Regression Testing

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

Appears in the *Encyclopedia of Software Engineering* (Taylor and Francis)

Keywords: test suite execution, test coverage monitoring, test suite selection, reduction and prioritization, efficiency and effectiveness trade-offs

Abstract. Regression testing techniques execute a test suite whenever the addition of defect fixes or new functionality changes the program under test. The repeated execution of a test suite aims to establish a confidence in the correctness of the software application and identify defects that were introduced by the program modifications. Industry experiences suggest that regression testing often improves the quality of the application under test. However, testing teams may not always perform regression testing because the frequent execution of the tests often incurs high time and space overheads. Test suite *selection* techniques try to reduce the cost of testing by running a subset of the tests, such as those that execute the modified source code, in order to ensure that the updated program still operates correctly. Alternatively, *reduction* methods decrease testing time overheads by discarding the tests that redundantly cover the test requirements. Approaches to test suite *prioritization* reorder the test cases in an attempt to maximize the rate at which the tests achieve a testing goal such as code coverage. After describing a wide variety of metrics for empirically evaluating different regression testing methods, this chapter considers the efficiency and effectiveness trade-offs associated with these techniques. The conclusion of this article summarizes the state-of-the-art in the field of regression testing and then offers suggestions for future work and resources for further study.

1 INTRODUCTION

Regression testing is an important software maintenance activity that involves repeatedly running a test suite whenever the program under test and/or the program's execution environment changes. Executing a regression test suite upon the introduction of either a defect fix or a new feature ensures that the modification of the program does not negatively impact the overall correctness. Recent industry reports suggest that (i) software engineers often use regression testing techniques [1] and (ii) employing regression testing methods often leads to a software application with high observed quality [2]. However, regression testing can be prohibitively expensive, particularly with respect to time [3], and thus accounts for as much as half the cost of software maintenance [4, 5]. In Rothermel et al. [5], an industrial collaborator reported that for one of its products of approximately 20,000 lines of code, the entire test suite required seven weeks to run.

Since several well-known software failures, such as the Ariane-5 rocket and the 1990 AT&T outage, can be blamed on not testing changes in a software system [6], many techniques have been developed to support efficient and effective regression testing. For instance, a *test suite execution* (TSE) component (e.g., JUnit, CppUnit, or NUnit) runs a large test suite in an automated and repeatable manner. As the test suite executes, a *test coverage monitor* (TCM) tracks how the test cases cover the test requirements that normally correspond to the program's state or structure (e.g., methods, statements, or definition-use associations). Whenever coverage information is available, reduction, prioritization, and selection algorithms can analyze the relative contribution of each test case in order to improve the regression test suite.

Regression test suite *reduction* techniques aim to control both the size and the execution time of a test suite by discarding the tests that redundantly cover the test requirements. In an attempt to improve the effectiveness of testing, approaches to test suite *prioritization* reorder the test cases according to an established priority metric. For instance, the prioritizer may rearrange the tests so that they cover the test requirements at a faster rate than the original ordering. Alternatively, a test prioritization method may addresses the challenge of running a test suite in a constrained environment where computational resources such as time and memory are limited. Test suite *selection* techniques try to reduce the cost of testing by only running those test cases that are most likely to ensure that the modified program still operates correctly (e.g., the tests that exercise the modified source code of the program).

There are many costs and benefits associated with the regression testing process. Both practitioners and researchers must conduct experiments in order to ascertain the trade-offs between the efficiency and the effectiveness of the chosen method(s) for regression testing. For instance, it is important to measure the time overhead associated with executing the tests and monitoring the coverage of the requirements. Experiments must also determine the time and space costs of using a selection, reduction, or prioritization algorithm and then evaluate these overheads in the context of the potentially diminished cost and increased effectiveness of the modified test suite. For example, the empirical characterization of a reduction technique frequently measures the decrease in the size and execution time of the test suite [7, 8], preservation of the original test suites' coverage [8, 9], or the amount of tests that are found in common for test suites produced by different reduction methods [10]. After characterizing the coverage density of a test suite [11], experimental studies of test prioritization schemes typically focus on evaluating the change in metrics such as coverage effectiveness (CE) [9], average percentage of blocks covered (APBC) [12], and average percentage of faults detected (APFD) [4, 13].

In summary, the important contributions of this chapter are as follows:

1. An overview of a model for the regression testing process (Section 2.1).
2. A description of the reduction, prioritization, and selection techniques that are often used during regression testing (Sections 2.2 through 2.7).
3. The definition of the metrics that evaluate the efficiency and effectiveness of different approaches to regression testing (Section 3).
4. A review of the important advances and the current state of the art in the field of regression testing (Sections 4 and 5).

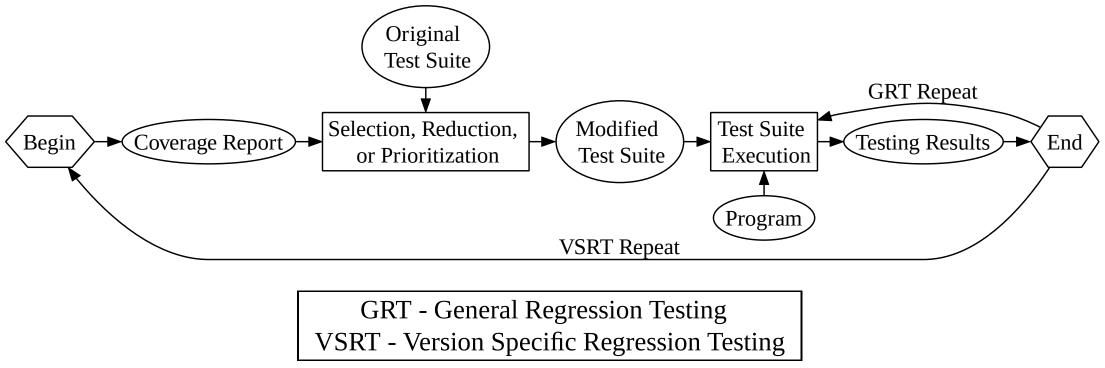


Figure 1: Overview of the Regression Testing Process.

2 REGRESSION TESTING TECHNIQUES

2.1 REGRESSION TESTING MODEL

Figure 1 provides an overview of a model for the regression testing process. In the *general* regression testing (GRT) framework, we apply selection, reduction, and/or prioritization to the test suite and then use the modified suite during many subsequent rounds of test suite execution [5]. This cost-effective approach to testing is motivated by empirical studies demonstrating that the adequacy of a test suite does not markedly change across subsequent versions of a program [14]. Alternatively, the *version specific* regression testing (VSRT) model suggests that the test suite should be re-analyzed after each modification to the program under test [5]. VSRT requires efficient implementations of the (i) test suite executor, (ii) test coverage monitor, and (iii) selection, reduction, and prioritization techniques. If the method for constructing the modified test suite is expensive, then the GRT framework supports the amortization of this cost over many executions of the tests. Yet, VSRT is more likely to improve the effectiveness of regression testing because it always leverages the most current information about the program and the tests. Furthermore, testers should consider the VSRT approach whenever the program and/or the test suite undergo a series of substantial changes. Of course, any regression testing approach that can efficiently operate in a version specific fashion should also enable GRT.

Regression testing establishes a confidence in the correctness of and isolates defects within a program by running a collection of tests known as a *test suite*. This chapter defines a test suite $T = \langle t_1, \ldots, t_n \rangle$ as a tuple (i.e., an ordered list) of n individual test cases. Intuitively, each test case $t_i \in T$ invokes one or more of the methods under test and inspects the output(s) in order to see if the operations worked correctly. We require that each test in T be *independent* so that we can guarantee that there are no test execution ordering dependencies [4, 15]. Many real world test suites exhibit test independence because the most popular test automation frameworks (e.g., JUnit, CppUnit, or NUnit) provide `setUp` and `tearDown` methods that respectively execute before and after each test case. Test case independence also makes it more likely that regression testing schemes (e.g., selection, reduction, and prioritization) will create a modified test suite that is both efficient and effective. For instance, test independence enables a prioritizer to reorder tests in any sequence that maximizes the suite's ability to cover the test requirements and find

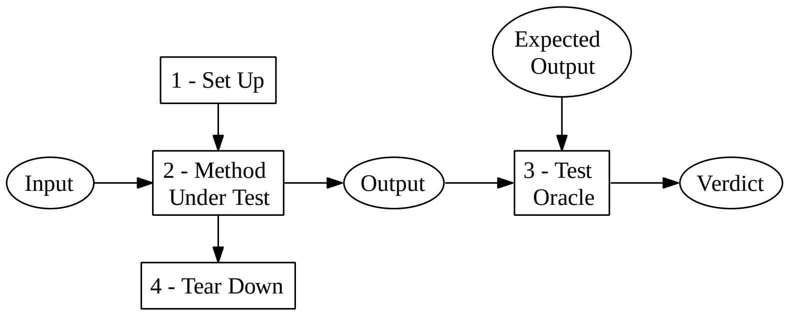


Figure 2: Overview of the Test Suite Execution Process.

the defects. All of the testing techniques described in this chapter may also be applied, albeit in a potentially diminished capacity, to non-independent regression test suites.

2.2 TEST SUITE EXECUTION

Figure 2 describes the process of executing a test case. As previously mentioned in Section 2.1, a `setUp` operation runs before the invocation of the method under test in order to perform any required initializations. For example, if the program interacts with a file system or a database, then `setUp` may populate these components with files or data. Alternatively, if the program uses a network, then `setUp` could establish a new network connection. Upon completion of the initialization procedure, the test case calls the program's method with the input that the test constructs. The test case captures the output of the method and provides the return value to the *test oracle* that determines whether the test passed or failed.

While tools may automatically generate oracles in certain circumstances (e.g., when it is possible to predict the output of new tests based upon the input and output of existing test cases [16]), often the tester manually implements the oracles. The oracle returns a failing verdict when the expected output does not match the actual output and a passing verdict results when the two outputs are equivalent. Finally, `tearDown` cleans up after the test case and thus ensures that it is independent. Depending on the configuration of the test suite executor, the regression testing process continues until either all of the n tests have executed or an oracle indicates that a test case failed (this behavior is the default for most versions of the JUnit test automation framework).

In order to make the discussion of test suite execution more concrete and to illustrate the challenges of testing, Figure 3 summarizes the outcomes associated with testing a Java class called `Kinetic` [4]. As shown in Figure 4, this class contains a `computeVelocity` method designed to calculate the velocity of an object based on its kinetic energy and mass. Since the kinetic energy of an object, K, is defined as $K = \frac{1}{2}mv^2$, it is clear that `computeVelocity` contains a defect on line 10. That is, line 10 should have the assignment statement `velocity_squared = 2 * (kinetic / mass)`. Furthermore, Figure 5 gives a test suite that will run in the JUnit 3.8.1 framework for automated test execution (a slightly modified version of this suite will fullfil the requirements of the more recent 4.8.1 version of JUnit). Interestingly, the results in Figure 3 reveal that only one of the four tests, t_4, reveals the fault in the `Kinetic` class.

Test Case	kinetic	mass	expected	actual	Verdict
`testOne` - t_1	5	0	Undefined	Undefined	Pass
`testTwo` - t_2	0	5	0	0	Pass
`testThree` - t_3	8	1	4	4	Pass
`testFour` - t_4	1000	5	20	24	Fail

Figure 3: Summarizing the Outcomes of Test Suite Execution for `computeVelocity`.

For instance, t_1 cannot isolate the fault in `computeVelocity` because it does not execute line 10 of the program. Test t_2 is also incapable of detecting the defect since an input of `kinetic = 0` results in `velocity_squared = velocity = final_velocity = 0` and thus leads to an inadvertently passing test case. Test t_3 passes even after the method incorrectly assigns `velocity_squared = 24` and subsequently computes `Math.sqrt(velocity_squared) = 4.898979` instead of the correct value of `Math.sqrt(velocity_squared) = 4.0`. Test t_3's inability to find the defect is due to the fact that line 11 of `computeVelocity` masks the faulty computation by casting the `velocity` variable as an `int` and arriving at the expected result of `velocity = 4`. Yet, Figure 3 shows that t_4 is capable of isolating the defect because it executes the faulty location, changes the value of the `final_velocity` variable, and returns the incorrect result to the calling test case. In summary, this example demonstrates that tests often do not have the same fault detection effectiveness. We also see that a program fault only manifests itself in a test failure when the input(s) (i) cause the execution of the defective location, (ii) change the values of the program's variables, and (iii) force the method to return an erroneous answer [17]. It is also evident that an effective regression testing process must use a test suite executor that can (i) repeatedly run the test cases, (ii) capture the output of the method under test, and (iii) issue a final verdict after using an oracle to compare the value of the `expected` and `actual` variables.

2.3 TEST ADEQUACY CRITERIA

Ideally, a regression testing technique would utilize knowledge about program faults as it reordered and reduced the test cases. Since it is normally difficult to collect information about the existence of faults within the program under test, regression testing methods must use a proxy for this type of complete knowledge. After identifying a type of test requirement that "good" test cases should aim to exercise, testers can calculate the *adequacy* of a test case as the ratio between the covered requirements and the total number of requirements. For example, a criterion that concentrates on the *control flow* of the program under test does so with the realization that a defect cannot be detected unless the test case executes the faulty location in the program's source code. Alternatively, a *data flow* adequacy criterion focuses on the definition and use of variables because a program will only be able to determine if it assigned the correct value to a variable when it subsequently uses the variable. An adequacy criterion could also consider the coverage of the program's methods and the context in which the methods were invoked during testing.

Current regression testing methods are often motivated by empirical investigations of the effectiveness of test adequacy criteria which indicate that the low adequacy tests are often unlikely to reveal program defects [18]. If adequacy information is available, then a test prioritizer could execute highly adequate tests before those with lower adequacy. As a concrete illustration of the concept of test adequacy, this chapter briefly examines criteria that focus on definition-use

```java
import java.lang.Math;
public class Kinetic
{
  public static String computeVelocity(int kinetic, int mass)
  {
    int velocity_squared, velocity;
    StringBuffer final_velocity = new StringBuffer();
    if( mass != 0 )
    {
      velocity_squared = 3 * (kinetic / mass);
      velocity = (int)Math.sqrt(velocity_squared);
      final_velocity.append(velocity);
    }
    else
    {
      final_velocity.append("Undefined");
    }
    return final_velocity.toString();
  }
}
```

Figure 4: The `Kinetic` Class that Contains a Fault in the `computeVelocity` Method.

```java
import junit.framework.*;
public class TestKinetic extends TestCase
{
  public TestKinetic(String name)
  {
    super(name);
  }
  public static Test suite()
  {
    return new TestSuite(TestKinetic.class);
  }
  public void testOne()
  {
    String expected = new String("Undefined");
    String actual = Kinetic.computeVelocity(5,0);
    assertEquals(expected, actual);
  }
  public void testTwo()
  {
    String expected = new String("0");
    String actual = Kinetic.computeVelocity(0,5);
    assertEquals(expected, actual);
  }
  public void testThree()
  {
    String expected = new String("4");
    String actual = Kinetic.computeVelocity(8,1);
    assertEquals(expected, actual);
  }
  public void testFour()
  {
    String expected = new String("20");
    String actual = Kinetic.computeVelocity(1000,5);
    assertEquals(expected, actual);
  }
}
```

Figure 5: A JUnit 3.8.1 Test Suite for the Faulty `Kinetic` Class.

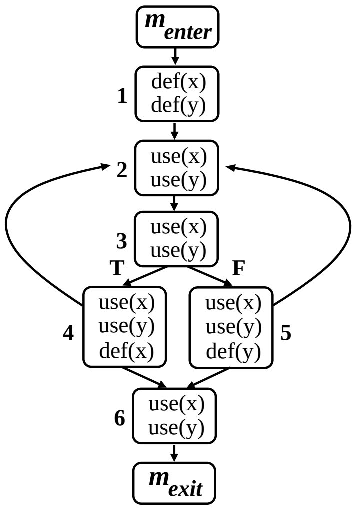


Figure 6: Example of a Graph-Based Representation for a Simple Program Under Test.

associations [19] and call tree paths [7, 8]. Data flow-based criteria are frequently very effective because they consider both the values stored in the program's variables and the structure of the program itself. Since data flow-based adequacy criteria often require the use of a potentially expensive algorithm to enumerate the definition-use associations, these criteria may not support the version specific model of regression testing [9, 20]. Furthermore, it is difficult to apply data flow criteria to programs for which testers lack access to the source code. Alternatively, this chapter considers call tree paths, a efficient-to-compute criterion that operates without source code access by focusing on the contextual coverage of the methods invoked during testing.

In data flow-based adequacy criteria, the occurrence of a variable on the left hand side of an assignment statement is called a *definition* of this variable. The *use* of a variable takes place when it appears on the right hand side of an assignment statement or in the predicate of a conditional logic statement or an iteration construct [18]. For example, the assignment statement `x = x + y` uses the variables `x` and `y` and then defines `x`. Figure 6 furnishes an intuitive depiction of a graph-based representation for a method under test. In this diagram, a node represents a computation and an edge stands for the transfer of control between two separate statements within the program [4]. The node labeled with a "4" in Figure 6, denoted here as N_4, would represent the definition and uses of program variables for the statement `x = x + y`.

A data flow-based adequacy criterion called *all-DUs* requires a test suite to cover a *definition-use association* $\langle N_d, N_u, var \rangle$ where the definition of variable *var* occurs in graph node N_d and a use of *var* occurs in node N_u [18]. For instance, the coverage report in Figure 7 reveals that $\langle N_1, N_4, x \rangle$ is one of the sixteen definition-use associations within the graph provided by Figure 6. Figure 7 also shows that a test case exercising the path $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6$ will cover seven of the associations and thus lead to an adequacy score of .4375. Yet, we see that a test case that also follows the edge $N_4 \rightarrow N_2$ will increase its measure of adequacy to .5625. Various approaches to regression testing use data flow information, such as the definition-use associations described in Figure 7, during the reorganization of a test suite. For example, a prioritizer may order the suite so that the first tests to run have the highest data flow adequacy.

$R_1 = \langle N_1, N_2, \mathtt{x} \rangle$	$R_9 = \langle N_4, N_2, \mathtt{x} \rangle$
$R_2 = \langle N_1, N_2, \mathtt{y} \rangle$	$R_{10} = \langle N_5, N_2, \mathtt{y} \rangle$
$R_3 = \langle N_1, N_3, \mathtt{x} \rangle$	$R_{11} = \langle N_5, N_3, \mathtt{y} \rangle$
$R_4 = \langle N_1, N_3, \mathtt{y} \rangle$	$R_{12} = \langle N_4, N_3, \mathtt{x} \rangle$
$R_5 = \langle N_1, N_4, \mathtt{x} \rangle$	$R_{13} = \langle N_4, N_6, \mathtt{x} \rangle$
$R_6 = \langle N_1, N_4, \mathtt{y} \rangle$	$R_{14} = \langle N_5, N_6, \mathtt{y} \rangle$
$R_7 = \langle N_1, N_5, \mathtt{x} \rangle$	$R_{15} = \langle N_4, N_5, \mathtt{x} \rangle$
$R_8 = \langle N_1, N_5, \mathtt{y} \rangle$	$R_{16} = \langle N_5, N_4, \mathtt{y} \rangle$

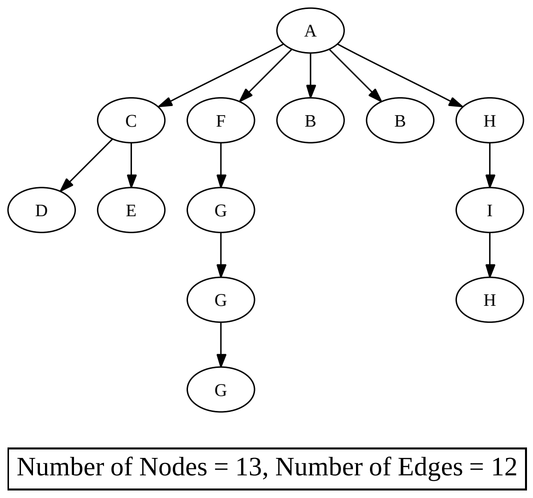
Path	Covered Associations	Adequacy
$N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6$	R_1, R_2, R_3, R_4, R_5, R_6, R_{13}	$\frac{7}{16} = .4375$
$N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6$	R_1, R_2, R_3, R_4, R_5, R_6, R_9, R_{12}, R_{13}	$\frac{9}{16} = .5625$

Figure 7: Test Requirements for the *all-DUs* Adequacy Criterion.

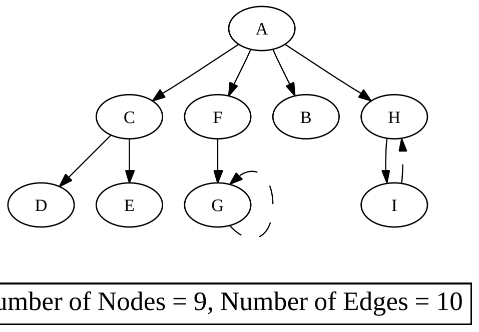
Alternatively, McMaster and Memon present a test adequacy criterion that obviates the need for source code access by measuring method coverage in the context in which the methods were invoked during testing [7]. A coverage report for this criterion corresponds to either a *dynamic call tree* (DCT) or a *calling context tree* (CCT) representing the dynamic behavior of the program while a test suite runs. Each node in a DCT or 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. Finally, a call tree path from the root node to a leaf node forms a test requirement. Regression testing methods may use call tree paths as a test adequacy criterion because these trees are efficient to collect and store, thus enabling the modification of a test suite each time the program under test changes [9, 21]. Although DCTs and CCTs may be criticized for not incorporating either the source code or parameters of the methods under test or the state of the program, they have been shown to perform closely to other criteria with respect to common fault detection metrics [22].

Figure 8(a) provides an example of a DCT with thirteen nodes and twelve edges. In this tree a node with the label "A" corresponds to the invocation of the method A and the edge A $\rightarrow$ B indicates that method A invokes method B. The existence of the two DCT edges A $\rightarrow$ B reveals that method A repeatedly invokes method B. In the example from Figure 8(a), the DCT represents the recursive invocation of method G by chaining together edges of the form G $\rightarrow$ G. In an attempt to reduce the size of the coverage report, the CCT in Figure 8(b) coalesces the DCT nodes and yields a 30.8% reduction in the number of nodes and a 16.7% decrease in the number of edges. For example, the CCT combines the two B nodes in the DCT into a single node. The CCT also coalesces nodes and introduces back edges when a method calls itself recursively (e.g., the DCT path G $\rightarrow$ G $\rightarrow$ G) or a method is run repeatedly (e.g., the DCT path H $\rightarrow$ I $\rightarrow$ H).

Using call tree paths as a test requirement enables regression testing techniques to determine which test cases may be redundant [7]. For instance, comparing a test that only causes method



(a)



(b)

Figure 8: Examples of the (a) DCT and (b) CCT for Use in the Call Tree Adequacy Criterion.

A to invoke method B to another test that yields one of the call trees in Figure 8 suggests that the first test case is potentially redundant. If testing time is constrained, then a test suite reduction method may discard a test case that does not cause the program under test to create unique sequences of methods invocations. As an alternative, a prioritizer may reorder a test suite so that it covers all of the unique method contexts as quickly as is possible.

2.4 TEST COVERAGE MONITORING

Given a test suite T, a test coverage monitor identifies a set of covered requirements $\mathscr{R}(T) = \{R_1, R_2, \ldots, R_m\}$. Each test t_i is associated with a non-empty subset of requirements $\mathscr{R}(t_i) \subseteq \mathscr{R}(T)$ 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 $\mathscr{T}(R_j) \subseteq T$ such that R_j is covered by each test in $\mathscr{T}(R_j)$. For instance, if test t_i creates the first path in Figure 7 and $\mathscr{R}(T) = \{R_1, \ldots, R_{16}\}$ is the set of requirements, then we know that t_i covers the requirements $\mathscr{R}(t_i) = \{R_1, R_2, R_3, R_4, R_5, R_6, R_{13}\}$. Since each of the requirements in $\mathscr{R}(t_i)$ are covered by t_i, we can also write that $t_i \in \mathscr{T}(R_j)$ for every $R_j \in \mathscr{R}(t_i)$. The test coverage monitor tracks what occurs during the execution of the regression test suite T in order to populate the set $\mathscr{R}(T)$ and for each t_i

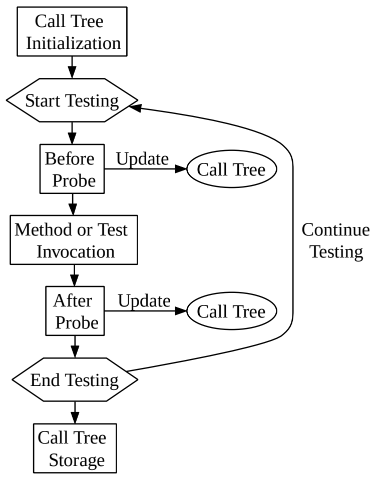


Figure 9: Call Tree Construction Probes for Test Coverage Monitoring.

and R_j construct the respective sets $\mathscr{R}(t_i)$ and $\mathscr{T}(R_j)$. Test coverage monitoring techniques place instrumentation *probes* into the program under test in order to report which test requirements are covered during the execution of T. Among other goals, the instrumentation must efficiently track coverage without changing the behavior of the program and the test suite [9, 20, 21].

This chapter primarily uses the call tree path coverage criterion to support the discussion of the instrumentation and test coverage monitoring process. As shown in Figure 9, the use of call tree-based test adequacy requires probes to execute before and after the execution of both a test case and a method. Each time a probe executes, it must update the call tree so that it correctly reflects the test execution history and eventually results in a tree like the ones in Figure 8. Since these probes do not initially exist in the program under test, the coverage monitor must place them into the methods of the program. Using aspect-oriented programming (AOP) techniques and tools such as AspectJ, a call tree constructor inserts instrumentation probes in either a static or dynamic fashion [8, 21]. A *static* instrumentor places the probes into the program before test suite execution whereas *dynamic* instrumentation methods insert the probes as the tests run. While static instrumentation must take place each time the program under test changes, dynamic instrumentors modify the program during testing, thus improving the flexibility of the monitor at the cost of a potential increase in run-time overheads.

As an example, test coverage monitors for Java programs can use either the Java virtual machine tools interface (JVMTI) or a custom class loader in order to perform dynamic instrumentation at class load-time [9]. While this approach is simple and easy to implement, it may insert probes that are not necessary and it cannot support the gathering of information for certain types of adequacy criteria (e.g., definition-use associations for program variables). Alternatively, Misurda et al. present the Jazz test coverage monitor that records information about the execution of control flow-based (e.g., edges and nodes) and data flow-based (e.g., definition-use associations) test requirements [20]. While more complicated than the use of AspectJ to construct call trees, this instrumentation scheme is unique because it incrementally removes the probes after a test case exercises the associated test requirements. In summary, instrumentation methods vary in their ability to capture various aspects of program behavior and they are commonly tailored to track the coverage of requirements for one or more specific test adequacy criteria.

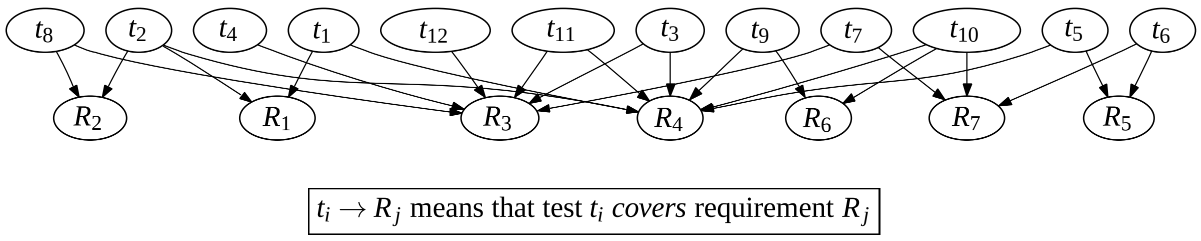


Figure 10: An Example of Overlap in the Coverage of the Test Requirements.

2.5 REDUCING AND PRIORITIZING TEST SUITES

Figure 10 visualizes a coverage report that a coverage monitor constructed after running a test suite T consisting of tests $\langle t_1, \ldots, t_{12} \rangle$ and requirements $\mathscr{R}(T) = \{R_1, \ldots, R_7\}$. Using the notation established in Section 2.4, this example illustrates coverage relationships such as $\mathscr{R}(t_1) = \{R_1, R_4\}$ and $\mathscr{T}(R_1) = \{t_1, t_2\}$. Since the test suite in Figure 10 contains a significant amount of overlap in test requirement coverage, it is a candidate for reduction. In fact, inspection of Figure 10 reveals that executing a reduced test suite containing $\langle t_2, t_3, t_6, t_9 \rangle$ instead of the original twelve tests will still cover all of the seven test requirements (other reductions are also possible for this test suite). Even though test suite reduction maintains complete coverage of the requirements, it does not guarantee the same fault detection capabilities as the original test suite [7, 22, 23]. If a tester is concerned that test suite reduction might compromise the fault detection effectiveness of the suite, then it may be reasonable to reorder the tests. For instance, a test suite prioritizer could construct a test sequence that runs the high coverage test cases (i.e., $\mathscr{R}(t_{10}) = \{R_4, R_6, R_7\}$) before the tests that cover few requirements (i.e., $\mathscr{R}(t_{12}) = \{R_3\}$).

Reduction methods attempt to produce a new test suite 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 a completely reordered suite. As shown in Figure 11, it is possible to prioritize a test suite by repeatedly invoking a reduction algorithm on successively smaller subsets of the tests [8]. Given a test suite T and test coverage set $\mathscr{R}(T)$ as input, the *PrioritizationViaRepeatedReduction* algorithm initializes T_p to the empty set and assigns $\mathscr{R}(T)$ as the set of live requirements $\mathscr{R}_\ell(T)$. While there are still tests remaining in T, the algorithm repeatedly uses a reduction technique, such as the *GreedyReductionWithOverlap* algorithm described in Section 2.5.1, to find a reduced suite T_r. Each iteration of the loop starting on line 2 of Figure 11 uses the order preserving union operator, denoted $\uplus$, to add the tests from the resulting T_r to T_p and then recalculates the live requirements $\mathscr{R}_\ell(T)$.

Figure 12 furnishes an example of this process for the small test suite that is provided to the right of the diagram. The checkmarks in this coverage report reveal that t_1 covers four requirements (i.e., R_1, R_2, R_3, and R_4) while test case t_4 covers only two requirements (i.e., R_1 and R_4). When given the original test suite $T = \langle t_1, t_2, t_3, t_4 \rangle$ the reduction algorithm produces the first output $T_{r1} = \langle t_1, t_4 \rangle$ and two residual tests t_2 and t_3. In this situation, the reduction algorithm incrementally picks the test case that covers the most currently uncovered requirements. After the first iteration, the residual tests are then once again passed to the reduction technique, yielding the second output $T_{r2} = \langle t_2, t_3 \rangle$. Using the $\uplus$ operator to concatenate T_{r1} and T_{r2} creates the prioritized test suite $T_p = \langle t_1, t_4, t_3, t_2 \rangle$ that *PrioritizationViaRepeatedReduction* ultimately returns.

```
Algorithm PrioritizationViaRepeatedReduction($T$,$\mathscr{R}(T)$)
Input: Test Suite $T = \langle t_1, \ldots, t_n \rangle$;
    Test Coverage Set $\mathscr{R}(T)$
Output: Prioritized Test Suite $T_p$
1.  $T_p \leftarrow \emptyset$, $\mathscr{R}_\ell(T) \leftarrow \mathscr{R}(T)$
2.  while $T \neq \emptyset$
3.      do $T_r \leftarrow ReductionTechnique(T, \mathscr{R}_\ell(T))$
4.          $T_p \leftarrow T_p \uplus T_r$
5.          $\mathscr{R}_\ell(T) \leftarrow \emptyset$
6.          for $t_i \in T$
7.              do $\mathscr{R}_\ell(T) \leftarrow \mathscr{R}_\ell(T) \cup \mathscr{R}(t_i)$
8.  return $T_p$
```

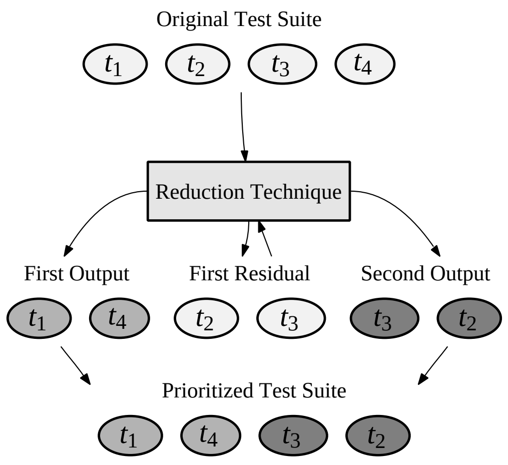
Figure 11: The *PrioritizationViaRepeatedReduction* Algorithm.

Figure 13's classification scheme for reduction and prioritization methods reveals that this chapter considers approaches involving greedy choices, the use of heuristic search, or the reversal or random shuffling of a test suite. As shown in this diagram, this chapter describes an *overlap-aware* greedy technique that is based on the approximation algorithm for the minimal set cover problem [24]. Greedy reduction with overlap awareness iteratively selects the most cost-effective test case for inclusion in the reduced test suite. During every successive iteration, the overlap-aware greedy algorithm re-calculates the cost-effectiveness for each leftover test according to how well it covers the remaining test requirements. This reduction technique terminates when the reduced test suite covers all of the test requirements that the initial tests cover.

Prioritization that is not overlap-aware re-orders the tests by sorting them according to a cost-effectiveness metric [5, 19]. When provided with a target size for the reduced test suite, the reducer that ignores overlap will sort the tests by cost-effectiveness and then pick test cases until the new test suite reaches the size limit. The overlap-aware reduction and prioritization techniques have the potential to identify a new test suite that is more effective than the suite that was created by methods that ignore the overlap in requirement coverage. However, a method that considers overlap may require more execution time than one that disregards this information.

There are also a wide variety of *custom* greedy algorithms for test suite reduction and prioritization. For instance, 2-OPT is an all-pairs greedy approach that compares each pair of tests to all other pairs and picks the best according to a cost-effectiveness metric [12]. The Harrold, Gupta, Soffa (HGS) algorithm constructs a reduced test suite by leveraging the *covered by* information available in the set $\mathscr{T}(R_j)$ for each requirement R_j [25]. The delayed greedy (DGR) method consults both $\mathscr{R}(t_i)$ and $\mathscr{T}(R_j)$ in order to identify the (i) tests that will not improve the reduced suite and (ii) requirements that the best tests already cover [26]. While both HGS and DGR were initially designed to support reduction, it is easy to integrate both of these methods into the *PrioritizationViaRepeatedReduction* algorithm shown in Figure 11 [8].

Given a suitable objective function that evaluates test suite quality, it is often possible to employ *heuristic search* techniques (e.g., hill climbing, genetic algorithms, tabu search, and simulated annealing) to reorder or reduce the tests [12, 27]. Figure 13 also indicates that a regression testing method may prioritize the test suite by simply *reversing* the initial test sequence [15]. This scheme may be useful if a tester always adds new, and possibly more effective, tests to



Coverage Report for
$T = \langle t_1, t_2, t_3, t_4 \rangle$

	R_1	R_2	R_3	R_4	R_5
t_1	✓	✓	✓	✓	
t_2				✓	
t_3	✓	✓			
t_4	✓				✓

Figure 12: An Example of Test Suite Prioritization by Repeated Reduction.

the end of the test suite. Test reduction via reversal selects tests from the reversed test suite until reaching the provided target size. During the evaluation of different testing strategies, both researchers and practitioners may also employ *random* reduction and prioritization as a form of experimental control [5, 15, 28]. Recent empirical studies demonstrate that these approaches to reduction and prioritization often improve the testing process. For instance, in the context of JUnit tests for Java programs, like those in Section 2.2, Do et al. draw the following conclusion: “the worst thing that JUnit users can do is not practice some form of prioritization” [28].

2.5.1 GREEDY METHODS

Figure 14 provides the *GreedyReductionWithOverlap* (GRO) algorithm that produces the reduced test suite T_r after repeatedly analyzing how each remaining test covers the requirements in $\mathscr{R}(T)$. As evidence by line 13 of Figure 14, this algorithm also uses $\uplus$, the order preserving union operator, to build up the final suite. GRO initializes the reduced test suite, denoted T_r, to the empty set and iteratively adds to it the most cost-effective test. Equation (1) defines the greedy cost-effectiveness ratio ρ_i for test case t_i.[1] This equation uses the $time(\langle t_i \rangle)$ function to calculate the execution time of the singleton test tuple $\langle t_i \rangle$. More generally, we require $time(\langle t_1, \ldots, t_n \rangle)$ to return the time overhead associated with executing all of the n tests in the input tuple. According to Equation (1), ρ_i is the average cost at which test case t_i covers the $|\mathscr{R}(t_i) \setminus \mathscr{R}(T_r)|$ requirements that are not yet covered by T_r [24]. Therefore, each iteration of GRO's outer **while** loop finds the test case with the lowest cost-effectiveness value and places it into T_r.[2]

$$\rho_i = \frac{time(\langle t_i \rangle)}{|\mathscr{R}(t_i) \setminus \mathscr{R}(T_r)|} \tag{1}$$

[1]Without loss of generality, this chapter focuses on using the cost to coverage ratio during test case evaluation. It is also possible to reduce and prioritize the test suite by exclusively focusing on either the cost or the coverage information. However, we chose this definition of ρ_i because recent empirical studies suggest that the cost-effectiveness ratio may lead to better orderings and reductions of the test cases [8].

[2]While many different implementations are acceptable, this chapter assumes that all of the greedy regression testing methods use a random choice to resolve a tie in the test case effectiveness scores.

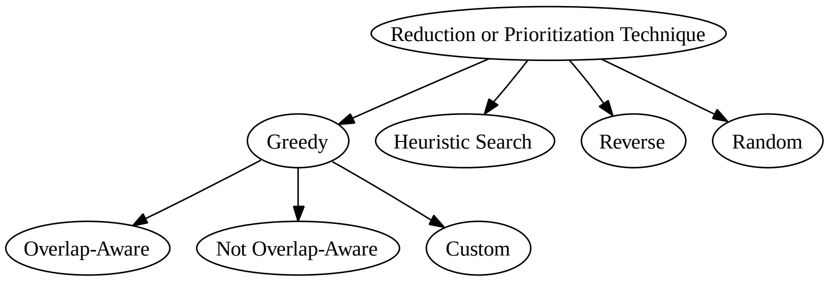


Figure 13: Classifying Several Approaches to Test Suite Reduction and Prioritization.

GRO initializes the temporary test suite $\hat{T}$ to contain all of T's tests and then selects test cases from $\hat{T}$. Line 2 of Figure 14 shows that GRO terminates when $\mathscr{R}(T_r) = \mathscr{R}(T)$. Lines 5 through 12 are responsible for (i) identifying t_k, the next test that GRO will opt to keep in T_r, and (ii) removing any non-viable test t_i that does not cover at least one of the un-covered requirements (i.e., t_i is *non-viable* when $\mathscr{R}(t_i) \setminus \mathscr{R}(T_r) = \emptyset$). Lines 13 and 14 respectively place t_k into T_r and then remove this test from $\hat{T}$ so that it is not considered during later executions of GRO's outer **while** loop. Finally, line 15 augments $\mathscr{R}(T_r)$ so that this set contains $\mathscr{R}(t_k)$, the set of requirements that t_k covers. Since we want GRO to support prioritization via successive invocations of the reducer, line 16 updates T so that it no longer contains any of the tests in T_r. We know that *GreedyReductionWithOverlap* is $O(m \times n)$ because the algorithm contains a **for** loop nested within a **while** loop and it analyzes n tests that cover a total of m requirements [5, 24].

Figure 15 gives the *GreedyPrioritizationWithOverlap* (GPO) algorithm that uses the GRO algorithm to re-order test suite T according to its coverage of the requirements in $\mathscr{R}(T)$. GPO initializes the prioritized test suite T_p to the empty set and uses $\mathscr{R}_\ell(T)$ to store the live test requirements. We say that a requirement is *live* as long as it is covered by a test case that remains in T after one or more calls to *GreedyReductionWithOverlap*. Each invocation of GRO yields both (i) a new reduced T_r that we place into T_p and (ii) a smaller number of residual tests in the original T. After each round of reduction, lines 5 through 7 reinitialize $\mathscr{R}_\ell(T)$ to the empty set and insert all of the live requirements into this set. GPO uses the newly populated $\mathscr{R}_\ell(T)$ during the next call to GRO. Line 2 shows that the prioritization process continues until $T = \emptyset$. The worst-case time complexity of *GreedyPrioritizationWithOverlap* is $O(n \times (m \times n) + n^2)$ or $O(n^2 \times (1 + m))$. The $n \times (m \times n)$ term in the time complexity stands for GPO's repeated invocation of *GreedyReductionWithOverlap* and the n^2 term corresponds to the cost of iteratively populating $\mathscr{R}_\ell(T)$ during each execution of the outer **while** loop. Since overlap-aware greedy prioritization must re-order the entire test suite, it is more expensive than GRO in the worst case.

Figure 16 describes the *GreedyReductionWithoutOverlap* (GR) algorithm that reduces a test suite T to the target size $n^* \in \{0, \ldots, n-1\}$. GR uses the ρ_i metric, as defined in Equation (1), when it sorts the tests in T in ascending order. Figure 16 shows that GR stores the output of *Sort*(T, ρ) in $\hat{T}$ and then creates T_r so that it contains $\hat{T}$'s first n^* tests (i.e., we use the notation $\hat{T}[1, n^*]$ to denote the sub-tuple $\langle t_1, \ldots, t_{n^*} \rangle$). Finally, Figure 18 demonstrates that *GreedyPrioritizationWithoutOverlap* (GP) returns the test suite that results from sorting T according to ρ. If we assume that the enumerating $T[1, n^*]$ takes linear time, then GR is $O(n \times \log_2 n + n^*)$ and GP is $O(n \times \log_2 n)$. These time complexities both include an $n \times \log_2 n$ term because they use a variant

```
**Algorithm** *GreedyReductionWithOverlap*$(T, \mathscr{R}(T))$
**Input:** Test Suite $T = \langle t_1, \dots, t_n \rangle$;
       Test Coverage Set $\mathscr{R}(T)$
**Output:** Reduced Test Suite $T_r$
1.   $T_r \leftarrow \emptyset$, $\mathscr{R}(T_r) \leftarrow \emptyset$, $\hat{T} \leftarrow T$
2.   while $\mathscr{R}(T_r) \neq \mathscr{R}(T)$
3.       do $\rho \leftarrow \infty$
4.           $t_k \leftarrow$ null
5.           for $t_i \in \hat{T}$
6.               do if $\mathscr{R}(t_i) \setminus \mathscr{R}(T_r) \neq \emptyset$
7.                   then $\rho_i \leftarrow \frac{time(\langle t_i \rangle)}{|\mathscr{R}(t_i) \setminus \mathscr{R}(T_r)|}$
8.                       if $\rho_i < \rho$
9.                           then $t_k \leftarrow t_i$
10.                               $\rho \leftarrow \rho_i$
11.                  else
12.                      $\hat{T} \leftarrow \hat{T} \setminus \langle t_i \rangle$
13.          $T_r \leftarrow T_r \uplus \langle t_k \rangle$
14.          $\hat{T} \leftarrow \hat{T} \setminus \langle t_k \rangle$
15.          $\mathscr{R}(T_r) \leftarrow \mathscr{R}(T_r) \cup \mathscr{R}(t_k)$
16.  $T \leftarrow T \setminus T_r$
17.  return $T_r$
```

Figure 14: The *GreedyReductionWithOverlap* (GRO) Algorithm.

```
**Algorithm** *GreedyPrioritizationWithOverlap*$(T, \mathscr{R}(T))$
**Input:** Test Suite $T = \langle t_1, \dots, t_n \rangle$;
       Test Coverage Set $\mathscr{R}(T)$
**Output:** Prioritized Test Suite $T_p$
1.   $T_p \leftarrow \emptyset$, $\mathscr{R}_\ell(T) \leftarrow \mathscr{R}(T)$
2.   while $T \neq \emptyset$
3.       do $T_r \leftarrow GreedyReductionWithOverlap(T, \mathscr{R}_\ell(T))$
4.           $T_p \leftarrow T_p \uplus T_r$
5.           $\mathscr{R}_\ell(T) \leftarrow \emptyset$
6.           for $t_i \in T$
7.               do $\mathscr{R}_\ell(T) \leftarrow \mathscr{R}_\ell(T) \cup \mathscr{R}(t_i)$
8.   return $T_p$
```

Figure 15: The *GreedyPrioritizationWithOverlap* (GPO) Algorithm.

```
**Algorithm** *GreedyReductionWithoutOverlap*$(T, n^*, \rho)$
**Input:** Test Suite $T = \langle t_1, \dots, t_n \rangle$;
       Test Suite Target Size $n^*$;
       Test Cost-Effectiveness Metric $\rho$
**Output:** Reduced Test Suite $T_r$
1.   $\hat{T} \leftarrow Sort(T, \rho)$
2.   $T_r \leftarrow \hat{T}[1, n^*]$
3.   return $T_r$
```

Figure 16: The *GreedyReductionWithoutOverlap* (GR) Algorithm.

Test Case	Test Cost	Test Coverage	Cost to Coverage Ratio
t_1	1	5	$1/5 = .2$
t_2	2	5	$2/5 = .4$
t_3	2	6	$2/6 = .33$

Initial Test Suite	$T = \langle t_1, t_2, t_3 \rangle$
Prioritized Test Suite	$T_p = \langle t_1, t_3, t_2 \rangle$

Figure 17: Using GP to Prioritize a Test Suite According to the Cost-Effectiveness Ratio.

of Bentley et al.'s method to sort the input test suite T and respectively create T_r and T_p [29]. The n^* term in GR's time complexity corresponds to running line 2 in Figure 16.

Using GR to perform reduction requires the selection of the target size parameter n^*. When provided with a testing time limit and the average time overhead of a test case, a tester could pick n^* so that test execution roughly fits into the time budget. In contrast to GRO and GPO, the GR technique may require the tuning of n^* in order to ensure that the modified test suite is both efficient and effective. Furthermore, GR and GP ignore the overlap in coverage and thus they may be less effective if a test suite contains tests that cover some of the same requirements. Yet, since most modern programming languages have built-in functions for efficient sorting, both GR and GP are easy to implement and they tend to be efficient for large test suites [9]. Finally, Figure 17 demonstrates how the GP algorithm would prioritize a simple test suite. This example shows that prioritization by the cost to coverage ratio creates the test suite $T_p = \langle t_1, t_3, t_2 \rangle$.

Figures 19 and 20 furnish the *ReverseReduction* (RVR) and *ReversePrioritization* (RVP) algorithms. RVR and RVP differ from GR and GP in that they use *Reverse* instead of *Sort*. Since reversal of the test tuple $T[1, n^*]$ is $O(n^*)$, we know that RVR is $O(2n^*)$ and RVP is $O(n)$. Figures 21 and 22 give the *RandomReduction* (RAR) and *RandomPrioritization* (RAP) algorithms. These algorithms are different than reduction and prioritization by reversal because they invoke *Shuffle* instead of *Reverse*. However, RAR and RAP also have respective worst-case case time complexities of $O(2n^*)$ and $O(n)$ where n stands for the number of tests. This result is due to the fact that *Reverse* and *Shuffle* are both linear time algorithms. Interestingly, recent experimental studies reveal that both the random and reverse orderings of a test suite are often more effective than the initial arrangement. Smith and Kapfhammer [8] and Do et al. [28] attribute this result to the fact that developers often add new tests after the last test case. These new tests are more likely to reveal faults than the existing tests because they frequently combine the capabilities of previous tests and/or invoke recently added features.

Since several recent experiments with regression testing methods use the Harrold, Gupta, Soffa (HGS) algorithm (e.g., [7, 22]), this chapter focuses on it as an example of a custom approach to reduction and prioritization. Since the goal of most reduction methods is to ensure that T_r covers every requirement, HGS starts to construct T_r by identifying each requirement R_j such that $|\mathscr{T}(R_j)| = 1$ [25]. After adding every test $\mathscr{T}(R_j) = \{t_i\}$ to the reduced test suite T_r, HGS considers each remaining uncovered requirement R_j when $|\mathscr{T}(R_j)| = 2$ and it uses a greedy choice metric (GCM), such as the coverage of the test, $\mathscr{R}(t_i)$ [25], or the cost-effectiveness value ρ_i from Equation (1) [8], to choose between the covering test cases. The HGS reducer continues by iteratively examining the $\mathscr{T}(R_j)$ of increasing cardinality until all of the requirements are

```
**Algorithm** *GreedyPrioritizationWithoutOverlap*$(T, \rho)$
**Input:** Test Suite $T = \langle t_1, \ldots, t_n \rangle$;
    Test Cost-Effectiveness Metric $\rho$
**Output:** Prioritized Test Suite $T_p$
1.  $T_p \leftarrow Sort(T, \rho)$
2.  **return** $T_p$
```

Figure 18: The *GreedyPrioritizationWithoutOverlap* (GP) Algorithm.

```
**Algorithm** *ReverseReduction*$(T, n^*)$
**Input:** Test Suite $T = \langle t_1, \ldots, t_n \rangle$;
    Test Suite Target Size $n^*$
**Output:** Reduced Test Suite $T_r$
1.  $\hat{T} \leftarrow T[1, n^*]$
2.  $T_r \leftarrow Reverse(\hat{T})$
3.  **return** $T_r$
```

Figure 19: The *ReverseReduction* (RVR) Algorithm.

```
**Algorithm** *ReversePrioritization*$(T)$
**Input:** Test Suite $T = \langle t_1, \ldots, t_n \rangle$
**Output:** Prioritized Test Suite $T_p$
1.  $T_p \leftarrow Reverse(T)$
2.  **return** $T_p$
```

Figure 20: The *ReversePrioritization* (RVP) Algorithm.

```
**Algorithm** *RandomReduction*$(T, n^*)$
**Input:** Test Suite $T = \langle t_1, \ldots, t_n \rangle$;
    Test Suite Target Size $n^*$
**Output:** Reduced Test Suite $T_r$
1.  $\hat{T} \leftarrow T[1, n^*]$
2.  $T_r \leftarrow Shuffle(\hat{T})$
3.  **return** $T_r$
```

Figure 21: The *RandomReduction* (RAR) Algorithm.

```
**Algorithm** *RandomPrioritization*$(T)$
**Input:** Test Suite $T = \langle t_1, \ldots, t_n \rangle$
**Output:** Prioritized Test Suite $T_p$
1.  $T_p \leftarrow Shuffle(T)$
2.  **return** $T_p$
```

Figure 22: The *RandomPrioritization* (RAP) Algorithm.

```
Algorithm SearchPrioritizeWithHillClimber($T, \mathscr{R}(T)$)
Input: Test Suite $T = \langle t_1, \ldots, t_n \rangle$;
    Test Coverage Set $\mathscr{R}(T)$
Output: Prioritized Test Suite $T_p$
1.  $\hat{T} \leftarrow Shuffle(T)$
2.  $T' \leftarrow \emptyset$
3.  while $\hat{T} \neq T'$
4.      do $T' \leftarrow \hat{T}$
5.          for $\bar{T} \in Neighborhood(\hat{T})$
6.              do if $Score(\bar{T}, \mathscr{R}(T)) > Score(\hat{T}, \mathscr{R}(T))$
7.                  then $\hat{T} \leftarrow \bar{T}$
8.  $T_p \leftarrow \hat{T}$
9.  return $T_p$
```

Figure 23: The *SearchPrioritizeWithHillClimber* (PHC) Algorithm.

```
Algorithm SwapFirstNeighborhood($T$)
Input: Test Suite $T = \langle t_1, \ldots, t_n \rangle$
Output: Test Neighborhood Set $\mathscr{N}(T)$
1.  $\mathscr{N}(T) \leftarrow \emptyset$
2.  for $t_i \in T[2, n]$
3.      do $T' \leftarrow Swap(T, t_1, t_i)$
4.          $\mathscr{N}(T) \leftarrow \mathscr{N}(T) \cup \{T'\}$
5.  return $\mathscr{N}(T)$
```

Figure 24: The *SwapFirstNeighborhood* (SRN) Algorithm.

covered. When the choice metric does not enable HGS to disambiguate between the tests in $\mathscr{T}(R_j)$ for $|\mathscr{T}(R_j)| = \mathscr{L}$, the algorithm "looks ahead" in order to determine how the tests fare in covering requirements with $\mathscr{L} + 1$ covering tests. If HGS performs the chosen maximum number of allowed look aheads without identifying the best test case, then the algorithm arbitrarily selects from those tests that remain [25]. Prior experiments reveal that while HGS is able to efficiently and effectively reduce test suites, the use of HGS in the *PrioritizationViaRepeatedReduction* algorithm may not always yield effective test orderings [8].

2.5.2 SEARCH-BASED TECHNIQUES

Search-based prioritizers use traditional heuristic search techniques (e.g., hill climbing, genetic algorithms, tabu search, and simulated annealing) to find a test ordering that maximizes an objective function denoted $Score(T, \mathscr{R}(T))$.[3] As further discussed in Section 3, this chapter focuses on *Score* functions that assign high values to test orderings that rapidly cover the test requirements. Yet, search-based prioritizers can leverage other types of *Score* functions as long as they clearly disambiguate between good and bad test orderings. For example, *Score* could focus on the fault detection effectiveness of a test suite as defined by the APFD metric that is also explained in Section 3. In light of its simplicity, ease of implementation, and efficiency [12, 27], this chapter describes a hill climbing local search algorithm that iteratively explores the neighborhood of the test ordering that currently has the best *Score*. In particular, we describe a method that uses hill climbing and one of three possible neighborhood generators as it prioritizes a test suite.

[3]This section only consider prioritization since there is a relative dearth of search-based reduction methods.

```
Algorithm SwapLastNeighborhood(T)
Input: Test Suite $T = \langle t_1, \ldots, t_n \rangle$
Output: Test Neighborhood Set $\mathscr{N}(T)$
1.  $\mathscr{N}(T) \leftarrow \emptyset$
2.  for $t_i \in T[1, n-1]$
3.      do $T' \leftarrow Swap(T, t_n, t_i)$
4.          $\mathscr{N}(T) \leftarrow \mathscr{N}(T) \cup \{T'\}$
5.  return $\mathscr{N}(T)$
```

Figure 25: The *SwapLastNeighborhood* (SLN) Algorithm.

```
Algorithm SwapFullNeighborhood(T)
Input: Test Suite $T = \langle t_1, \ldots, t_n \rangle$
Output: Test Neighborhood Set $\mathscr{N}(T)$
1.  $\mathscr{N}(T) \leftarrow \emptyset$
2.  for $t_i \in T[1, n-1]$
3.      do for $t_k \in T[i+1, n]$
4.          do $T' \leftarrow Swap(T, t_i, t_k)$
5.              $\mathscr{N}(T) \leftarrow \mathscr{N}(T) \cup \{T'\}$
6.  return $\mathscr{N}(T)$
```

Figure 26: The *SwapFullNeighborhood* (SFN) Algorithm.

Figure 23 gives the *SearchPrioritizeWithHillClimber* (PHC) algorithm that tries to use the requirement information in $\mathscr{R}(T)$ to find a prioritized test suite T_p that is better than the initial ordering in T. As shown on line 1 of Figure 23, PHC starts the hill climbing process by storing a random ordering of T in test suite $\hat{T}$. Next, PHC uses the *Neighborhood*$(\hat{T})$ function to enumerate all of the test orderings that are "near" the current test suite $\hat{T}$. As the search-based prioritizer examines each $\overline{T} \in$ *Neighborhood*$(\hat{T})$, lines 6 and 7 show that PHC performs the assignment $\hat{T} \leftarrow \overline{T}$ whenever the neighbor $\overline{T}$ earns a higher score than the current best ordering $\hat{T}$. The *SearchPrioritizeWithHillClimber* algorithm terminates when $\hat{T} = T'$ signals that the current iteration did not make progress towards the goal of finding a better ordering.

As formally described in Figures 24 through 26 and visualized in Figures 27 and 28, PHC generates a set of neighbors, denoted $\mathscr{N}(T)$, by performing a series of swaps. For instance, the *SwapFirstNeighborhood* (SRN) algorithm in Figure 24 generates a $\mathscr{N}(T)$ set with $n-1$ items by swapping the first test t_1 with each $t_i \in T[2, n]$. Figure 27(a) shows that when SRN starts with the initial test suite $\langle t_1, t_2, t_3, t_4, t_5 \rangle$ it yields an $\mathscr{N}(T)$ containing test orderings such as $\langle t_2, t_1, t_3, t_4, t_5 \rangle$. Alternatively, the *SwapLastNeighborhood* (SLN) method in Figure 25 constructs $\mathscr{N}(T)$ so that it includes the $n-1$ test suites that result from swapping t_n with each $t_i \in T[1, n-1]$. Finally, Figures 26 and 28 illustrate the *SwallFullNeighborhood* (SFN) algorithm that populates $\mathscr{N}(T)$ with the $(n \times (n-1))/2$ test orderings that result from swapping all possible pairs of test cases. Interestingly, Figures 27 and 28 demonstrate that the SRN, SLN, and SFN neighborhood generators can create an $\mathscr{N}(T)$ that contains the same test ordering (e.g., $\langle t_5, t_2, t_3, t_4, t_1 \rangle$ is a member of the neighborhood created by all three of the algorithms). Yet, it is important to observe that the SFN generator constructs neighborhoods that contain orderings, such as $\langle t_1, t_2, t_4, t_3, t_5 \rangle$ and $\langle t_1, t_3, t_2, t_4, t_5 \rangle$, that SRN and SLN do not produce.

There are many variations to the *steepest ascent* hill climber described in Figures 23 through 26. For instance, PHC could perform *first ascent* hill climbing by immediately starting to explore a new neighborhood whenever it encounters a $\overline{T}$ that is better than the current $\hat{T}$. The *random*

$$\begin{tikzcd}
 & \langle t_4,t_2,t_3,t_1,t_5\rangle & \\
\langle t_5,t_2,t_3,t_4,t_1\rangle & & \langle t_3,t_2,t_1,t_4,t_5\rangle \\
 & \langle t_1,t_2,t_3,t_4,t_5\rangle \arrow[ul] \arrow[uu] \arrow[ur] \arrow[r] & \langle t_2,t_1,t_3,t_4,t_5\rangle
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & \langle t_1,t_5,t_3,t_4,t_2\rangle & \\
\langle t_5,t_2,t_3,t_4,t_1\rangle & & \langle t_1,t_2,t_5,t_4,t_3\rangle \\
 & \langle t_1,t_2,t_3,t_4,t_5\rangle \arrow[ul] \arrow[uu] \arrow[ur] \arrow[r] & \langle t_1,t_2,t_3,t_5,t_4\rangle
\end{tikzcd}$$

(b)

Figure 27: Small Neighborhood Generation Using (a) SRN and (b) SLN.

restart hill climber repeatedly executes the PHC algorithm from multiple starting points and returns the best overall test ordering as T_p. In situations when it is difficult for the hill climber to find a highly effective T_p (i.e., there are many local maxima in the search space created by T, $\mathscr{R}(T)$, and *Score*), then search-based prioritization with simulated annealing, tabu search, or genetic algorithms may produce superior results. These alternative approaches to heuristic search employ additional methods (e.g., cooling factors, tabu lists, and crossover and mutation operators) that may increase computational cost while attempting to avoid returning a poor ordering. For more details about using genetic algorithms to prioritize test suites, please refer to [12, 27].

In comparison to the greedy methods described in Section 2.5.1, search-based techniques tend to incur higher prioritization time overheads. Yet, reordering a test suite with either a hill climber (HC) or a genetic algorithm (GA) has three potential advantages over the use of greedy techniques [27]. First, previous theoretical and empirical studies have shown that genetic algorithms are often amenable to parallelization [30, 31]. Given the rise in multi-core central processing units (CPUs) and the increasing use of graphics processing units (GPUs) for general computation [32], parallelization has the potential to effectively reduce the cost of search-based methods, thereby making their performance comparable to or better than that of the greedy techniques [15]. HC and GA methods can also be interrupted during their execution, thus enabling the identification of the test ordering that is currently the best and the use of a "human in the loop" prioritization model where an intelligent human effectively guides the search algorithm [33].

A third advantage of the search-based methods concerns the degree to which they can construct diverse test orderings that achieve equivalent coverage effectiveness scores [34]. If the test coverage report and the execution time of the tests does not change, then multiple prioritizations of a given test suite produced by a greedy algorithm will always be identical. In contrast, a hill climber or a genetic algorithm is likely to yield different test orderings, assuming the use of a different initial test suite order. It is more desirable to use different orderings of tests to cover the

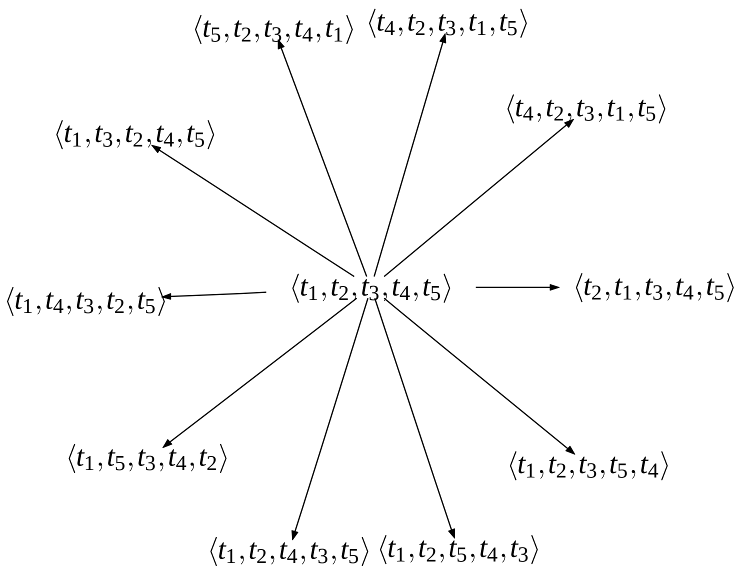

$\langle t_5, t_2, t_3, t_4, t_1 \rangle$
$\langle t_4, t_2, t_3, t_1, t_5 \rangle$
$\langle t_1, t_3, t_2, t_4, t_5 \rangle$
$\langle t_4, t_2, t_3, t_1, t_5 \rangle$
$\langle t_1, t_4, t_3, t_2, t_5 \rangle$
$\langle t_1, t_2, t_3, t_4, t_5 \rangle$
$\langle t_2, t_1, t_3, t_4, t_5 \rangle$
$\langle t_1, t_5, t_3, t_4, t_2 \rangle$
$\langle t_1, t_2, t_3, t_5, t_4 \rangle$
$\langle t_1, t_2, t_4, t_3, t_5 \rangle$
$\langle t_1, t_2, t_5, t_4, t_3 \rangle$

Figure 28: Large Neighborhood Generation Using SFN.

same requirements than it is to repeatedly use an unchanged test ordering. This activity ensures that latent properties of the tests that are not reflected in the requirements will be brought to bear on the application, possibly increasing the test suite's capability to find faults not connected to the chosen adequacy criterion [34]. Search-based prioritizers are ideally suited for this task because they can produce different test orderings that have similar effectiveness scores.

Much like greedy methods, search-based prioritizers often exhibit many interesting trade-offs in efficiency and effectiveness. For example, the SRN and SLN neighborhood generators lead to lower prioritization time overheads since they create a smaller $\mathscr{N}(T)$ than SFN. Yet, it is possible that the costly SFN may yield a large neighborhood containing test ordering(s) that are more effective than the test suites that are part of the neighborhoods generated by SRN and SLN. Similarly, at the cost of an increase in execution time, prioritization with random restart hill climbing may also lead to better orderings than those that are produced by a traditional hill climber. Using the terminology established in Section 2.1, the computationally expensive search-based prioritizers (e.g., random restart hill climbing with SFN) are best suited for general regression testing environments where testers run the same test ordering over many modifications to the program under test. Alternatively, efficient search-based prioritizers, such as hill climbing with SRN or SLN, can better support the version specific approach to regression testing.

2.6 PERFORMING TEST SUITE SELECTION

As the program under test changes, test suite selection methods aim to reduce the cost of regression testing by only re-running those test cases that have the potential to reveal defects. A selection method is called *safe* if it can always construct test suites that are capable of detecting the same faults as the original tuple of tests [3, 35, 36]. If developers adhere to the *controlled regression testing assumption* and only make modifications to the source code of the program, then a test case selector can guarantee defect isolation by identifying and running those tests that exercise the changed modules in the program [35]. Selection methods often use coverage moni-

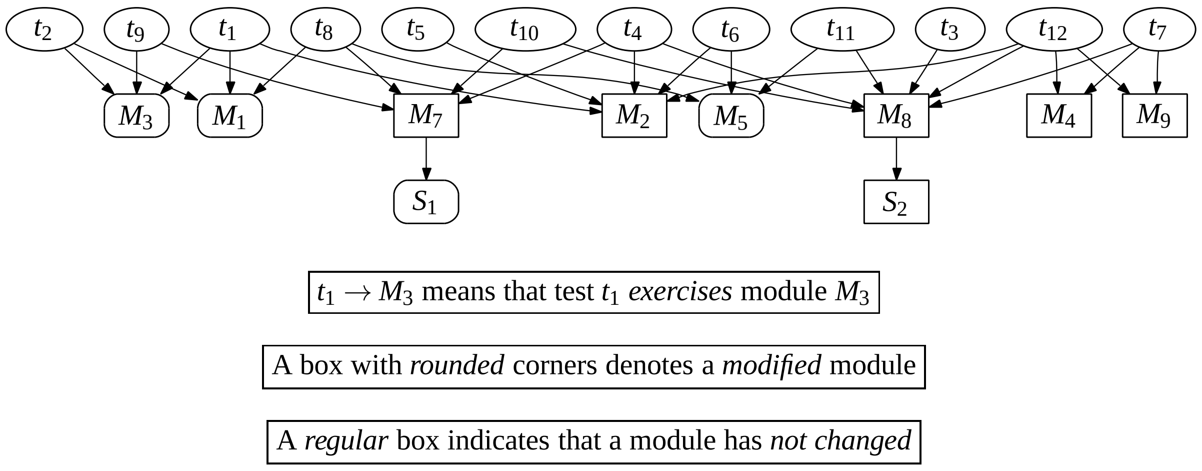

t_2
t_9
t_1
t_8
t_5
t_{10}
t_4
t_6
t_{11}
t_3
t_{12}
t_7
M_3
M_1
M_7
M_2
M_5
M_8
M_4
M_9
S_1
S_2
$t_1 \rightarrow M_3$ means that test t_1 *exercises* module M_3
A box with *rounded* corners denotes a *modified* module
A *regular* box indicates that a module has *not changed*

Figure 29: Test Suite Selection in the Presence of Changed Program Modules.

toring information to determine how the modules of the program are exercised by the tests. After recording a coverage report and determining which modules were recently changed, the selector executes a potentially smaller test suite that only focuses on these updated modules. Selection methods normally concentrate on program modifications involving the change, deletion, or addition of a source code location. In practice, test suite selection techniques also need to track the changes that the developers make to external resources (e.g., configuration files and databases).

Figure 29 depicts a test suite for a modified program containing a total of nine unique modules that may be Java methods or classes. As an example, this diagram uses the notation $t_1 \rightarrow M_3$ to indicate that the test case t_1 *exercises* module M_3. For each of the modules in Figure 29, a box with rounded corners highlights a module that recently underwent modification (e.g., M_3) while a standard box means that developers did not change the module (e.g., M_9). Furthermore, this example uses S_1 and S_2 to respectively denote external resources that have and have not been changed by developers. A regression test selection mechanism analyzes coverage and change reports like the one in Figure 29 in order to determine which tests do not need to run because they did not exercise any modified modules (i.e., $t_3, t_4, t_5, t_7,$ and t_{12}). After finding the tests that (i) interact with methods using modified resources (e.g., $t_4, t_8, t_9,$ and t_{10}) and (ii) directly exercise the changed modules (i.e., $t_1, t_2, t_6, t_8, t_9,$ and t_{11}), a selection method can create and run a smaller test suite such as $\langle t_1, t_6, t_8 \rangle$. In this instance, the selection mechanism must choose a test like t_8 in order to ensure the testing of module M_7's interaction with the modified resource S_1. Furthermore, the technique picks tests such as t_1 and t_6 in order to ensure the isolation of defects that may arise from changes in modules $M_1, M_3,$ and M_5.

Results from analytical studies, empirical evaluations, and practical experience suggest that there are interesting trade-offs in the efficiency and effectiveness of test suite selection techniques. For instance, experiments conducted by Rothermel and Harrold indicate that test suite design can have a substantial impact of the effectiveness of selection methods [36]. That is, selection may not be cost-effective when the tests execute rapidly, the test suite is small, or there are certain modules that are exercised by many test cases [36]. However, for situations like the one depicted in Figure 29, selection can often reduce testing time because each test focuses on a small number

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
t_1	✓	✓		✓	✓	✓	✓	✓
t_2	✓							
t_3	✓				✓			
t_4		✓	✓				✓	
t_5				✓		✓		✓
t_6		✓		✓		✓		

	Faults	Cost (Mins)	Avg. Faults/Min
t_1	7	9	0.778
t_2	1	1	1.0
t_3	2	3	0.667
t_4	3	4	0.75
t_5	3	4	0.75
t_6	3	4	0.75

(a)

	Time Limit: 12 minutes			
	Fault (T_p)	Time ($\hat{T_p}$)	Avg. Faults/Min. ($\bar{T_p}$)	Intelligent ($\check{T_p}$)
	t_1	t_2	t_2	t_5
		t_3	t_1	t_4
		t_4		t_3
		t_5		
Total Faults	7	8	7	8
Total Time	9	12	10	11

(b)

Figure 30: An Example of Time-Aware Test Suite Prioritization.

of modules (e.g., t_{12} only interacts with four modules while many tests, such as t_5 and t_6, use just one or two). Furthermore, the use of the Testar test selection tool at Google reveals that "the smaller your changes are (or the more frequently you run Testar), and the more tests you have, the bigger are the relative savings" [37]. While conceding that test suite selection may not be capable of decreasing testing time for some applications, Graves et al. observe that a safe selection method found all of the faults for which fault-identifying tests existed and discarded 60% of the tests on the median [3]. Finally, selection methods may still identify small and useful test suites in circumstances when the controlled regression testing assumption does not hold.

2.7 RESOURCE-AWARE REGRESSION TESTING

Several new regression testing methods aim to handle the challenges associated with running tests in constrained environments where computational resources such as time, memory, or power are limited [15]. This chapter considers the concrete example of time-aware test suite prioritization since time is a concern for organizations that rely upon nightly builds or perform regression testing each time source code changes are committed to a version control repository. As an example of time constrained testing, suppose that a tester wants to reorder $T = \langle t_1, t_2, t_3, t_4, t_5, t_6 \rangle$, as shown in Figure 30. For the purposes of illustration, this example assumes a priori knowledge of the faults detected by T in the program P. As given in Figure 30(a), test case t_1 can find seven faults, $\{f_1, f_2, f_4, f_5, f_6, f_7, f_8\}$, in nine minutes, t_2 finds one fault, $\{f_1\}$, in one minute, and t_3 isolates two faults, $\{f_1, f_5\}$, in three minutes. Test cases t_4, t_5, and t_6 each find three faults in four minutes: $\{f_2, f_3, f_7\}$, $\{f_4, f_6, f_8\}$, and $\{f_2, f_4, f_6\}$, respectively.

Suppose that the time budget for regression testing is twelve minutes. Because we want to find as many faults as possible early on, we order the test cases by only considering the number of faults that they can detect. Without a time budget, the test suite $T = \langle t_1, t_4, t_5, t_6, t_3, t_2 \rangle$ would execute. Out of this, only the test suite $T_p = \langle t_1 \rangle$ can run under a twelve minute time constraint, and it would find a total of seven faults, as noted in Figure 30(b). Since time is a principal concern, it may also seem logical to order the test cases with regard to their execution time. In the time constrained environment, a time-based prioritization $\hat{T_p} = \langle t_2, t_3, t_4, t_5 \rangle$ could be executed and find eight defects, as shown in Figure 30(b). Another option would be to consider the time budget and fault information together. To do this, we could order the test cases according to the average percent of faults that they can detect per minute. Under the time constraint, the execution of the ordering $\overline{T_p} = \langle t_2, t_1 \rangle$ finds a total of seven faults.

If the time budget and the fault information are both considered intelligently, that is, in a way that accounts for overlapping fault detection, the test cases could be better prioritized and thus increase the overall number of faults found in the desired time period. In this example, the test cases are intelligently reordered so that the suite $\check{T_p} = \langle t_5, t_4, t_3 \rangle$ is executed, revealing eight errors in less time than $\hat{T_p}$. It is also clear that $\check{T_p}$ can reveal more defects than T_p and $\overline{T_p}$ in the specified testing time. Finally, it is important to note that the first two test cases of $\hat{T_p}$, t_2 and t_3, find a total of two faults in four minutes whereas the first test case in $\check{T_p}$, t_5, detects three defects in the same time period. The time-aware prioritization, $\check{T_p}$, is favored over $\hat{T_p}$ because it is able to detect more faults earlier in test execution.

There are several different approaches to implementing a time-aware test prioritizer [15, 38, 39]. For example, Walcott et al. present a genetic algorithm based method that reorganizes test suites so that the new order will (i) always run within a time limit and (ii) have the highest possible potential for defect detection based upon the information in the coverage report [15]. Alternatively, Alspaugh et al. describe an approach to efficient time-aware prioritization that uses uses solvers for the 0/1 knapsack problem to reorder the test suite [38]. In comparison to the genetic algorithm, the knapsack solvers do not consider the overlap in test coverage, thus quickly producing a test suite that is often less effective than the one constructed by the GA. Zhang et al. introduce a time-aware prioritizer that uses an integer linear programming (ILP) method to solve the time and coverage constraints introducing by a restrictive testing time budget [39].

Recent experimental results indicate that higher levels of coverage and fault detection are obtained when time-aware prioritizers explicitly consider time constraints [15, 39]. Even when a severe time restriction forces testers to reduce the time allotted to testing by 75%, Walcott et al. report that their search-based technique preserves on average 94% of the original test suite's code coverage [15]. Zhang et al. also find that certain traditional regression testing methods, such as those described in Section 2.5, may create reasonably effective test orderings when the testing time budget is not too constrained. In these situations, it may make sense to use the traditional greedy prioritizers since the non-time-aware techniques are normally cheaper than those that explicitly consider the testing time constraints. Finally, the experiments of Zhang et al. reveal that the ILP-based solvers are often the most efficient and effective approach to time-aware prioritization, suggesting that this method may also be useful in quickly handling other resource constraints such as those related to memory and battery consumption [39].

3 EVALUATION OF REGRESSION TESTING TECHNIQUES

During the use of regression testing in either an industrial environment or an experimental study, it is important to gauge the efficiency and effectiveness of the techniques that are described in Section 2. Since it is always desirable for a testing technique to run with low time and space overheads, this section focuses on methods for measuring the effectiveness of approaches to selection, reduction, and prioritization. Equation (2) defines $\text{RFFS}(T,T_r) \in [0,1)$, the *reduction factor for size* given a test suite T and it's reduced form T_r [7]. Since the RFFS reflects the percent of original tests that remain after selection or reduction, an RFFS of 0 means that the algorithm removed none of the tests while an RFFS near 1 means that the reducer discarded many tests (an RFFS of 1 is not possible because testers often mandate that T_r must contain at least one test to cover at least some of the requirements). As stated by Equations (3) and (4), $\text{RFFT}(T,T_r) \in [0,1)$ is the *reduction factor for time* for test suites T and T_r [9]. 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., $time(T) - time(T_r) = time(T)$).

$$\text{RFFS}(T,T_r) = \frac{|T| - |T_r|}{|T|} \quad (2) \qquad \text{RFFT}(T,T_r) = \frac{time(T) - time(T_r)}{time(T)} \quad (3)$$

$$time(T) = \sum_{t_i \in T} time(t_i) \quad (4)$$

The majority of prior empirical research calculates the decrease in fault detection effectiveness for a reduced test suite after seeding faults into the program under test (e.g., [7]). Yet, it is also important to use effectiveness metrics that do not require fault information since fault seeding may be time consuming and error-prone. To this end, Equation (5) defines the *reduction factor for test requirements* as $\text{RFFR}(T,T_r) \in [0,1]$. Unlike the RFFS and RFFT metrics, we prefer low values for $\text{RFFR}(T,T_r)$ because this indicates that a reduced test suite T_r covers the majority of the requirements that the initial tests cover. To avoid confusion during the comparison of different reduction techniques, Equation (6) defines the *preservation factor for test requirements*. If a reduced test suite has a high value for $\text{PFFR}(T,T_r) \in [0,1]$, then we also know that it covers most of the requirements that the original tests cover. Since the overlap-aware and custom greedy reduction algorithms defined in Section 2, such as GRO, HGS, and DGR, always create a T_r that covers all of the test requirements we know that $\text{PFFR}(T,T_r) = 1$ for these methods. The other reduction techniques (e.g., GR, RVR, and RAR) may not construct a test suite that covers all $R_j \in \mathscr{R}(T)$ and thus these methods may yield test suites with PFFR values that are less than one.

$$\text{RFFR}(T,T_r) = \frac{|\mathscr{R}(T)| - |\mathscr{R}(T_r)|}{|\mathscr{R}(T)|} \quad (5)$$

$$\text{PFFR}(T,T_r) = 1 - \text{RFFR}(T,T_r) \quad (6)$$

In order to facilitate the comparison between different approaches to reduction, we use the tuple $E_r = \langle \text{RFFS}, \text{RFFT}, \text{PFFR} \rangle$ to organize the evaluation metrics for test suite T_r.[4] Figure 31

[4]Without loss of generality, this chapter concentrates on using RFFS, RFFT, and PFFR during the comparison of reduction techniques. However, it is possible to apply the same approach if E contains scores from different metrics.

E_r	$\hat{E_r}$	Comparison
$\langle .4, .5, .8 \rangle$	$\langle .4, .5, .9 \rangle$	$\hat{E_r} \gg E_r$
$\langle .4, .5, 1 \rangle$	$\langle .4, .55, 1 \rangle$	$\hat{E_r} \gg E_r$
$\langle .6, .5, 1 \rangle$	$\langle .4, .5, 1 \rangle$	$E_r \gg \hat{E_r}$
$\langle .4, .5, 1 \rangle$	$\langle .55, .75, .9 \rangle$	$E_r \sim \hat{E_r}$

Figure 31: Evaluation Tuples Used to Compare Reduced Test Suites.

summarizes the four different examples of evaluation tuples that this chapter uses to explain the process of comparing different reduction algorithms. Suppose that two reduction techniques create T_r and $\hat{T_r}$ that are respectively characterized by the evaluation tuples $E_r = \langle .4, .5, .8 \rangle$ and $\hat{E_r} = \langle .4, .5, .9 \rangle$, found in the first row of the table in Figure 31. As a further aid in comparing reduction methods, we use the notation $e \in E_r$ and $\hat{e} \in \hat{E_r}$ to clarify the tuple membership of scores e and $\hat{e}$ (i.e., $\widehat{\text{RFFS}}$ refers to the reduction factor for size score in $\hat{E_r}$). In this example, a tester would prefer $\hat{E_r}$ because it (i) has the same values for RFFS and RFFT (i.e., the reduction factors for the number of tests and the overall testing time) and (ii) preserves the coverage of more test requirements since $\widehat{\text{PFFR}} > \text{PFFR}$. If $E_r = \langle .4, .5, 1 \rangle$ and $\hat{E_r} = \langle .4, .55, 1 \rangle$, then we would favor the reduced suite with $\hat{E_r}$ because it fully preserves requirement coverage while yielding a larger value for RFFS (i.e., $.55 > .5$). Next, suppose that $E_r = \langle .6, .5, 1 \rangle$ and $\hat{E_r} = \langle .4, .5, 1 \rangle$. In this situation, we would favor E_r's reduction algorithm since it yields the smallest test suite (i.e., $\text{RFFS} > \widehat{\text{RFFS}}$). This choice is sensible because it will control testing time if there is an increase in the costs of starting up and shutting down an individual test case.

During the evaluation of reduction algorithms, it may not always be clear which technique is the most appropriate for a given program and its test suite. For example, assume that $E_r = \langle .4, .5, 1 \rangle$ and $\hat{E_r} = \langle .55, .75, .9 \rangle$, as provided by Figure 31. In this case, $\hat{E_r}$ shows that $\hat{T_r}$ is (i) better at reducing testing time and (ii) worse at preserving requirement coverage when we compare it to T_r. In this circumstance, a tester must choose the reduction technique that best fits the current regression testing process. For instance, it may be prudent to select $\hat{E_r}$ when the test suite is executed in a time and/or memory constrained environment (e.g., [15, 40]) or the tests are repeatedly run during continuous testing (e.g., [41]). If the correctness of the application is the highest priority, then it is advisable to use the reduction technique that leads to T_r and E_r.

For the reduced test suites T_r and $\hat{T_r}$ and their respective evaluation tuples E_r and $\hat{E_r}$, we write $E_r \gg \hat{E_r}$ when the logical predicate in Equation (7) holds (i.e., we *prefer* T_r to $\hat{T_r}$). If $E_r \gg \hat{E_r}$, then we know that T_r is as good as $\hat{T_r}$ for all three evaluation metrics and better than $\hat{T_r}$ for at least one metric.[5] For instance, the first row of Figure 31 shows that $\hat{E_r} \gg E_r$ because $\hat{T_r}$ yields (i) RFFS and RFFT values that are equal to the scores for T_r and (ii) a PFFR value that is greater than that of T_r. If Equation (7) does not hold for test suites T_r and $\hat{T_r}$ that were produced by two different reduction techniques (i.e., $E_r \not\gg \hat{E_r}$ and $\hat{E_r} \not\gg E_r$), then we write $E_r \sim \hat{E_r}$ (i.e., T_r and $\hat{T_r}$ are *similar*). Since Equation (7) does not dictate a preference between T_r and $\hat{T_r}$ when $E_r \sim \hat{E_r}$, as shown in the final row of Figure 31, a tester must use the constraints inherent in the testing

[5]Equation (7)'s definition of the $\gg$ operator is based on the concept of *pareto efficiency* that is employed in the fields of economics and multi-objective optimization (please refer to [42] for more details about these areas).

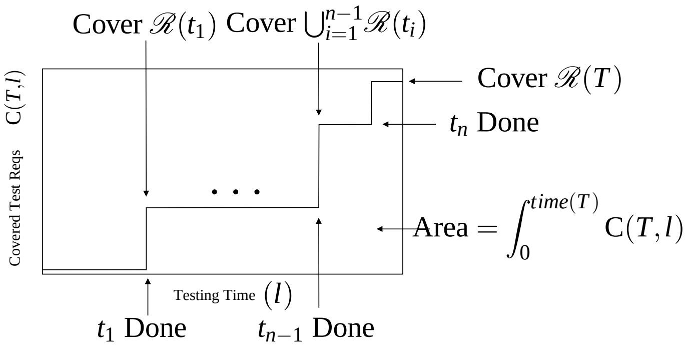


Figure 32: The Coverage Effectiveness of a Test Suite.

process to inform the choice of the best reduction technique. For instance, a tester may pick the fastest reducer when $E_r \sim \hat{E}_r$ and the developers wants to perform version specific testing.

$$\forall e \in E_r, \hat{e} \in \hat{E}_r : (e \geq \hat{e}) \;\wedge\; \exists\, e \in E_r, \hat{e} \in \hat{E}_r : (e > \hat{e}) \tag{7}$$

Any metric for evaluating test suite prioritizers must be able to assess how well a test ordering uses the full time allotted to testing. For instance, testers often prefer an ordering that rapidly covers the test requirements since this could reduce the time required to find the first fault in the program. When provided with cost and coverage information, as given in Figure 33, it is possible to calculate the *coverage effectiveness* of test suite T, denoted $\mathrm{CE}(T)$. If test cost information is not available, then testers can assume that each test consumes a single unit of time and subsequently compute $\mathrm{CE_u}(T)$, the *unit coverage effectiveness* of T. Suppose that a regression testing tool creates test suites T_p and $\hat{T}_p$ after applying two different prioritization techniques to the original test suite T. If $\mathrm{CE}(T_p) > \mathrm{CE}(\hat{T}_p)$, then we know that T_p is more coverage effective than $\hat{T}_p$ and thus a tester would prefer the first approach to prioritization instead of the second. In situations where $\mathrm{CE}(T_p) = \mathrm{CE}(\hat{T}_p)$, the tester may pick the most efficient test suite prioritizer.

The coverage effectiveness metric evaluates a prioritized test suite by determining the cumulative coverage of the tests over time [8]. As defined in Equation (8) and depicted in Figure 32, 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. Following the definition of $time(T)$ given in Equation (4), the formulation of $C(T,l)$ uses the *time* function to compute the execution time of the tests in T (e.g., $time(\langle t_1 \rangle)$ returns the running time of the first test and $time(\langle t_1,\ldots,t_{n-1}\rangle)$ determines the time required to run T's first $n-1$ tests). We define $C(T,l)$ as an $(n+1)$-part piecewise function when $T = \langle t_1,\ldots,t_n \rangle$. Equation (8) reveals that $C(T,l) = 0$ until the completion of test case t_1 (i.e., $l < time(\langle t_1 \rangle)$). In the time period after the execution of t_1 and during the running of t_2 (i.e., $l \in [time(\langle t_1 \rangle), time(\langle t_1, t_2 \rangle))$), the value of C shows that T has covered a total of $|\mathscr{R}(t_1)|$ requirements. The function C maintains the maximum height of $|\mathscr{R}(T)|$ for all time points $l \geq time(T)$, as graphically depicted in Figure 32.

$$C(T,l) = \begin{cases} 0 & l < time(\langle t_1 \rangle) \\ |\mathscr{R}(t_1)| & l \in [time(\langle t_1 \rangle), time(\langle t_1, t_2 \rangle)) \\ \vdots & \vdots \\ |\bigcup_{i=1}^{n-1} \mathscr{R}(t_i)| & l \in [time(\langle t_1,\ldots,t_{n-1} \rangle), time(T)) \\ |\mathscr{R}(T)| & l \geq time(T) \end{cases} \tag{8}$$

To formulate $\mathrm{CE}(T) \in (0,1)$, the integral of $C(T,l)$ is divided by the integral of the ideal cumulative coverage function $\bar{C}(T,l)$ that Equation (9) defines to immediately cover all of the requirements. Equation (10) shows that CE considers test requirement coverage throughout the execution time of T by taking the integrals within the closed interval from 0 to $time(T)$. Since any prioritization of a test suite should always cover the same requirements as the original ordering (i.e., $\mathscr{R}(T) = \mathscr{R}(T_p)$), our statement of coverage effectiveness forbids the case of $\mathscr{R}(T_p) = \emptyset$ that would lead to $\mathrm{CE}(T) = 0$. Since it is impossible for T_p to instantaneously cover all of the test requirements, Equation (10)'s expression of coverage effectiveness also precludes $\mathrm{CE}(T) = 1$. Finally, $\mathrm{CE_u}(T)$ is defined in a similar manner to Equations (8) through (10), except for the fact that we assume all tests have unit cost and thus $time(t_i) = 1$ for all $t_i \in T$.

$$\bar{C}(T,l) = |\mathscr{R}(T)| \qquad (9)$$

$$\mathrm{CE}(T) = \frac{\int_0^{time(T)} \mathrm{C}(T,l)}{\int_0^{time(T)} \bar{\mathrm{C}}(T,l)} \qquad (10)$$

Since many test coverage monitoring tools do not record the point in time when a test case covers a requirement [20, 21], CE conservatively credits a test with the coverage of its requirements when it finishes execution. While CE does furnish a time sensitive measurement of effectiveness, it may be unfair to high coverage tests with extended running times. One approach to handling this issue is to *linearly interpolate* between the points in the piece-wise coverage function. For instance, the linear interpolant between the time when testing begins and the first test case finishes execution is the straight line between the points $(0,0)$ and $(time(\langle t_1 \rangle), |\mathscr{R}(t_1)|)$. Between the time when an arbitrary t_j ends and the following test t_{j+1} completes, Equation (11) defines $C_\delta(T,l)$, the coverage function that performs linear interpolation between all of the points $(time(\langle t_1, \ldots, t_j \rangle), |\cup_{i=1}^{j} \mathscr{R}(t_i)|)$ and $(time(\langle t_1, \ldots, t_{j+1} \rangle), |\cup_{i=1}^{j+1} \mathscr{R}(t_i)|)$. The $C(T,l)$ function is similar to $C_\delta(T,l)$, except for the fact that C exhibits abrupt "jumps" in height at the completion of each test case whereas C_δ uses a straight line to approximate the increase in coverage as a test runs. Finally, Equation (12) calculates $\mathrm{CE}_\delta(T)$ in an analogous fashion to the one used for $\mathrm{CE}(T)$, with the exception of the fact that the numerator uses C_δ instead of C.

$$C_\delta(T,l) = \begin{cases} l \times \left(\frac{|\mathscr{R}(t_1)|}{time(\langle t_i \rangle)} \right) & l < time(\langle t_1 \rangle) \\ |\mathscr{R}(t_1)| + & \\ (l - time(t_1)) + \left(\frac{|\mathscr{R}(t_1) \cup \mathscr{R}(t_2)| - |\mathscr{R}(t_1)|}{time(\langle t_1, t_2 \rangle) - time(\langle t_1 \rangle)} \right) & l \in [time(\langle t_1 \rangle), time(\langle t_1, t_2 \rangle)) \\ \vdots & \vdots \\ |\bigcup_{i=1}^{n-1} \mathscr{R}(t_i)| + & \\ (l - time(\langle t_1, \ldots, t_{n-1} \rangle)) \times \left(\frac{|\mathscr{R}(T)| - |\bigcup_{i=1}^{n-1} \mathscr{R}(t_i)|}{time(T) - time(\langle t_1, \ldots, t_{n-1} \rangle)} \right) & l \in [time(\langle t_1, \ldots, t_{n-1} \rangle), time(T)) \\ |\mathscr{R}(T)| & l \geq time(T) \end{cases} \qquad (11)$$

$$\mathrm{CE}_\delta(T) = \frac{\int_0^{time(T)} \mathrm{C}_\delta(T,l)}{\int_0^{time(T)} \bar{\mathrm{C}}(T,l)} \qquad (12)$$

Test Case	Cost (seconds)	Requirements				
		R_1	R_2	R_3	R_4	R_5
t_1	5	✓	✓			
t_2	10	✓	✓	✓		✓
t_3	4	✓			✓	✓

$\boxed{\text{Total Testing Time = 19 seconds}}$

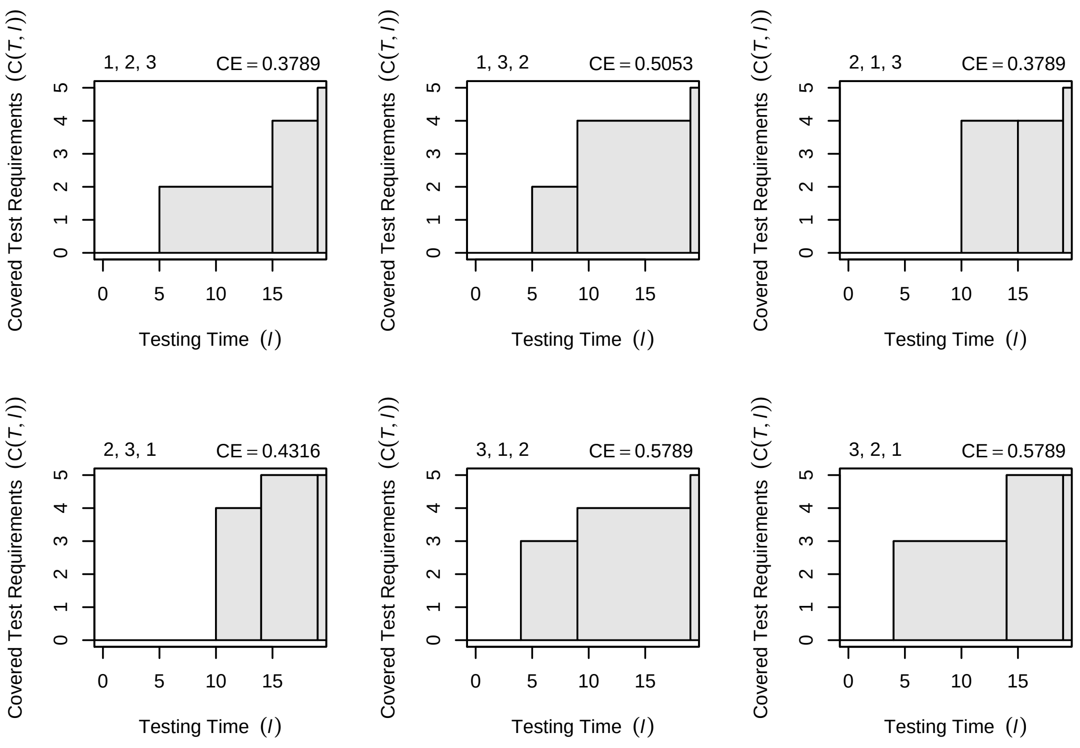
Figure 33: The Cost and Coverage Characteristics of a Test Suite.

As an example of calculating coverage effectiveness, suppose that test suite $T = \langle t_1, t_2, t_3 \rangle$ covers a total of five requirements while testing program P. Figure 33 characterizes test suite T according to execution time and requirement coverage (e.g., test t_2 takes ten seconds to execute while t_1 and t_3 respectively consume five and four seconds). Figure 34 visualizes the coverage curves and gives the CE and CE_u values for the $3! = 3 \times 2 \times 1 = 6$ different orderings of test suite T. These graphs demonstrate that the inclusion of test case execution costs does impact the measurement of effectiveness. For instance, CE_u equivalently ranks the orderings $T_p = \langle t_1, t_2, t_3 \rangle$ and $\hat{T}_p = \langle t_1, t_3, t_2 \rangle$ while CE classifies the latter as more effective. This result is due to the fact that $\hat{T}_p$'s first two tests cover four requirements in nine seconds while the corresponding tests in T_p take fifteen seconds to cover four requirements.

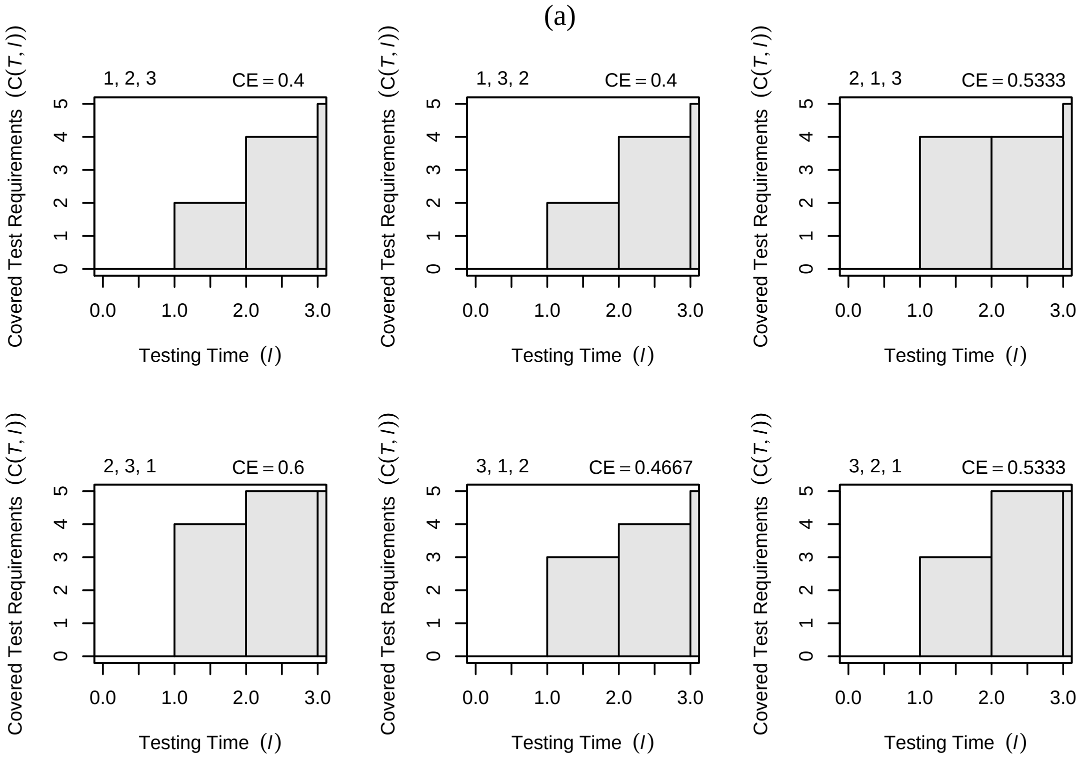
For a given test order, CE may be higher than CE_u or vice-versa. For instance, $\langle t_2, t_1, t_3 \rangle$ results in a low value for CE and a high CE_u score because CE incorporates the substantial cost of running t_2 and CE_u assumes that t_2's running time is equivalent to the other tests. Figure 35 furnishes the graphs for the linearly interpolating coverage function C_δ and the corresponding CE_δ values. A comparison of Figure 34(a) and Figure 35 reveals that CE_δ's use of interpolation uniformly increases the coverage effectiveness score. Finally, these eighteen graphs illustrate that different test orderings are more or less effective at using the amount of time devoted to testing.

In contrast to CE, other evaluation metrics such as the *average percentage of faults detected* (APFD) [5] do not factor time into the evaluation of a test suite prioritizer. Unlike existing evaluation metrics that incorporate time (e.g., APFD_c [43]), CE obviates the need to use fault information when calculating effectiveness. However, if per-test fault information, like that in Figure 36, is available from testing records or seeded faults, then testers may pursue the calculation of APFD. For a set of known faults F, Equation (13) defines the $\text{APFD}(T,F) \in [\frac{1}{2\times|T|}, 1 - \frac{1}{2\times|T|}]$ when $\mathit{reveal}(f,T)$ denotes the position within T of the first test that reveals fault $f \in F$ [44]. Using this formulation of APFD, the minimum value of $\frac{1}{2\times|T|}$ corresponds to the circumstance in which the last test case in the ordering is the first to expose all of the faults. Moreover, the maximum value of $1 - \frac{1}{2\times|T|}$ is evident when the first test can find all of the faults. If we use the test orderings $T_p = \langle t_1, t_2, t_3, t_4, t_5 \rangle$ and $\hat{T}_p = \langle t_3, t_4, t_1, t_2, t_5 \rangle$ and the fault information in Figure 36 as an example, then we have $\text{APFD}(T_p, F) = 1 - .4 + .1 = .7$ and $\text{APFD}(\hat{T}_p, F) = 1 - .2 + .1 = .82$. This result suggests that, according to the APFD metric, $\hat{T}_p$ is a better prioritization than T_p.

$$\text{APFD}(T,F) = 1 - \frac{\sum_{f \in F} \mathit{reveal}(f,T)}{|T| \times |F|} + \frac{1}{2 \times |T|} \tag{13}$$



(a)



(b)

Figure 34: The (a) CE(T) and (b) CE$_u(T)$ Scores for $T = \langle t_1, t_2, t_3 \rangle$.

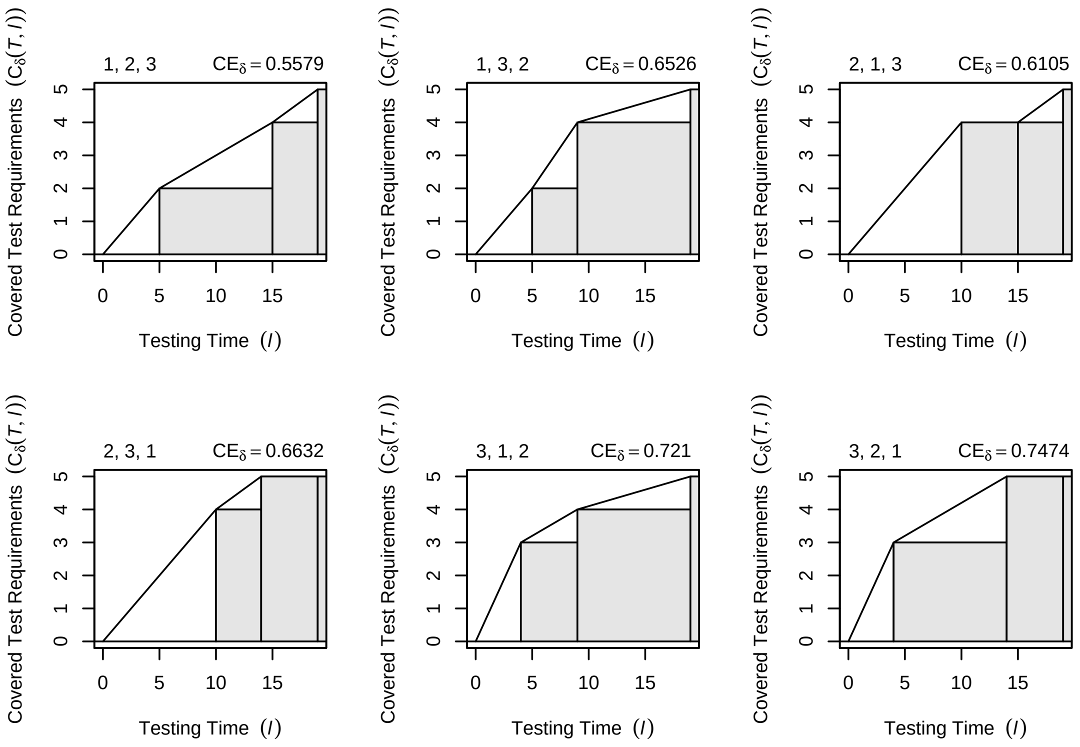


Figure 35: The $\mathrm{CE}_{\delta}(T)$ Scores for $T = \langle t_1, t_2, t_3 \rangle$.

Several recent empirical studies have shown that certain types of test suites are relatively easy to prioritize [8, 9, 11]. If a regression test suite T contains many tests that cover a substantial number of the requirements, then a random reordering of T may often have a CE value that is greater than the CE score for the initial ordering [8, 9]. Similarly, a random prioritization may improve the APFD of a test suite containing many tests that detect a large number of faults [11]. Equation (14) defines $\mathrm{D}(T, \mathscr{R}) \in (0, 1]$, the *coverage density* of test suite T for requirement set $\mathscr{R}$. A low value for $\mathrm{D}(T, \mathscr{R})$ suggests that many test cases cover a small number of requirements, while $\mathrm{D}(T, \mathscr{R}) = 1$ indicates that each test case covers all of the requirements. If we assume that each of the requirements in $\mathscr{R}(T)$ must be covered by one of the tests within T, then every test suite will have a density value greater than zero. A random prioritizer may be sufficient for test suites that have a high value for D, whereas a low D value indicates the need for the greedy and search-based approaches. For example, the test suite in Figure 10 yields the relatively low value of $\mathrm{D}(T, \mathscr{R}) = .2619$. Finally, Figure 37 furnishes an empirical cumulative distribution function (ECDF) that visualizes the range of $|\mathscr{R}(t_i)|$ values for the test suite in Figure 10. An ECDF gives the cumulative percentage of the data set whose values fall below a specific value [45]. As a confirmation of the test suite's low density value, this ECDF reveals that 80% of the test cases in Figure 10 have an $\mathscr{R}(t_i)$ set containing two or fewer requirements.

$$\mathrm{D}(T, \mathscr{R}) = \frac{\sum_{t_i \in T} |\mathscr{R}(t_i)|}{|T| \times |F|} \tag{14}$$

Test Case	Faults				
	f_1	f_2	f_3	f_4	f_5
t_1			✓	✓	
t_2	✓	✓			
t_3	✓	✓	✓		
t_4			✓	✓	✓
t_5		✓	✓		

Figure 36: The Faults Detected by a Test Suite.

4 CONCLUSIONS AND FUTURE WORK

This chapter examines an important software maintenance activity known as regression testing. Methods for regression testing focus on running a test suite whenever the addition of defect fixes or new functionality causes the program under test to change. Even though the use of regression testing techniques often leads to software applications with high observed quality [2], the repeated execution of test cases can be so costly that it accounts for half the cost of maintaining a software system [3]. This chapter focuses on reduction methods that decrease the cost of testing by discarding those tests that redundantly cover the test requirements. Additionally, the chapter describes approaches to test suite prioritization that reorder a test suite in an attempt to improve the rate at which tests achieve certain testing goals such as requirement coverage. Finally, we report on test selection techniques that reduce the cost of testing by only running those tests that exercise the recently modified modules of the program. These methods for enhancing regression test suites have different trade-offs in efficiency and effectiveness, thus making them more or less useful in the general and version specific models of regression testing.

After furnishing a detailed description of greedy and search-based techniques for prioritization and reduction, this chapter highlights safe test selection methods. We also note that the reduction factors for size and time and the preservation factor for test requirements are three metrics that facilitate the comparison between different approaches to reduction. As long as a method incurs minimal time and space overheads while preserving requirement coverage, a tester would normally pick a test reducer that substantially decreases the size and running time of a test suite. In the context of test suite prioritization, evaluation metrics such as coverage effectiveness and the average percentage of faults detected must determine how well a regression test suite uses the given time to cover requirements or detect faults.

In light of the wide range of advances in this field, technology transfer represents an important step for further work in regression testing. To date, many important methods for running and analyzing test suites have not transitioned into practice due to the lack of properly documented and supported tools. The existence of freely available and open source testing tools that integrate with existing frameworks such as JUnit and CppUnit will be beneficial to both testing researchers and practitioners. However, as noted by Frederick P. Brooks Jr., the greatest hope that the software engineering community has for solving the crisis of low quality software is exceptional engineers and testers [46]. As such, the proper education and training of testing students and professionals is an important area for further investigation. In summary, future progress in regression testing research, the development of new testing tools, and the education of testers all stand to substantially benefit a modern society that is increasingly reliant upon software.

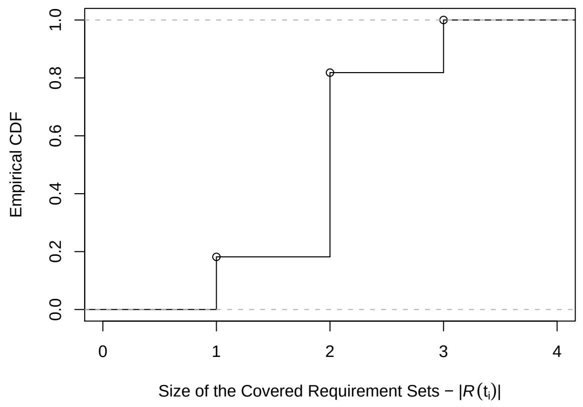


Figure 37: An ECDF of the Covered Requirement Sets for the Test Suite in Figure 10.

5 FURTHER INFORMATION

The ACM/IEEE International Conference on Software Engineering, the ACM SIGSOFT Symposium on the Foundations of Software Engineering, the ACM SIGSOFT International Symposium on Software Testing and Analysis, and the ACM SIGAPP Symposium on Applied Computing's Software Engineering Track are all important forums for new research in the area of regression testing. Other important conferences include: IEEE/ACM Automated Software Engineering, IEEE International Conference on Software Maintenance, IEEE International Symposium on Software Reliability Engineering, IEEE/ACM International Symposium on Empirical Software Engineering and Measurement, IEEE/NASA Software Engineering Workshop, and IEEE Computer Software and Applications Conference. The *IEEE Transactions on Software Engineering* and the *ACM Transactions on Software Engineering and Methodology* are two noteworthy journals that publish regression testing papers. Other journals include: *Software Testing, Verification, and Reliability*, *Software: Practice and Experience*, *Software Quality Journal*, *Automated Software Engineering: An International Journal*, *Empirical Software Engineering: An International Journal*, and *Information and Software Technology*. Magazines that publish software testing articles include *Communications of the ACM*, *IEEE Software*, *IEEE Computer*, and *Better Software* (formerly known as *Software Testing and Quality Engineering*). ACM SIGSOFT also sponsors the bi-monthly newsletter called *Software Engineering Notes*.

Acknowledgments. The biblical verse "Test everything. Hold on to the good" from Thessalonians 5:21 (New International Version) has served as a constant source of motivation during the completion of this chapter. The constructive feedback and valuable assistance from Adam Smith, Alexander Conrad, Zachary Williams, and Arpan Agrawal have better enabled me to produce a chapter that will hopefully "pass the test" and ultimately be "held on to" by readers.

References

[1] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5), 1998.

[2] Mechelle Gittens, Hanan Lutfiyya, Michael Bauer, David Godwin, Yong Woo Kim, and Pramod Gupta. An empirical evaluation of system and regression testing. In *Proceedings of the Conference*

of the Centre for Advanced Studies on Collaborative Research, 2002.

[3] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2), 2001.

[4] Gregory M. Kapfhammer. *The Computer Science Handbook*, chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.

[5] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), 2001.

[6] Dick Hamlet and Joe Maybee. *The Engineering of Software*. Addison Wesley, Boston, MA, 2001.

[7] Scott McMaster and Atif M. Memon. Call stack coverage for test-suite reduction. In *Proceedings of the 21st International Conference on Software Maintenance*, 2005.

[8] Adam M. Smith and Gregory M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 24th Symposium on Applied Computing*, 2009.

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

[10] Hao Zhong, Lu Zhang, and Hong Mei. An experimental study of four typical test suite reduction techniques. *Information and Software Technology*, 50(6), 2008.

[11] Sreedevi Sampath, Renee C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and A. Gunes Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, 2008.

[12] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4), 2007.

[13] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2), 2002.

[14] David S. Rosenblum and Elaine J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3), 1997.

[15] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time-aware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2006.

[16] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[17] Jeffrey M. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–735, 1992.

[18] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[19] Matthew Rummel, Gregory M. Kapfhammer, and Andrew Thall. Towards the priortiziation of regression test suites with data flow information. In *Proceedings of the 20th Symposium on Applied Computing*, 2005.

[20] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[21] Gregory M. Kapfhammer and Mary Lou Soffa. Database-aware test coverage monitoring. In *Proceedings of the 1st India Software Engineering Conference*, 2008.

[22] Scott McMaster and Atif Memon. Call stack coverage for GUI test-suite reduction. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, 2006.

[23] Scott McMaster and Atif M. Memon. Fault detection probability analysis for coverage-based test suite reduction. In *Proceedings of the 23rd International Conference on Software Maintenance*, 2007.

[24] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[25] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3), 1993.

[26] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[27] Alexander Conrad, Robert S. Roos, and Gregory M. Kapfhammer. Empirically studying the role of selection operators during search-based test suite prioritization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2010.

[28] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.

[29] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software, Practice, and Experience*, 23(11):1249–1265, 1993.

[30] Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, 2000.

[31] Brian Zorman, Gregory M. Kapfhammer, and Robert S. Roos. Creation and analysis of a JavaSpace-based genetic algorithm. In *Proceedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2002.

[32] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the Conference on Supercomputing*, 2007.

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

[34] Shin Yoo, Mark Harman, and Shmuel Ur. Measuring and improving latency to avoid test suite wear out. In *Proceedings of the Workshop on Search-Based Software Testing*, 2009.

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

[36] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[37] Testar - selective testing tool for Java. 2009. http://google-testar.sourceforge.net/.

[38] Sara Alspaugh, Kristen R. Walcott, Michael Belanich, Gregory M. Kapfhammer, and Mary Lou Soffa. Efficient time-aware prioritization with knapsack solvers. In *Proceedings of the Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, 2007.

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

[40] Gregory M. Kapfhammer, Mary Lou Soffa, and Daniel Mosse. Testing in resource constrained execution environments. In *Proceedings of the 20th International Conference on Automated Software Engineering*, 2005.

[41] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2004.

[42] Eckart Zitzler and Lothar Thiele. Multi-objective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4), 1999.

[43] 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.

[44] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. Technical Report 03-01-01, Department of Computer Science and Engineering, University of Nebraska – Lincoln, January 2003.

[45] Shirley Dowdy, Stanley Wearden, and Daniel Chilko. *Statistics for Research*. Wiley-Interscience, third edition, 2004.

[46] Jr. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts, 1995.