An Empirical Study of Incorporating Cost into Test Suite Reduction and Prioritization

Adam M. Smith
Department of Computer Science
University of Pittsburgh
ams292@cs.pitt.edu

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
gkapfham@allegheny.edu

ABSTRACT

Software developers use testing to gain and maintain confidence in the correctness of a software system. Automated reduction and prioritization techniques attempt to decrease the time required to detect faults during test suite execution. This paper uses the Harrold Gupta Soffa, delayed greedy, traditional greedy, and 2-optimal greedy algorithms for both test suite reduction and prioritization. Even though reducing and reordering a test suite is primarily done to ensure that testing is cost-effective, these algorithms are normally configured to make greedy choices with coverage information alone. This paper extends these algorithms to greedily reduce and prioritize the tests by using both test cost (e.g., execution time) and the ratio of code coverage to test cost. An empirical study with eight real world case study applications shows that the ratio greedy choice metric aids a test suite reduction method in identifying a smaller and faster test suite. The results also suggest that incorporating test cost during prioritization allows for an average increase of 17% and a maximum improvement of 141% for a time sensitive evaluation metric called coverage effectiveness.

Categories and Subject Descriptors: D.2.5 [Software Engineering][Testing and Debugging][Testing tools]

General Terms: Experimentation, Performance

Keywords: regression testing, reduction, prioritization

1. INTRODUCTION

Since developers inevitably introduce errors while implementing software systems, they often use software testing to detect and isolate these defects. As the source code grows in size, test cases are written for the new functionality. However, these new tests do not obviate the old ones. In an attempt to ensure both the correctness of new code and its proper integration into the system, every old test is executed in a regression test suite $T = \langle t_1, t_2, t_3, \ldots, t_n \rangle$. The repeated execution of T is known as regression testing [4, 16, 17]. When a test suite becomes too large to run in a cost-effective fashion, reduction [4] and prioritization [16] methods can alter the test suite in order to address this issue. Reduction produces a smaller test suite T_r by removing tests from T, with the intent of decreasing its execution time while preserving the coverage of the test requirements. Test suite prioritization produces T_p by reordering T and thus allowing for an execution order that is more likely to find defects faster than the original test suite.

This paper describes the extension and empirical evaluation of the Harrold Gupta Soffa (HGS) [4], delayed greedy (DGR) [18], traditional greedy (GRD) [9], and 2-optimal greedy (2OPT) [9] algorithms. HGS and DGR use coverage information to make intelligent decisions before they make a greedy choice, GRD is a standard greedy algorithm, and 2OPT is a greedy algorithm that performs all-pairs comparisons. HGS and GRD are commonly used techniques for reduction while the DGR reducer can determine whether or not its reduced test suite is optimally small. Beyond incorporating test case cost (e.g., test execution time), our implementations repeatedly invoke a reduction algorithm in order to prioritize test suites, as described in Section 2.

Traditional implementations of test suite reduction and prioritization methods may not incorporate test cost into a greedy choice metric (GCM). For instance, Li et al. describe a greedy prioritizer that iteratively selects the test case that covers the most uncovered test requirements [9] and Harrold et al. reduce a test suite by exclusively focusing on coverage information [4]. Yet, variability in test case cost suggests that the test with the most coverage may not be the best option for either inclusion in a reduced test suite or early placement in a prioritized one [5, 6, 10, 17]. Since the purpose of both reduction and prioritization is to save software testers time by finding faults sooner, not in less test cases, our extended algorithms use GCMs that consider test coverage, test case cost, and the ratio of coverage to cost (hereafter referred to as ratio). Our work is distinguished from [10] because we (i) use coverage effectiveness (CE) [6] to evaluate a prioritized test suite, (ii) consider four different reduction algorithms (e.g., HGS, DGR, GRD, and 2OPT) and extend their functionality to use cost, coverage, and ratio for greedy choices, (iii) perform both reduction and prioritization on the JUnit test suites for object-oriented Java programs, and (iv) do not require the use of faults in either the test adequacy criterion or the evaluation metrics.

The test suite in Table 1(a) furnishes a motivating example for incorporating test case execution time into greedy choices. In this table each row represents a test, a column corresponds to a test requirement, and an "X" shows when a test covers a requirement. Table 1(b) shows that using GRD to reduce this test suite based on coverage alone yields a test

	r_1	r_2	r_3	r_4	r_5	Execution Time
t_1	X	X	X	X		4
t_2			X	X		1
t_3		X				1
t_4	X				X	1

(a)

Greedy-by	T_r	$time(T_r)$	T_p	CE
coverage	$\langle t_1, t_4 \rangle$	5	$\langle t_1, t_4, t_2, t_3 \rangle$	0.400
time	$\langle t_2, t_3, t_4 \rangle$	3	$\langle t_2, t_3, t_4, t_1 \rangle$	0.714
ratio	$\langle t_2, t_4, t_3 \rangle$	3	$\langle t_2, t_4, t_3, t_1 \rangle$	0.743

(b)

Table 1: Reduction and Prioritization Example.

suite that contains only two test cases and executes in five time units. However, reducing based on ratio or time produces test suites containing three test cases that execute in only three units of time. If testing time is limited, then often it is better for developers to use a test suite with more test cases and a short execution time than a smaller test suite that takes longer to run. Therefore, reducing with time or ratio may produce test suites that are more desirable than reducing with just coverage. Similarly for prioritization, using time or ratio for greedy choices allows for better prioritized test suites as defined by the "higher is better" coverage effectiveness metric explained in Section 3. In summary, the important contributions of this paper include:

1. The implementation and extension of four existing algorithms for regression testing. Each algorithm originally reduced using only coverage, but this paper also uses test cost and the ratio of coverage to test cost.
2. An experimental analysis of the effectiveness of each regression testing technique on eight case study applications. The evaluation metrics, reduction factor for time (RFFT) and CE (see Section 3 for more details), identify reduced and prioritized test suites that decrease testing time while still preserving the coverage of test requirements. The experiments reveal fundamental trade-offs associated with incorporating test cost into regression testing:
   (a) HGS creates reduced test suites with the greatest average RFFT, but DGR, 2OPT, and GRD each perform closely to HGS.
   (b) Similarly for the reduction factor for size (RFFS) metric, DGR, HGS, and GRD obtain the highest average values, but 2OPT was not far behind.
   (c) Using the intrinsically coverage-based techniques (i.e., HGS and DGR) often leads to orderings with modest coverage effectiveness values.
   (d) The 2OPT and GRD prioritizers obtain the highest average CE values, with a 28% improvement over HGS and a 17% increase over DGR.

2. REGRESSION TESTING

Test Adequacy. Test suites allow developers to execute application code in a selective manner, with each test normally designed for an individual aspect of the program. In the context of this paper, a test suite refers to a Java class that is executed by a Java testing framework called JUnit (http://www.junit.org) and a test case refers to a method in that test class. Given a test suite T, a test coverage monitor identifies a set of covered requirements $\mathcal{R}(T) = \{r_1, r_2, \ldots, r_m\}$. Each test t_i is associated with a subset of requirements $\mathcal{R}(t_i) \subseteq \mathcal{R}(T)$ that t_i is said to *cover*. A coverage monitor also determines the *covered by* relationship that associates a requirement r_j with a set of tests $\mathcal{T}(r_j) \subseteq T$ such that r_j is covered by each test in $\mathcal{T}(r_j)$.

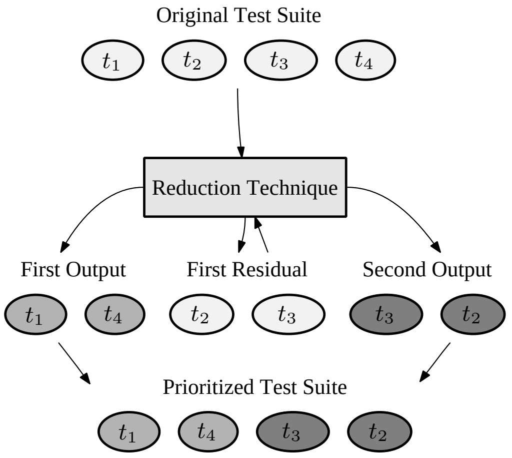

Figure 1: Prioritization by Repeated Reduction.

McMaster and Memon present a test adequacy criterion that measures method coverage in the context in which the methods were invoked during testing [12]. A coverage report for this criterion corresponds to a calling context tree (CCT) that represents the dynamic behavior of the program while a test suite runs. Each node in a CCT stands for a method that was called during the execution of a test case. An edge from a parent to a child node signifies that the parent method called the child method during testing and a path in the CCT from the root node to a leaf node forms a requirement r_j. This paper uses call tree paths as a test adequacy criterion because they are efficient to collect and store, thus enabling the reordering or reducing of a test suite each time the program under test changes [5, 7].

Although CCTs may be criticized for not incorporating the (i) source code or parameters of the methods under test and (ii) state of the program, they have been shown to perform closely to other test adequacy criterion with respect to common fault detection metrics [13]. In fact, in a recent empirical study, test suites that had been reduced using call trees as a coverage metric were 97-100% likely to detect each known fault in the evaluated program [14]. However, as part of future work, we intend to replicate the experiments with different test adequacy criteria.

Reduction and Prioritization. When more than one test covers the same requirement, the tests have an overlap in coverage that suggests the possibility of removing some tests while maintaining test suite completeness. Equations 1 through 3 define the coverage, cost, and ratio greedy choice metrics used by HGS, DGR, GRD, and 2OPT. Although this paper uses execution time for the cost and ratio GCMs, the algorithms may employ any "lower is better" quantification of cost. The techniques also work properly with any type of test requirement even though we pick CCT paths.

$$coverage(t_i) = |\mathcal{R}(t_i)| - |\mathcal{R}(t_i) \bigcap \mathcal{R}(T_r)| \quad (1)$$

$$cost(t_i) = time(t_i) \quad (2)$$

$$ratio(t_i) = \frac{coverage(t_i)}{cost(t_i)} \quad (3)$$

Name	$\|T\|$	$\|\mathcal{R}(T)\|$	CCN	NCSS
DS	110	40	1.35	1243.00
GB	51	88	2.60	1455.00
JD	54	783	1.64	2716.00
LF	13	6	1.40	215.00
RM	13	19	2.13	569.00
SK	27	117	2.00	628.00
TM	27	46	2.21	748.00
RP	76	221	2.65	6822.00

Table 2: Case Study Applications.

Even though test suite reduction maintains 100% coverage of the requirements, it does not guarantee the same fault detection capabilities as the original test suite [12, 13, 14]. For this reason, some developers prefer test suite prioritization techniques that retain all tests while finding an ordering T_p that rapidly covers all of the requirements in $\mathcal{R}(T)$. The same approach to reduction can also be leveraged by a prioritization mechanism. The reduction methods attempt to produce a T_r that is smaller than the input test suite T. While reducers ignore the redundant tests, a prioritizer repeatedly inputs the surplus tests into the reduction algorithm until all of the tests have been added to T_p. As shown in Figure 1, when given the original test suite T the reduction algorithm produces the first output $T_{r1} = \langle t_1, t_4 \rangle$ and two residual tests t_2 and t_3. These tests are then once again passed to the reduction technique, resulting in the second output $T_{r2} = \langle t_3, t_2 \rangle$. The concatenation of T_{r1} and T_{r2} creates the prioritized test suite $T_p = \langle t_1, t_4, t_3, t_2 \rangle$.

The GRD algorithm analyzes the set of tests based on the cost, coverage, or ratio GCM [9]. When reducing based on coverage, the algorithm picks the tests that cover the most additional requirements. For the cost GCM, the algorithm finds the tests with the lowest execution time. Ratio choices allow the algorithm to pick the tests that cover the most requirements per unit cost. The 2OPT algorithm is an all-pairs greedy approach that compares each pair of tests to all of the other pairs and selects the best according to a GCM [9]. Both the GRD and 2OPT algorithms proceed in an iterative fashion by adding tests to T_r until $\mathcal{R}(T) = \mathcal{R}(T_r)$.

Since every requirement must be covered by the reduced test suite, HGS starts to construct T_r by identifying each requirement r_j such that $|\mathcal{T}(r_j)| = 1$ [4]. After adding every test $\mathcal{T}(r_j) = \{t_i\}$ to the reduced test suite T_r, HGS considers each remaining uncovered requirement r_j when $|\mathcal{T}(r_j)| = 2$ and it uses a GCM to choose between the covering test cases. The HGS reducer continues by iteratively examining the $\mathcal{T}(r_j)$ of increasing cardinality until all of the requirements are covered. When the GCM does not enable HGS to disambiguate between the tests in $\mathcal{T}(r_j)$ for $|\mathcal{T}(r_j)| = \ell$, the algorithm "looks ahead" in order to determine how the tests fare in covering requirements with $\ell + 1$ covering tests. If HGS performs the maximum number of allowed look aheads without identifying the best test case, the algorithm arbitrarily selects from those tests that remain.

DGR consults both $\mathcal{R}(t_i)$ and $\mathcal{T}(r_j)$ in order to identify the tests that that will not benefit T_r and the requirements that the reduced suite already covers [18]. If $\mathcal{R}(t_i) \subseteq \mathcal{R}(t_k)$, then t_i does not need to be analyzed for placement in T_r because as long as t_k is included, $\mathcal{R}(t_i)$ will be covered. Also, if $\mathcal{T}(r_h) \subseteq \mathcal{T}(r_j)$ then r_h will be covered as long as r_j is covered. Therefore, r_h no longer needs to be considered during the analysis of T. After test and requirement eliminations have been made, DGR adds all tests t_i where for any j,

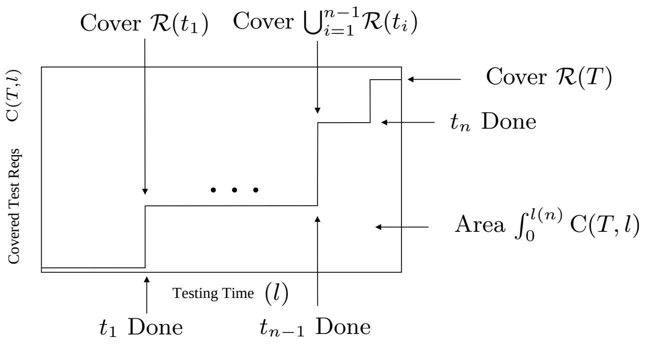

Figure 2: Coverage Effectiveness.

$\mathcal{T}(r_j) = \{t_i\}$. If $|\mathcal{T}(r_j)| \neq 1$ for all j after eliminations have been made, then DGR greedily chooses a test to add to T_r based on a GCM. These reductions and greedy choices are repeated until $\mathcal{R}(T) = \mathcal{R}(T_r)$. If DGR never uses a GCM, then it returns an optimally small T_r [18].

3. EXPERIMENT GOALS AND DESIGN

Case Study Applications. Each case study application contains a JUnit test suite whose test coverage information was captured using instrumentation probes [17]. Table 2 shows the number of test cases ($|T|$), number of requirements ($|\mathcal{R}(T)|$), average cyclomatic complexity number (CCN) across all methods [11], and non-commented source statements (NCSS) for each of the case study applications. JDepend (JD) is a Java source code analyzer that measures several quality attributes of a program. The Transaction-Manager (TM) is an ATM machine that interacts with a bank database while the Sudoku (SK) application solves sudoku puzzles. The DataStructures (DS) application includes implementations and tests for different data structures (e.g., Stacks and LinkedLists). The Reduction and Prioritization (RP) package contains the implementation and tests for all of the algorithms described in this paper. LoopFinder (LF) searches a graph for cycles and Reminder (RM) creates prompts for specific dates, adds them to a database, and performs database operations such as queries. Finally, the GradeBook (GB) application allows for the input, editing, and statistical analysis of student grades.

Evaluation Metrics. Equation 4 defines RFFS$(T, T_r) \in [0, 1)$, the reduction factor for size (RFFS) given a test suite T and it's reduced form T_r [12]. Since the RFFS reflects the percent of original test cases that remain after reduction, an RFFS of 0 means that the algorithm removed none of the tests while an RFFS near 1 means that the reducer removed many tests (an RFFS of 1 is not possible because T_r must contain at least one test that covers all of the requirements).

$$\text{RFFS}(T, T_r) = \frac{|T| - |T_r|}{|T|} \tag{4}$$

As stated by Equations 5 and 6, RFFT$(T, T_r) \in [0, 1)$ is the reduction factor for time (RFFT) for test suites T and T_r [5]. An RFFT of 0 signifies that T and T_r execute for the same length of time (i.e., $time(T) - time(T_r) = 0$) while an RFFT of 1 is the impossible case when T_r executes instantaneously (i.e., $time(T) - time(T_r) = time(T)$).

$$time(T) = \sum_{i=1}^{|T|} time(t_i) \tag{5}$$

$$\text{RFFT}(T, T_r) = \frac{time(T) - time(T_r)}{time(T)} \tag{6}$$

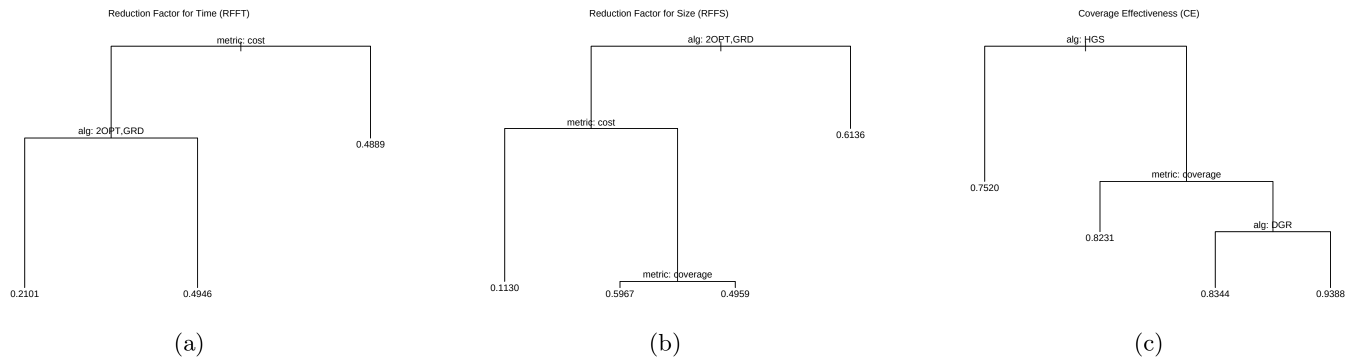

Figure 3: Tree Models for the RFFT, RFFS, and CE Evaluation Metrics.

The coverage effectiveness (CE) metric evaluates a prioritized test suite by determining the cumulative coverage of the tests over time [6]. As defined in Equation 7 and depicted in Figure 2, the cumulative coverage function $C(T, l)$ takes the input of a test suite T and a time l and returns the total number of requirements covered by T after running for l time units. To formulate the CE metric, the integral of $C(T, l)$ is divided by the integral of the ideal cumulative coverage function $\bar{C}(T, l)$ that Equation 8 defines to immediately cover all of the requirements. Equation (9) shows that these integrals are taken within the closed interval 0 to $l(n)$ where $l(i)$ is the time required to execute $t_1, t_2, ..., t_i$ and there are n test cases in the suite. Since many test coverage monitoring tools do not record the point in time when a test case covers a requirement [7, 15], CE conservatively credits a test with the coverage of its requirements when it finishes execution. While CE may be unfair to high coverage tests with extended running times, the metric does furnish a time sensitive measurement of effectiveness. In contrast, prior metrics such as APFD [16], APBC, APDC, and APRC [9], did not factor time into the evaluation of a prioritization technique. Unlike existing evaluation metrics that do incorporate time (e.g., APFD$_c$ [10]), CE obviates the need to use fault information when calculating effectiveness.

$$C(T,l) = \begin{cases} 0 & l < l(1) \\ |\mathcal{R}(t_1)| & l \in [l(1), l(2)) \\ \vdots & \vdots \\ |\bigcup_{i=1}^{n-1} \mathcal{R}(t_i)| & l \in [l(n-1), l(n)) \\ |\mathcal{R}(T)| & l \geq l(n) \end{cases} \tag{7}$$

$$\bar{C}(T,l) = |\mathcal{R}(T)| \tag{8}$$

$$\mathrm{CE}(T) = \frac{\int_0^{l(n)} \mathrm{C}(T,l)}{\int_0^{l(n)} \bar{\mathrm{C}}(T,l)} \tag{9}$$

Analysis Techniques. As evidenced by Figure 3, this paper uses automatically generated tree models to describe the trends in the data sets. In particular, we use the recursive partitioning algorithm [1] to determine how the explanatory variables (i.e., greedy choice metric and regression testing technique) impact the evaluation metrics. We selected this type of hierarchical model because it furnishes a clear view of the interactions between the GCMs and algorithms. The root of a tree corresponds to the most important explanatory variable for the given data set. By following a path from the root to a leaf node, it is possible to determine the mean value for the specified subset of the data. In the split points of our trees (e.g., "metric:cost" in the first model of Figure 3), the word before the colon is a categorical explanatory variable and the word(s) to the right are the descriptors associated with the left sub-tree. Moreover, the right sub-tree always corresponds to the data values that are not included in the split's descriptor. For instance, the root of the first tree model indicates that the left sub-tree describes the RFFT of the cost GCM while the right sub-tree explains the RFFT for the coverage and ratio metrics. The first tree in Figure 3 also reveals that 2OPT and GRD have an average RFFT of .2101 when they use the cost GCM.

Figures 4 through 6 use scatter plots in order to visualize how the evaluation metrics vary for different case study applications, algorithms, and GCMs. Since certain combinations of a GCM and regression testing technique lead to the same value for RFFT, RFFS, or CE, we add a small amount of horizontal spacing in order to avoid overlapping data points. Figures 7 and 8 use box and whisker plots to depict the variation in execution time for each of the reduction and prioritization algorithms. These plots show the median value as a filled circle and the inter-quartile range as a box. Furthermore, these graphs use whiskers to represent the minimum and maximum execution times.

4. EXPERIMENTAL RESULTS

Reduction. The outcomes in Figure 4 reveal that the RFFT values vary across case study application. For instance, RM exhibits uniformly low reduction factors while the reducers are able to discard many of the tests within the suites of DS, GB, and LF. This phenomenon is due to the fact that RM has many test cases that perform unique testing tasks. In contrast, the tests for DS and LF repeatedly exercise the same methods with slightly different input values and GB contains several expensive tests that have redundant database interactions. Figure 4 also shows that the coverage and ratio GCMs yield reduced test suites with similar RFFT values while the cost GCM gives acceptable results for a few applications (e.g., GB and LF). However, the cost GCM performs poorly for some case study applications (e.g., RP, SK, DS, and JD), thus suggesting that the myopic focus on test execution time may mislead a reducer. In particular, the tree model in Figure 3(a) confirms that the combination of either 2OPT or GRD and the cost GCM leads to an average RFFT of 0.2101.

Overall, the reduction techniques exhibit relatively similar values for RFFT. For example, Figure 3(a) reveals that pairing the cost GCM with DGR or HGS leads to an average RFFT value of 0.4946 while the combination of any algorithm with either the coverage or ratio GCM gives an average RFFT of 0.4889. Furthermore, Table 3 furnishes the average RFFT values across all of the case study applica-

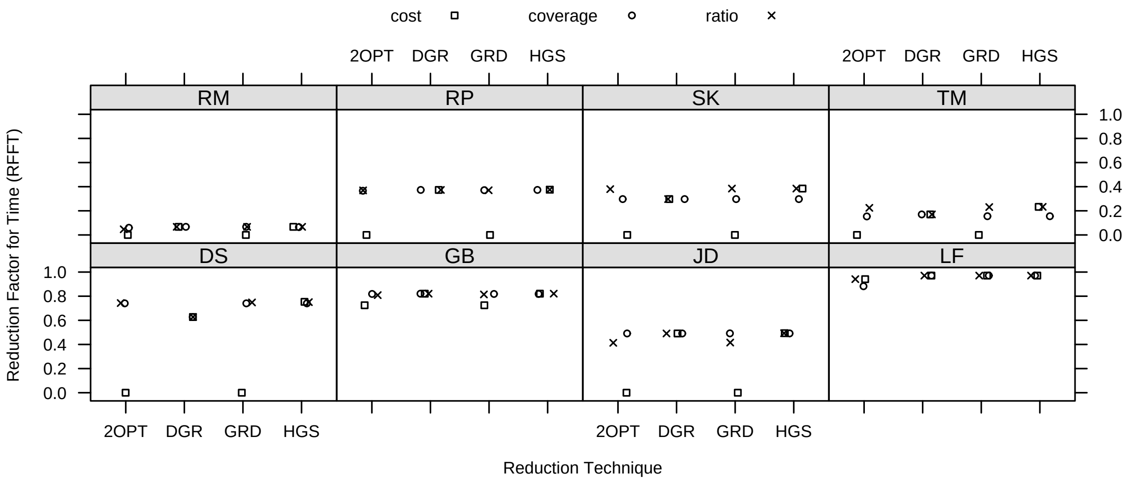

Figure 4: Reduction Factor for Time (RFFT) Across All Applications.

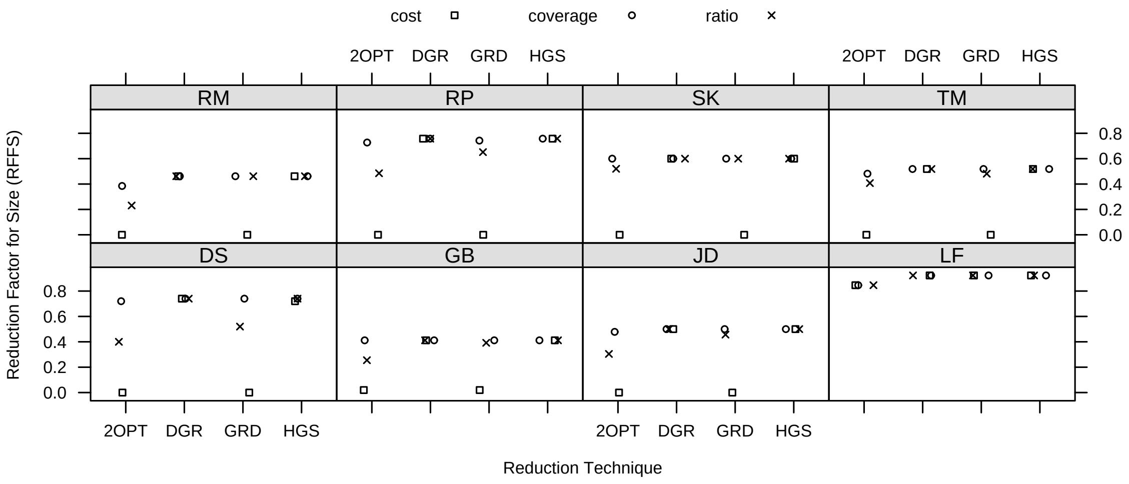

Figure 5: Reduction Factor for Size (RFFS) Across All Applications.

tions. This summary table and the scatter plots in Figure 4 support the conclusion that incorporating test case cost improves RFFT. We also note that 2OPT has RFFT values that increase from 0.208 to 0.476 or 0.491 when we replace the cost GCM with coverage or ratio, respectively. Even though 2OPT and GRD see the most improvement from using ratio instead of coverage, the intrinsically coverage-based HGS still exhibits a minor increase in RFFT.

As demonstrated by the scatter plots in Figures 4 and 5, many applications have higher values for RFFS than RFFT. This result suggests that it is easier to find a small reduced test suite than it is to identify a T_r that will decrease testing time. In fact, the DGR algorithm displays invariant performance across all three choice metrics because it identifies optimally small test suites for each case study application and thus avoids the invocation of a GCM. Interestingly, Table 3 shows that DGR's optimal RFFS of 0.614 corresponds to an RFFT of 0.477 that is less than the 0.512 RFFT associated with using HGS and the ratio GCM. Thus, the empirical results highlight the fact that an optimally small test suite may not be the fastest collection of test cases.

The tree model in Figure 3(b) and the plots in Figure 5 also establish that for all applications except LF, the cost GCM leads to low RFFS values for the 2OPT and GRD algorithms. In contrast, HGS is impervious to the misleading cost GCM because it focuses on coverage information during each iteration. We also see that pairing 2OPT or GRD with the cost and ratio GCMs yields varying results. For instance, the ratio GCM leads to lower RFFS values for RP, DS, GB, and JD and similar levels for RM, SK, TM, GB, and LF. Overall, the tree models in Figures 3(a) and (b) confirm that it is reasonable to select the ratio GCM when using 2OPT and/or GRD because this choice metric gives relatively high values for both RFFT and RFFS. Using 2OPT or GRD with the coverage GCM or picking either DGR or HGS is most appropriate for situations in which it is important to maximize RFFS (e.g., when test cost data is unavailable or tests run in a memory constrained environment [8]). Moreover, the tree in Figure 3(b) substantiates the claims made by Harrold et al. [4] and Tallam and Gupta [18]: with an average RFFS of .6136, HGS and DGR are the best methods for reducing the size of a regression test suite.

Prioritization. Figure 6 highlights the fact that, with the exception of the smallest application called LF, the prioritizers create test orderings that exhibit some variability in their CE scores. For different configurations of the prioritizers, RM, RP, and SK have variation in the CE values because their test cases display the most trade-offs in test cost and coverage. We also observe that RM, RP, and SK have lower CE values when the algorithm uses the coverage

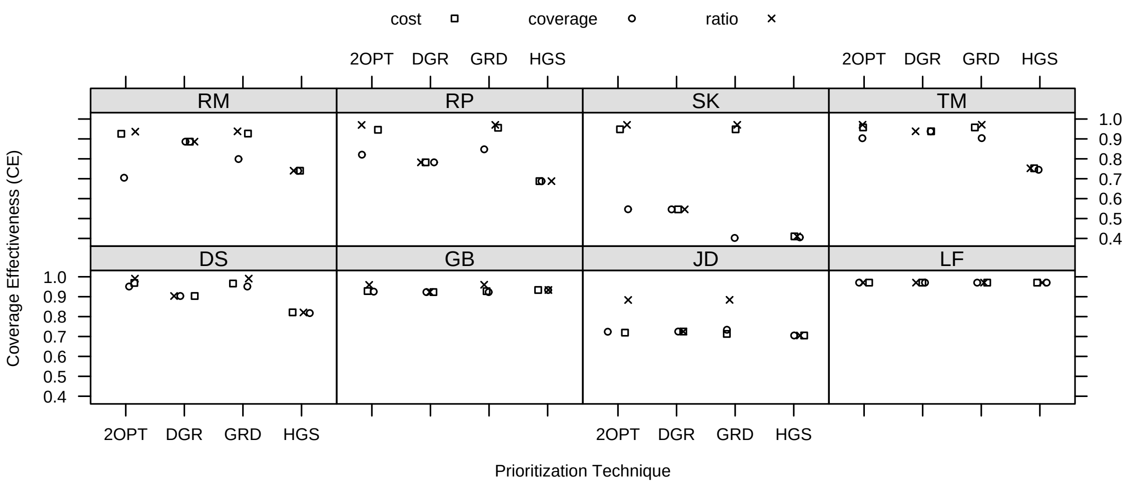

Figure 6: Coverage Effectiveness (CE) Across All Applications.

GCM, further suggesting that the applications have high cost tests that also cover many requirements. Interestingly, when the ratio GCM combines cost and coverage information, it moderates the shortsighted focus on coverage and leads to high CE values for all applications. For instance, Figure 6 shows that when 2OPT and GRD use the ratio GCM instead of cost or coverage, the CE values for JD increase markedly. Table 3 also points out that, across all of the applications, the CE values for 2OPT and GRD increase from 0.921 to 0.957 when ratio replaces the cost GCM.

The scatter plots in Figure 6 and the summary values in Table 3 further reveal that 2OPT and GRD consistently obtain the highest CE scores. We see that for all but GB, JD, and LF, the DGR prioritizer ties or does better than HGS. The tree model in Figure 3(c) furnishes additional evidence of this empirical trend: HGS has an average CE value of 0.7520 while the other configurations yield average CE values ranging from 0.8231 to 0.8388. Finally, Table 4 shows each technique and GCM that results in a lower CE than the initial (INIT) and reverse (REV) orderings for each application. Notice that 2OPT and GRD combined with ratio are the only techniques that never create a prioritized test suite that is worse than the REV or INIT orderings.

Figures 7 and 8 give the execution time (in milliseconds) for the reduction and prioritization methods. These box and whisker plots clearly demonstrate that the regression testing methods execute efficiently for all of the case study applications. For example, no run of a reducer or prioritizer consumes more than 150 milliseconds. However, we note that GRD and HGS tend to be the most efficient techniques with 2OPT and DGR exhibiting both higher time overheads and greater variability across trial runs and GCMs. Overall, these efficiency and effectiveness results indicate that 2OPT and GRD are ideal algorithms for development environments that only use prioritization techniques. Alternatively, DGR is a good candidate for both reduction and prioritization since it leads to moderate values for all three of the evaluation metrics. Finally, the experiments demonstrate that while HGS performs well when it rapidly reduces a test suite, it exhibits lackluster CE scores when it is invoked repeatedly as part of test prioritization routine.

5. RELATED WORK

Due to space constraints, this paper briefly reviews the most relevant research. While Malishevsky et al. also describe prioritizers that incorporate test case cost [10], Section 1 notes that our work differs in several key ways (e.g., we focus on both reduction and prioritization). Similar to the present work, Do et al. consider the prioritization of JUnit test suites yet without including test case costs [2]. Although both Do et al. and Elbaum et al. include many different algorithms in a empirical study of prioritizers [2, 3], their investigations do not handle reduction or evaluate either 2OPT or the repeated invocation of HGS and DGR.

Technique	Greedy-by	RFFT	RFFS	CE
2OPT	cost	0.208	0.108	0.921
2OPT	coverage	0.476	0.581	0.818
2OPT	ratio	0.491	0.431	0.957
DGR	cost	0.477	0.614	0.834
DGR	coverage	0.477	0.614	0.834
DGR	ratio	0.477	0.614	0.834
GRD	cost	0.212	0.118	0.921
GRD	coverage	0.489	0.612	0.817
GRD	ratio	0.500	0.561	0.957
HGS	cost	0.512	0.612	0.753
HGS	coverage	0.489	0.614	0.751
HGS	ratio	0.512	0.614	0.753

Table 3: Average Values for RFFT, RFFS, and CE.

Technique	GCM	REV	INIT
2OPT	cost	JD	-
2OPT	coverage	RP, RM, JD	SK
2OPT	ratio	-	-
DGR	all	RM, RP, JD	SK
GRD	cost	JD	-
GRD	coverage	RM, RP, JD	SK
GRD	ratio	-	-
HGS	all	RM, RP, JD	RP, SK, TM

Table 4: Comparison to Reverse and Initial.

6. CONCLUSIONS AND FUTURE WORK

This paper describes the extension and evaluation of four existing algorithms for regression testing (e.g., HGS, DGR, GRD, and 2OPT). Even though the reduction and prioritization techniques typically focus on the coverage of a test suite, we enable them to use greedy choice metrics that consider both test case cost and the ratio of coverage to cost. Using a variety of visualizations and analysis techniques (e.g., automatically generated tree models, scatter plots, and box and whisker plots), we identify fundamental trade-offs

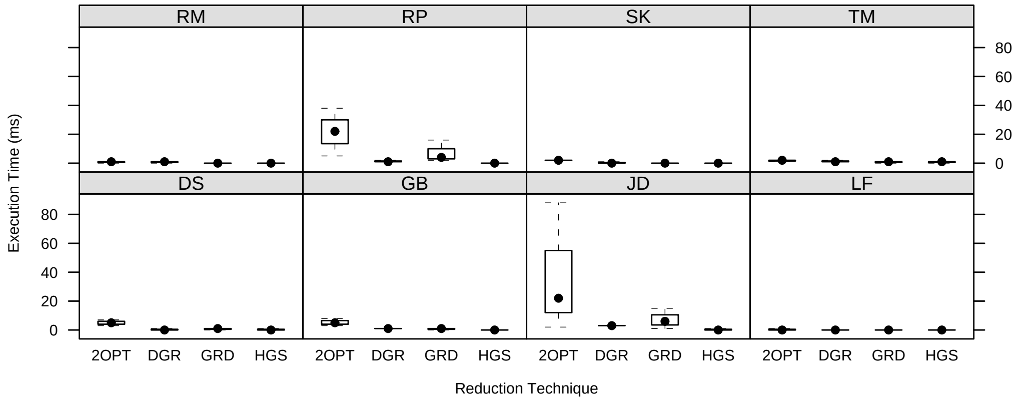

Figure 7: Reduction Time Across All Applications.

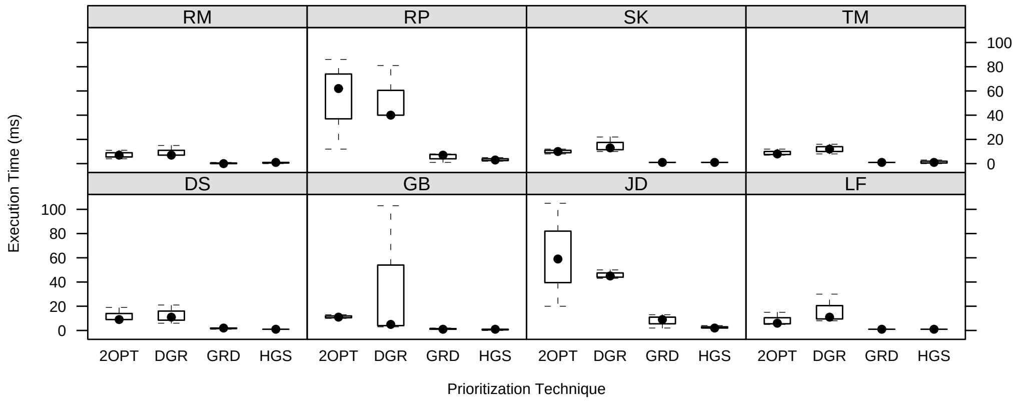

Figure 8: Prioritization Time Across All Applications.

in the efficiency and effectiveness of the testing methods. For instance, the experiments revealed the fact that an optimally small test suite may not contain the fastest set of test cases. We also found that pairing the 2OPT and GRD algorithms with the ratio GCM leads to test prioritizations with high coverage effectiveness values that are also greater than the scores associated with the reverse and initial orderings. Moreover, the empirical study demonstrates that the algorithms operate efficiently for each case study application. As part of future work, we intend to evaluate the reduction and prioritization mechanisms with larger case study applications. Furthermore, we will use fault databases and mutation testing tools to determine how the techniques fare in creating test suites that effectively detect defects. Finally, we plan to improve the evaluation process by employing additional statistical analyses (e.g., multiple comparisons with an analysis of variance and the Tukey post-hoc test).

7. REFERENCES

[1] M. J. Crawley. *The R Book*. John Wiley & Sons, Inc., 2007.

[2] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. of ISSRE*, 2004.

[3] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, 28(2), 2002.

[4] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM TOSEM*, 2(3), 1993.

[5] G. M. Kapfhammer. *A Comprehensive Framework for Testing Database-Centric Applications*. PhD thesis, University of Pittsburgh, Pittsburgh, Pennsylvania, 2007.

[6] G. M. Kapfhammer and M. L. Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *Proc. of WEASELTech*, 2007.

[7] G. M. Kapfhammer and M. L. Soffa. Database-aware test coverage monitoring. In *Proc. of ISEC*, 2008.

[8] G. M. Kapfhammer, M. L. Soffa, and D. Mosse. Testing in resource constrained execution environments. In *Proc. of ASE*, 2005.

[9] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE TSE*, 33(4), 2007.

[10] A. G. Malishevsky, J. Ruthruff, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, University of Nebraska - Lincoln, 2006.

[11] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *CACM*, 32(12), 1989.

[12] S. McMaster and A. Memon. Call stack coverage for test suite reduction. In *Proc. of ICSM*, 2005.

[13] S. McMaster and A. Memon. Call stack coverage for GUI test-suite reduction. In *Proc. of ISSRE*, 2006.

[14] S. McMaster and A. M. Memon. Fault detection probability analysis for coverage-based test suite reduction. In *Proc. of ICSM*, 2007.

[15] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proc. of ICSE*, 2005.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE TSE*, 27(10), 2001.

[17] A. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa. Test suite reduction and prioritization with call trees. In *Proc. of ASE*, 2007.

[18] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proc. of PASTE*, 2005.