are often still able to detect these difficult faults relatively early in the regression testing process, especially in comparison to random test case orderings.

It is important to recognize that our choice to consider averages of 20 random runs lets us control for random variance that might distort our results. In practice, software engineers using a randomized prioritization technique would likely use the test case ordering yielded by that technique over and over again, limiting their control over the random variance of the randomized technique. Practitioners must therefore be prepared for the possibility of varying results when utilizing randomized techniques, as the true potential for variance in random test case orderings is much greater than for other techniques.

When comparing general categories of techniques, the strongest result of this study may be the performance of the additional coverage prioritization techniques in comparison to the random prioritization techniques. As mentioned earlier, a random test case ordering occasionally outperformed a cost-cognizant technique, in terms of cost-cognizant rate of fault detection. However, when all faults were considered, in no case did a random test case ordering outperform an additional coverage test case ordering; additional function prioritization, additional fault index prioritization, additional DIFF prioritization and additional binary DIFF prioritization always outperformed random prioritization. One explanation for the small difference in the rate of fault detection of these test suites is the high $APFD_c$ values of each prioritized suite, prioritized or not. Mathematically speaking, when the $APFD_c$ values of a prioritized test suite are very high, there is a diminished opportunity for a cost-cognizant technique to significantly outperform a randomized prioritization technique.

However, while the differences in the rate of detection between additional coverage techniques and random test case orderings were not as significant as in previous studies, the results of this study nonetheless support the results of these earlier studies in concluding that additional coverage prioritization techniques outperform a random test case prioritization. Thus, while our results may seem contrary to previous work concerning the effectiveness of random test case orderings, our results support the findings of these earlier works concerning the effectiveness of additional coverage techniques, relative to random test case orderings.

The performance of the total coverage prioritization techniques, however, was not as favorable as that of the additional coverage prioritization techniques. When all faults were considered, each total coverage technique was either outperformed or equaled, in terms of rate of fault detection, by its corresponding additional coverage technique. Furthermore, the $APFD_c$ values of 72.5% (29 out of 40) of the total cov-

erage techniques, across versions v1–v10, were lower than the $APFD_c$ values of the random test case orderings. This result suggests that, when cost-cognizant metrics are factored into test case prioritization, software practitioners may want to choose an inexpensive random test case ordering over a possibly more expensive total coverage prioritization technique. However, as mentioned earlier, when the $APFD_c$ calculations were limited to "hard to detect faults", with the exception of version v10, all prioritization techniques, including the total prioritization techniques, outperformed the random test case ordering of that version. Consequently, if a software practitioner is confident that the "easy to detect faults" were uncovered during unit testing by programmers and that only "hard to detect faults" remain in the software system, he or she may favor a total coverage prioritization technique over a random test case ordering.

Finally, it is important to note that, in practice, the significance in the rate of fault detection between two test suites is directly dependent upon the execution time of each test suite. If a test suite takes a long time to execute—for example, on the order of weeks—a test suite with a high rate of fault detection may be considered favorable to a test suite with a lower rate of fault detection, even if the difference in the rate of fault detection is small, as faults may be detected days in advance with the faster detecting test suite. However, even a significant difference in the rate of fault detection between two test suites may not be significant to the software practitioner, if these test suites have fast execution times, as faults may only be detected minutes in advance. This might be especially applicable if the cost of generating a prioritized suite is high, or if practitioners are choosing between an expensive technique, such as a fault index prioritization technique, and an inexpensive technique, such as a binary DIFF technique.

Chapter 5

CONCLUSIONS AND FUTURE WORK

Through this thesis, we have provided a further examination of cost-cognizant test case prioritization techniques. We explored the ability of these techniques to make the regression testing process more efficient by investigating their capability to improve the rate of fault detection that might otherwise be achieved by a random test case ordering. Our results suggest that cost-cognizant additional coverage prioritization techniques can improve a test suite's rate of fault detection, but that cost-cognizant total coverage prioritization techniques may not provide this improvement. We also explored the capabilities of cost-cognizant techniques in providing an improved rate of fault detection when the faults in a software system are very difficult to detect, where we found that cost-cognizant test case prioritization techniques generally provide a significant improvement over random test case orderings as faults grow increasingly difficult to detect.

While this study was designed to provide additional empirical evidence concerning the effectiveness of cost-cognizant prioritization techniques, it has also raised several possibilities for future research. First, our results concerning the $APFD_c$ of randomized test case orderings seem to indicate that prioritization techniques may not provide an improved rate of fault detection when cost-cognizant metrics are factored into the techniques. These findings, however, are contrary to previous research investigating the rate of fault detection of non-cost-cognizant test case prioritization techniques. Yet, since these results were gathered using a metric where fault severi-

ties are considered uniform, further research could provide additional clarity on this matter.

Second, our results indicate that cost-cognizant test case prioritization techniques yield test suites with higher rates of fault detection than random test case orderings when the faults of a software system are "hard to detect". Furthermore, the superiority of test case prioritization techniques appears to be amplified as these faults grow increasingly difficult to detect. This is a relatively untested finding, however, and further research should be committed towards investing this issue further.

Finally, an additional strategy for investigating the effectiveness of cost-cognizant prioritization techniques might be to compare the $APFD_c$ values of the cost-cognizant adaptions of our techniques with those values of the non-cost-cognizant adaptions. A study designed similar to that of this thesis could be effective in investigating cost-cognizant techniques through this comparison, provided that non-cost-cognizant implementations of each prioritization technique are available. Conducting such a study on multiple test subjects with multiple test suites, both of varying sizes, would increase the power of such a study.

It is our hope that the results of this study will contribute to the advancement of efficient methods of regression testing through test case prioritization, and will help to improve the overall quality of future software.

BIBLIOGRAPHY

[1] IEEE Standards Association. Software engineering standards. In *Std. 1061: Standard for Software Quality Methodology*, volume 3. Institute of Electrical and Electronics Engineers, 1999 edition, 1992.

[2] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–716, September 1995.

[3] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. Philosophy for software measurement. *J. Sys. Software*, 12(3):277–281, 1990.

[4] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proceedings of the 21st International Conference on Software Engineering*, pages 345–354, May 1999.

[5] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems.* Prentice Hall, Upper Saddle River, New Jersey, 2000.

[6] T. Chen and M. Lau. Dividing strategies for the optimization of a test suite. *Info. Proc. Let.*, 60(3):135–141, March 1996.

[7] Y. Chen, D. Rosenblum, and K. Vo. Testtube: A system for selective regression testing. In *Proc. Int'l Symp. Softw. Testing and Analysis*, pages 102–112, August 2000.

[8] 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.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, May 2001.

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

[11] S. G. Elbaum and J. C. Munson. A standard for the measurement of C complexity attributes. Technical Report TR-CS-98-02, University of Idaho, February 1998.

[12] S. G. Elbaum and J. C. Munson. Code churn: A measure for estimating the impact of code changes. In *Proceedings of the International Conference on Software Maintenance*, pages 24–31, November 1998.

[13] S. G. Elbaum and J. C. Munson. Software evolution and the code fault introduction process. 4(3):241–262, September 1999.

[14] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering*, pages 188–197, April 1998.

[15] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

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

[17] R. Hildebrandt and A. Zeller. Minimizing failure-inducing input. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 135–145, August 2000.

[18] R. A. Johnson and D. W. Wichorn. *Applied Multivariate Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 3rd edition, 1992.

[19] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using complexity metrics. *J. on Selected Areas in Comm.*, 8(2):253–261, February 1990.

[20] H. Leung and L. White. Insights into regression testing. In *Proc. Conf. Softw. Maint.*, pages 60–69, October 1989.

[21] J. C. Munson. Software measurement: Problems and practice. *Annals of Software Engineering*, 1(1):255–285, 1995.

[22] J. Musa. *Software Reliability Engineering*. McGraw-Hill, New York, N.Y., 1999.

[23] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability, Measurement, Prediction, Application*. McGraw-Hill, New York, 1987.

[24] A. P. Nikora and J. C. Munson. Software evolution and the fault process. In *Proceedings of the 23rd Annual Software Engineering Workshop*. NASA/Goddard Space Flight Center, 1998.

[25] J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, pages 111–123, June 1995.

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

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

[28] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002 (to appear).

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

[30] 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, September 1999.

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

[32] G. Schulmeyer and J. McManus. *Handbook of Software Quality Assurance*. Prentice Hall, New York, N.Y., 3rd edition, 1999.

[33] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Related Engineering*, pages 230–238, November 1997.

Appendix A

CASE STUDY DESIGN FIGURE