AN ABSTRACT OF THE DISSERTATION OF

Chaoqiang Zhang for the degree of Doctor of Philosophy in Computer Science presented on June 10, 2015.

Title: Test Case Reduction, Prioritization, and Selection for Symbolic Execution

Abstract approved: ______________________________

Alex David Groce

Software testing is of critical importance for the success of software projects. Current inefficient testing methods often still take up half or more of a software project's budget. Automatic test data generation is the most promising way to lower the software testing cost. Manually creating testing data is expensive and often needs deep domain knowledge. Therefore, both industry and academia are always highly interested in automatic approaches to generating test data.

Symbolic execution has been one of the most promising and exciting areas of automated testing research for many years now. In principle, symbolic execution "runs" a program, replacing concrete inputs with symbolic variables that represent all possible values. When a program branches, the execution takes both paths (if they are feasible under current constraints) and a set of path conditions on symbolic variables is modified for each path to record the new constraints on the symbolic values.

However, scaling symbolic execution to large programs or programs with complex inputs remains difficult due to path explosion and complex constraints, as well as external method calls. Additionally, creating an effective test structure with symbolic inputs can be difficult. A popular symbolic execution strategy in practice is to perform symbolic execution not “from scratch” but based on existing test cases. This dissertation explores the idea that the effectiveness of this approach to symbolic execution can be enhanced by (1) reducing the size of seed test cases, (2) prioritizing seed test cases to maximize exploration efficiency, and (3) selecting a subset tests for symbolic execution without specifying the number of tests. The proposed test case reduction strategy is based on a recently introduced generalization of delta-debugging, and our prioritization and selection techniques include novel methods that, for this purpose, can outperform some traditional regression testing algorithms. Our results show that applying these methods can significantly improve the effectiveness of symbolic execution based on existing test cases.

Test Case Reduction, Prioritization, and Selection for Symbolic Execution

by

Chaoqiang Zhang

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 10, 2015
Commencement June 2015

ACKNOWLEDGEMENTS

First of all I would like to express my thanks to professor Alex Groce for his mentorship , encouragement, and patience which have helped me grow into a more confident researcher over the past six years. I am very grateful to him for providing an excellent research group and for giving me the freedom to explore various research ideas.

During my time at Oregon State University I have collaborated with many wonderful researchers. I would especially like to thank Mohammad Amin Alipour, Ali Aburas, Arpit Christi, Rahul Gopinath, and Iftekhar Ahmed for discussing some interesting research topics together; thank Lu Zhao for mentoring me when I was a summer intern at HP Fortify; thank Milos Gligoric, Darko Marinov, and Rohan Sharma at UIUC for the collaboration on coverage criteria comparison work; thank Yang Chen and John Regehr at University of Utah for the pleasure of working with and learning from each of you, as well as my Ph.D. committee for your feedback and insights: Weng-Keen Wong, Carlos Jensen, Xiaoli Fern, Maggie Niess, and Martin Erwig.

Finally, none of this work would have been possible without the support of family and friends. I'd particularly like to acknowledge Jeffery Wolf, Hongmei Wolf, Ethan Wolf, Haejin Kim, Paul Cheong, Sarah Song, the Griffiths family, Piawah Sim, Kok Meng Lam, Craig Hanson, Liping Hu, Sherman Juang, Catherine Fang, and Gary Ferngren for their long-lasting support and encouragement; and family, for endless support to my study. Thank you.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF ALGORITHMS

LIST OF APPENDIX TABLES

LIST OF APPENDIX TABLES (Continued)

LIST OF APPENDIX TABLES (Continued)

Chapter 1: Introduction

1.1 Motivation

Software testing plays a crucial role in a software project. After completing a software project, software developers spend significant effort testing it. This includes planning the test cases, generating test inputs for exploring program behaviors, etc.

According to a 2002 NIST report[1], it is estimated that over \$22 billion of the costs of software errors could be eliminated with better software testing techniques. Current inefficient testing methods often still take up half or more of a software project's budget. Out of all those testing costs, generating test inputs for running a program takes a huge amount of time. Those include, but are not limited to, generating inputs for exploring every possible program behavior, valid inputs, invalid inputs, and generating performance testing data.

To generate these data, software firms need to hire professionals who know the application domain well to manually generate inputs. These are expensive and sometimes some important program behaviors could be missed.

Automatic test data generation is the most promising way to lower the software testing cost. Manually creating test input data is expensive and often needs deep

[1] `http://www.nist.gov/director/planning/upload/report02-3.pdf`.

domain knowledge. Therefore, both industry and academia are always highly interested in automatic approaches to generating test data.

Random testing is one of the most effective ways of generating large amounts of testing data. By defining input ranges and structures, software testers build up a random testing framework. This framework can generate millions of test inputs in a very short time [Hamlet, 1994]. But its disadvantages are also clear: 1) defining a random testing framework needs deep domain knowledge, 2) random testing can miss some important cases. For example, a program with a single integer input only crashes when the input is 1000; if the random testing framework allows any input, the probability of the program crashing is very low.

For the last several decades, researchers have been trying to reason about a program's inputs from its source code. The basic motivation behind symbolic execution is to produce inputs to explore program behaviors [King, 1976]. Since it was proposed in the 1970s, symbolic execution has been one of the most promising and exciting areas of automated testing research [Godefroid et al., 2005]. Symbolic execution "runs" a program, replacing concrete inputs with symbolic variables that represent all possible values. When a program branches, the execution takes both paths (if they are feasible under current constraints) and a set of path conditions on symbolic variables is modified for each path to record the new constraints on the symbolic values. Constraint solvers are used to produce a concrete input from a symbolic path or to check safety of operations (for example, determining if current path constraints ensure that a division is applied to a non-zero value).

Dynamic symbolic execution combines symbolic execution with concrete execution in order to improve scalability [Godefroid et al., 2005]. However, the promise of (dynamic) symbolic execution-based testing remains to be challenged by scalability issues, including the path explosion problem and the challenge of solving for complex constraints. In terms of program paths generated per unit of testing time, symbolic execution can be a much less efficient approach to automated testing than explicit-state model checking or random testing [Groce et al., 2014b]. For some large programs and generalized symbolic harnesses, symbolic execution basically does not yet work.

One popular method for addressing these problems is to not perform symbolic execution "from scratch" using a highly generalized test harness, but to seed symbolic execution from existing inputs, augmenting an existing suite [Xu et al., 2010, Xu et al., 2011, Kim et al., 2012, Kim et al., 2014]. For example, the `zesti` (Zero-Effort Symbolic Test Improvement) extension of `KLEE` (an open-source symbolic execution engine) is based on the observation that symbolic execution can take advantage of regression test cases [Marinescu and Cadar, 2012]. Other work, motivated by a desire to efficiently reproduce field failures, has shown that symbolic execution can scale better based on a "test case" (a call sequence) than when using only a point-of-failure or call stack [Jin and Orso, 2012]. The best-known and most successful large-scale application of symbolic execution to real-world testing, Microsoft's SAGE, is based on seeded exploration from test inputs [Bounimova et al., 2013].

Directed symbolic execution, since it was proposed, has been an active research topic in software testing and verification field. Many research efforts have addressed reducing the cost of constraint solving, external method calls, and distributed symbolic execution.

For a typical software project, either developers already created some test inputs or they built some automated tests generation tools for generating test cases automatically. Seeded symbolic execution, on the other hand, needs to explore input tests in a one-by-one manner. Current research just uses some of the existing tests. No research has focused on how to choose tests when facing a large test pool. In addition to the selection problem, some tests are designed for exploring complicated program behavior and thus are very complicated. They might need a long time to run and cause very long program execution paths. An execution path can be understood as a sequence of executed program statements. Those complicated tests are not suitable for seeding symbolic execution. Therefore, a natural question is how we can simplify these tests before feeding them to a symbolic executor.

In this dissertation, I investigate how traditional software testing techniques can be used to address these problems. Particularly, I show how test case reduction, prioritization, and selection can be used to help seeded symbolic execution when facing large number of complicated existing test cases. Figure 1.1 shows the main problems we address in this dissertation and the main techniques for them.

1.2 Contributions

The main goal of this dissertation project is to investigate how we can improve the efficiency of seeded symbolic execution based on a suite of existing test cases. In

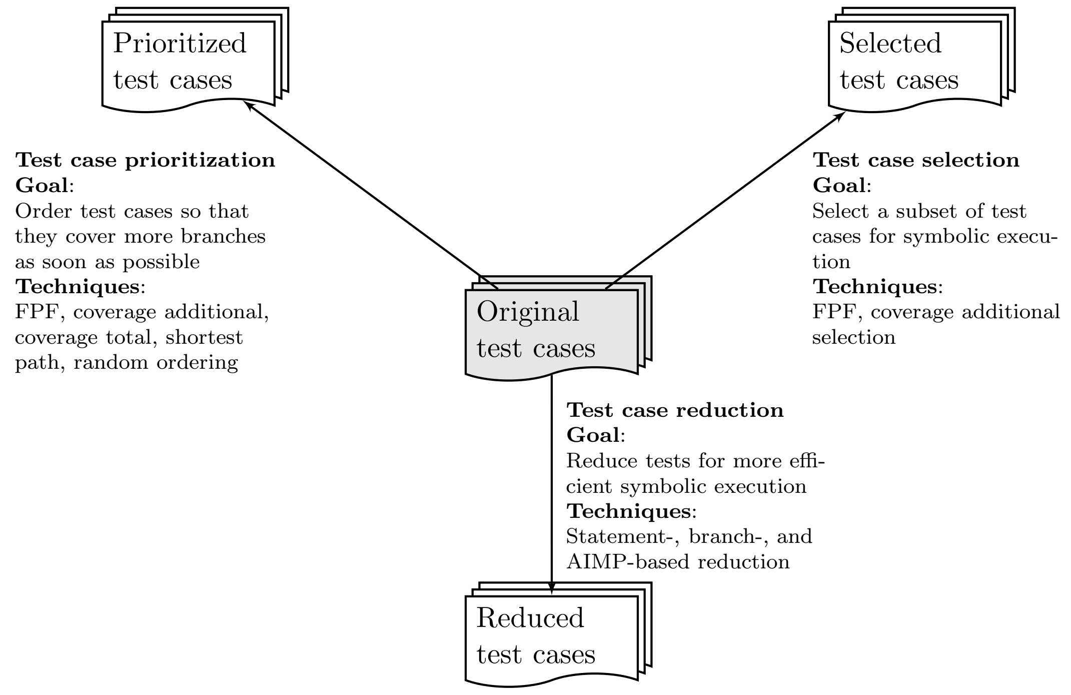

Prioritized test cases
Selected test cases
Test case prioritization
Goal:
Order test cases so that they cover more branches as soon as possible
Techniques:
FPF, coverage additional, coverage total, shortest path, random ordering
Test case selection
Goal:
Select a subset of test cases for symbolic execution
Techniques:
FPF, coverage additional selection
Original test cases
Test case reduction
Goal:
Reduce tests for more efficient symbolic execution
Techniques:
Statement-, branch-, and AIMP-based reduction
Reduced test cases

Figure 1.1: Dissertation framework.

particular, we investigate (1) how to *transform* some existing test cases so that a symbolic executor can utilize them better for exploring more program behaviors, (2) given a large number of existing test cases, how should we *rank* them such that a symbolic executor can explore more program behaviors in a given time, and (3) given a large number of existing test cases, how we can *choose* a small number of tests for symbolic execution but do not lose too many branches.

For these research questions, we explore the idea that the effectiveness of the seeded approach to symbolic execution can be enhanced by (1) reducing the size of seed test cases, (2) prioritizing seed test cases to maximize exploration efficiency, and (3) selecting a subset of tests for symbolic execution without specifying the number of tests. The proposed test case reduction strategy is based on a recently introduced generalization of delta-debugging, and our prioritization and selection techniques include novel methods that, for this purpose, can outperform some traditional regression testing algorithms. Our experimental results show that applying these methods can significantly improve the effectiveness of symbolic execution based on existing test cases.

1.3 Organization

This dissertation is organized as follows. Chapter 2 introduces some software testing concepts and techniques used in the rest of dissertation and summarizes some related work. Chapter 3 describes our methodology and experimental results for test case reduction. Chapter 4 presents test case prioritization and selection methods and experimental results for them. I conclude this dissertation in Chapter 5.

Chapter 2: Background and Literature Review

To improve the accessibility of this dissertation, I will give a brief introduction for all the important, but not widely spread or talked about, concepts, techniques, and tools used in this dissertation. It will include, but is not limited to, (1) what is symbolic execution, how it works, how seeded symbolic execution works, what is path constraints and different symbolic execution search strategies. With some examples, I will show why symbolic execution is hard to scale up to large-scale software systems, (2) the concepts of test case, test case minimization, test case prioritization and selection, and general techniques for them, (3) different structural coverage criteria, such as statement, branch, and acyclic intra-method path coverage, and (4) delta-debugging technique for test case reduction and its implementation introduced in Zeller's paper [Hildebrandt and Zeller, 2000].

2.1 Software Testing Concepts

Test Cases: For a Software Under Test (SUT), a test case is a program input which is designed for exercising a certain program behavior. In addition to test input, it also includes the expected output derived from the input. A *test suite* includes a set of test cases for a particular testing purpose. For example, a test suite for an *absolute(int x)* (In Figure 2.1) function might have three test cases (as

```
int absolute(int x) {
  if(x>0) {
    return x;
  else if(x==0)
    return 0;
  else
    return -x;
}
```

Figure 2.1: absolute as a running example.

shown below) for exercising its behaviors for positive, negative, and zero inputs.

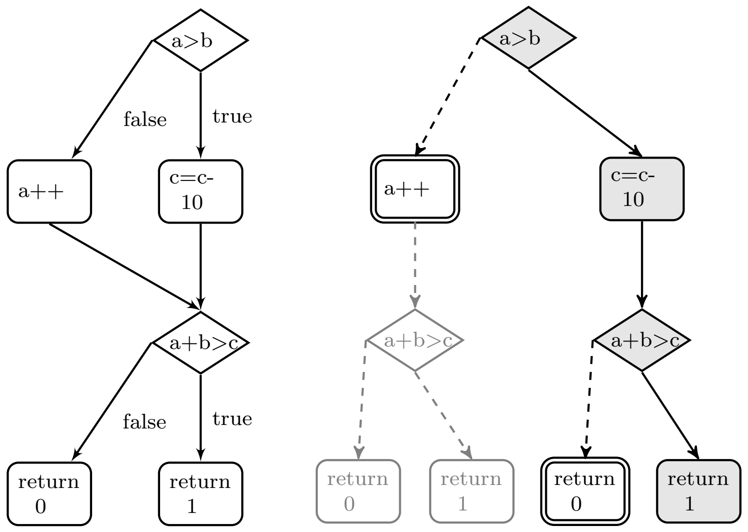
Test case 1: input “3”, expected output “3”;

Test case 2: input “-2”, expected output “2”;

Test case 3: input “0”, expected output “0”.

Coverage Criteria: A fundamental question in software testing research is how to compare test suites, often as a means for comparing test-generation techniques. Researchers frequently compare test suites by measuring their coverage. A coverage criterion C provides a set of test requirements and measures how many requirements a given suite satisfies. In this dissertation, we consider both traditional (statement and branch) and recently used (based on program paths) coverage criteria. Statement and branch coverage are well-known structural criteria [Ammann and Offutt, 2008]. They measure how many different program statements and branch decisions are hit.

```
int foo(int a,int b,
        int c)
{
  if( a>b ) {
    c=c-10;
  } else {
    a++;
  }
  if( a+b>c ) {
    return 1;
  } else {
    return 0;
  }
}
```



(a) (b) (c)

Figure 2.2: A C program `foo`, its control flow graph, and its symbolic execution tree.

2.2 Symbolic Execution

Symbolic execution is a static program analysis technique that "runs" a program with symbolic instead of concrete inputs. In software testing, it is used for generating test inputs to drive a program along all feasible *paths*. A program *path* is a sequence of branches or basic blocks traversed during a program execution.

Figure 2.2 shows a program and its control flow graph (CFG). This program accepts three integer inputs a, b, and c, and has four possible program paths formed by taking `true` or `false` branches of the two `if` statements. Manually generating test inputs (i.e., the values of a, b, and c) is not difficult for this particular program due to its simplicity. However, for a larger program with thousands of branches, manually specifying inputs to exercise a particular path becomes very challenging if it is not impossible. Symbolic execution was designed for accomplishing this data generation task automatically with the help of *constraint solvers*. For the CFG in Figure 2.2 (b), a symbolic executor first identifies a program path (e.g., ⟨`true`, `true`⟩) in its control flow graph. Along this particular path, the executor collects the path's *path constraint*:

$$a > b \bigwedge c' = c - 10 \bigwedge a + b > c' \tag{2.1}$$

A path constraint (PC) is a boolean formula over the input symbolic variables that the inputs must be satisfied for an execution to follow a path. Note that the variable c's value is changed by the assignment statement $c = c - 10$, the executor has to give it a new name c' after the assignment. The executor then asks a constraint solver if the collected constraint has a solution. The underlying solver either answers "yes" and gives a possible assignment of the variables (in this case, $a = 1$, $b = 0$, $c = 10$), or answers "not" (proving the corresponding path is not *feasible*). By traversing all the four possible paths in the CFG and solving all the constraints, the executor can generate four test inputs for this program.

Godefroid et al. [Godefroid et al., 2005] introduced a dynamic technique to direct symbolic execution. It first uses a random concrete input to execute the underlying program along a path p. Then, for each branch decision on p, it negates the branch and asks a solver if the resulting constraint is satisfiable if the input is symbolic. In Figure 2.2 (c), assume the random input is {a=1, b=0, c=10},

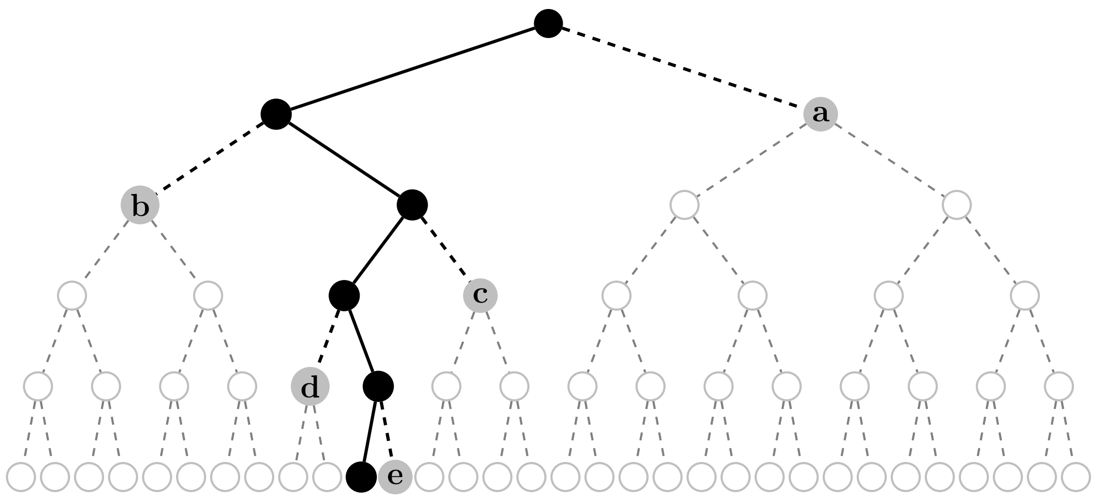


Figure 2.3: A symbolic execution tree. The black circles show the trace of a concrete execution. The gray circles are feasible states after the seeding stage.

the program execution will follow the path ⟨`true`, `true`⟩ (highlighted). For the resulting path constraint:

$$a > b \bigwedge c' = c - 10 \bigwedge \underline{a + b > c'} \tag{2.2}$$

It may negate the branch decision $a + b > c'$ and get a new path constraint:

$$a > b \bigwedge c' = c - 10 \bigwedge \underline{a + b <= c'} \tag{2.3}$$

The underlying constraint solver will answer "yes" and give a solution to this new constraint.

Symbolic Execution Search Strategies: Typically, a program has many possible execution paths. A symbolic executor might not be able to explore them all in a given time budget. In this case, it has to decide which paths should be explored first to achieve a particular goal (e.g., covering more code structure, discovering more buffer overflows). Various strategies have been investigated and implemented in some research tools [Cadar et al., 2008, Burnim and Sen, 2008, Marinescu and Cadar, 2012]. For example, in Equation (2.1), there are two path conditions (i.e., $a > b$ and $a + b > c'$) that can be negated. Depending on different search strategies, a symbolic executor can choose the deepest condition to negate first, the shallowest one first, random one, or the one based on a heuristic strategy as shown in Figure 2.3.

Previous investigations have considered the question of how best to symbolically augment a test suite for purposes of dealing with *changes* to the code [Xu et al., 2010, Marinescu and Cadar, 2013]. However, the literature does not, to our knowledge, address the problem of seeded symbolic execution in general, aimed at improving an existing test suite for a fixed program.

Classical symbolic execution was first introduced in 1970s [King, 1976, Boyer et al., 1975, Clarke, 1976]. Recently, Godefroid et al. [Godefroid et al., 2005] introduced dynamic automated random testing (a.k.a., concolic testing) for testing real-world software. Its main advantage over classical symbolic execution is that it uses concrete values of pointers, hash functions, system calls, and other values that constraint solvers do not handle well [Cadar et al., 2008, Godefroid et al., 2005, Sen et al., 2005] . Since exploring all the paths of a real-world program is generally impossible, different search strategies have been investigated for achieving various symbolic execution goals. Cadar et al. [Cadar et al., 2008] used random-path strategy for achieving high code coverage. Burnim and Sen investigated several strategies for symbolic execution and proposed a static code structure-based strategy [Burnim and Sen, 2008]. Li et al. introduced an unified strategy to guide symbolic execution to less explored parts of a program [Li et al., 2013]. Recently, Seo and Kim presented a context-guided search strategy in concolic testing [Seo and Kim, 2014]. Some exhaustive surveys on symbolic execution can be found in the papers by Păsăreanu and Visser [Păsăreanu and Visser, 2009], Cadar et al. [Cadar et al., 2011], and Cadar and Sen [Cadar and Sen, 2013].

One of the popular implementations for classical symbolic execution is described in Algorithm 2.1. A program state s is used to track each possible program path during symbolic execution. It normally includes the current path constraints (pc), the current instruction ($instr$), memory data mapping which maps a variable to its symbolic expression value (M), etc.

2.2.1 Using Existing Test Cases for Seeding Symbolic Execution

Seeded dynamic symbolic execution takes an initial test case and tries to cover a branch that test has not covered. This initial test case is called the seed. In the literature, the seed is usually chosen arbitrarily from a pool of test cases [Godefroid et al., 2005, Sen et al., 2005, Garg et al., 2013, Majumdar and Sen, 2007]. Some methods choose seeds for regression purposes [Xu et al., 2010]. `KATCH` [Marinescu and Cadar, 2013] chooses the test case closest (by a distance metric based both on static code structure and weakest preconditions) to a target which is a code

Algorithm 2.1 Classical symbolic execution $SymbolicExe(P, S, I_s)$

```
Let  $P$    be the program to be symbolically executed
     $S$    be a set of initial symbolic states
     $I_s$  be symbolic inputs

 1: $states \leftarrow S$
 2: while $states \neq \emptyset$ do
 3:     $s \leftarrow$ choose a state from $states$ with different search strategies
 4:     $instr \leftarrow$ the current instruction of $s$
 5:     if $typeof(instr)$ is v:=e then
 6:         $s \leftarrow s.M[v \leftarrow e]$        ▷ $M$ maps a variable to its symbolic value
 7:     else if $typeof(instr)$ is if $C$ then $stmt_1$ else $stmt_2$ then
 8:         if $\neg C(I_s)$ is satisfiable then
 9:             $sc \leftarrow$ clone state $s$
10:             $sc.pc \leftarrow sc.pc \wedge \neg C$                ▷ $pc$: path constraint
11:             $states \leftarrow states \cup \{sc\}$
12:         end if
13:         if $C(I_s)$ is satisfiable then
14:             $s.pc \leftarrow s.pc \wedge C$
15:         end if
16:     else if $typeof(instr)$ is exit then
17:         generate a test case by solving constraint $s.pc$
18:         remove $s$ from $states$
19:     else if $typeof(instr)$ is assert($e$) then
20:         generate a failure-inducing test case by solving constraint $s.pc$
21:         remove $s$ from $states$
22:     end if
23: end while
```

modification in a patch. Our work differs from previous efforts partly in goal but primarily in that we manipulate the test cases in the pool prior to performing symbolic execution and use novel prioritization and selection methods.

Instead of running a program symbolically from scratch, seeded symbolic execution works in a two-stage way: In stage I, a program is executed concretely and

Algorithm 2.2 Seeded symbolic execution $SeededSymbolicExe(P, I_c, I_s)$

```
Let  $P$    be the program to be symbolically executed
     $I_c$  be concrete inputs
     $I_s$  be symbolic inputs

1: run program concretely and symbolically in parallel
2: $states \leftarrow$ {states on all diverge points}
3: $SymbolicExe(P, states, I_s)$                    ▷ Invoke Algorithm 2.1
```

each diverge point is checked to see if its negated branch is feasible if the inputs are symbolic. If its negated branch is feasible, this diverge point will be added to the to-be-explored states for stage II. In stage II, from the residual states from stage I, a seeded symbolic executor works in exactly the same way as a classical symbolic executor does. Algorithm 2.2 describes this two-stage symbolic execution.

A comprehensive literature review for symbolic execution is out of range of this dissertation. Saswat Anand collects a list of papers at this website [SymbExeBib,].

2.3 Test Case Reduction

Large test cases, especially those generated by random testing, usually include redundant behaviors. Understanding such long test cases is difficult for developers. Delta debugging [Hildebrandt and Zeller, 2000] is a general technique for reducing the size of a failing test case while holding the fact that the test case fails constant. Test case reduction has been traditionally applied only to failing test cases. The most relevant work considers delta debugging in random testing [Lei and Andrews, 2005, Groce et al., 2014a, Groce et al., 2014b, Leitner et al., 2007], which tends to produce complex, essentially unreadable, failing test cases [Lei and Andrews, 2005]. Random test cases are also highly redundant, and the typical reduction for random test cases in the literature ranges from 75% to well over an order of magnitude [Lei and Andrews, 2005, Leitner et al., 2007, Groce et al., 2007, Regehr et al., 2012, Chen et al., 2013]. Reducing highly-redundant test cases to enable debugging is an essential enough component of random testing that some form of automated reduction seems to have been applied even before the publication of the delta debugging algorithm, e.g. in McKeeman's early work [McKeeman, 1998], and reduction for compiler testing is an active research area [Regehr et al., 2012].

Recently, we proposed using delta debugging based on preserving *code coverage* to reduce both failing and successful test cases for purposes of building very fast regression suites [Groce et al., 2014a]. This dissertation adapts the same technique to the problem of improving the scalability of symbolic execution. Recent work has shown that reduction has other uses: Chen et. al showed that reduction was required for using machine learning to rank failing test cases to help users sort out different underlying faults in a large set of failures [Chen et al., 2013].

Delta Debugging: Delta debugging (DD) was first proposed to reduce the size of a failing test case. Software developers often use failing test cases to debug the SUT and try to figure out the *root cause* of a defect. However, complicated test cases often cause complicated program states, which make themselves hard to be understood. Ideally, a failing test case should be small enough for understanding and still exposes the same software failures. Delta debugging is a technique that systematically “slices away” irrelevant parts of a test case and makes it as small as possible. For example, for an input C program that causes a compiler crash, a DD tool can remove some source code lines from the program, and see if the reduced program still causes the crash. It uses a variation of binary search to remove individual lines from the program until it cannot remove more. The resulting input program should be smaller than the original one but also exposes the same crash. With this reduced C program, compiler developers can check the program behavior in a much easier way.

2.4 Test Case Prioritization

A mature software usually has many test cases. Running all the test cases may take a long time. To discover faults in the software earlier, a possible way is to rank all the test cases so that the faults in the software can be exposed as early as possible. Using the $absolute(x)$ function as an example (In Figure 2.1), for a list of test inputs {0, 3, 4, 5, -1, -200, 20}, a “good” prioritization in terms of branch coverage can be {0, 3, -200, 20, 4, 5, -1} since the first three test inputs exercise all the branch decisions of the program.

Test case prioritization [Yoo and Harman, 2012] ranks test cases such that faults can be discovered earlier in the testing process. Rothermel et al. [Rothermel et al., 1999] have proposed and experimented with different orderings of test cases based on statement and branch coverage. Statement (branch) total prioritization ranks test cases based on their absolute statement (branch) coverage; the higher the statement (branch) coverage of a test case, the sooner it is executed. Statement (branch) delta prioritization ranks test cases by the *new* coverage over all previously ranked test cases. Zhang et al. [Zhang et al., 2013] have explored the wide variety of sophisticated algorithms for prioritization. While those approaches have focused on minimization [McMaster and Memon, 2006, Offutt et al., 1995, Hsu and Orso, 2009, Chen and Lau, 1996a], selection [Bible et al., 2001], and prioritization [Elbaum et al., 2000, Elbaum et al., 2004, Walcott et al., 2006] at the granularity of entire test suites, this dissertation uses the reduction of the size of the test cases composing the suite, a “finer-grained” approach.

In addition to various prioritization strategies, test case prioritization can have many variations depending on the underlying coverage criteria. Researchers have been using many criteria, such as statement coverage, block coverage [Do et al., 2004], branch coverage [Rothermel et al., 1999], function coverage [Elbaum et al., 2000], and modified condition/decision coverage [Jones and Harrold, 2001] for prioritization.

2.5 Test Case Selection

Test case selection systematically removes test cases from a test suite and make the reduced test suite still meet a certain testing requirement. Again, using the $absolute(x)$ function as an example (In Figure 2.1), for the test suite which includes inputs {0, 3, 4, 5, -1, -200, 20}, if the testing requirement is to cover all branches, instead of running all test inputs, one can just choose {0, -200, 20} for running since these three test inputs can drive the program along all possible branches. By only running a subset of all the tests, for a complicated programs with huge amount of test cases, regression testing can be done in a much shorter time.

Since test case selection is a NP-complete problem [Yoo and Harman, 2012], most selection techniques are heuristic-based. Chen and Lau [Chen and Lau, 1996b] employed GE and GRE heuristics for test case selection and compared the results to HGS heuristic [Harrold et al., 1993]. Marré and Bertolino considered test case selection as a spanning set over a graph problem [Marré and Bertolino, 2003]. Yoo and Harman take the selection as a multi-objective optimization problem [Yoo and Harman, 2007].

A comprehensive literature review of test case selection is out of the scope of this dissertation. Yoo and Harman [Yoo and Harman, 2012] gives a complete survey on test case selection.

Chapter 3: Test Case Reduction for Symbolic Execution

3.1 Introduction

Previous investigations have considered the question of how best to symbolically augment a test suite for purposes of dealing with *changes* to the code [Xu et al., 2010, Marinescu and Cadar, 2013]. However, the literature does not, to our knowledge, address the problem of seeded symbolic execution in general, aimed at improving an existing test suite for a fixed program. Because of its high cost as a pure technique and its improved effectiveness given a base of test cases, dynamic symbolic execution may best fit into a **two-stage** testing process. In the first stage, a much less expensive automated testing approach, such as random testing [Pacheco et al., 2007, Groce et al., 2007] is used to generate an initial suite of test cases until coverage saturates. Remaining uncovered branches at this point are likely to be difficult enough to cover to justify the cost of symbolic execution. In many cases, the probability of coverage for some reachable branches by randomized methods is essentially zero. The goal of the two-stage approach is to maximize total code coverage as rapidly as possible, under the assumption that this will result in effective and efficient fault detection.

The first stage of this two-stage approach to testing is difficult to generalize: the optimal approach depends heavily on the testing method used and the structure of the test space. This dissertation considers the more well-scoped problem of improving the effectiveness of the second stage: *given a suite of existing test cases, how can we improve the efficiency of seeded symbolic execution based on those test cases?* The evaluation criteria for proposed methods is simple: given a total testing budget B, how can additional code coverage over the starting suite be maximized?

3.2 Methodology

Two basic approaches are possible. First, the test cases themselves may be modified to improve their suitability as a basis for symbolic exploration. Symbolic techniques are generally more expensive to apply to larger seeds, for the obvious reasons (the set of paths is larger, the program state becomes more complex, and there are more symbolic inputs to solve for). It is therefore reasonable to expect that reducing the size of input test cases should improve the scalability of symbolic exploration.

Principally, seeded symbolic execution works in a two-stage manner. We use `KLEE` symbolic executor as an example to explain the process. In stage I, the SUT is executed concretely and related symbolic information is collected for stage II. For each conditional statement, the executor first uses the concrete input values to decide which branch should be taken. If it finds that the `true` branch should be taken, instead of ignoring the `false` branch, the executor will ask a constraint solver if the `false` branch is feasible in the case of that the input values are symbolic. If the `false` branch is feasible, the executor then records all the related information (e.g., path constraints and memory state) into a data structure (i.e., state) for the stage II exploration. After this recording work, the symbolic executor continues the execution along the `true` branch until it finishes the whole concrete execution. In stage II, the symbolic executor explores all states from stage I in a pure symbolic manner.

Seeded symbolic execution was designed for reaching deep program states under the direction of concrete inputs. From the process we explained earlier, one obvious problem of seeded symbolic execution is that the stage I (i.e., seeding stage) may take too long time if the concrete inputs cause a long program execution, such as some loops are executed thousands of times. Although `KLEE` has a "timeout" command line option for seeding stage, a premature termination of concrete execution may prevent the program from reaching deep states. To address this problem, our first proposed technique is to make a test input smaller but with the same code coverage. By simplifying a test input, a symbolic executor can spend less time on seeding stage, thus more time on stage II exploration in a given time budget.

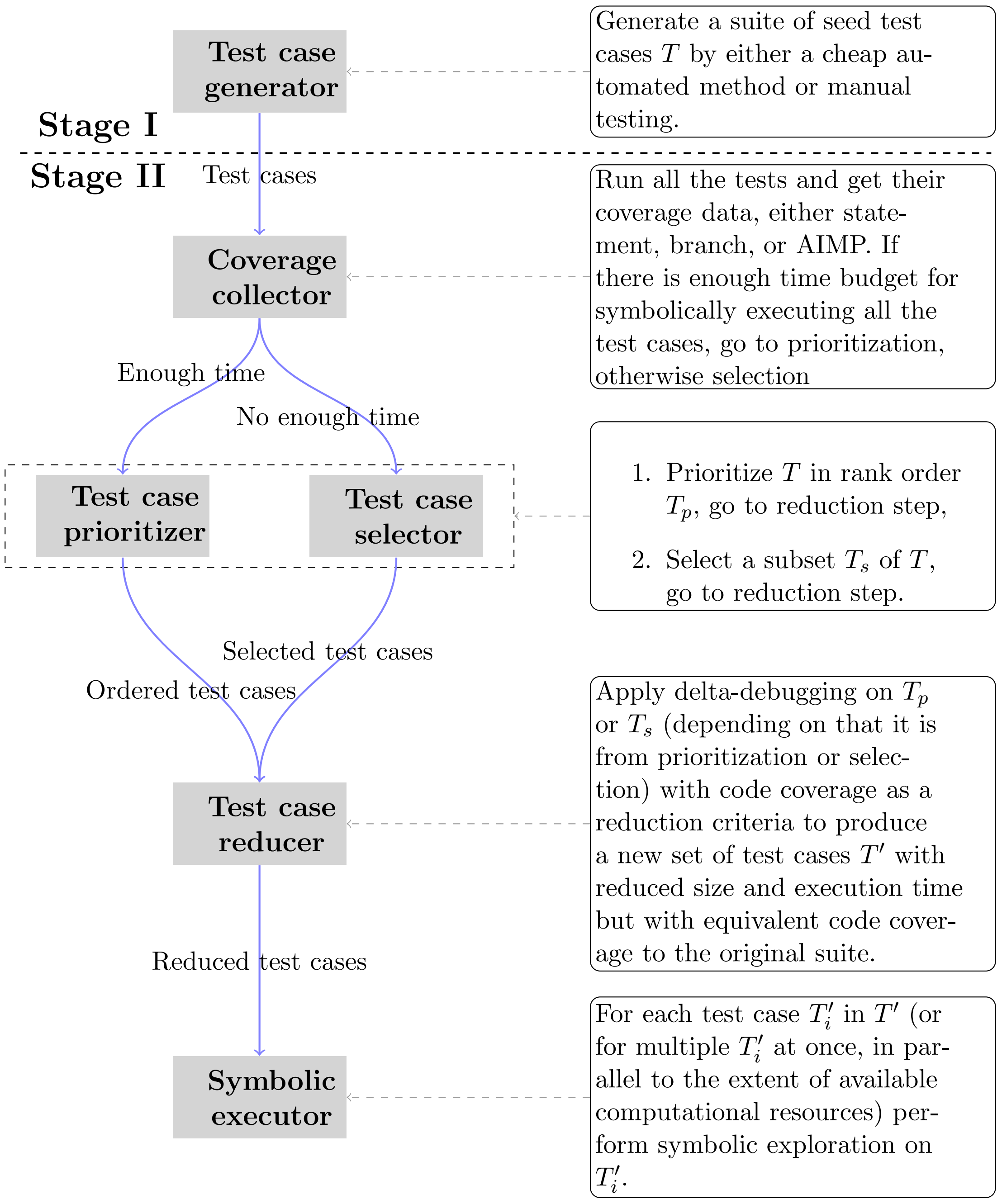
Research Hypothesis I: The basic purpose of seeded symbolic augmentation is arguably to explore the "coverage neighborhood" of a test input, which leads to the hypothesis that a test input of smaller size but with the same code coverage will almost always lead to more efficient symbolic exploration than a larger test that does not increase coverage over the smaller test. Delta-debugging [Zeller and Hildebrandt, 2002] is a well-known technique for reducing the size of test cases, but it is only useful for failing test cases, and often reduces test cases so much that they are a poor basis for symbolic exploration. However, recent work has shown that delta-debugging using *preservation of source code coverage* rather than failure as a property to be preserved can produce extremely efficient regression suites based on existing tests [Groce et al., 2014a].

Second, seeded symbolic exploration can be seen as a powerful variation of regression testing, which suggests the use of test case prioritization and selection [Yoo and Harman, 2012]. For example, an obvious approach is to attempt to run regression tests in such an order that total code coverage is maximized as quickly as possible.

Research Hypothesis II: The hypothesis is therefore that better ordering of the test cases used as symbolic seeds can improve the efficiency of symbolic execution as well. Because symbolic exploration is much more computation-demanding than simple replay of regression tests, some prioritizations that are not obviously useful for traditional regression testing (in that they do not aim for high code coverage) are considered. The `KATCH` tool [Marinescu and Cadar, 2013] uses a novel prioritization for seeded exploration based on both CFG distance and weakest preconditions to target changed code in patches; our aim is similar, but intended for the general case of increasing overall coverage over an existing test suite base.

The basic workflow[1] for seeded symbolic execution introduced in this dissertation is therefore (as Figure 3.1 illustrates):

[1]This is different from the workflow we introduced in our ISSTA 2014 paper in terms of the order of reduction and prioritization. In that paper, we only considered reduction and prioritization. Once we introduce selection into this process, we do not need to reduce all the tests since some (or most) of tests may not be selected and executed at all.



Test case generator
Generate a suite of seed test cases T by either a cheap automated method or manual testing.
Stage I
Stage II
Test cases
Coverage collector
Run all the tests and get their coverage data, either statement, branch, or AIMP. If there is enough time budget for symbolically executing all the test cases, go to prioritization, otherwise selection
Enough time
No enough time
Test case prioritizer
Test case selector
1. Prioritize T in rank order T_p, go to reduction step,
2. Select a subset T_s of T, go to reduction step.
Selected test cases
Ordered test cases
Test case reducer
Apply delta-debugging on T_p or T_s (depending on that it is from prioritization or selection) with code coverage as a reduction criteria to produce a new set of test cases T' with reduced size and execution time but with equivalent code coverage to the original suite.
Reduced test cases
Symbolic executor
For each test case T'_i in T' (or for multiple T'_i at once, in parallel to the extent of available computational resources) perform symbolic exploration on T'_i.

Figure 3.1: Workflow of this approach.

1. Generate a suite of seed test cases T by either a cheap automated method or manual testing. If there is enough time budget for symbolically executing all the test cases, go to Step 2, otherwise Step 3,

2. Prioritize T in rank order T_p, go to Step 4,

3. Select a subset T_s of T,

4. Apply delta-debugging on T_p or T_s (depending on that it is from prioritization or selection) with code coverage as a reduction criteria to produce a new set of test cases T' with reduced size and execution time but with equivalent code coverage to the original suite. For each test case T'_i in T' (or for multiple T'_i at once, in parallel to the extent of available computational resources) perform symbolic exploration on T'_i. Symbolic exploration consists of two steps:

   (a) Execute T'_i and collect all possible coverage divergence points (untaken branches),

   (b) Apply a symbolic execution search strategy based on these divergence points using T'_i as a seed.

This dissertation focuses on examining how steps 2, 3, and 4 of this process impact the effectiveness of the symbolic execution effort, measured in terms of improved incremental branch coverage for a time budget and the rate of coverage improvement. The primary contributions of this dissertation are (1) introducing methods for reducing, selecting, and prioritizing test cases for seeded symbolic execution and (2) experimentally demonstrating that these methods are effective in improving the efficiency of symbolic execution for 6 real C programs of moderate to large size. A secondary contribution is the dissertation that in general a two-stage framework may address some of the difficulties with scaling symbolic execution.

3.3 Test Case Reduction

The goal in our framework of using symbolic execution based on test cases is to improve the total program coverage. Seeded symbolic execution takes a test case and attempts to cover new program behavior based on it; for example, if a conditional is `false` in the original test case, and the symbolic engine finds the `true` branch to be feasible, it explores to find a test case covering the `true` branch. The essential quality of the seed test suite is therefore its code coverage. In general, given two test cases similar other than in length, symbolic execution has more difficulty with the longer test case, due to path explosion and increasingly complex constraints as the program executes. Therefore, given two test cases with the same code coverage, it seems likely that symbolic exploration based on the "smaller" test case will be more effective, though it is possible that the change *could* reduce the value of the seed test case (for example constraints may change and branch feasibility in context may change). *Cause reduction* allows us, given a test case T to use delta-debugging [Hildebrandt and Zeller, 2000] to produce a new test case T' such that T and T' have the same code coverage, and removing any component of T' will result in lower code coverage [Groce et al., 2014a]. Cause reduction uses

Algorithm 3.1 Test case reduction algorithm $ddmin(P, I, C, n)$

Let	P	be the program
	I	be a test input to be simplified
	C	be the coverage of test input I
	n	be the number of partitions of I

```
 1: split $I$ into $n$ components $I_1, I_2, \ldots, I_n$
 2: if $\exists I_i$ such that $Cov(I_i) = C$ then
 3:     return $ddmin(P, I_i, C, 2)$
 4: else if $\exists I_i$ such that $Cov(I - I_i) = C$ then
 5:     return $ddmin(P, I - I_i, C, max(n - 1, 2))$
 6: else if $n < size(I)$ then
 7:     return $ddmin(P, I, C, min(2n, sizeof(I)))$
 8: else
 9:     return $I$
10: end if
```

the usual *ddmin* algorithm of delta debugging, but replaces the check that reduced tests fail with a check that a reduced test covers at least all statements covered by the original test case. In experiments with the `SpiderMonkey` JavaScript engine, a suite of tests where each test was reduced while preserving statement coverage surprisingly showed *better* fault detection than the original suite, and statement coverage-based reduction was shown to preserve other important properties well (test case failure, branch coverage, and mutation kills) [Groce et al., 2014a].

Our test case reduction algorithm is described in Algorithm 3.1. For a given program P, a test case input I to be simplified, the coverage of test input I, and the number of components n of test input I to be split, the algorithm first divides input I into n components $I_1, I_2, \ldots, I_n$. If *ddmin* finds that one of the components I_i has the same coverage with the original coverage C, it runs *ddmin* on component

Table 3.1: Test case reduction strategies

Reduction strategy	Coverage constraint between unreduced test t_u and the reduced test t_r	
Statement-based reduction	$\mathrm{COV}_{\mathrm{Statement}}(t_u)$	$= \mathrm{COV}_{\mathrm{Statement}}(t_r)$
Branch-based reduction	$\mathrm{COV}_{\mathrm{Branch}}(t_u)$	$= \mathrm{COV}_{\mathrm{Branch}}(t_r)$
AIMP-based reduction	$\mathrm{COV}_{\mathrm{AIMP}}(t_u)$	$= \mathrm{COV}_{\mathrm{AIMP}}(t_r)$

I_i recursively to reduce I_i further. By contrast, if none of the components has the same coverage with the C, *ddmin* checks each $I - I_i$ in the same manner. If both I_i and $I - I_i$ cannot be reduced for all the n components, *ddmin* increases the granularity (i.e., the value of n) of partitioning by a factor of 2, and invokes itself on this finer granularity. The algorithm is first invoked with $n = 2$.

3.3.1 Variations of Test Case Reduction

Depending on the underlying coverage criteria, test case reduction implementation can have several variations. Table 3.1 lists all the reduction strategies we investigated in our experiment. Statement coverage-based reduction is an effective strategy that could be easily applied to current symbolic execution workflows. In addition to statement-based reduction, we implement and evaluate the reduction strategies based on branch and AIMP coverage criteria. Although we speculate that statement coverage-based reduction is an effective strategy that could be easily applied to current symbolic execution workflows, and may be difficult to improve on, given the need to balance (1) preserving the value of the existing test case while (2) obtaining enough reduction to aid symbolic execution, we need to evaluate them before we draw any definitive conclusions.

Our test case reduction experiments apply cause reduction to an initial seed test suite $(T_1, T_2, \ldots, T_n)$ to produce $(T'_1, T'_2, \ldots, T'_n)$ such that $\forall i.COV(T_i) = COV(T'_i)$, where COV is statement, branch, or AIMP coverage. It is possible for T_i to be equal to T'_i, but usually T'_i is smaller and executes faster than T_i, as results below shown. Cause reduction of a test case often takes a relatively long time to produce a 1-minimal [Hildebrandt and Zeller, 2000] test case. Usually the majority of the reduction achieved is obtained early, and the remaining computation produces little reduction. Unlike traditional delta-debugging, the purpose is not to remove all redundancy to aid human readers, but to make testing more efficient, so total minimality is not important. All experiments therefore use a limited budget for cause reduction, and return the smallest coverage-equivalent test discovered before timeout. The details of test case components and cause reduction implementation differ from program to program. In truth, any mature automated testing infrastructure *should* include a delta-debugging module to enable effective debugging [Lei and Andrews, 2005], and a delta-debugger is easily modified to implement coverage-based cause reduction.

For example, a test case for the `grep` utility consists of an input pattern string and an input file:

```
grep ‘patternstring’ inputfile.inp
```

To apply cause reduction to `grep`, our implementation first holds `inputfile.inp` constant, and uses simple character-based delta-debugging to find a (smaller) `patternstring` that results in the same statement (branch, and AIMP) coverage. The process is then repeated, holding `patternstring` constant and reducing `inputfile.inp`. The pattern string is reduced first because it is typically shorter and therefore less likely to result in a timeout. For test cases that are sequences of API calls (e.g. the `YAFFS2` file system or a container class), we can simply use each API call as a component and run one reduction. Note that while we count cause reduction against test budgets in our experiments, in practice projects might maintain cause reduced tests as fast regressions ("quick tests") [Groce et al., 2014a]. In all cases, we used statement, branch, and AIMP coverage-based reduction, even though symbolic execution is based on branches. In Section 3.7, we present our work for evaluating the effectiveness of test case reduction and prioritization [Zhang et al., 2014]. The major result of this prior work is that applying test case reduction and test case prioritization techniques to the initial set of test cases from which symbolic execution proceeds can improve the results of symbolic execution, both in terms of total additional branches covered and in terms of how quickly additional branches are covered. In several cases, test reduction leads to an improvement in additional branch coverage of 40-100%. The value of prioritization is a substantial improvement in the discovery curves for new branches as symbolic execution proceeds. For three of the six subjects, applying our methods not only improves the results for the best symbolic search strategy for the program, it improves a previously inferior strategy's results such that it becomes the best way to explore the program's behavior.

3.4 Experimental Methodology

As the first step for evaluating the methodology introduced in Section 3.2, this work considers three related research questions. First, in seed-based symbolic execution, the effectiveness of exploration depends on the seed test case itself, and some complex/large seeds can perform poorly. Second, each exploration takes significant time, and we would like to quickly improve coverage. Third, given a limited exploration time budget, we want to choose a subset of test cases for symbolic execution. The research questions are therefore:

- **RQ1:** *Can test case reduction improve the effectiveness of seeded symbolic execution?*
- **RQ2:** *Given a fixed search time for each seed test case, can ranking seed tests improve the efficiency of symbolic execution?*
- **RQ3:** *Given a large number of test cases, can we just select small amount of them and still do symbolic execution effectively?*

Because measuring actual faults detected is highly sensitive to the set of faults used, and makes it difficult to obtain statistical significance for results [Arcuri and Briand, 2011], our basic measure is incremental gain in branch coverage over an initial suite, for a fixed testing budget. There is considerable evidence that when branch coverage for two test suites differs by a significant amount, the suite with higher coverage is usually better at fault detection as well [Gligoric et al., 2013]. This approach combines both effectiveness (since the absolute coverage numbers can be compared given the fixed computation effort) and efficiency (in that results are for a given time), and matches the practical needs of testing, where budgets are not infinite and finding bugs sooner rather than later is critical. For prioritization, we follow previous practice and modify this basic evaluation measure to examine the average incremental coverage, thus capturing the climb of the discovery curve for new branches produced by a prioritization strategy (since once all tests have been completed, all prioritizations have the same final coverage results). In all of the experiments, the computation time required for evaluated approaches is counted against the total budget for testing; e.g., if a test case is minimized, the time to minimize is taken away from the time spent in symbolic execution. Because of the large number of test cases involved in each experiment (100) and the large number of experimental treatments run, we limited our time budgets to 10 and 20 minutes per test case (and used a large compute cluster to run symbolic execution instances in parallel).

All experiments are based on the `zesti` version of `KLEE`, which can combine seeded symbolic execution with `KLEE` search strategies. For a given test input, this approach executes the test case concretely and symbolically, and for each branch determines feasibility of the negated branch in the symbolic execution. Feasible branches are recorded along with their path constraints for a second-stage pure symbolic search using a chosen search strategy. Because the effectiveness of symbolic execution can vary depending on the search strategy, four `KLEE`-supported strategies are used in all experiments [Cadar et al., 2008, Li et al., 2013]. These are:

1. **Depth-First Search (DFS)**: DFS always continues from the latest execution state when possible. DFS, as in model checking, has the advantage of very low overhead in state selection but can become trapped in parts of the state space, e.g., when there are loops with few statements but that may execute many times.

2. **Random State Search (RSS)**: As the name suggests, RSS selects a random state for exploration. The advantages are uniform exploration and avoiding the traps of DFS; however, RSS can repeatedly generate test cases similar to those already produced.

3. **Random Path Selection (RPS)**: RPS uses a binary execution tree (where nodes are fork points and leaves are current states) to record explored parts of a program. RPS randomly traverses this structure, which advantages leaves high in the tree, motivated by the probability that these have fewer constraints and may be more likely to hit uncovered behavior. RPS can, like RSS, produce many similar test cases.

4. **Minimum Distance to Uncovered (MD2U)**: This approach uses heuristics to prioritize states that are "likely" to cover new code soon, based using on factors including minimum distance to an uncovered instruction and query cost to produce a weight for each state. MD2U may not perform well for every program, like any heuristic strategy.

Table 3.2: Subject programs used in the evaluation

Subject	NBNC	LLVM instructions	Size of test pool
Sed	13,359	48,684	370
Space	6,200	24,843	500
Grep	10,056	43,325	469
Gzip	5,677	21,307	214
Vim	107,926	339,292	1,950
YAFFS2	10,357	30,319	500

3.5 Subject Programs

Programs: Table 3.2 summarizes the programs used in our experiments, showing the name and number of NBNC (non-blank, non-comment) lines of code (measured by CLOC [ClocWebPage,]) for each program. We used a total of six C programs, five of which are taken from the SIR repository [Do et al., 2005]. The other subject, `YAFFS2` [YAFFS2WebPage,], is a widely used open-source flash file system for embedded devices (the default image format for earlier versions of Android). In addition to CLOC, we also show the number of LLVM instructions for each subject program.

Tests Cases: Table 3.2 also shows the total number of test cases in the test pools from which various test suites are composed. We use the SIR pools for the programs from SIR. For `YAFFS2`, we generated random tests using feedback [Groce et al., 2007]. We did not use swarm testing [Groce et al., 2012b] (our usual approach) due to concern swarming would make tests too easy to reduce and diversify, and thus unfairly advantage our methods. For `YAFFS2`, we first generated 500 random API-call sequences of length 50, then changed all input parameters into symbolic ones. In our initial experiments, we found that `KLEE` can finish the seeding stage only for 48 or 55 out of these 500 test cases (for 10 or 20 minutes search respectively). This difficulty means that `KLEE` could not perform its regular search for most of the test cases. We limited the maximum seeding duration for `YAFFS2` to half of the total symbolic execution time. For the remainder of the programs, seeding time was unlimited (up to the total symbolic execution budget).

3.6 Experimental Setup

The program inputs for `sed`, `grep`, `space`, and `gzip` are command line parameters. Each parameter is either a command line option or an input file. We use `zesti` [Marinescu and Cadar, 2012] for our symbolic execution, which can automatically accept command line parameters as symbolic inputs. The symbolic input size is decided by the concrete input size. Take `grep` as an example:

```
klee --zest grep.bc ‘patternstring’ file.inp
```

Given this command, `zesti` will use one symbolic string and a symbolic file as program inputs, and use `‘patternstring’` and `file.inp`’s contents as search seeds. With these original inputs, we use `KLEE` to do symbolic search around the seeds with the four search strategies noted above. For `vim` seed inputs are 1,950 script files taken from the SIR repository, called as `vim -s scriptfile`. We have `zesti` take the concrete script files as search seeds.

3.7 Experimental Results: RQ1 (Test Case Reduction)

3.7.1 Test Case Reduction Rate

The first question is whether test cases can be reduced significantly while preserving coverage; if there is no redundancy with respect to coverage in test paths, it is unlikely the answer to **RQ1** will be affirmative. Previous work on such reduction used only randomly generated tests, which are known to be highly redundant, but SIR subjects include many test cases produced by other methods, including by human authors. Our subject programs also have various test case structures, including method call sequences, pattern search strings, text files, and other structures, while previous work was essentially based on API call sequences. We use path length as the measurement of the size of each test case. We used a 20% of exploration time timeout (2 minutes for 10 minute budget, 4 minutes for 20) for all subjects. We define

$$\text{reduction rate} = \frac{PL_u - PL_r}{PL_u}. \tag{3.1}$$

where PL_u and PL_r denote the path lengths of a test case before and after reduction, respectively.

Table 3.3 shows path lengths before reduction, after statement-, branch-, and AIMP-based reduction and reduction rates for all the subject programs. The column “Original” for each subject program are the mean path lengths over all its test cases. The columns “Length” and “Rate” under “Statement”, “Branch”, and

Table 3.3: Test case reduction rates (%) for statement-, branch-, and AIMP-based reduction (mean values over all the test cases)

Subject	Original	10m*20%=2m reduction budget						20m*20%=4m reduction budget					
		Statement		Branch		AIMP		Statement		Branch		AIMP	
		Length	Rate	Length	Rate	Length	Rate	Length	Rate	Length	Rate	Length	Rate
Sed	88,657	7,034	81.39	8,036	80.40	88,657	8.75	6,370	85.19	6,894	83.34	88,483	9.40
Space	6,364	5,739	15.75	5,731	15.60	5,866	11.96	5,601	18.19	5,564	18.01	5,803	13.08
Grep	103,097	25,853	72.61	29,282	69.85	103,033	0.76	25,841	73.79	28,878	72.30	100,851	3.13
Gzip	752,098	320,285	50.67	352,334	32.78	488,332	22.58	239,690	57.67	368,484	31.46	488,324	25.01
Vim	189,165	182,191	3.01	182,191	2.99	189,093	0.05	182,191	3.02	182,191	2.99	189,092	0.05
YAFFS2	53,139	41,652	20.50	41,707	20.68	41,707	20.68	40,520	23.76	40,851	22.62	40,954	22.37

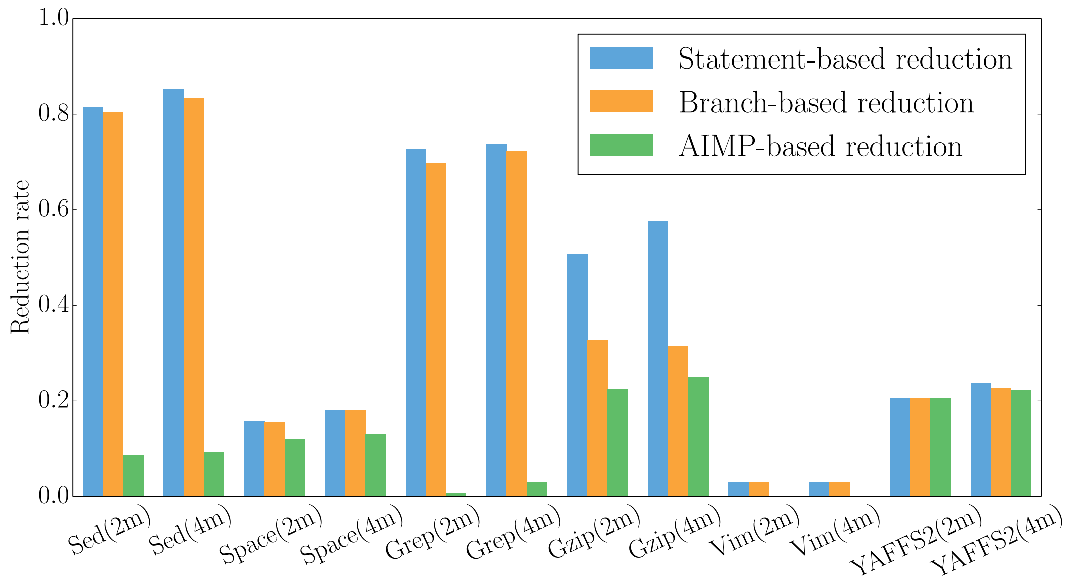


Figure 3.2: Reduction rates by subject, reduction budget, and reduction type.

“AIMP” are the path lengths and reduction rates of the test cases after applying various reductions on those original test cases for a given reduction time budget. All the values are the mean values over all the test cases of each subject program.

Figure 3.2 compares the reduction rates of statement-, branch-, and AIMP-based reduction strategies. Each bar group is for one subject program and one reduction time budget (i.e., 2 minutes or 4 minutes). As can be seen from Figure 3.2, (1) tests can be considerably reduced clearly and (2) statement-based test case reduction outperforms other strategies in terms of reduction rates. Does this reduction translate into more efficient symbolic exploration?

3.7.2 Test Case Reduction Effectiveness

Table 3.4 shows that reduced test cases can in fact improve symbolic execution efficiency compared to the unreduced test cases. For each input test case, we asked `KLEE` to perform symbolic search for 10 and 20 minutes. From the test case pool, we randomly chose 100 test cases as a test suite 150 times, for each subject. For each such suite, C_o is the total branch coverage of the original 100 test cases. The set of all branches explored during symbolic execution of the 100 original unreduced test cases is C_u, and $C_u \setminus C_o$ denotes the set difference of C_u and C_o — the new branches discovered during symbolic exploration. The same approach is applied to the reduced test cases and the branches hit are referred to as C_r. To see if reduced test cases improve symbolic execution, we simply compare $|C_u \setminus C_o|$ and $|C_r \setminus C_o|$.

In Table 3.4, the column "Original" for each search strategy are the sizes of $C_u \setminus C_o$. The columns "BranchΔ" under "Statement-reduced tests", "Branch-reduced tests", and "AIMP-reduced tests" are the sizes of $C_r \setminus C_o$ for each different test case reduction method based on various coverage criteria. The columns "R(%)" denote the coverage improvement percentage comparing to unreduced tests, which

Table 3.4: Branch coverage increment (BranchΔ, mean values over 150 test suites) on 100 random tests with reduction rate (R), Cohen's d effect sizes (es) between reduced and unreduced tests, and rounded Wilcoxon p-values

Subject	Time	Search	Original	Statement-reduced tests				Branch-reduced tests				AIMP-reduced tests			
			BranchΔ	BranchΔ	R(%)	*es*	*p*	BranchΔ	R(%)	*es*	*p*	BranchΔ	R(%)	*es*	*p*
Sed	10m	DFS	154.33	223.88	45.06	1.99	0.00	216.93	40.56	1.69	0.00	195.03	26.37	0.96	0.00
		RP	266.21	274.73	3.20	0.53	0.00	274.94	3.28	0.55	0.00	257.41	-3.31	-0.56	0.00
		RS	191.86	247.98	29.25	3.24	0.00	250.93	30.79	3.09	0.00	221.41	15.40	1.58	0.00
		MD2U	190.79	237.35	24.40	2.28	0.00	245.85	28.86	2.78	0.00	218.67	14.61	1.36	0.00
	20m	DFS	168.65	229.82	36.27	1.86	0.00	223.15	32.31	1.48	0.00	197.30	16.99	0.71	0.00
		RP	286.31	288.69	0.83	0.13	0.03	287.43	0.39	0.06	**0.25**	282.88	-1.20	-0.16	0.00
		RS	220.53	258.84	17.37	1.57	0.00	259.20	17.54	1.59	0.00	239.90	8.79	0.71	0.00
		MD2U	229.25	256.71	11.98	1.00	0.00	255.87	11.61	0.96	0.00	241.01	5.13	0.40	0.00
Space	10m	DFS	2.18	10.79	394.80	3.00	0.00	3.85	76.45	1.30	0.00	3.85	76.45	1.30	0.00
		RP	4.14	4.28	3.38	0.08	0.00	4.14	0.00	0.00	0.00	4.14	0.00	0.00	0.00
		RS	2.71	6.05	123.10	1.88	0.00	3.99	47.17	0.82	0.00	3.89	43.24	0.76	0.00
		MD2U	2.71	6.95	156.02	1.86	0.00	3.89	43.24	0.77	0.00	3.89	43.24	0.76	0.00
	20m	DFS	2.15	12.03	458.51	3.39	0.00	3.81	76.78	1.33	0.00	3.77	74.92	1.31	0.00
		RP	3.91	4.67	19.25	0.42	0.00	3.91	0.00	0.00	0.00	3.91	0.00	0.00	0.00
		RS	2.53	6.43	154.62	1.95	0.00	3.71	46.97	0.79	0.00	3.71	46.97	0.79	0.00
		MD2U	2.53	7.57	199.74	2.07	0.00	3.71	46.97	0.79	0.00	3.68	45.65	0.78	0.00
Grep	10m	DFS	17.81	167.50	840.31	4.76	0.00	160.19	799.25	4.07	0.00	17.40	-2.32	-0.06	**0.14**
		RP	116.59	135.25	16.00	1.52	0.00	133.39	14.40	1.36	0.00	116.02	-0.49	-0.06	0.00
		RS	115.65	185.12	60.06	3.26	0.00	180.52	56.09	2.99	0.00	114.67	-0.85	-0.14	0.00
		MD2U	111.89	190.49	70.25	3.55	0.00	181.77	62.46	3.12	0.00	112.61	0.65	0.11	0.00
	20m	DFS	37.07	180.05	385.67	4.83	0.00	173.55	368.14	4.61	0.00	17.63	-52.45	-1.28	0.00
		RP	125.00	164.17	31.34	1.58	0.00	163.21	30.57	1.53	0.00	120.26	-3.79	-0.55	0.00
		RS	116.86	198.37	69.75	3.69	0.00	190.82	63.29	3.61	0.00	116.48	-0.33	-0.05	**0.21**
		MD2U	112.85	203.41	80.24	3.83	0.00	194.18	72.06	3.25	0.00	113.38	0.47	0.08	0.00
Gzip	10m	DFS	157.71	122.89	-22.08	-0.85	0.00	124.17	-21.27	-0.81	0.00	148.19	-6.04	-0.20	0.00
		RP	225.99	228.42	1.07	0.10	0.00	229.36	1.49	0.14	0.00	223.57	-1.07	-0.09	0.04
		RS	167.24	179.90	7.57	0.58	0.00	157.50	-5.82	-0.43	0.00	138.17	-17.38	-1.62	0.00
		MD2U	185.31	197.15	6.39	0.42	0.00	198.91	7.34	0.43	0.00	161.11	-13.06	-0.95	0.00
	20m	DFS	162.81	115.05	-29.34	-1.16	0.00	168.62	3.57	0.11	0.00	165.96	1.94	0.06	0.01
		RP	237.46	232.99	-1.88	-0.21	0.00	241.67	1.77	0.20	0.00	243.53	2.56	0.30	0.00
		RS	176.51	188.73	6.93	0.51	0.00	188.64	6.87	0.52	0.00	185.91	5.33	0.39	0.00
		MD2U	212.96	224.37	5.36	0.38	0.00	236.05	10.84	0.78	0.00	228.30	7.20	0.53	0.00
Vim	10m	DFS	340.59	342.89	0.67	0.02	**0.86**	337.06	-1.04	-0.03	**0.31**	330.11	-3.08	-0.07	**0.22**
		RP	116.11	128.63	10.78	0.34	0.00	121.73	4.84	0.17	0.00	115.97	-0.12	-0.01	0.00
		RS	274.29	292.09	6.49	0.34	0.00	277.55	1.19	0.07	**0.52**	262.81	-4.19	-0.26	0.00
		MD2U	322.78	340.89	5.61	0.33	0.00	326.05	1.01	0.07	**0.62**	317.95	-1.50	-0.10	0.00
	20m	DFS	610.50	668.41	9.49	0.31	0.00	666.46	9.17	0.32	0.00	667.32	9.31	0.31	0.00
		RP	118.40	145.51	22.90	0.53	0.00	127.86	7.99	0.27	0.00	123.73	4.50	0.16	0.00
		RS	335.52	355.18	5.86	0.38	0.00	340.39	1.45	0.12	**0.11**	333.11	-0.72	-0.06	**0.11**
		MD2U	398.32	427.64	7.36	0.50	0.00	416.79	4.64	0.37	0.00	407.25	2.24	0.18	0.00
YAFFS2	10m	DFS	72.68	73.85	1.61	0.07	**0.09**	77.41	6.50	0.29	0.00	79.38	9.22	0.42	0.00
		RP	88.49	90.15	1.88	0.12	0.00	90.07	1.79	0.12	0.00	90.21	1.94	0.13	0.00
		RS	87.16	93.40	7.16	0.45	0.00	93.52	7.30	0.46	0.00	92.97	6.66	0.42	0.00
		MD2U	87.95	96.82	10.09	0.65	0.00	96.46	9.68	0.62	0.00	96.65	9.89	0.63	0.00
	20m	DFS	77.98	79.75	2.27	0.10	0.01	87.55	12.27	0.53	0.00	86.45	10.86	0.48	0.00
		RP	89.85	91.73	2.09	0.12	0.00	93.01	3.51	0.20	0.00	93.54	4.10	0.23	0.00
		RS	87.60	96.21	9.82	0.51	0.00	97.20	10.96	0.57	0.00	94.35	7.70	0.41	0.00
		MD2U	89.01	100.09	12.45	0.68	0.00	101.35	13.86	0.75	0.00	97.71	9.77	0.52	0.00
The best results			1	30	-	-	12	-	-	4	-	-			

is defined by

$$\text{coverage improvement percentage} = \frac{|C_r \setminus C_o| - |C_u \setminus C_o|}{|C_u \setminus C_o|} \times 100\%. \tag{3.2}$$

In all experiments, the value in the table is the mean of 150 runs, and p-values are based on a Wilcox rank sum test. The few cases where p is not well below 0.00 (rounded) are shown in bold. The best results are highlighted. Additionally, to show if test case reduction has a strong effect on improving seeded symbolic execution, Cohen's d effect sizes are also shown in Table 3.4 (the columns es).

It is clear that in general, for 10 minute and 20 minute test budgets, spending some portion of the budget reducing test cases usually results in markedly improved incremental branch coverage. The best timeout for reduction is unclear, but the basic validity of using test case reduction to improve symbolic exploration is difficult to ignore. In many cases the difference in median incremental covered branches is also quite large in absolute or relative terms, 50+ branches or from 40-100% more branches, in addition to being statistically significant.

Figure 3.3 compares the relative performance of reduced tests and unreduced tests. Especially, Figure 3.3 shows that how many times reduced tests are better than the unreduced tests, how many times reduced tests are worse than the unreduced tests out of all the 48 different experiment configurations, which are all the combinations of six subject programs, two different reduction budgets, and four search strategies. The left-hand plot of Figure 3.3 shows that reduced tests with statement-, branch-, and AIMP-based reductions win the unreduced tests 43, 39,

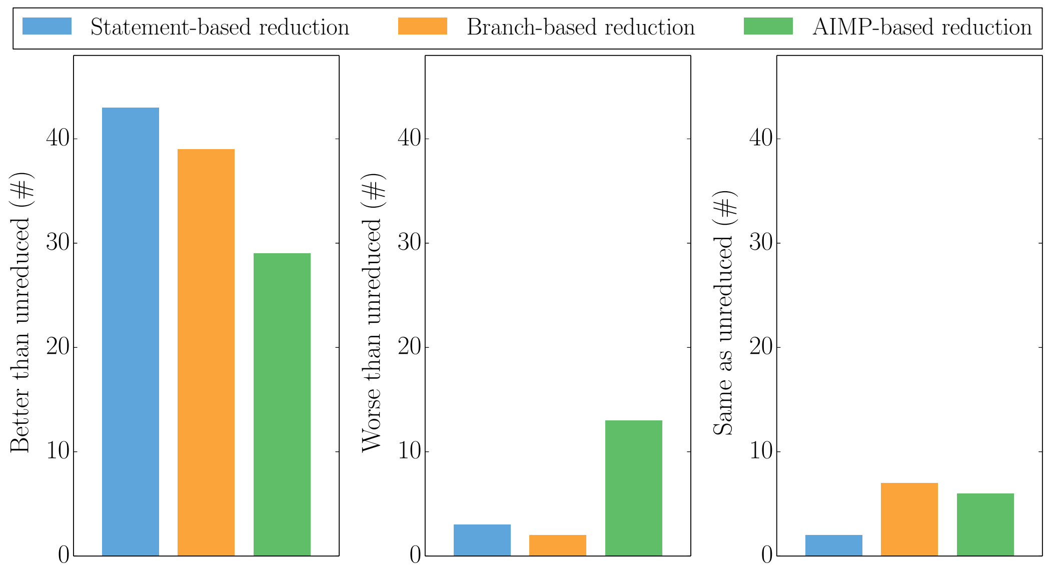


Figure 3.3: Performance comparison of statement-, branch-, and AIMP-based reduction tests and the unreduced tests.

and 29 times out of a total of 48 comparisons, respectively. It demonstrates that reduced tests are generally better than the unreduced tests as symbolic execution seeds.

In addition to the comparison of reduced tests and the unreduced tests, we also examine which strategy is the best out of all the three reduction strategies. Figure 3.4 shows the relative performance of the reduced tests with statement-, branch-, and AIMP-based reductions in terms of three measures: the number of best results, median branch increment rates, and mean branch increment rates, as defined by (3.2). Figure 3.4 clearly shows that the statement-based test case reduction performs the best out of all reduction methods inspected in our experiment.

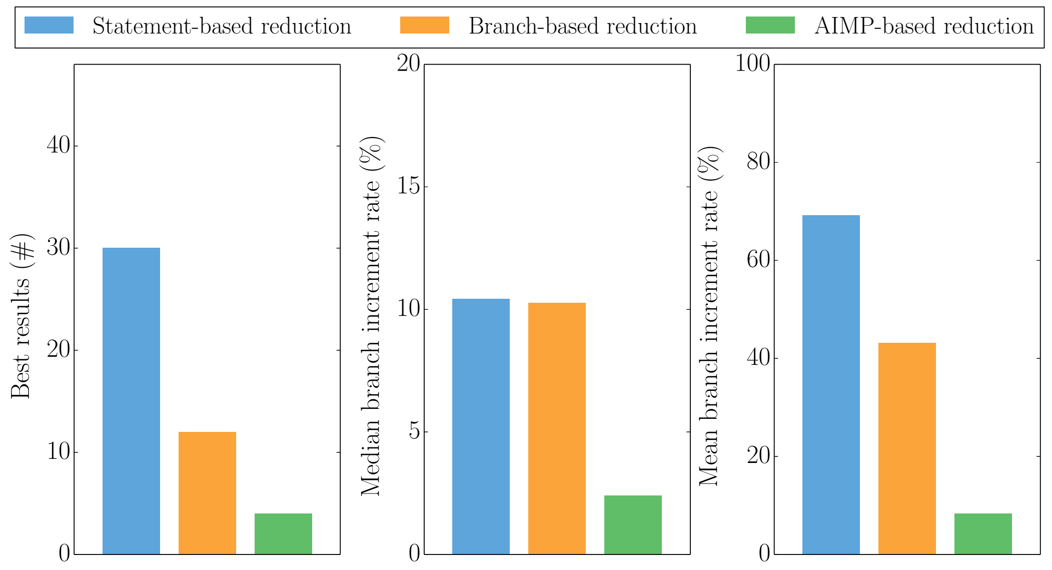


Figure 3.4: Effectiveness comparison of statement-, branch-, and AIMP-based reduction strategies.

Figure 3.5 shows the improvements (with random ordering of test cases) for `sed`.

3.7.3 Test Case Reduction Cost

Table 3.5 shows the reduction cost for each subject program and each reduction strategy in a given two or four minutes time budget.

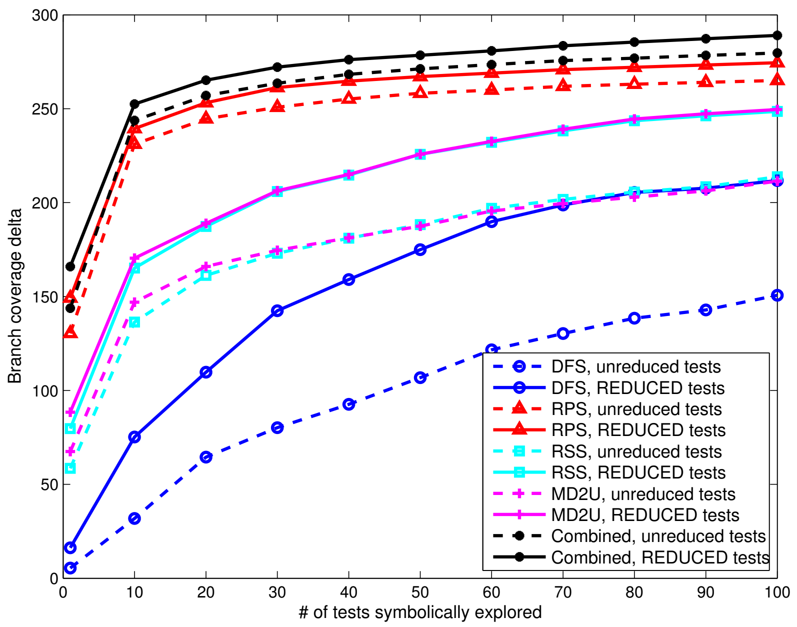


Figure 3.5: Additional branch coverage on `sed` program during seeded symbolic execution with unreduced tests and reduced tests (test cases are in random order).

Table 3.5: Test case reduction cost for statement-, branch-, and AIMP-based reduction (seconds for 100 test cases)

Subject	10m*20%=2m reduction budget			20m*20%=4m reduction budget		
	Statement	Branch	AIMP	Statement	Branch	AIMP
Sed	85.61	84.35	105.76	86.49	91.98	199.63
Space	94.21	87.90	65.14	116.46	106.62	74.09
Grep	94.45	100.38	118.81	156.18	182.01	237.05
Gzip	105.42	108.11	107.81	202.28	212.75	212.69
Vim	39.81	45.25	8.82	40.17	45.61	9.17
YAFFS2	37.28	35.42	15.20	32.99	30.78	14.86

3.8 Evaluation Metric (Comparing Non-adequate Test Suites using Coverage Criteria)

So far, we have used incremental gain in *branch coverage* over an initial suite as our basic measure in our discussion of effectiveness of test case reduction, we now show that branch coverage is a better evaluation metric for measuring test case effectiveness [Gligoric et al., 2013, Gligoric et al.,] by turning to a discussion of our coverage criteria comparison work (joint work with researchers at UIUC).

Our experimental evaluation suggests that due to high effectiveness and low overhead, researchers should use branch coverage to compare suites whenever possible. A variation of acyclic intra-method path coverage performed best of all non-branch coverage criteria, and has desirable simplicity, ease of implementation, and reasonable overhead. There is considerable evidence that when branch coverage for two test suites differs by a significant amount, the suite with higher coverage is usually better at fault detection as well [Gligoric et al., 2013]. This approach combines both effectiveness (since the absolute coverage numbers can be compared given the fixed computation effort) and efficiency (in that results are for a given time), and matches the practical needs of testing, where budgets are not infinite and finding bugs sooner rather than later is critical. For prioritization, we follow previous practice and modify this basic evaluation measure to examine the average incremental coverage, thus capturing the climb of the discovery curve for new branches produced by a prioritization strategy (since once all tests have been completed, all prioritizations have the same final coverage results).

To evaluate the effectiveness of our proposed test case reduction technique, a fundamental problem is how to compare the coverage acquired by symbolically exploring the original test suite $(T_1, T_2, \ldots, T_n)$ to the coverage acquired by symbolically exploring the reduced test suite $(T'_1, T'_2, \ldots, T'_n)$ (with reduced size and execution time but with equivalent code coverage to the original suite). Our major research question for this work is therefore: for the common case of *non-adequate* test suites: given criteria C and C', which one is better to use to compare test suites? Namely, if suites $T_1, T_2 \ldots T_n$ have coverage values $c_1, c_2 \ldots c_n$ for C and $c'_1, c'_2 \ldots c'_n$ for C', is it better to compare suites based on $c_1, c_2 \ldots c_n$ or based on $c'_1, c'_2 \ldots c'_n$? We evaluate a large set of plausible criteria, including statement and branch coverage, as well as stronger criteria used in recent studies. Two criteria perform best: branch coverage and an acyclic intra-method path coverage.

3.8.1 Coverage Criteria

Intra-Method Path Coverages: We first describe two forms of path-based coverage used in my proposal. Whole program path coverage was proposed over 20 years ago to measure how many different paths tests execute from the beginning to the end of a program. Even for loop-free programs, whole program paths result in a number of test requirements exponential in the number of branches in a program, so more recent work [Chilimbi et al., 2009a, Wang and Roychoudhury, 2005] used more scalable intra-method paths (IMP), where each path is for a single method execution only (similar to Godefroid's notion of compositional path coverage [Godefroid, 2007]). An intra-method path starts at the beginning of a method, includes the IDs of the executed basic blocks, does not include nested method invocations, and ends when the execution returns from the method. IMP subsumes branch coverage (and thus statement coverage) but faces the problem that loops introduce an unbounded number of test requirements.

Another variant of path coverage, acyclic intra-method paths (AIMP), retains subsumption of branch coverage but bounds the total number of requirements by considering only acyclic paths in intra-method control-flow graphs [Ball and Larus, 1996]. The number of AIMP paths is therefore bounded by $m \cdot 2^k$ where m is the number of methods in a program and k is the maximum number of branches in a single method. The paths to be covered have no repeated IDs, i.e., AIMP modifies IMP such that a repeated basic block ID ends the current path and starts a new path.

Predicate-Complete Test Coverage (PCT): Predicate-complete test coverage (PCT) [Ball, 2005, Ball, 2004] was introduced by Ball as a finite-state alternative to path coverage, inspired by predicate abstraction in model checking [Ball and Rajamani, 2001]. Like path coverage, PCT subsumes both branch and statement coverage, but unlike some versions of path coverage, PCT does not face the problem that loops introduce an unbounded number of test requirements. PCT is incomparable to (i.e., neither subsumes nor is subsumed by) path coverages such as IMP and AIMP, even for loop-free programs. Several research studies [Visser et al., 2006, Pacheco et al., 2007], compared test suites using PCT, but with code hand-instrumented for measuring PCT; we refer to this version as PCT(MS).

PCT defines coverage using Boolean predicates extracted from the program source, in particular from branch conditions, implicit run-time checks, and assertions. These predicates are evaluated at many program points, e.g., at all statements or all starts of basic blocks, potentially far from where the predicates appear in the program source. The test requirements for PCT are to cover all (feasible) combinations of predicate values at all the points. In the limit, for n predicates at p points, there are $p \cdot 2^n$ combinations (many often infeasible, and not every point has all n predicates). The PCT coverage for a test suite is measured as the number of combinations of predicate values obtained during the execution of the test suite.

3.8.2 Experimental Methodology

Testing literature does not have one agreed upon methodology for comparing test coverage criteria, so we motivate and describe the methodology we use. Coverage values are used to evaluate suites, typically as predictors for finding real faults. Intuitively, if a good criterion deems one suite better than another suite, then we expect the better suite to find, *on average*, more faults. While older studies on comparing coverage criteria used (a small number of) real faults [Frankl and Weiss, 1993, Hutchins et al., 1994, Frankl and Iakounenko, 1998], more recent studies use (a large number of) systematically seeded faults [Cai and Lyu, 2005, Andrews et al., 2006, Namin and Andrews, 2009]. Several studies establish that mutation scores, i.e., how many mutants a suite kills, are effective for predicting detection

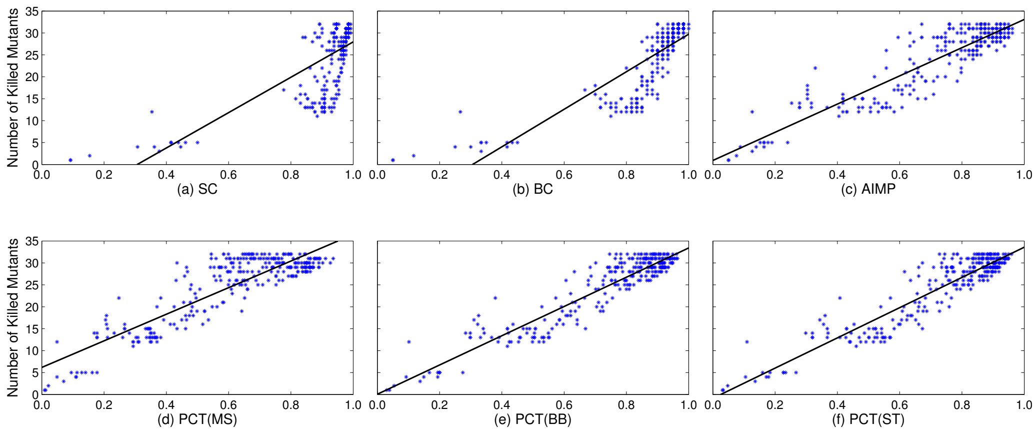


Figure 3.6: Coverage criteria values and mutation scores correlation for BinomialHeap.

of real faults [Andrews et al., 2006, Andrews et al., 2005, Zhang et al., 2010]. However, for large programs or suites, performing mutation testing is very costly.

Specifically we examine the ability of coverage values to *predict (the relative ordering or absolute values of) mutation scores*. We compare two traditional criteria (statement and branch) and two sets of recently used criteria based on paths (IMP and AIMP) and predicates (various PCT coverages). To visualize this concept, Figure 3.6 shows six plots (for six coverage criteria) that relate coverage values and mutation scores for `BinomialHeap`. Each point represents one of 300 suites (selected as explained in Section 3.8.2). The X-axis shows coverage, normalized between 0.0 and 1.0, and the Y-axis shows mutation score[2]. It is clear in all six plots that if a suite A has a higher coverage than a suite B, then the suite A also

[2]The mutation score is not normalized, but dividing by a constant never changes values for our two correlations.

Table 3.6: Subject programs used in the evaluation

Subject	NBNC	Size of test pool	Mutation	BC	PCT						
			killed/mutants	branches	predicates			points		states	
				exe/static	MS	BB,ST	MS	BB	ST	BB	ST
language: Java											
AvlTree	344	11,041	51/335	20/104	4	87	104	189	167	153	156
BinomialHeap	264	8,423	37/205	60/60	9	49	60	109	150	335	419
BinTree	100	13,825	16/55	32/32	7	26	32	51	54	224	228
FibHeap	264	12,842	38/186	44/60	14	67	56	98	160	132	228
FibonacciHeap	397	4,478	74/295	45/66	14	58	62	100	156	139	252
HeapArray	98	4,064	61/122	30/32	3	19	32	50	60	205	235
IntAVLTreeMap	213	17,072	38/199	47/56	4	52	56	100	112	242	277
IntRedBlackTree	296	20,419	210/279	83/90	6	76	90	149	177	479	534
JFreeChart$^{1.0.14}$	72,490	2,217	14,932/45,409	12,083/17,866	-	13,536	-	32,907	42,372	34,899	45,406
JodaTime$^{2.0.0}$	27,472	3,828	16,478/24,956	6,364/7,357	-	2,913	-	9,476	10,570	16,673	18,723
LinkedList	245	1,307	5/167	8/36	4	40	36	78	107	34	53
NodeCachLList	234	1,776	16/159	14/34	4	34	34	68	103	82	129
SinglyLList	98	1,762	10/57	20/26	3	22	26	39	55	67	95
TreeMap	449	14,076	106/463	101/147	6	102	119	239	280	749	837
TreeSet	323	17,400	82/360	83/93	6	69	94	150	183	462	521
language: C											
Printtokens	479	4,130	442/536	63/66	-	70	-	73	265	292	1,050
Printtokens2	401	4,115	343/343	159/162	-	108	-	133	282	908	2,339
Replace	512	5,542	530/613	169/180	-	190	-	177	345	1,041	1,968
Schedule	292	2,650	125/140	55/58	-	52	-	64	176	545	1,554
Schedule2	297	2,710	251/300	83/88	-	54	-	75	190	705	1,751
SglibRbtree	476	5,000	193/443	238/378	-	426	-	350	720	3,794	9,841
Space	6,200	1,350	753/1,142	1,014/1,190	-	1,552	-	884	3,927	5,708	25,100
SQLite$^{3.7.13}$	81,934	117,240	19,294/52,367	15,676/17,304	-	21,285	-	13,786	37,313	529,272	1,432,590
Totinfo	340	917	511/511	79/88	-	55	-	76	238	977	3,109
Tcas	135	1,608	311/311	61/66	-	45	-	72	133	1,311	2,603
YAFFS2	11,760	5,000	4,186/10,674	1,852/4,274	-	4,149	-	3,520	8,273	27,501	755,42

likely has a higher mutation score than the suite B. The purpose of our statistical evaluation is to quantify the degree to which this relationship holds for each criterion, and thus to compare criteria. We apply two different standard statistical tools, Kendall τ_b rank correlation and the R^2 coefficient of determination for linear regression, discussed in detail in Section 3.8.2. Intuitively, Kendall τ_b measures how well coverage values predict the relative ordering of mutation scores, and R^2 correlates coverage values with mutation scores using a linear model.

Experimental Subjects **Programs:** Table 3.6 summarizes the programs used in our experiments, showing the name and number of NBNC (non-blank, non-comment) lines of code (measured by CLOC [ClocWebPage,]) for each program.

We used a total of 26 programs, 15 Java programs and 11 C programs. All Java programs but two are implementations of data structures that have been used in numerous previous studies, primarily on comparing different testing techniques [Visser et al., 2006, Galeotti et al., 2010, Sharma et al., 2010, Sharma et al., 2011, Groce, 2011, Groce et al., 2012a]. `JFreeChart` [JFreeChartWebPage,] is an open-source library for both interactive and non-interactive manipulation of charts; `JodaTime` [JodaTimeWebPage,] is an open-source library for manipulating date and time. For C, seven programs are from the Siemens suite from the SIR repository [Do et al., 2005, Hutchins et al., 1994], `Space` [Vokolos and Frankl, 1998, Do et al., 2005] is a bigger program from the same repository, `SglibRbtree` [Vittek et al., 2006] is the red-black tree implementation from the Sglib library, `YAFFS2` [YAFFS2WebPage,] is a widely used open-source flash file system for embedded devices (the default image format for older versions of Android), and `SQLite` [SQLiteWebPage,] is a widely deployed database engine.

Tests: Table 3.6 also shows the total number of tests in the test pools from which various test suites are composed. For Java data structures, we use test pools *automatically generated* in previous studies [Sharma et al., 2011, Groce, 2011, Groce et al., 2012a] using three test-generation techniques: random (*Random*), shape abstraction (*ShapeAbs*) [Visser et al., 2006], and adaptation-based programming (*ABP*) [Groce, 2011, Groce et al., 2012a]. Table 3.6 shows the total number of tests generated by all three techniques. For `JFreeChart` and `JodaTime`, we use the large, publicly available pool of *manually written* JUnit tests. For C programs, we use the Siemens/SIR test pools for the programs from SIR. For

`SglibRbtree` and `YAFFS2`, we generated random tests (feedback-directed [Groce et al., 2007] for `YAFFS2`). For `SQLite` we use manually written tests available from the `SQLite` repository [SQLiteWebPage,].

Mutants: Table 3.6 also tabulates for each program the number of mutants created and the total number of mutants killed by the entire test pool (while different suites selected from the pool kill different number of mutants). The percentage of killed mutants is low because we mutated *all* the methods in the code but automatically generated tests execute only *some* core methods for the smaller subjects [Sharma et al., 2011]. Low absolute mutation scores are suitable for our purpose of examining non-adequate suites, the typical case for suites for large programs. Non-adequate suites will seldom attain extremely high mutation scores. Additionally, we did not investigate which mutants are equivalent, as this does not affect our analysis (because compensating for equivalent mutants is equivalent to dividing mutation score by a constant, which does not affect τ_b or R^2).

For Java programs, we used Javalanche [Schuler and Zeller, 2009] to create mutants. Because the number of mutants may be lower than one would expect, it should be noted that Javalanche uses selective mutation [Offutt et al., 1993] to reduce the cost of mutation testing. Selective mutation applies only a subset of mutation operators that are empirically shown to approximate the results that would be achieved if all operators were used. In particular, Javalanche uses only the following operators: replace numerical constants, negate jump condition, replace arithmetic operator, replace method calls, and remove method calls. Still, Javalanche created over 45K and 24K mutants for `JFreeChart` and `JodaTime`, respectively.

For C programs, we created mutants using the tool implemented by Andrews et al. [Andrews et al., 2005], which produces mutants based on a set of operators selected through an empirical study on selective mutation [Siami Namin et al., 2008].

Branch Coverage Information: The BC column in Table 3.6 provides information for branch coverage: "static" shows the number of branches in the code, and "exe" shows the number of branches executed by at least one test.

PCT Information: Table 3.6 also provides PCT-specific information, i.e., the total number of predicates used in the instrumentation, the number of program points at which these predicates are inserted, and the number of executed states (i.e., encountered states during the execution) by all tests. *MS* ("Manually Selected") denotes a set of predicates and points that were first selected for four data structures by Visser et al. [Visser et al., 2006] and then similarly selected for the remaining structures by Sharma et al. [Sharma et al., 2011]. These programs, manually instrumented for PCT coverage are publicly available [CoverageWebPage,]. *BB* ("Basic Blocks") and *ST* ("Statements") denote the results of automatic instrumentation by our PCT coverage tools. Recall that our tools select (almost) all predicates from the code and insert each predicate at all program points where the variables from the predicate are in scope.

Test Suites and Metrics We used two methods for selecting test suites, to see if results are robust in the face of different suite compositions. The bounds in our methods (e.g., 100 suites) were chosen before experimentation, to limit computation time while providing sufficiently many samples for statistical analysis, or were chosen to match previous papers.

Coverage Method: For each program, to ensure test suites of varying coverage and size, we created suites by first uniformly selecting a coverage level between 1% and 100% and then randomly selecting tests from the test pool until they reached the selected level of *PCT(BB)* coverage. We picked PCT(BB) as one strong criterion but could have used any other criterion. For the Java data structures we selected 100 suites from the pool for each of the three techniques (Random, ShapeAbs, and ABP), giving a total of 300 test suites. For JFreeChart and JodaTime we used 100 suites. For all C programs except `SQLite` we used 300 suites. For `SQLite` each "test" in the pool is essentially a large suite of tests that must run together, so we treated each of the 592 "tests" as a suite.

Size Method: We also followed another suite selection method, used in previous studies of coverage criteria [Namin and Andrews, 2009, Hassan and Andrews, 2013]. For each program, we created 100 random suites for each size (number of tests) between 1 and 50, which gives 5,000 suites per program, but with less varied coverage than Coverage Method. Also, this method creates many suites that are near adequate in at least one criterion and does not include suites based on different test generation techniques, which most closely reflect the intended purposes of our evaluation. `SQLite` was handled as for the Coverage Method.

We collected several metrics for the selected test suites.

Coverage Criteria: For each suite, we measured several coverage values. For

Java suites, we measured statement coverage (SC), branch coverage (BC), IMP, AIMP, PCT(MS) (except for `JFreeChart` and `JodaTime` programs), PCT(BB), PCT(ST), and mutation score. For C suites, we measured basic block coverage[3], BC, IMP, AIMP, PCT(BB), PCT(ST), and mutation score.

Runtime Overhead: We separately ran each coverage measurement so that we could measure the runtime overhead. We performed all Java experiments on a machine with a 4-core Intel Core i7 2.70GHz processor and 4GB RAM, running Linux version 3.2.0 and Java OpenJDK 64-Bit Server VM, version 1.7.0_04. We performed all C experiments on a machine with a 4-core Intel Xeon E5400 2.83GHz processor and 4GB RAM, running Linux version 2.6.32.

Correlation Analysis To evaluate the relationship between coverages and mutation scores, we computed two correlation measures.

Kendall τ_b: One core question of this work is whether (and which) coverage criteria can be used to effectively predict the *rank order* of suites' mutation scores. This is the primary use of coverage in recent studies; authors have tended to focus on claiming that some testing technique is "better", and relatively small differences in coverage values have been used to justify a claim of "better" [Visser et al., 2006, Groce et al., 2012a]. The most robust and usefully interpreted statistical measure for this question is the *Kendall τ rank correlation coefficient* [Kendall, 1938, Cliff, 1996].

Consider the coverage and mutation score data as a set of pairs (C, M), where

[3]We use slightly different criteria in Java (statements) and C (basic blocks), but the two have highly similar results.

C is the coverage value for a suite and M is the mutation score for that suite. Two pairs (C_1, M_1) and (C_2, M_2) are called *concordant* if the ordering of C_1 and C_2 matches the ordering of M_1 and M_2, i.e., $C_1 < C_2$ and $M_1 < M_2$ or $C_1 > C_2$ and $M_1 > M_2$. The pairs are called *discordant* if $C_1 < C_2$ and $M_1 > M_2$ or $C_1 > C_2$ and $M_1 < M_2$. Kendall's τ is the ratio of the difference between the number of concordant and discordant pairs and the total number of pairs. Kendall's original τ does not handle ties well, and thus was not suitable for our study, where BC and SC had 30% or more ties among suites for some subjects.

Kendall τ_b, used in our study, is a standard adaptation that adjusts for ties [Costner, 1965]. Using a non-parametric rank correlation allows us to avoid the difficult question of whether the relationship between any criterion and mutation score is linear; τ_b does *not* make any assumption about the underlying functional relationships. A final attractive feature of τ_b is that in the absence of ties, the value can be intuitively interpreted: $0.5 + |\frac{\tau}{2}|$ is the probability of correctly predicting the ordering of mutation scores using the ordering of coverage values [Costner, 1965]. Despite these desirable features of τ_b, our study is among the first to use τ_b in comparison of multiple criteria[4]. (A few studies [Namin and Andrews, 2009, Wong et al., 1994, Wong et al., 1995] only mention τ or use it for other purposes.) Values for τ_b range from -1.0 (which would indicate that the coverage values are always opposite of the mutation score) to 1.0 (which would indicate a perfect predictive

[4]A statistic similar to τ or τ_b is Spearman ρ; we prefer using τ_b to the more frequently used ρ due to interpretive ease and handling of ties and small sample sizes; the primary arguments for ρ are tradition and ease of calculation. In many cases, ρ and τ/τ_b are very similar in value.

power for a criterion); a τ_b of 0.0 indicates there is no relationship between the rank ordering by the criterion and rank ordering by mutation score.

$\mathbf{R^2}$: We also formed linear regression models for each criterion and obtained the R^2 *coefficient of determination* for the fits of those models to our data. It is well known that mutation scores do *not* depend linearly on coverage values [Cai and Lyu, 2005, Andrews et al., 2006, Namin and Andrews, 2009, Hassan and Andrews, 2013], but R^2 still gives an indication of correlation. Intuitively, it attempts to answer the question: if one suite has $X\%$ higher coverage value than another suite, does it have a $c * X\%$ higher mutation score? More precisely, it shows how well a linear model fits the actual data points, with 1.0 indicating a perfect fit and 0.0 indicating there is no relationship between the coverage and mutation score. Figure 3.6 shows lines that best fit the observed data.

3.8.3 Experimental Results

Rank Correlation Table 3.7 shows Kendall τ_b correlation values for all subjects and most criteria we examined, for both methods for test-suite selection. Each section highlights the best (darker/green) and worst (lighter/red) values. Values for PCT(MS) are missing where manual instrumentation was not used, and values for `SQLite` are repeated for both methods. The first key result is that most criteria had τ_b values over 0.5, often over 0.7, for most subjects. Using any of the criteria studied would correctly predict mutation score rankings for a large fraction of all suite pairs. Based on the standard Guilford scale [Guilford, 1956], we would say

Table 3.7: τ_b values for each subject program and criteria

Subject	Test-Suite Selection with Coverage Method							Test-Suite Selection with Size Method						
	SC	BC	IMP	AIMP	PCT			SC	BC	IMP	AIMP	PCT		
					MS	BB	ST					MS	BB	ST
language: Java														
JFreeChart	0.962	0.966	0.845	0.964		0.951	0.936	0.777	0.818	0.768	0.792	-	0.818	0.776
JodaTime	0.966	0.972	0.965	0.964	-	0.959	0.961	0.808	0.835	0.836	0.840	-	0.826	0.815
AvlTree	0.773	0.774	0.783	0.785	0.756	0.789	0.816	0.301	0.301	0.556	0.492	0.494	0.520	0.530
BinomialHeap	0.617	0.775	0.487	0.585	0.527	0.637	0.631	0.624	0.629	0.367	0.521	0.409	0.467	0.450
BinTree	0.132	0.220	0.341	0.351	0.491	0.417	0.510	0.271	0.510	0.587	0.696	0.564	0.658	0.656
FibHeap	0.759	0.807	0.278	0.395	0.509	0.634	0.515	0.566	0.637	0.475	0.641	0.676	0.622	0.617
FibonacciHeap	0.494	0.512	0.539	0.527	0.497	0.480	0.478	0.409	0.419	0.492	0.487	0.440	0.389	0.395
HeapArray	0.803	0.801	0.761	0.726	0.638	0.771	0.703	0.728	0.723	0.519	0.742	0.646	0.592	0.583
IntAVLTreeMap	0.777	0.770	0.788	0.815	0.786	0.728	0.762	0.684	0.682	0.633	0.677	0.665	0.621	0.617
IntRedBlackTree	0.710	0.741	0.712	0.751	0.697	0.748	0.737	0.671	0.726	0.757	0.803	0.755	0.778	0.758
LinkedList	0.756	0.746	0.713	0.716	0.746	0.705	0.701	0.353	0.849	0.132	0.154	0.849	0.157	0.155
NodeCachLList	0.737	0.724	0.527	0.670	0.693	0.531	0.495	0.404	0.355	0.343	0.393	0.404	0.377	0.380
SinglyLList	0.577	0.586	0.451	0.495	0.492	0.571	0.634	0.494	0.494	0.419	0.824	0.385	0.667	0.699
TreeMap	0.747	0.772	0.690	0.748	0.721	0.743	0.755	0.680	0.700	0.759	0.777	0.746	0.741	0.738
TreeSet	0.755	0.784	0.696	0.770	0.737	0.752	0.772	0.703	0.739	0.736	0.774	0.732	0.764	0.754
language: C														
Space	0.930	0.929	0.913	0.929	-	0.917	0.911	0.841	0.858	0.815	0.881	-	0.769	0.759
SQLite	0.904	0.904	0.837	0.909	-	0.906	0.904	0.904	0.904	0.837	0.909	-	0.906	0.904
YAFFS2	0.700	0.702	0.501	0.690	-	0.667	0.680	0.625	0.640	0.466	0.655	-	0.640	0.632
Printtokens	0.797	0.781	0.901	0.916	-	0.794	0.855	0.638	0.627	0.730	0.829	-	0.617	0.688
Printtokens2	0.848	0.845	0.826	0.831	-	0.839	0.844	0.690	0.695	0.548	0.605	-	0.655	0.679
Replace	0.701	0.699	0.691	0.697	-	0.677	0.681	0.498	0.504	0.566	0.539	-	0.485	0.493
Schedule	0.778	0.776	0.747	0.766	-	0.716	0.711	0.753	0.720	0.546	0.653	-	0.731	0.745
Schedule2	0.674	0.767	0.683	0.749	-	0.691	0.751	0.489	0.493	0.588	0.532	-	0.529	0.548
SglibRbtree	0.784	0.793	0.680	0.698	-	0.765	0.762	0.632	0.627	0.581	0.583	-	0.628	0.647
Totinfo	0.721	0.758	0.743	0.748	-	0.671	0.711	0.558	0.554	0.492	0.517	-	0.478	0.478
Tcas	0.779	0.773	0.739	0.739	-	0.766	0.749	0.721	0.720	0.703	0.703	-	0.747	0.729
Standard deviation	0.164	0.147	0.172	0.158	0.116	0.134	0.133	0.166	0.156	0.170	0.172	0.157	0.166	0.163
Geometric mean	0.705	0.735	0.660	0.709	0.627	0.711	0.717	0.583	0.624	0.555	0.624	0.577	0.593	0.595
Arithmetic mean	0.738	0.757	0.686	0.728	0.638	0.724	0.729	0.609	0.645	0.587	0.655	0.597	0.622	0.624
The best results	8	10	1	4	0	0	3	4	4	4	11	3	2	1
The worst results	2	1	11	1	3	4	5	7	1	12	1	1	4	3

that the mean values often showed high (>0.7) or nearly high (>0.6) correlation, and almost all correlations were at least moderate (>0.4). All values below 0.4, for criteria other than IMP and manual PCT, came from just 4 simple Java data-structure classes.

The second key result is that the absolute values and relative effectiveness of criteria vary with subject and test-suite selection method, in a few cases by a wide range. However, considering all subjects and both methods, it is clear that BC coverage performs very well, and AIMP seems to perform best of the non-branch criteria (though PCT(BB) and PCT(ST) have slightly higher means for

Coverage Method). For large subjects, coverage and mutation score ties were rare enough that the values in the table can be reasonably interpreted as indicating these criteria predict mutation score rank successfully 80% or more of the time. We additionally note that our results support, to a considerable extent, previous studies that used newer path and predicate criteria to evaluate test suites/techniques [Ball, 2004, Visser et al., 2006, Pacheco et al., 2007, Sharma et al., 2011, Groce, 2011, Groce et al., 2012a, Chilimbi et al., 2009b, Wang and Roychoudhury, 2005, Chaki et al., 2004]: while PCT criteria were not our best, the hand-coded PCT(MS) performed well, and PCT performed better than IMP, which was used in fewer studies. Our results also indicate the benefit of using multiple criteria to evaluate suites, as is common practice in studies: while the worst correlation for some subjects is below 0.5, the best is over 0.5 in all but two subjects. Agreement between multiple criteria should increase confidence in a ranking.

Linear Regression Table 3.8 shows R^2 values for our subjects and criteria. For the primary research question of this work (the validity of using criteria to predict ranking of mutation scores), R^2 is less relevant than τ_b, and the validity of relative R^2 values may be compromised by non-linear relationships. However, the overall picture of the correlation between criteria and mutation scores changes from τ_b only in that R^2 suggests that AIMP is often *better* than BC coverage for quantitative prediction. This confirms the claim that AIMP is the most useful non-BC criteria. We also note that in some cases R^2 for a coverage criterion is too low to suggest

Table 3.8: R^2 values for each subject program and criteria

Subject	Test-Suite Selection with Coverage Method							Test-Suite Selection with Size Method						
	SC	BC	IMP	AIMP	PCT			SC	BC	IMP	AIMP	PCT		
					MS	BB	ST					MS	BB	ST
language: Java														
JFreeChart	0.992	0.995	0.836	0.998	-	0.989	0.989	0.875	0.916	0.417	0.892	-	0.900	0.863
JodaTime	0.990	0.994	0.999	0.998	-	0.997	0.998	0.914	0.935	0.934	0.937	-	0.929	0.918
AvlTree	0.801	0.790	0.778	0.753	0.867	0.916	0.927	0.390	0.418	0.575	0.622	0.674	0.627	0.605
BinomialHeap	0.520	0.690	0.520	0.824	0.771	0.875	0.863	0.617	0.766	0.369	0.866	0.782	0.881	0.878
BinTree	0.248	0.271	0.198	0.310	0.454	0.393	0.485	0.172	0.276	0.600	0.667	0.665	0.606	0.700
FibHeap	0.825	0.884	0.124	0.277	0.599	0.713	0.536	0.735	0.805	0.414	0.652	0.703	0.796	0.752
FibonacciHeap	0.473	0.497	0.441	0.517	0.472	0.493	0.478	0.222	0.300	0.377	0.415	0.439	0.396	0.398
HeapArray	0.743	0.870	0.506	0.679	0.581	0.846	0.679	0.834	0.897	0.577	0.862	0.828	0.911	0.846
IntAVLTreeMap	0.888	0.860	0.800	0.896	0.767	0.785	0.827	0.891	0.872	0.793	0.884	0.766	0.863	0.865
IntRedBlackTree	0.637	0.659	0.807	0.834	0.769	0.833	0.813	0.486	0.462	0.817	0.815	0.744	0.793	0.782
LinkedList	0.583	0.757	0.423	0.818	0.757	0.658	0.546	0.751	0.904	0.042	0.463	0.904	0.362	0.373
NodeCachLList	0.492	0.730	0.566	0.694	0.702	0.550	0.440	0.725	0.707	0.122	0.691	0.618	0.357	0.343
SinglyLList	0.325	0.359	0.176	0.304	0.302	0.399	0.468	0.446	0.456	0.484	0.567	0.492	0.607	0.741
TreeMap	0.799	0.829	0.781	0.889	0.875	0.897	0.903	0.686	0.695	0.851	0.895	0.872	0.875	0.873
TreeSet	0.762	0.776	0.777	0.874	0.824	0.827	0.875	0.683	0.662	0.827	0.869	0.824	0.818	0.847
language: C														
Space	0.986	0.989	0.839	0.993	-	0.985	0.972	0.954	0.963	0.900	0.974	-	0.899	0.896
SQLite	0.942	0.950	0.051	0.981	-	0.965	0.960	0.942	0.950	0.051	0.981	-	0.965	0.960
YAFFS2	0.802	0.804	0.137	0.802	-	0.770	0.779	0.785	0.798	0.397	0.826	-	0.793	0.775
Printtokens	0.812	0.834	0.745	0.976	-	0.799	0.899	0.798	0.764	0.700	0.969	-	0.740	0.882
Printtokens2	0.852	0.854	0.724	0.827	-	0.856	0.856	0.657	0.653	0.455	0.642	-	0.651	0.639
Replace	0.767	0.771	0.537	0.751	-	0.746	0.749	0.642	0.652	0.560	0.669	-	0.635	0.642
Schedule	0.739	0.813	0.558	0.837	-	0.826	0.821	0.773	0.816	0.494	0.825	-	0.845	0.849
Schedule2	0.702	0.705	0.574	0.732	-	0.735	0.760	0.396	0.434	0.503	0.545	-	0.540	0.739
SglibRbtree	0.867	0.877	0.660	0.773	-	0.842	0.835	0.828	0.834	0.648	0.765	-	0.823	0.827
Totinfo	0.661	0.667	0.610	0.695	-	0.664	0.637	0.666	0.681	0.420	0.691	-	0.694	0.674
Tcas	0.795	0.819	0.790	0.790	-	0.828	0.791	0.751	0.768	0.770	0.770	-	0.803	0.772
Standard deviation	0.160	0.130	0.244	0.149	0.156	0.139	0.151	0.167	0.162	0.253	0.144	0.124	0.163	0.154
Geometric mean	0.746	0.791	0.503	0.782	0.679	0.787	0.775	0.676	0.712	0.430	0.760	0.723	0.725	0.738
Arithmetic mean	0.765	0.804	0.585	0.800	0.701	0.801	0.792	0.698	0.731	0.526	0.774	0.734	0.746	0.758
The best results	0	6	1	10	0	3	7	3	4	1	8	3	4	4
The worst results	4	0	20	2	1	0	1	8	2	14	0	1	0	1

it as a valid predictor of mutation score, but Kendall τ_b shows that the criterion nonetheless manages to have a high probability to correctly predict rank order of mutation scores.

Chapter 4: Test Case Prioritization and Selection for Symbolic Execution

4.1 Introduction

Seeded symbolic exploration can be seen as a powerful variation of regression testing, which suggests the use of test case prioritization and selection [Yoo and Harman, 2012]. For example, an obvious approach is to attempt to run regression tests in such an order that total code coverage is maximized as quickly as possible.

Since each different seed can direct symbolic execution into a different state, it is reasonable to assume that there are some better ordering of seeds to explore than in a random order. For example, by symbolically exploring diverse test cases first, a symbolic executor can have a higher probability of reaching more diverse program states.

In the previous chapter, we experimentally evaluate the effectiveness of test case reduction. We now elaborate test case prioritization and selection techniques and examine how they impact the effectiveness of the symbolic execution effort, measured in terms of improved incremental branch coverage for a time budget and the rate of coverage improvement.

4.2 Test Case Prioritization

Test case prioritization orders test cases with the goal of finding faults in regression testing as quickly as possible [Yoo and Harman, 2012]. For seed-based symbolic execution, we can also order the seeds, with the goal of gaining incremental coverage as quickly as possible. A large body of work exists on prioritization methods for regression testing, and it is not our purpose to explore all possible strategies. Instead, we aim to demonstrate first, that ordering is important and second, that some approaches that might not be obviously useful in traditional regression testing are effective for seed-based symbolic execution.

4.2.1 Ranking Strategies

We considered five ranking strategies, some drawn from the simpler regression approaches and two that were devised with symbolic execution in mind:

1. **Random Ordering (Random):** Our first prioritization is not in fact a proposed solution but a baseline to measure the importance of ordering. Random ordering simply chooses a random permutation of test cases.

2. **Coverage Total (CT):** Prioritize test cases by their statement, branch, or AIMP coverage such that the test with the highest coverage is used as a seed first. Ties are broken randomly. For example, for three test cases A, B, and C with coverage {2, 3}, {4}, and {1, 2, 3} respectively, the CT ordering of them is [C, A, B].

Algorithm 4.1 Coverage Additional prioritization algorithm $CA(T)$

Let T be test cases $\{t_1, t_2, \ldots, t_n\}$ to be prioritized

```
 1: for $i \leftarrow 1, n$ do
 2:     $C_i \leftarrow$ coverage of $t_i$          ▷ It can be statement, branch, or AIMP.
 3: end for
 4: $C \leftarrow \emptyset$
 5: $R \leftarrow ()$
 6: while $T \neq \emptyset$ do
 7:     $imax \leftarrow \underset{i \in indices(T)}{\text{argmax}} (|C_i \setminus C|)$
 8:     if $C_{imax} \setminus C = \emptyset$ then
 9:         $C \leftarrow \emptyset$
10:     end if
11:     $C \leftarrow C \cup C_i$
12:     $T \leftarrow T \setminus \{t_{imax}\}$
13:     $R \leftarrow R + (t_{imax})$
14: end while
15: return $R$
```

3. **Coverage Additional (CA):** Prioritize by incremental statement, branch, or AIMP coverage over all tests previously ranked. The first test chosen is therefore the same as with the Coverage Total method, but further tests are selected by the additional coverage provided, not absolute coverage. The details of this method is depicted in Algorithm 4.1. For the previous example, the CA ordering of A, B, and C is [C, B, A].

4. **Furthest-Point-First (FPF)**: The first test ranked is chosen randomly. Future test cases are selected by computing the minimum distance to all already-ranked tests, and adding the test case with the largest minimum distance to the ranking. The distance function in our case is the Hamming

Algorithm 4.2 Furthest-Point-First (FPF) prioritization algorithm $FPF(T)$

Let T be test cases $\{t_1, t_2, \dots, t_n\}$ to be prioritized

```
 1: for $i \leftarrow 1, n$ do
 2:     $C_i \leftarrow$ coverage of $t_i$            ▷ It can be statement, branch, or AIMP.
 3: end for
 4: for $i \leftarrow 1, n$ do
 5:     for $j \leftarrow 1, n$ do
 6:         $D_{ij} \leftarrow distance(C_i, C_j)$            ▷ As described in (4.1)
 7:     end for
 8: end for
 9: $R \leftarrow$ randomly choose one test from $T$
10: while $T \neq \emptyset$ do
11:     $imax \leftarrow \underset{i \in indices(T)}{\text{argmax}} (MD_{iR})$ where $MD_{iR} = \min_{j \in indices(R)} (D_{ij})$
12:     $T \leftarrow T \setminus \{t_{imax}\}$
13:     $R \leftarrow R + (t_{imax})$
14: end while
15: return $R$
```

distance between the coverage vectors of two test cases; even if a test does not produce incremental coverage it can therefore be ranked highly if it executes a very different set of coverage targets than any other test. The details of this method is described in Algorithm 4.2.

5. **Shortest Path (SP):** Tests are ranked in order of increasing path length. The number of branches taken during execution is counted, and paths that execute fewer total branches (where the same branch executed multiple times counts each time) in the source of the tested program are chosen first.

While CT and CA are selected as examples of simple coverage-based prioritizations from the regression literature, FPF and SP require more justification, since in theory they might well tend to select test cases with poor total or incremental coverage. In fact, SP seems guaranteed to prioritize test cases that have worse branch coverage, on average!

The motivation for FPF [Gonzalez, 1985] is that while branch coverage is important to symbolic execution based on a seed test case (if a branch is not covered, its divergences will obviously be hard to explore), the context in which branches are executed is also important. We would like to explore from tests that, in some sense, have very different behavior. Unfortunately, defining a generalized, cheap-to-compute, behavioral distance metric for program executions is difficult [Groce, 2004]. The hope is that simply executing different branches together can indicate behavioral difference in test cases well enough to serve as a useful ordering. The use of the FPF algorithm is inspired by efforts in *fuzzer taming*, which seeks to rank a set of failing test cases such that test cases exposing different underlying faults are ranked highly [Chen et al., 2013]. Basically, FPF aims to maximize the *diversity* (as defined by a distance metric) of test cases for the first N ranked test cases. The chosen distance metric is different than any used in fuzzer taming [Chen et al., 2013] because the aim is general diversity, not focused on failing test cases (and the inputs do not have such meaningful tokens as program source code). The hypothesis is that despite not necessarily prioritizing high coverage tests, FPF will produce good results by somewhat diversifying the behavior of executions.

The shortest path method is motivated by a simpler concern. Seeded symbolic execution can scale very poorly to long test cases, in our experience. It may be that simply choosing short tests, as measured in terms of potential divergence points, is the best possible ordering, since each symbolic exploration run will have more chance of exploring a large neighborhood of a test case if path explosion is limited and constraints are simpler.

Obviously, there are a great many plausible ranking strategies; our FPF ranking alone is simply one example from a vast family of different rankings based on different feature vectors and distance functions.

The purpose of this dissertation is not to exhaustively explore the space, but to establish that ranking is useful, even with possibly sub-optimal ranking techniques, and to show whether prioritizations that are not similar to ones used (to our knowledge) in regression prioritization can be useful due to the nature of symbolic execution.

In general, a distance function maps any pair of test cases to a real number that serves as a measure of similarity. This is useful because our goal is to present symbolic executors with a collection of highly dissimilar test cases. Because there are many ways in which two test cases can be similar to each other—e.g., the functions they covered can be similar, their branch coverage can be similar. In this dissertation, we use Hamming distance as the distance function of two test case vectors. Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. It is formally defined

Table 4.1: Test case prioritization metric

	Prioritization method		
Coverage criteria	(Furthest Point First)	(Coverage Additional)	(Coverage Total)
Statement	FPF(Stmt)	CA(Stmt)	CT(Stmt)
Branch	FPF(Branch)	CA(Branch)	CT(Branch)
AIMP	FPF(AIMP)	CA(AIMP)	CT(AIMP)

by

$$HMD(v_1, v_2) = \sum_{i=1..n} |v_1[i] - v_2[i]| \tag{4.1}$$

For constructing vectors from test cases, we use runtime profiling information of executing those test cases. For example, the *statement coverage* of a test case is a list of the statements that it executes while running this test case. The length of the vector is the total number of statements of the underlying program. If a statement is executed when running a test case, we take the corresponding element in the vector as one, and zero otherwise. Other possible feature vectors include, branch coverage, acyclic intra-method method path (AIMP) coverage, and so forth.

Besides the variations of prioritization methods, since there are various selections of coverage criteria when doing Coverage Total, FPF, and Coverage Additional prioritization, we also explore the best combinations of test case feature vectors and prioritization methods.

Table 4.1 shows all the combinations of prioritization methods and coverage criteria we investigated in our experiment. In addition to those, we also explore shortest path based prioritization and random prioritization.

Table 4.2: Ranking cost for 100 test cases for all subject programs in seconds (FPF: Furthest Point First, SP: Shortest Path, BA: Branch Additional)

Subject	Coverage collection	FPF	SP	BA
Sed	17.3	1.3	< 0.1	1.0
Space	12.0	1.1	< 0.1	0.8
Grep	19.5	1.5	< 0.1	1.3
Gzip	43.7	0.7	< 0.1	0.6
Vim	104.8	8.7	< 0.1	7.7
YAFFS2	21.9	1.7	< 0.1	1.5

4.2.2 Experimental Results: RQ2 (Test Case Prioritization)

Test Case Prioritization Cost Table 4.2 shows the ranking cost for 100 randomly chosen test cases for all subject programs. The cost consists of two parts: the first part is from collecting branch coverage information for each test case. We use `gcov` to obtain branch coverage. The second cost is in using the coverage vector to compute the distance between each pair of test cases and rank all test cases using relative distances. As Table 4.2 shows, ranking test cases is a low cost operation compared with symbolic execution, a necessary requirement for an affirmative answer to **RQ2**.

Test Case Prioritization Effectiveness To evaluate if prioritizations perform better than random ordering, we adapt the Average Percentage Faults Detected (APFD) [Rothermel et al., 1999, Rothermel et al., 2001] measure, which is used extensively in test case prioritization evaluations. Although we examine branch coverage rather than fault detection, the same method works well for comparing two *curves* to see which one converges faster. Average Percentage Branches Discovered (APBD), our measure, is computed as [Rothermel et al., 2001]:

$$\text{APBD} = \frac{\sum_{i=1}^{n-1} BC_i}{nm} + \frac{1}{2n}. \tag{4.2}$$

Here, n is the number of the test cases, m is the total newly covered branches from symbolic execution, and BC_i is the number of branches newly covered by symbolic exploration of at least one test case in the first i test cases. Note that because APBD compares curve gains rather than absolute values, it is only useful within orderings of the same total result. For example, the APBD might be better for a 10 minute symbolic execution time limit than for a 20 minute symbolic execution time limit because while the final total branches explored by the 20 minute execution will almost certainly be higher, the *percent* gain might be better for the lower total gain. This is important to keep in mind when reading results below, as often APBD will be "better" for some less effective explorations; however our results only compare prioritizations across the same final total.

For each subject program, we generated 150 test suites, each including 100 randomly chosen test cases from the pool. We computed APBD values for each test suite with all ranking techniques and search strategies and averaged the results (we also performed the same experiment using reduced test cases).

Table 4.3 and 4.4 show the major results of our prioritization evaluation. All the prioritization methods are compared with median, mean APBD values, the

Table 4.3: Median, mean, standard deviation, Wilcoxon p-values, and Cohen's d effect sizes (es) of APBD values by search strategy over all programs, search times, and reduced and original test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement, STD: Standard deviation, DFS: Depth first search, RP: random path search, RS: random state search, MD2U: Minimum distance to an uncovered instruction search)

Search		Prioritization method										
		Random	Shortest	FPF			CA			CT		
strategy	Statistics		path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
DFS	Median(%)	74.35	88.67	84.17	82.96	82.03	82.49	80.90	79.78	49.83	51.92	52.61
	Mean(%)	74.26	85.47	84.30	83.33	81.97	83.86	83.31	82.26	49.79	49.81	49.20
	STD(%)	6.35	10.41	8.82	9.62	9.79	7.59	7.60	7.94	20.55	20.89	20.93
	p	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	es	-	1.30	1.31	1.11	0.93	1.37	1.29	1.11	-1.61	-1.58	-1.62
RP	Median(%)	92.25	93.28	94.81	94.23	94.08	94.29	94.42	94.63	92.49	92.52	90.99
	Mean(%)	90.57	92.65	94.91	94.71	94.61	94.74	94.78	94.79	81.93	82.00	81.75
	STD(%)	6.22	4.67	2.58	2.72	2.93	2.73	2.61	2.76	21.30	21.21	20.98
	p	-	**0.36**	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.00
	es	-	0.38	0.91	0.86	0.83	0.87	0.88	0.88	-0.55	-0.55	-0.57
RS	Median(%)	84.85	92.16	93.11	92.50	91.50	90.80	90.49	91.32	68.01	67.39	66.46
	Mean(%)	84.79	91.03	92.91	92.47	91.51	91.07	90.91	90.67	68.98	68.59	68.04
	STD(%)	4.93	4.81	2.96	3.23	3.76	3.35	3.59	3.90	18.00	17.77	17.74
	p	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	es	-	1.28	2.00	1.84	1.53	1.49	1.42	1.32	-1.20	-1.24	-1.29
MD2U	Median(%)	84.69	92.38	92.98	92.51	91.50	90.43	91.30	91.53	70.28	70.55	67.69
	Mean(%)	84.30	91.02	92.61	92.25	91.28	90.86	90.82	90.46	68.58	68.27	67.53
	STD(%)	5.33	4.94	3.40	3.59	4.22	3.45	3.62	4.03	19.45	19.26	19.04
	p	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	es	-	1.31	1.86	1.75	1.45	1.46	1.43	1.31	-1.10	-1.13	-1.20

number of times that one outperforms all the other methods, etc. Specifically, Table 4.3 shows APBD values of all prioritization methods we evaluated on each symbolic execution search strategy. The values are mean, median, and standard deviation over all the subjects and search time budget. Additionally, to show if the results are statistically significant and how strong the effects of different prioritization methods over random ordering are, Wilcoxon p-values and Cohen's

Table 4.4: APBD comparison by search strategy over all programs, search times, and reduced and original test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement)

Search strategy	#	Prioritization method										
		Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
DFS	The best results	2	19	8	6	2	10	1	0	0	0	0
	Win random	-	44	40	40	38	42	40	40	8	8	8
	Lost to random	-	4	8	8	10	6	8	8	40	40	40
RP	The best results	3	6	9	6	6	6	4	7	0	0	1
	Win random	-	23	43	42	38	41	42	43	20	19	19
	Lost to random	-	25	5	3	7	4	3	2	25	26	26
RS	The best results	0	15	13	9	2	0	3	6	0	0	0
	Win random	-	44	48	48	46	46	42	44	11	11	10
	Lost to random	-	4	0	0	2	2	6	4	37	37	38
MD2U	The best results	0	13	14	11	5	2	3	0	0	0	0
	Win random	-	45	48	48	46	44	45	45	11	13	14
	Lost to random	-	3	0	0	2	4	3	3	37	35	34

d (the columns es) are also shown in Table 4.3.

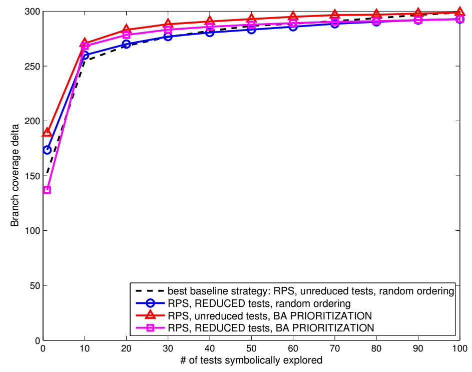
A green highlighted value is the best value in each row while the red highlighted is the worst value. Coverage Total (CT) prioritization generally performed worst in almost all experiments, and in fact performed worse than random ordering. We are unclear why in these experiments and in the cause reduction experiments [Groce et al., 2014a] coverage total ordering performed so poorly, but we do not suggest its use in prioritization for symbolic execution. The best prioritization varies by subject and strategy, but a few general points are clear. First, all prioritizations outperform random ordering in general. Second, the non-traditional prioritizations are competitive, despite not maximizing branch coverage, and work differently than the regression-prioritization based method. Finally, shortest path (SP) was most effective in two cases: it benefited DFS searches in general, and was often the best prioritization for `YAFFS2`, the most complex and difficult to test of our subjects. We suspect that SP may prove even more valuable as we attempt to apply symbolic execution to increasingly complex programs. Another observation is that FPF-based prioritization on statement coverage outperforms all the other prioritization methods.

Another observation from these two tables is that FPF prioritization based on statement coverage is even better than SP prioritization method in terms of the times it wins random baseline and outperforms all the other methods.

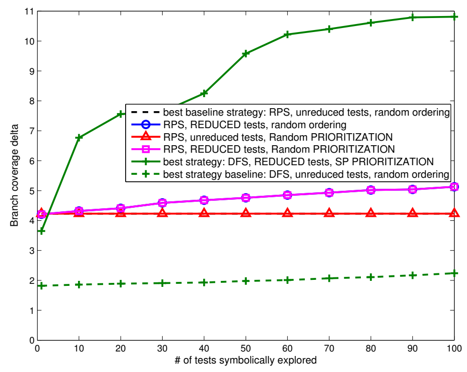
Table A.1— A.4 show APBD values of all prioritization techniques we investigated for each different symbolic search strategy. Values in *italics* indicate cases where the difference between a technique and random ordering was not statistically significant (using the same test as with reduction results).

Table A.5— A.8 show Cohen's *d* effect sizes for for all prioritization methods. Larger effect sizes indicate that the corresponding prioritization methods have a stronger effect over random ordering.

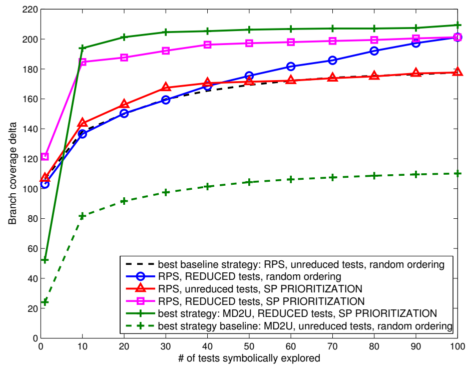
To show how good a prioritization method is comparing to random ordering, we can either compare the APBD values or compare the average absolute branch increment between a prioritization method and random ordering. Table A.9— A.12 show average absolute branch increment values.



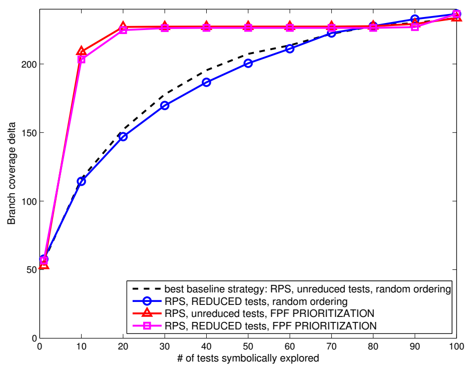
(a) sed



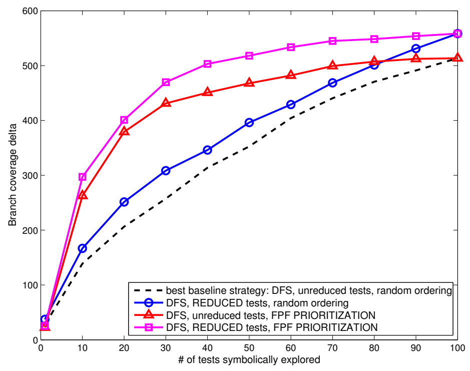
(b) space



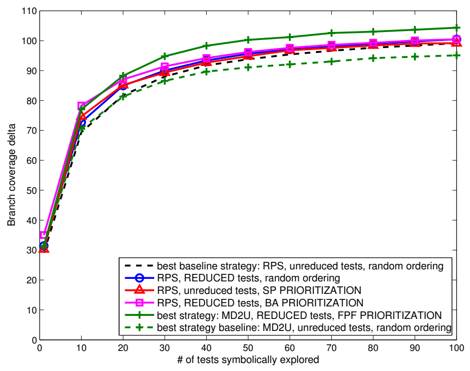
(c) grep



(d) gzip



(e) vim



(f) yaffs2

Figure 4.1: Ranking methods comparison for all subject programs.

4.2.3 Overall Effects of Test Case Reduction and Prioritization

Figure 4.1 graphically shows the contributions of reduction and prioritization. Each graph shows, for one subject, (1) the average incremental branch discovery curve for the search strategy that performs best (as measured by APBD) for the baseline case of no reduction and random ordering, (2) the curve for that same strategy with test case reduction and random ordering, (3) the curve for that strategy with the best prioritization but no reduction, (4) the curve for that strategy with the best prioritization and with test case reduction, (5) the best APBD curve for that subject over all experiments, if this is not already included in 1-4, and (6) the baseline for 5 if it is not already included. These graphs show, first, that the value of reduction vs. prioritization varies with subject and strategy. Second, in three cases using reduction and prioritization not only improves a method, but changes the most effective strategy for search. The `space` and `grep` examples are particularly compelling: the baselines for DFS and MD2U respectively in these examples are strikingly different than the curves for the same search strategy with reduction and prioritization, so much so that these become the optimal strategies for exploring the state space by a large margin over the best strategy without reduction and prioritization. The same thing happens with `YAFFS2`, though the difference in baseline and final curves is less dramatic.

4.3 Test Case Selection

As we have shown before, symbolically executing a test case is time-consuming. Given a large amount of existing test cases of a program and a limited time budget, one natural question is that if we can just choose a subset of test cases for symbolic execution while not losing too many coverage targets? This has the same goal with the test case selection problem in regression testing. In regression testing, test cases are selected out of a large tests pool while the same testing requirements are still met. For the test case selection problem for symbolic execution, the testing requirement becomes the overall branch coverage after symbolically augmenting a suite of test cases.

Formally, test case selection problem for symbolic execution can be depicted as: for a suite of seed test cases $S : (T_1, T_2, \ldots, T_n)$, find a subset of S_1, S_2 such that by symbolically augmenting S_2 we get the same coverage targets (branch coverage) as we get by symbolically augmenting S_1.

In regression testing, researchers have used various greedy strategies for finding a minimal subset of test cases. The difference between selection problem in symbolic execution and the one in regression testing is that we cannot know how many additional targets one test case can discover before we actually symbolically execute it.

In regression testing, testers can run all test cases and collect coverage and running time information for each test case. With those information, test case selection algorithm then can choose a subset of test cases. But, in seeded symbolic execution, it takes too much time for running all tests symbolically. In fact, before we actually run a test case symbolically for a given time budget, we cannot precisely predict how many branches targets will be covered for the given time interval since it depends on many factors, such as search strategies, constraints solving time, etc.

Given that it is often infeasible to symbolically execute all the test cases due to the demanding computation and memory requirements, we might need to relax the goal of "discover all branch targets that can be discovered by executing all the tests symbolically" to "discover as many as branch targets" by only exploring a small subset of test cases. Our third research question is therefore:

RQ3: *Given a large number of test cases, can we just choose small amount of them and still do symbolic execution effectively?*

4.3.1 Selection Strategies

Unlike test case prioritization problem, test case selection problem needs to know how many test cases are needed. Once we have this number, we can use any "good" prioritization method to rank all the test cases, and choose the top tests in the rank to execute symbolically. A selection algorithm can ask users for this input, but it will impose an unnecessary burden on its users since users might not be able to give a reasonable number. Ideally, A selection algorithm should be able to analyze the characteristics of all the test cases and give a reasonable number of test cases and which test cases should be selected to reach a certain coverage goal.

To choose a subset of test cases automatically, we consider two approaches.

They are all adapted from our prioritization strategies introduced in Section 4.2:

1. **Coverage Additional Selection (CAS):** Instead of arbitrarily choosing a fixed number of test cases, coverage additional method gives a natural stop rule.

    - Choose a test case with highest structural coverage,
    - Choose next test cases which can increase the existing coverage in a largest amount,
    - Repeat step 1) and 2) until no more test cases are needed.

2. **Furthest-Point-First Selection (FPFS):** Recall from Algorithm 4.2 that FPF method calculates a maximum minimal distance for each new test case. The distances will get shorter and shorter with more test cases selected. For a given test suite, one heuristic selection method with FPF method could be that we stop choosing more once the max-min distance get small enough or do not change much.

Similar to Coverage Additional test case prioritization, Coverage Additional Selection chooses test cases in such an order that the total code coverage of selected test cases is maximized as quickly as possible. Unlike its counterpart in prioritization, the selection process terminates once the total coverage cannot grow anymore. Algorithm 4.3 shows the detailed steps of test case selection method we used in our experiment.

Algorithm 4.3 Coverage Additional selection algorithm $CAS(T)$

Let T be test cases $\{t_1, t_2, \ldots, t_n\}$ to be selected

```
1: for $i \leftarrow 1, n$ do
2:     $C_i \leftarrow$ coverage of $t_i$          ▷ It can be statement, branch, or AIMP.
3: end for
4: $C \leftarrow \emptyset$
5: $R \leftarrow ()$
6: while $T \neq \emptyset$ do
7:     $imax \leftarrow \underset{i \in indices(T)}{\text{argmax}} (|C_i \setminus C|)$
8:     if $C_{imax} \setminus C = \emptyset$ then
9:         return $R$
10:    end if
11:    $C \leftarrow C \cup C_i$
12:    $T \leftarrow T \setminus \{t_{imax}\}$
13:    $R \leftarrow R + (t_{imax})$
14: end while
15: return $R$
```

FPF-based test case selection is another greedy method we investigated in our experiment. We first use an example from sed program to explain the main idea of this selection algorithm. Figure 4.2 shows that the maximum min distances decrease when more `sed` program tests are added to the rank sequence. The x-axis is the number of test cases, and the y-axis is the max-min distance when we compute FPF ranking. Our greedy strategy is that once the distance is small enough, which means that the unselected test cases are very close to the selected test cases, the selection stops. The detailed steps are depicted in Algorithm 4.4. The algorithm first invokes FPF prioritization algorithm for computing a ranking of all the test cases. In addition to the rank, it also records all the maximum

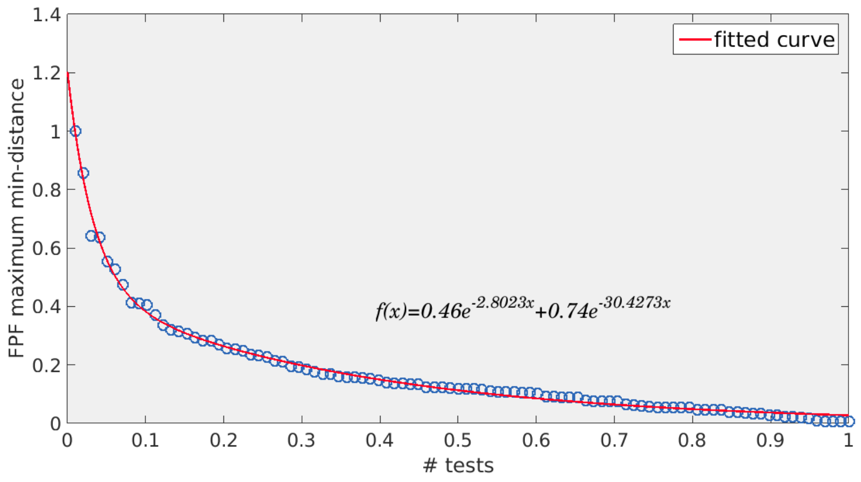


Figure 4.2: Illustration of the decrease of FPF distance with more tests added. The x-axis and y-axis values are normalized. The red curve is a fitted curve with function $f(x) = ae^{bx} + ce^{dx}$ (a.k.a. biexponential decay function).

minimum-distance for each test case. With the data points (i.e., pairs of test case number and distance), it fits a curve f and find a point p on this curve such that the slope on point p is -1.0 or -0.5. The number p later is used to decide how many test cases will be chosen from the whole test cases.

Similar to prioritization methods, for Coverage Additional and FPF based selection, there are three variations for each of them, depending on the underlying coverage criteria (i.e., statement, branch, and AIMP). We evaluate all of them in our selection strategy experiment.

For test case selection, two questions must be answered are that how many test cases are needed to reach a certain goal and how should we evaluate a test case selection result. We need a measure to evaluate our test case selection method. Ideally, we want to choose as few as test cases since symbolically executing a test

Algorithm 4.4 Furthest-Point-First (FPF) selection algorithm $FPFS(T, slope)$

```
Let $T$     be test cases $\{t_1, t_2, \ldots, t_n\}$ to be selected
    $slope$ termination slope
1: $R \leftarrow FPF(T)$                                        ▷ Invoke Algorithm 4.2.
2: $D \leftarrow$ maximum minimum distances when computing FPF rank
3: $X \leftarrow (1, 2, \ldots, n)/n$
4: $Y \leftarrow D/max(D)$
5: $f \leftarrow CurveFit(X, Y)$                                ▷ $f(x) = ae^{bx} + ce^{dx}$
6: $p \leftarrow x_0$ such that $\left.\frac{df(x)}{dx}\right|_{x=x_0}$ = slope.
7: $p \leftarrow \lfloor p \rfloor$
8: return $R[1, 2, ..., p]$
```

case is expensive as we shown in previous sections while we do not want to lose "good" test cases which can direct symbolic execution to the program states that other tests cannot direct to.

4.3.2 Experimental Results: RQ3 (Test Case Selection)

We evaluate Coverage Additional- and FPF-based test case selection methods with statement, branch, and AIMP coverage criteria. The selection termination rule varies for these two methods. For additional based method, once a coverage criteria is saturated and cannot grow any more, the selection stops.

Same as the evaluation of test case reduction and prioritization, we asked `KLEE` to perform symbolic search for 10 and 20 minutes. From the test case pool, we randomly chose 100 test cases as a test suite 150 times, for each subject. For each such suite, we first apply various selection method to generate a subset of the test suite. We define four measures below for explaining how we evaluate the quality of the chosen subset.

1. C_o**:** The set of all branches covered by the original 100 test cases,

2. C_s**:** The set of all branches explored during symbolic execution of the 100 original test cases,

3. C_{ss}**:** The set of all branches explored during symbolic execution of the chosen subset of the 100 original test cases.

Based on these measures, we define a ratio R as shown below:

$$R = \frac{|C_{ss} \setminus C_o|}{|C_s \setminus C_o|}. \tag{4.3}$$

The R here denotes how "good" the chosen subset is. Ideally, we want the size of the chosen subset be as small as possible, and R is as close as to 1. In the case of that R is 1, we discovered all the branches discovered by the whole test suite without symbolically running all of them.

Table 4.5 gives a comparison of FPF selection method with Coverage Additional selection method. The "#" columns under "Statement","Branch", and "AIMP" columns are the number of tests selected with FPF- or Coverage Additional-based selection method based on statement, branch, and AIMP coverage criteria. The "%" columns are the values R. All the values in the table are the means of 150 runs

Table 4.5: Test case selection values with different prioritization methods on 100 and 200 randomly selected test cases (FPF: Furthest Point First, CA: Coverage Additional)

	FPF (termination slope: -1)						FPF (termination slope: -0.5)						CA					
Search	Statement		Branch		AIMP		Statement		Branch		AIMP		Statement		Branch		AIMP	
strategy	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%	#	%
Test suite size: 100																		
DFS	14.3	70.9	16.1	70.5	15.9	67.3	27.9	80.7	33.5	82.7	36.5	83.1	28.7	82.0	32.0	84.2	43.9	89.4
RP		91.3		91.2		87.6		95.2		96.1		96.2		93.4		94.5		96.6
RS		87.3		87.2		82.0		93.6		94.3		94.3		92.7		94.0		97.2
MD2U		87.1		87.0		81.7		93.1		94.0		94.0		92.6		94.0		97.0
Test suite size: 200																		
DFS	22.8	70.9	24.9	70.9	22.9	66.3	45.2	81.4	53.5	82.8	57.2	81.3	40.8	79.4	45.1	81.5	62.8	84.9
RP		91.6		90.9		86.5		95.1		95.3		95.9		92.5		94.0		95.8
RS		87.5		87.0		80.6		93.6		94.1		93.6		90.8		92.5		95.8
MD2U		87.7		87.1		80.7		93.6		94.4		93.9		91.3		93.0		96.3

over all subject programs. We evaluate the selection methods with two different test suite sizes 100 and 200.

From the table, we notice that AIMP-based coverage additional selection performs the best in term of the newly acquired branch coverage. However, it also chooses the largest number of tests. Another observation is that FPF-based selection choose less tests and also achieve 90+% coverage additional except for DFS search strategy.

Tables B.1–B.8 show further details for the experiment.

Chapter 5: Conclusion

Symbolic execution technique was proposed in the 1970s but it is until recently that it found its practical applications in software testing and verification due to the advance of constraint solvers and the faster computing power of modern computers. Test case directed or seeded symbolic execution further improves its adoption in large-scale industrial environments. By mixing concrete and symbolic execution, seeded symbolic execution can handle native code. This flexibility is especially crucial for real software projects which depend on some third-party closed-source libraries.

When applying seeded symbolic execution to a real software project, one outstanding problem is which test cases should be used for seeding symbolic execution, since a typical software project has many either randomly generated or manually created existing test cases already. Those test cases either are too complicated or too many. Complicated tests are very challenging for symbolic execution.

To address those challenges, we introduced three related techniques: reduction, prioritization, and selection of test cases. These three techniques addressed three different problems in applying symbolic execution in the presence of a large number of existing complicated test cases.

Test case reduction, built on top of a delta debugging tool, is used to simplify a complicated test case before seeding symbolic execution. Test case prioritization is implemented by adopting a traditional coverage incremental-based prioritization method. A furthest point first (FPF) method is also investigated and it demonstrates its superiority for finding diverse test cases for symbolic execution. For test case selection, we study FPF-based and coverage incremental methods and find that FPF-based method can achieve 90+% branch incremental by selecting only 20% test cases.

Although our experiment shows that test case reduction, prioritization, and selection can improve the effectiveness and efficiency of seed symbolic execution, we believe that many problems remain open, especially the test case selection problem.

This dissertation introduces and proposes solutions to three related research questions concerning improving test case seed-based symbolic execution, in the context of a proposed two-stage framework for testing. Our experimental results show that test case reduction, prioritization, and selection can be used to preprocess existing test cases for improving seeded symbolic execution. We also explore multiple variations of reduction, prioritization, and selection methods and make a systematic comparison of them.

Bibliography

[Ammann and Offutt, 2008] Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press.

[Andrews et al., 2005] Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411.

[Andrews et al., 2006] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *Trans. Softw. Eng.*, 32:608–624.

[Arcuri and Briand, 2011] Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1–10, New York, NY, USA. ACM.

[Ball, 2004] Ball, T. (2004). A theory of predicate-complete test coverage and generation. Technical Report MSR-TR-2004-28, Microsoft Research.

[Ball, 2005] Ball, T. (2005). A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects*, pages 1–22.

[Ball and Larus, 1996] Ball, T. and Larus, J. R. (1996). Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57.

[Ball and Rajamani, 2001] Ball, T. and Rajamani, S. (2001). Automatically validating temporal safety properties of interfaces. In *Workshop on Model Checking of Software*, pages 103–122.

[Bible et al., 2001] Bible, J., Rothermel, G., and Rosenblum, D. S. (2001). A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183.

[Bounimova et al., 2013] Bounimova, E., Godefroid, P., and Molnar, D. (2013). Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 122–131, Piscataway, NJ, USA. IEEE Press.

[Boyer et al., 1975] Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). Select&mdash;a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA. ACM.

[Burnim and Sen, 2008] Burnim, J. and Sen, K. (2008). Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA. IEEE Computer Society.

[Cadar et al., 2008] Cadar, C., Dunbar, D., and Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA. USENIX Association.

[Cadar et al., 2011] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA. ACM.

[Cadar and Sen, 2013] Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90.

[Cai and Lyu, 2005] Cai, X. and Lyu, M. R. (2005). The effect of code coverage on fault detection under different testing profiles. In *International Workshop on Advances in Model-Based Testing*, pages 1–7.

[Chaki et al., 2004] Chaki, S., Groce, A., and Strichman, O. (2004). Explaining abstract counterexamples. In *Symposium on the Foundations of Software Engineering*, pages 73–82.

[Chen and Lau, 1996a] Chen, T. Y. and Lau, M. F. (1996a). Dividing strategies for the optimization of a test suite. *Inf. Process. Lett.*, 60(3):135–141.

[Chen and Lau, 1996b] Chen, T. Y. and Lau, M. F. (1996b). Dividing strategies for the optimization of a test suite. *Inf. Process. Lett.*, 60(3):135–141.

[Chen et al., 2013] Chen, Y., Groce, A., Zhang, C., Wong, W.-K., Fern, X., Eide, E., and Regehr, J. (2013). Taming compiler fuzzers. In *Proceedings of the 34th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 197–208, New York, NY, USA. ACM.

[Chilimbi et al., 2009a] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. (2009a). Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA. IEEE Computer Society.

[Chilimbi et al., 2009b] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. (2009b). Holmes: Effective statistical debugging via efficient path profiling. In *International Conference on Software Engineering*, pages 34–44.

[Clarke, 1976] Clarke, L. A. (1976). A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA. ACM.

[Cliff, 1996] Cliff, N. (1996). *Ordinal Methods for Behavioral Data Analysis*. Pyschology Press.

[ClocWebPage,] ClocWebPage. Count lines of code. `http://cloc.sourceforge.net/`.

[Costner, 1965] Costner, H. L. (1965). Criteria for measures of association. *American Sociological Review*, 3.

[CoverageWebPage,] CoverageWebPage. Instrumented container classes - predicate coverage. `http://mir.cs.illinois.edu/coverage/`.

[Do et al., 2005] Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435.

[Do et al., 2004] Do, H., Rothermel, G., and Kinneer, A. (2004). Empirical studies of test case prioritization in a junit testing environment. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 113–124.

[Elbaum et al., 2000] Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2000). Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, New York, NY, USA. ACM.

[Elbaum et al., 2004] Elbaum, S., Rothermel, G., Kanduri, S., and Malishevsky, A. G. (2004). Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3):185–210.

[Frankl and Iakounenko, 1998] Frankl, P. G. and Iakounenko, O. (1998). Further empirical studies of test effectiveness. In *Symposium on the Foundations of Software Engineering*, pages 153–162.

[Frankl and Weiss, 1993] Frankl, P. G. and Weiss, S. N. (1993). An experimental comparison of the effectiveness of branch testing and data flow testing. *Trans. Software Eng.*, 19:774–787.

[Galeotti et al., 2010] Galeotti, J. P., Rosner, N., López Pombo, C. G., and Frias, M. F. (2010). Analysis of invariants for efficient bounded verification. In *International Symposium on Software Testing and Analysis*, pages 25–36.

[Garg et al., 2013] Garg, P., Ivancic, F., Balakrishnan, G., Maeda, N., and Gupta, A. (2013). Feedback-directed unit test generation for C/C++ using concolic execution. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 132–141, Piscataway, NJ, USA. IEEE Press.

[Gligoric et al.,] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., and Marinov, D. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology*. accepted for publication.

[Gligoric et al., 2013] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., and Marinov, D. (2013). Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA. ACM.

[Godefroid, 2007] Godefroid, P. (2007). Compositional dynamic test generation. In *Symposium on Principles of Programming Languages*, pages 47–54.

[Godefroid et al., 2005] Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA. ACM.

[Gonzalez, 1985] Gonzalez, T. F. (1985). Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38:293–306.

[Groce, 2004] Groce, A. (2004). Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 108–122.

[Groce, 2011] Groce, A. (2011). Coverage rewarded: Test input generation via adaptation-based programming. In *International Conference on Automated Software Engineering*, pages 380–383.

[Groce et al., 2014a] Groce, A., Alipour, M. A., Zhang, C., Chen, Y., and Regehr, J. (2014a). Cause reduction for quick testing. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 243–252, Washington, DC, USA. IEEE Computer Society.

[Groce et al., 2012a] Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, M. A., Erwig, M., and Lopez, C. (2012a). Lightweight automated testing with adaptation-based programming. In *International Symposium on Software Reliability Engineering*, pages 161–170.

[Groce et al., 2014b] Groce, A., Havelund, K., Holzmann, G., Joshi, R., and Xu, R.-G. (2014b). Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349.

[Groce et al., 2007] Groce, A., Holzmann, G., and Joshi, R. (2007). Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 621–631, Washington, DC, USA. IEEE Computer Society.

[Groce et al., 2012b] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. (2012b). Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, New York, NY, USA. ACM.

[Guilford, 1956] Guilford, J. P. (1956). *Fundamental Statistics in Pyschology and Education*. McGraw-Hill.

[Hamlet, 1994] Hamlet, R. (1994). Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley.

[Harrold et al., 1993] Harrold, M. J., Gupta, R., and Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285.

[Hassan and Andrews, 2013] Hassan, M. M. and Andrews, J. H. (2013). Comparing multi-point stride coverage and dataflow coverage. In *International Conference on Software Engineering*, pages 172–181.

[Hildebrandt and Zeller, 2000] Hildebrandt, R. and Zeller, A. (2000). Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 135–145, New York, NY, USA. ACM.

[Hsu and Orso, 2009] Hsu, H.-Y. and Orso, A. (2009). Mints: A general framework and tool for supporting test-suite minimization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 419–429, Washington, DC, USA. IEEE Computer Society.

[Hutchins et al., 1994] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200.

[JFreeChartWebPage,] JFreeChartWebPage. JFreeChart Home Page. `http://www.jfree.org/jfreechart/`.

[Jin and Orso, 2012] Jin, W. and Orso, A. (2012). Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA. IEEE Press.

[JodaTimeWebPage,] JodaTimeWebPage. JodaTime Home Page. `http://joda-time.sourceforge.net/`.

[Jones and Harrold, 2001] Jones, J. A. and Harrold, M. J. (2001). Test-suite reduction and prioritization for modified condition/decision coverage. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 92–, Washington, DC, USA. IEEE Computer Society.

[Kendall, 1938] Kendall, M. (1938). A new measure of rank correlation. *Biometrika*, 1-2:81–89.

[Kim et al., 2012] Kim, Y., Kim, M., Kim, Y., and Jang, Y. (2012). Industrial application of concolic testing approach: A case study on Libexif by using CREST-BV and KLEE. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1143–1152, Piscataway, NJ, USA. IEEE Press.

[Kim et al., 2014] Kim, Y., Zu, Z., Kim, M., Cohen, M. B., and Rothermel, G. (2014). Hybrid directed test suite augmentation: An interleaving framework. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 263–272, Washington, DC, USA. IEEE Computer Society.

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

[Lei and Andrews, 2005] Lei, Y. and Andrews, J. H. (2005). Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE '05, pages 267–276, Washington, DC, USA. IEEE Computer Society.

[Leitner et al., 2007] Leitner, A., Oriol, M., Zeller, A., Ciupa, I., and Meyer, B. (2007). Efficient unit test case minimization. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 417–420.

[Li et al., 2013] Li, Y., Su, Z., Wang, L., and Li, X. (2013). Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 19–32, New York, NY, USA. ACM.

[Majumdar and Sen, 2007] Majumdar, R. and Sen, K. (2007). Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA. IEEE Computer Society.

[Marinescu and Cadar, 2012] Marinescu, P. D. and Cadar, C. (2012). make testzesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 716–726, Piscataway, NJ, USA. IEEE Press.

[Marinescu and Cadar, 2013] Marinescu, P. D. and Cadar, C. (2013). KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint*

Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 235–245, New York, NY, USA. ACM.

[Marré and Bertolino, 2003] Marré, M. and Bertolino, A. (2003). Using spanning sets for coverage testing. *IEEE Trans. Softw. Eng.*, 29(11):974–984.

[McKeeman, 1998] McKeeman, W. M. (1998). Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107.

[McMaster and Memon, 2006] McMaster, S. and Memon, A. (2006). Call stack coverage for gui test-suite reduction. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 33–44, Washington, DC, USA. IEEE Computer Society.

[Namin and Andrews, 2009] Namin, A. S. and Andrews, J. H. (2009). The influence of size and coverage on test suite effectiveness. In *International Symposium on Software Testing and Analysis*, pages 57–68.

[Offutt et al., 1995] Offutt, A. J., Pan, J., and Voas, J. M. (1995). Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pages 111–123.

[Offutt et al., 1993] Offutt, A. J., Rothermel, G., and Zapf, C. (1993). An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 100–107, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Pacheco et al., 2007] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA. IEEE Computer Society.

[Păsăreanu and Visser, 2009] Păsăreanu, C. S. and Visser, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353.

[Regehr et al., 2012] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. (2012). Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA. ACM.

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

[Rothermel et al., 1999] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–188, Washington, DC, USA. IEEE Computer Society.

[Schuler and Zeller, 2009] Schuler, D. and Zeller, A. (2009). Javalanche: efficient mutation testing for Java. In *Symposium on the Foundations of Software Engineering*, pages 297–298.

[Sen et al., 2005] Sen, K., Marinov, D., and Agha, G. (2005). CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA. ACM.

[Seo and Kim, 2014] Seo, H. and Kim, S. (2014). How we get there: A context-guided search strategy in concolic testing. In *Proceedings of the 2014 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2014. To appear.

[Sharma et al., 2011] Sharma, R., Gligoric, M., Arcuri, A., Fraser, G., and Marinov, D. (2011). Testing container classes: Random or systematic? In *Fundamental Approaches to Software Engineering*, pages 262–277.

[Sharma et al., 2010] Sharma, R., Gligoric, M., Jagannath, V., and Marinov, D. (2010). A comparison of constraint-based and sequence-based generation of complex input data structures. In *Software Testing, Verification, and Validation Workshops*, pages 337–342.

[Siami Namin et al., 2008] Siami Namin, A., Andrews, J. H., and Murdoch, D. J. (2008). Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, New York, NY, USA. ACM.

[SQLiteWebPage,] SQLiteWebPage. SQLite Home Page. `http://www.sqlite.org/`.

[SymbExeBib,] SymbExeBib. A bibliography of papers related to symbolic execution. `https://github.com/saswatanand/symexbib`.

[Visser et al., 2006] Visser, W., Pasareanu, C. S., and Pelánek, R. (2006). Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*, pages 37–48.

[Vittek et al., 2006] Vittek, M., Borovansky, P., and Moreau, P.-E. (2006). A simple generic library for C. In *International Conference on Software Reuse*, pages 423–426.

[Vokolos and Frankl, 1998] Vokolos, F. I. and Frankl, P. G. (1998). Empirical evaluation of the textual differencing regression testing technique. In *International Conference on Software Maintenance*, pages 44–53.

[Walcott et al., 2006] Walcott, K. R., Soffa, M. L., Kapfhammer, G. M., and Roos, R. S. (2006). Timeaware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 1–12, New York, NY, USA. ACM.

[Wang and Roychoudhury, 2005] Wang, T. and Roychoudhury, A. (2005). Automated path generation for software fault localization. In *International Conference on Automated Software Engineering*, pages 347–351.

[Wong et al., 1994] Wong, W., Horgan, J., London, S., and Mathur, A. (1994). Effect of test set size and block coverage on the fault detection effectiveness. In *International Symposium on Software Reliability*, pages 230–238.

[Wong et al., 1995] Wong, W., Horgan, J., London, S., and Mathur, A. (1995). Effect of test set minimization on fault detection effectiveness. In *International Conference on Software Engineering*, pages 41–50.

[Xu et al., 2011] Xu, Z., Kim, Y., Kim, M., and Rothermel, G. (2011). A hybrid directed test suite augmentation technique. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 150–159, Washington, DC, USA. IEEE Computer Society.

[Xu et al., 2010] Xu, Z., Kim, Y., Kim, M., Rothermel, G., and Cohen, M. B. (2010). Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 257–266, New York, NY, USA. ACM.

[YAFFS2WebPage,] YAFFS2WebPage. YAFFS: A flash file system for embedded use. `http://www.yaffs.net`.

[Yoo and Harman, 2007] Yoo, S. and Harman, M. (2007). Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 140–150, New York, NY, USA. ACM.

[Yoo and Harman, 2012] Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120.

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200.

[Zhang et al., 2014] Zhang, C., Groce, A., and Alipour, M. A. (2014). Using test case reduction and prioritization to improve symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 160–170, New York, NY, USA. ACM.

[Zhang et al., 2013] Zhang, L., Hao, D., Zhang, L., Rothermel, G., and Mei, H. (2013). Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201.

[Zhang et al., 2010] Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., and Mei, H. (2010). Is operator-based mutant selection superior to random mutant selection? In *International Conference on Software Engineering*, pages 435–444.

APPENDICES

Appendix A: Tables of Test Case Prioritization Effects

Table A.1: APBD values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (percentage %) (Search strategy: Depth First Search)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	65.1	96.8	84.1	75.1	81.6	85.4	79.7	75.7	16.6	16.9	15.9
	20m	67.6	94.6	83.4	77.3	82.6	82.0	79.1	75.4	33.0	30.9	32.5
Space	10m	87.9	98.5	96.6	97.5	98.7	92.4	93.9	91.6	80.7	82.1	81.9
	20m	88.2	98.4	96.9	97.6	98.2	91.1	92.7	88.6	77.7	77.8	75.5
Grep	10m	74.5	88.9	89.6	88.1	86.9	87.5	86.4	87.7	47.8	47.4	45.5
	20m	67.2	88.7	87.6	87.6	86.8	88.9	86.4	82.7	31.2	30.7	29.2
Gzip	10m	76.1	94.4	96.3	96.4	94.0	97.0	96.8	96.1	47.2	47.5	49.9
	20m	75.9	94.4	96.5	96.7	94.1	97.2	97.0	96.2	46.3	47.7	47.3
Vim	10m	69.3	*71.8*	82.6	82.1	81.0	*77.3*	77.2	77.3	77.3	76.1	75.9
	20m	67.7	*69.6*	79.4	79.4	77.2	76.1	76.5	75.4	78.1	77.8	75.8
YAFFS2	10m	74.3	*74.5*	69.7	67.0	65.0	*73.1*	*73.3*	71.2	55.9	57.5	55.7
	20m	75.1	*75.6*	68.7	67.5	65.7	73.5	73.0	70.8	54.8	55.6	54.7
After statement reduction												
Sed	10m	75.9	95.2	79.5	*77.0*	*78.0*	80.6	79.7	*78.8*	19.1	18.7	18.5
	20m	77.9	95.0	86.8	86.5	81.7	81.5	83.3	*79.3*	36.4	31.8	32.0
Space	10m	67.1	76.1	70.1	69.7	64.5	80.5	79.3	78.0	34.8	36.8	37.8
	20m	69.3	75.1	71.0	72.0	67.4	83.0	81.8	80.5	34.4	36.7	37.6
Grep	10m	64.2	81.7	89.5	85.6	84.0	84.0	82.9	83.7	20.6	19.5	16.9
	20m	67.7	84.2	89.9	84.7	86.4	84.7	82.3	84.6	25.5	23.1	19.3
Gzip	10m	83.4	93.4	96.2	96.2	95.1	96.6	96.4	96.0	61.0	61.3	62.8
	20m	86.0	93.0	96.4	96.4	95.7	96.8	96.4	96.5	65.1	65.0	65.1
Vim	10m	69.9	*71.9*	80.3	81.5	80.3	78.1	78.6	78.8	78.0	77.3	78.8
	20m	68.2	*65.8*	80.4	81.0	79.9	78.0	78.1	77.4	80.6	80.5	79.4
YAFFS2	10m	74.8	*75.8*	69.7	66.6	63.9	*73.6*	*73.6*	71.5	54.6	55.6	54.3
	20m	74.7	*75.5*	69.1	67.5	64.7	*73.4*	*73.1*	70.5	57.2	57.8	56.5
After branch reduction												
Sed	10m	74.4	95.5	78.8	*75.3*	*76.5*	79.7	79.7	78.2	18.3	17.5	17.4
	20m	74.3	95.7	84.3	83.1	78.7	77.6	79.5	*76.7*	24.8	23.3	25.1
Space	10m	81.0	95.9	88.1	89.6	84.8	85.4	85.0	84.3	49.0	51.8	52.7
	20m	86.9	98.1	91.5	94.5	90.6	90.8	88.6	88.7	46.2	46.3	45.7
Grep	10m	63.4	84.0	90.0	86.2	85.1	86.0	83.8	85.1	16.7	14.9	13.3
	20m	64.3	87.3	88.9	83.5	86.0	83.5	81.1	85.1	17.4	17.4	15.5
Gzip	10m	83.1	93.4	96.2	96.2	95.0	96.6	96.4	96.2	61.1	61.7	63.4
	20m	77.4	93.8	95.8	96.0	93.7	96.5	96.2	95.5	50.6	52.0	51.8
Vim	10m	70.4	*70.6*	82.9	82.8	82.2	79.3	80.7	80.3	78.0	78.0	77.9
	20m	68.5	*67.6*	81.4	82.7	80.9	80.3	80.3	78.3	81.3	80.7	79.2
YAFFS2	10m	74.1	*74.1*	*73.5*	69.9	67.7	*73.0*	*73.1*	71.6	56.4	57.5	55.6
	20m	75.8	77.7	71.4	72.2	68.6	*76.4*	*75.5*	73.5	63.8	64.4	61.5
After AIMP reduction												
Sed	10m	72.1	94.8	80.0	76.2	77.4	80.8	77.2	78.3	17.3	16.5	16.5
	20m	71.8	94.7	80.3	77.7	79.6	79.4	79.9	78.6	17.8	17.2	16.9
Space	10m	81.0	95.9	88.1	89.6	84.8	85.4	84.9	84.2	48.9	51.7	52.5
	20m	84.3	97.6	88.0	91.0	88.6	87.5	86.9	85.4	46.7	47.0	45.6
Grep	10m	73.9	88.6	89.9	90.5	85.6	88.8	88.1	85.4	49.0	49.2	45.2
	20m	74.5	86.5	88.5	89.2	87.7	88.2	86.9	87.2	48.2	45.2	42.6
Gzip	10m	78.7	93.7	95.2	95.5	93.1	95.6	95.8	94.9	54.1	54.8	57.4
	20m	77.5	93.9	95.9	96.0	93.8	96.6	96.3	95.6	51.7	53.4	53.8
Vim	10m	70.5	73.4	80.6	80.8	81.8	77.4	78.4	77.3	75.6	74.3	75.4
	20m	68.8	*70.6*	81.9	82.7	81.9	79.4	79.8	79.0	80.5	80.7	78.3
YAFFS2	10m	73.8	71.4	*72.9*	68.8	67.6	72.5	71.7	71.0	57.2	57.7	57.1
	20m	76.0	78.4	72.2	72.8	69.3	*76.4*	*75.4*	74.1	65.2	65.6	62.5
The best results		2	15	8	6	2	10	1	0	0	0	0
Win random		-	35	40	38	36	40	40	37	8	8	8
Lost to random		-	1	6	8	10	2	2	8	40	40	40

Table A.2: APBD values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (percentage %) (Search strategy: Random Path)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	92.9	92.2	*93.2*	*92.7*	*92.5*	*93.0*	92.2	91.6	84.7	84.9	84.4
	20m	94.1	92.5	94.9	*94.6*	*94.3*	*94.3*	*94.2*	*94.6*	87.9	88.1	87.6
Space	10m	99.5	99.3	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
	20m	99.5	99.3	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
Grep	10m	97.0	96.1	97.8	97.9	97.7	97.8	97.7	97.7	94.7	94.7	94.8
	20m	94.7	*95.3*	96.3	96.2	96.1	95.8	95.2	95.8	91.5	91.1	91.6
Gzip	10m	80.9	94.6	94.0	93.7	92.1	94.2	94.1	94.0	33.3	32.7	33.3
	20m	80.6	94.5	93.7	94.2	93.3	93.8	94.6	94.7	32.5	33.4	33.5
Vim	10m	86.9	85.6	96.1	96.3	95.9	95.1	95.4	95.2	95.2	95.4	95.2
	20m	87.6	84.8	95.2	95.3	95.7	94.4	94.0	94.5	95.2	94.9	94.6
YAFFS2	10m	91.9	92.4	92.7	93.0	93.3	92.8	92.9	93.0	92.4	92.9	*92.4*
	20m	92.1	93.2	93.4	93.4	93.6	93.1	93.2	93.5	92.8	93.0	92.9
After statement reduction												
Sed	10m	94.4	93.0	94.8	*94.6*	94.0	*94.3*	*94.8*	94.9	80.9	81.4	81.2
	20m	94.6	93.2	95.2	95.2	*94.9*	95.2	95.4	95.3	86.2	86.3	85.4
Space	10m	98.4	97.4	98.3	98.0	98.3	98.2	98.3	98.4	98.4	98.1	98.7
	20m	92.5	95.1	92.1	93.4	91.3	96.7	96.3	95.5	87.3	88.7	87.5
Grep	10m	91.8	93.9	95.6	94.4	96.0	96.0	95.6	96.1	83.6	82.7	84.4
	20m	85.7	89.6	94.6	90.8	93.9	95.2	94.2	95.2	73.1	73.8	73.0
Gzip	10m	81.1	93.6	91.6	91.1	90.5	91.6	91.5	92.0	38.0	37.4	37.8
	20m	80.8	93.6	92.0	92.6	92.1	91.7	92.6	93.0	37.7	38.4	38.7
Vim	10m	84.4	*83.1*	95.3	95.5	95.4	94.5	94.8	94.9	94.0	94.5	94.4
	20m	83.8	79.1	91.9	93.3	92.6	90.6	91.3	90.6	91.8	91.4	88.3
YAFFS2	10m	92.4	93.0	93.4	93.3	93.6	92.9	93.1	93.3	92.8	93.1	*92.7*
	20m	92.3	93.7	93.3	93.3	93.5	93.1	93.4	93.3	*92.6*	*92.6*	*92.3*
After branch reduction												
Sed	10m	94.1	93.3	94.6	*94.2*	93.7	*94.1*	*94.4*	*94.4*	80.2	80.7	80.5
	20m	94.3	92.8	95.1	95.2	94.9	95.2	95.5	95.5	85.9	85.9	85.1
Space	10m	99.5	99.3	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
	20m	99.5	99.3	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
Grep	10m	91.9	94.1	95.7	94.5	96.1	96.3	96.1	96.5	83.8	82.9	85.0
	20m	85.6	89.5	94.5	90.6	94.0	95.1	94.3	95.3	73.1	73.8	73.2
Gzip	10m	80.9	93.6	91.4	90.9	90.2	91.3	91.3	91.8	38.2	37.7	38.1
	20m	80.1	94.1	93.3	93.9	93.1	93.6	94.7	94.5	36.6	37.5	37.7
Vim	10m	85.7	*85.0*	96.1	96.1	95.7	95.2	95.3	95.1	94.9	95.1	94.8
	20m	85.9	82.9	94.9	94.1	94.1	93.8	92.6	92.4	94.3	92.7	90.2
YAFFS2	10m	92.4	93.1	93.4	93.4	93.6	93.1	93.3	93.6	92.9	93.3	*92.8*
	20m	92.0	93.3	93.1	93.3	93.3	92.9	93.6	93.2	92.7	93.1	*92.4*
After AIMP reduction												
Sed	10m	93.1	*92.3*	*93.4*	*92.7*	92.2	*93.3*	92.6	91.9	83.5	83.7	83.2
	20m	94.3	93.3	95.1	95.0	*94.3*	*94.2*	*94.3*	*94.5*	85.6	85.8	85.6
Space	10m	99.5	99.3	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
	20m	99.5	99.3	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5	99.5
Grep	10m	96.8	96.1	97.8	97.8	97.7	97.9	97.7	97.8	94.7	94.7	94.8
	20m	95.8	*95.8*	97.3	97.1	97.0	97.0	96.7	96.7	92.9	92.4	92.5
Gzip	10m	77.7	93.8	86.7	86.0	83.4	85.3	86.4	84.6	38.4	38.1	38.5
	20m	81.0	94.1	93.6	94.1	93.3	93.8	94.4	94.5	36.4	37.2	37.5
Vim	10m	86.6	*85.6*	95.8	96.1	95.7	94.7	95.2	94.7	94.6	95.0	94.5
	20m	86.9	83.6	94.8	94.2	94.2	93.8	92.5	92.2	94.6	93.0	90.4
YAFFS2	10m	92.4	*92.9*	93.3	93.4	93.6	*92.7*	93.0	93.4	*92.8*	93.2	*92.7*
	20m	92.2	93.2	93.4	93.6	93.5	93.1	93.7	93.4	92.9	93.5	92.7
The best results		3	6	7	6	6	6	4	7	0	0	1
Win random		-	20	41	39	35	37	38	40	18	18	14
Lost to random		-	21	5	1	6	1	3	2	25	26	25

Table A.3: APBD values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (percentage %) (Search strategy: Random State)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	86.0	91.0	93.1	90.9	92.5	91.9	90.3	89.0	71.6	72.3	71.8
	20m	84.5	92.1	91.9	89.9	90.4	90.7	89.1	87.9	69.2	68.6	68.8
Space	10m	85.8	99.4	98.4	99.0	98.7	90.4	91.7	87.9	75.3	75.2	73.2
	20m	85.7	99.4	98.6	98.8	98.4	88.7	90.3	84.5	71.2	70.3	66.7
Grep	10m	92.3	90.4	95.4	95.5	95.2	95.8	95.4	95.8	82.6	82.2	80.9
	20m	92.1	90.8	95.5	95.8	95.4	95.8	95.6	96.1	82.9	82.0	81.1
Gzip	10m	81.9	94.3	96.8	96.7	95.4	96.2	96.7	96.4	40.9	41.2	41.5
	20m	80.8	94.4	96.9	96.8	95.5	96.2	96.6	96.4	39.8	40.5	40.5
Vim	10m	76.1	83.9	89.6	89.1	87.9	83.8	83.1	84.7	83.4	81.7	79.6
	20m	81.8	*82.6*	92.2	92.3	91.5	89.6	89.6	89.6	90.0	88.4	86.9
YAFFS2	10m	93.3	*93.3*	93.9	94.0	94.0	93.8	93.9	93.7	*93.1*	*93.5*	92.6
	20m	93.7	94.5	94.5	94.7	94.6	94.3	94.5	94.6	*93.7*	*93.6*	*93.5*
After statement reduction												
Sed	10m	85.1	93.1	89.8	88.8	87.8	88.9	90.4	88.8	63.3	63.9	63.0
	20m	87.2	92.8	92.3	92.1	90.3	90.5	91.5	90.9	66.8	66.0	66.2
Space	10m	82.7	91.0	85.6	86.0	80.0	88.2	86.8	85.8	46.9	47.5	46.6
	20m	81.1	89.9	84.0	84.6	80.0	88.4	86.9	85.6	45.0	46.5	45.8
Grep	10m	79.6	85.4	92.4	90.2	90.1	91.4	90.4	91.8	55.3	54.0	54.1
	20m	78.2	85.6	92.3	89.2	91.2	91.5	90.9	92.4	52.2	51.1	50.6
Gzip	10m	82.3	94.1	96.1	96.1	94.8	96.0	96.3	95.9	44.6	44.5	44.6
	20m	81.0	93.9	96.5	96.1	95.1	95.8	96.0	95.9	43.4	44.1	44.4
Vim	10m	76.3	82.0	89.4	89.7	89.4	85.1	85.1	86.4	85.5	84.4	84.0
	20m	80.4	*80.8*	91.8	92.1	91.2	89.4	89.4	89.5	89.7	88.2	87.0
YAFFS2	10m	90.8	91.8	93.2	92.8	93.1	*90.9*	*90.3*	92.0	89.9	*90.1*	*90.6*
	20m	90.6	91.4	93.0	93.1	93.2	*90.7*	*91.0*	91.8	*90.8*	*90.7*	*91.5*
After branch reduction												
Sed	10m	84.6	92.8	88.8	87.6	86.4	87.7	89.4	87.8	62.5	63.0	62.0
	20m	86.8	92.9	92.0	91.6	89.4	89.6	90.9	90.1	66.8	66.2	66.1
Space	10m	86.3	97.5	93.2	92.6	87.6	88.6	86.9	86.0	51.9	51.5	50.2
	20m	86.5	98.1	93.2	92.4	89.7	87.7	86.3	84.1	50.0	48.4	46.4
Grep	10m	80.1	85.9	92.9	90.4	90.5	92.1	90.9	92.0	56.2	54.6	54.6
	20m	78.7	86.2	92.0	88.8	90.6	91.7	91.2	92.7	53.9	53.0	52.3
Gzip	10m	84.2	93.8	94.1	93.9	91.6	94.7	94.9	93.7	50.7	50.9	51.2
	20m	81.0	94.1	96.8	96.5	95.3	96.1	96.5	96.1	43.4	44.2	44.3
Vim	10m	77.2	82.6	90.0	90.3	89.9	86.1	86.0	87.2	86.5	85.6	85.0
	20m	80.9	*81.8*	91.8	91.9	91.1	89.2	89.2	89.2	89.8	88.0	86.6
YAFFS2	10m	90.9	91.9	93.2	92.8	93.1	*90.9*	*90.3*	92.0	89.9	90.0	*90.6*
	20m	90.5	92.3	93.0	93.1	93.1	*90.9*	*91.3*	92.4	91.4	*91.2*	91.9
After AIMP reduction												
Sed	10m	85.2	91.8	89.0	87.1	87.1	89.3	88.2	87.9	63.4	63.2	62.1
	20m	85.0	92.6	89.5	88.8	88.2	88.5	89.4	88.6	63.7	63.3	63.3
Space	10m	84.7	97.8	94.0	93.3	88.8	88.2	86.8	85.3	53.4	52.4	51.6
	20m	86.5	98.1	93.2	92.4	89.7	87.7	86.3	84.1	50.0	48.4	46.4
Grep	10m	92.1	90.0	94.8	95.1	94.9	95.1	94.9	95.4	82.9	82.6	81.7
	20m	92.0	90.6	95.0	95.3	95.1	95.3	95.2	95.9	83.1	82.2	81.5
Gzip	10m	85.1	93.4	94.7	95.0	92.1	95.1	95.2	93.9	55.3	55.7	56.6
	20m	81.8	93.9	96.9	96.8	95.7	96.2	96.6	96.2	44.2	44.8	45.0
Vim	10m	75.9	85.0	88.9	89.1	88.4	84.1	83.9	84.4	83.9	83.0	81.5
	20m	82.2	83.7	92.4	92.3	91.6	90.2	89.9	89.6	90.0	88.3	87.0
YAFFS2	10m	91.1	92.6	93.6	93.5	93.7	*91.0*	*90.5*	92.6	90.2	*90.5*	*91.0*
	20m	91.5	92.2	93.6	93.5	93.4	*91.1*	*91.2*	*91.8*	90.9	90.7	*91.4*
The best results		0	15	13	9	2	0	3	6	0	0	0
Win random		-	40	48	48	46	42	40	43	9	8	9
Lost to random		-	4	0	0	2	0	2	4	36	34	33

Table A.4: APBD values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (percentage %) (Search strategy: Min Distance to Uncovered)

Subject	Time	Random	Shortest	FPF			CA			CT		
			path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	88.3	91.2	94.1	92.0	93.6	93.9	92.5	91.3	76.3	77.0	75.9
	20m	85.1	91.6	93.0	91.3	91.6	91.6	90.2	89.4	73.6	72.7	72.9
Space	10m	85.8	99.4	98.4	99.0	98.7	90.4	91.8	88.0	75.3	75.3	73.3
	20m	85.7	99.4	98.6	98.8	98.4	88.7	90.3	84.5	71.2	70.3	66.7
Grep	10m	92.8	*92.9*	96.0	95.9	96.0	95.7	95.3	95.9	84.9	84.4	82.0
	20m	93.1	92.3	96.2	96.2	96.0	95.8	95.4	95.8	85.0	84.2	82.4
Gzip	10m	80.6	94.3	95.9	95.9	93.8	96.1	96.3	95.5	38.2	38.1	39.1
	20m	79.5	94.5	96.2	96.4	94.7	96.0	96.3	95.8	34.5	35.6	35.4
Vim	10m	76.9	84.1	89.3	88.7	87.5	84.4	83.6	84.1	84.0	82.0	79.9
	20m	81.2	83.7	91.0	90.4	89.8	88.0	87.5	88.0	88.5	86.5	85.7
YAFFS2	10m	93.4	93.8	94.5	94.8	94.5	93.7	93.9	94.0	92.8	*93.4*	92.7
	20m	93.5	94.0	94.7	94.8	94.7	93.9	94.1	94.3	*93.3*	*93.3*	*93.2*
After statement reduction												
Sed	10m	85.8	92.4	89.8	89.1	87.8	89.6	91.7	88.3	67.9	68.9	66.8
	20m	86.9	93.0	92.5	92.2	90.3	90.3	91.4	90.8	66.5	65.3	65.9
Space	10m	78.7	86.0	83.1	82.5	77.8	86.3	85.0	84.0	47.5	47.3	47.5
	20m	77.6	83.3	79.7	79.2	75.3	85.6	84.5	82.2	43.8	44.6	45.0
Grep	10m	79.8	86.1	92.8	90.8	90.7	92.0	91.2	92.4	52.2	50.8	50.5
	20m	78.2	86.4	92.8	89.3	91.3	92.0	91.4	92.7	50.4	49.8	49.1
Gzip	10m	81.3	94.3	96.4	96.3	94.6	96.3	96.4	95.8	41.6	41.5	42.0
	20m	78.6	94.3	95.9	95.5	93.6	95.3	95.3	95.1	37.8	38.7	39.1
Vim	10m	76.8	81.9	89.0	89.8	89.5	85.0	85.9	87.0	86.1	85.8	84.7
	20m	79.7	*81.0*	90.3	90.6	89.9	88.1	88.1	88.0	88.3	86.9	85.9
YAFFS2	10m	90.5	92.2	93.2	93.3	93.6	*90.4*	*90.4*	92.6	*90.4*	*90.9*	*91.5*
	20m	89.9	90.7	92.4	93.1	93.2	*90.0*	*90.7*	91.9	*90.3*	*90.7*	91.2
After branch reduction												
Sed	10m	85.0	92.4	89.4	89.0	87.2	88.3	90.8	88.3	65.6	67.0	65.0
	20m	86.8	92.7	91.9	91.7	89.3	89.8	91.4	90.0	66.7	65.4	66.0
Space	10m	84.4	98.7	92.1	91.8	87.2	88.1	86.8	85.9	53.4	52.9	51.4
	20m	86.0	97.8	93.1	92.3	89.4	87.7	86.2	84.0	49.8	48.1	46.2
Grep	10m	80.8	86.5	93.4	91.0	91.2	93.0	91.9	92.5	55.7	54.2	53.9
	20m	79.0	87.2	93.5	89.7	92.1	92.6	92.0	93.0	53.9	53.1	51.8
Gzip	10m	80.0	94.2	92.8	92.8	91.4	93.3	93.7	93.4	41.9	42.1	42.3
	20m	77.5	94.5	95.6	94.0	93.2	95.2	94.5	94.7	36.4	37.2	37.3
Vim	10m	77.0	84.1	89.5	89.3	88.9	86.0	85.5	86.2	86.5	85.5	84.4
	20m	80.6	*81.1*	90.9	90.9	90.3	88.5	88.4	88.5	89.5	87.7	86.9
YAFFS2	10m	91.1	92.8	93.4	93.6	93.6	*91.0*	*91.4*	93.2	*91.6*	*92.0*	92.2
	20m	89.9	91.3	92.6	93.3	93.1	90.7	91.4	92.4	91.2	91.4	91.4
After AIMP reduction												
Sed	10m	86.1	90.8	90.1	89.3	88.7	90.0	90.4	89.9	69.4	70.8	68.6
	20m	85.8	92.4	90.3	89.7	88.9	89.6	90.3	89.2	66.3	65.7	65.8
Space	10m	82.7	97.1	90.9	93.0	87.4	87.5	86.5	84.8	53.3	52.9	51.5
	20m	86.2	98.4	93.1	92.7	90.1	87.2	85.9	83.7	50.3	48.7	46.5
Grep	10m	92.7	*92.4*	96.0	96.1	95.9	95.6	95.2	95.4	85.0	84.7	82.7
	20m	92.7	*92.2*	95.6	95.6	95.5	95.4	94.7	95.2	84.9	84.1	82.4
Gzip	10m	83.1	93.5	92.9	93.7	91.6	93.3	94.1	93.0	49.0	49.7	50.2
	20m	79.4	94.1	96.4	96.4	94.6	95.8	96.2	95.5	37.6	38.5	38.3
Vim	10m	76.5	83.9	88.9	88.5	87.9	84.0	83.1	83.9	84.0	82.6	80.2
	20m	81.5	83.3	90.6	90.8	90.0	88.4	88.1	88.1	88.7	87.0	85.8
YAFFS2	10m	90.6	92.5	93.2	93.3	93.4	*90.1*	*90.3*	92.4	*90.2*	*90.8*	*90.9*
	20m	91.1	92.1	93.4	93.7	93.5	*90.9*	*91.1*	*91.8*	*90.8*	*91.0*	*91.3*
The best results		0	13	14	11	5	2	3	0	0	0	0
Win random		-	42	48	48	46	43	42	44	9	9	11
Lost to random		-	1	0	0	2	0	1	3	33	32	33

Table A.5: Cohen's *d* effect sizes of APBD values of different prioritization methods and random ordering on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Depth First Search)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	2.4	1.3	0.6	1.0	1.4	1.0	0.7	-3.6	-3.5	-3.7
	20m	-	2.3	1.2	0.7	1.0	1.1	0.9	0.6	-2.1	-2.3	-2.1
Space	10m	-	0.8	0.6	0.7	0.8	0.3	0.4	0.2	-0.3	-0.3	-0.3
	20m	-	0.8	0.6	0.7	0.8	0.2	0.3	0.0	-0.4	-0.4	-0.5
Grep	10m	-	1.3	1.4	1.2	1.1	1.2	1.0	1.2	-1.7	-1.8	-1.9
	20m	-	1.5	1.2	1.3	1.2	1.5	1.3	1.0	-1.8	-2.0	-2.1
Gzip	10m	-	1.7	1.9	1.9	1.6	1.9	1.9	1.9	-1.9	-1.8	-1.6
	20m	-	1.7	1.8	1.9	1.6	1.9	1.9	1.8	-1.9	-1.8	-1.9
Vim	10m	-	0.2	1.2	1.1	1.0	0.6	0.6	0.6	0.6	0.5	0.5
	20m	-	0.2	1.2	1.2	1.0	0.8	0.9	0.8	1.1	1.1	0.8
YAFFS2	10m	-	0.0	-0.6	-0.9	-1.1	-0.2	-0.1	-0.4	-2.2	-2.0	-2.2
	20m	-	0.1	-0.9	-1.0	-1.2	-0.2	-0.3	-0.6	-2.6	-2.7	-2.7
After statement reduction												
Sed	10m	-	2.0	0.3	0.1	0.2	0.4	0.3	0.2	-5.8	-5.8	-5.9
	20m	-	1.7	0.7	0.7	0.3	0.3	0.5	0.1	-2.2	-2.6	-2.5
Space	10m	-	0.8	0.3	0.2	-0.2	1.5	1.3	1.2	-3.3	-3.1	-2.9
	20m	-	0.6	0.2	0.3	-0.2	1.6	1.4	1.2	-3.9	-3.6	-3.4
Grep	10m	-	1.0	1.7	1.4	1.3	1.1	1.1	1.2	-2.9	-3.0	-3.2
	20m	-	1.2	2.1	1.5	1.7	1.4	1.1	1.4	-3.6	-3.8	-4.2
Gzip	10m	-	1.7	2.2	2.1	1.9	2.2	2.2	2.1	-2.6	-2.4	-2.3
	20m	-	1.3	1.9	1.9	1.8	2.0	1.9	1.9	-2.7	-2.7	-2.7
Vim	10m	-	0.2	0.8	1.0	0.8	0.6	0.7	0.7	0.6	0.6	0.8
	20m	-	-0.2	1.1	1.1	1.1	0.8	0.8	0.8	1.2	1.1	1.0
YAFFS2	10m	-	0.2	-0.7	-1.1	-1.3	-0.2	-0.2	-0.5	-2.4	-2.4	-2.4
	20m	-	0.1	-0.7	-1.0	-1.3	-0.2	-0.2	-0.6	-2.2	-2.3	-2.3
After branch reduction												
Sed	10m	-	2.1	0.4	0.1	0.2	0.5	0.4	0.3	-5.5	-5.5	-5.6
	20m	-	2.0	0.8	0.7	0.4	0.3	0.4	0.2	-3.8	-4.0	-3.6
Space	10m	-	1.1	0.5	0.6	0.2	0.4	0.3	0.3	-1.8	-1.5	-1.5
	20m	-	1.2	0.4	0.7	0.3	0.4	0.2	0.2	-3.1	-3.1	-2.8
Grep	10m	-	1.1	1.7	1.4	1.3	1.2	1.1	1.2	-2.8	-3.0	-3.1
	20m	-	1.5	1.8	1.4	1.6	1.2	1.0	1.5	-3.6	-3.4	-3.4
Gzip	10m	-	1.8	2.2	2.2	2.0	2.3	2.3	2.2	-2.5	-2.3	-2.2
	20m	-	1.7	1.9	1.9	1.6	1.9	1.9	1.8	-1.9	-1.8	-1.9
Vim	10m	-	0.0	1.2	1.2	1.1	0.8	1.0	0.9	0.6	0.6	0.7
	20m	-	-0.1	1.3	1.5	1.3	1.1	1.1	0.9	1.3	1.3	1.1
YAFFS2	10m	-	-0.0	-0.1	-0.5	-0.7	-0.1	-0.1	-0.3	-2.2	-2.1	-2.2
	20m	-	0.3	-0.6	-0.5	-0.9	0.1	-0.0	-0.3	-1.3	-1.3	-1.6
After AIMP reduction												
Sed	10m	-	2.2	0.6	0.3	0.4	0.7	0.4	0.5	-5.4	-5.4	-5.5
	20m	-	2.1	0.7	0.5	0.6	0.6	0.7	0.6	-5.0	-5.1	-5.1
Space	10m	-	1.1	0.5	0.6	0.2	0.4	0.3	0.2	-1.8	-1.5	-1.5
	20m	-	1.2	0.3	0.5	0.3	0.3	0.2	0.1	-2.5	-2.5	-2.5
Grep	10m	-	1.2	1.3	1.4	0.8	1.2	1.2	0.9	-1.5	-1.4	-1.7
	20m	-	1.0	1.2	1.3	1.1	1.1	1.0	1.1	-1.6	-1.8	-2.1
Gzip	10m	-	1.7	1.9	1.9	1.6	1.9	1.9	1.8	-1.9	-1.8	-1.6
	20m	-	1.6	1.8	1.8	1.6	1.9	1.9	1.8	-1.7	-1.6	-1.6
Vim	10m	-	0.2	0.8	0.9	1.0	0.6	0.7	0.6	0.4	0.3	0.4
	20m	-	0.1	1.2	1.3	1.3	0.9	1.0	1.0	1.1	1.2	0.9
YAFFS2	10m	-	-0.3	-0.1	-0.6	-0.7	-0.2	-0.3	-0.4	-2.1	-2.0	-2.0
	20m	-	0.3	-0.5	-0.4	-0.8	0.1	-0.1	-0.3	-1.2	-1.3	-1.5
The best results		0	20	9	5	2	10	1	0	1	0	0

Table A.6: Cohen's *d* effect sizes of APBD values of different prioritization methods and random ordering on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Random Path)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	-0.3	0.1	-0.1	-0.2	0.1	-0.3	-0.6	-3.9	-3.7	-3.7
	20m	-	-0.6	0.4	0.2	0.1	0.1	0.1	0.2	-2.7	-2.8	-3.0
Space	10m	-	-0.8	0.0	0.1	-0.1	0.2	0.2	0.2	0.2	0.2	0.2
	20m	-	-0.9	-0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Grep	10m	-	-0.7	0.9	1.0	0.7	0.9	0.7	0.8	-1.9	-2.0	-1.9
	20m	-	0.3	0.8	0.8	0.7	0.5	0.2	0.5	-1.1	-1.2	-1.1
Gzip	10m	-	1.9	1.7	1.6	1.3	1.7	1.7	1.6	-6.2	-6.3	-6.3
	20m	-	2.1	1.7	1.9	1.7	1.8	2.0	2.0	-7.1	-6.9	-6.9
Vim	10m	-	-0.2	1.5	1.5	1.5	1.3	1.4	1.3	1.3	1.4	1.3
	20m	-	-0.3	1.0	1.0	1.2	0.9	0.8	0.9	1.1	1.1	1.0
YAFFS2	10m	-	0.2	0.4	0.5	0.7	0.4	0.4	0.5	0.2	0.5	0.2
	20m	-	0.6	0.6	0.6	0.7	0.4	0.5	0.6	0.3	0.4	0.4
After statement reduction												
Sed	10m	-	-0.7	0.3	0.1	-0.2	-0.1	0.3	0.3	-5.4	-6.0	-6.1
	20m	-	-0.6	0.4	0.4	0.2	0.4	0.5	0.5	-3.5	-3.4	-4.2
Space	10m	-	-0.2	-0.0	-0.1	-0.0	-0.0	-0.0	0.0	0.0	-0.1	0.1
	20m	-	0.3	-0.0	0.1	-0.1	0.5	0.4	0.3	-0.4	-0.3	-0.4
Grep	10m	-	0.5	0.9	0.6	1.0	1.0	0.9	1.0	-1.3	-1.4	-1.1
	20m	-	0.5	1.1	0.5	1.1	1.3	1.1	1.3	-1.0	-0.9	-1.0
Gzip	10m	-	1.8	1.3	1.2	1.1	1.3	1.3	1.3	-5.8	-6.0	-6.0
	20m	-	2.0	1.4	1.6	1.4	1.5	1.6	1.6	-6.4	-6.2	-6.1
Vim	10m	-	-0.1	1.4	1.4	1.5	1.3	1.3	1.4	1.2	1.3	1.3
	20m	-	-0.4	0.7	1.0	0.8	0.6	0.7	0.6	0.8	0.7	0.4
YAFFS2	10m	-	0.3	0.5	0.5	0.6	0.3	0.3	0.4	0.2	0.3	0.2
	20m	-	0.7	0.5	0.5	0.6	0.4	0.5	0.5	0.1	0.1	-0.0
After branch reduction												
Sed	10m	-	-0.5	0.3	0.1	-0.3	-0.0	0.2	0.2	-5.6	-6.0	-6.4
	20m	-	-0.6	0.5	0.5	0.3	0.5	0.7	0.7	-3.5	-3.5	-4.3
Space	10m	-	-0.8	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
	20m	-	-0.9	-0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Grep	10m	-	0.5	0.9	0.5	1.0	1.0	1.0	1.0	-1.2	-1.4	-1.0
	20m	-	0.5	1.1	0.5	1.1	1.3	1.2	1.3	-1.0	-0.9	-1.0
Gzip	10m	-	1.8	1.3	1.2	1.1	1.3	1.3	1.3	-5.8	-5.9	-5.9
	20m	-	2.1	1.7	1.9	1.7	1.8	2.1	2.0	-6.3	-6.0	-6.1
Vim	10m	-	-0.1	1.7	1.7	1.6	1.5	1.5	1.5	1.4	1.5	1.4
	20m	-	-0.3	1.1	1.0	1.0	0.9	0.7	0.7	1.1	0.8	0.4
YAFFS2	10m	-	0.3	0.5	0.5	0.6	0.3	0.4	0.5	0.2	0.4	0.2
	20m	-	0.6	0.5	0.6	0.6	0.4	0.8	0.6	0.3	0.5	0.2
After AIMP reduction												
Sed	10m	-	-0.3	0.2	-0.1	-0.4	0.1	-0.2	-0.6	-4.3	-4.1	-4.1
	20m	-	-0.4	0.4	0.4	0.0	-0.0	0.0	0.1	-4.6	-4.7	-4.5
Space	10m	-	-0.8	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
	20m	-	-0.9	-0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Grep	10m	-	-0.6	1.0	1.0	0.8	1.1	0.8	1.0	-1.7	-1.8	-1.7
	20m	-	0.0	1.2	1.0	0.9	0.9	0.6	0.7	-1.8	-2.0	-1.9
Gzip	10m	-	2.0	0.8	0.8	0.5	0.8	0.9	0.7	-4.5	-4.6	-4.6
	20m	-	2.1	1.8	1.9	1.8	1.9	2.0	2.0	-6.9	-6.7	-6.7
Vim	10m	-	-0.1	1.6	1.6	1.6	1.4	1.5	1.4	1.4	1.4	1.3
	20m	-	-0.3	1.0	0.9	0.9	0.8	0.6	0.6	1.0	0.7	0.3
YAFFS2	10m	-	0.2	0.5	0.5	0.6	0.2	0.3	0.5	0.2	0.4	0.2
	20m	-	0.5	0.6	0.7	0.7	0.4	0.7	0.6	0.4	0.7	0.3
The best results		0	9	7	8	8	5	3	5	2	0	1

Table A.7: Cohen's *d* effect sizes of APBD values of different prioritization methods and random ordering on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Random State)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	1.3	1.8	1.1	1.5	1.6	1.1	0.8	-3.3	-2.9	-3.0
	20m	-	1.4	1.2	0.9	0.9	1.1	0.8	0.6	-2.1	-2.2	-2.2
Space	10m	-	0.8	0.7	0.7	0.7	0.2	0.3	0.1	-0.3	-0.3	-0.4
	20m	-	0.8	0.7	0.7	0.7	0.1	0.2	-0.1	-0.4	-0.4	-0.5
Grep	10m	-	-0.8	1.7	1.8	1.4	1.9	1.6	2.0	-3.3	-3.6	-4.1
	20m	-	-0.4	1.5	1.7	1.4	1.7	1.6	1.9	-2.9	-3.5	-3.5
Gzip	10m	-	1.8	2.2	2.2	2.0	2.1	2.2	2.1	-5.5	-5.4	-5.4
	20m	-	2.1	2.5	2.4	2.2	2.4	2.4	2.4	-5.7	-5.6	-5.6
Vim	10m	-	0.9	1.9	1.9	1.6	1.0	0.9	1.2	0.9	0.7	0.4
	20m	-	0.1	2.1	2.0	1.9	1.4	1.4	1.5	1.5	1.1	0.9
YAFFS2	10m	-	0.0	0.3	0.4	0.4	0.3	0.3	0.2	-0.1	0.1	-0.4
	20m	-	0.6	0.6	0.7	0.7	0.4	0.6	0.6	0.0	-0.0	-0.1
After statement reduction												
Sed	10m	-	1.7	0.8	0.6	0.5	0.7	1.0	0.6	-3.9	-3.7	-4.0
	20m	-	1.3	1.1	1.0	0.6	0.7	1.0	0.8	-4.4	-4.9	-4.8
Space	10m	-	0.9	0.3	0.3	-0.2	0.6	0.5	0.3	-2.3	-2.4	-2.4
	20m	-	0.8	0.2	0.3	-0.1	0.7	0.6	0.4	-2.3	-2.2	-2.3
Grep	10m	-	0.7	1.8	1.4	1.4	1.6	1.5	1.8	-2.6	-3.0	-2.7
	20m	-	0.9	2.0	1.5	1.8	1.8	1.8	2.0	-2.9	-3.0	-3.0
Gzip	10m	-	2.0	2.4	2.4	2.1	2.4	2.4	2.3	-5.8	-5.6	-5.7
	20m	-	2.0	2.4	2.3	2.1	2.3	2.3	2.3	-5.0	-4.9	-4.7
Vim	10m	-	0.7	1.9	2.0	2.0	1.2	1.2	1.4	1.2	1.1	1.0
	20m	-	0.1	2.0	2.1	1.9	1.5	1.5	1.6	1.6	1.3	1.1
YAFFS2	10m	-	0.3	0.7	0.6	0.7	0.0	-0.1	0.3	-0.2	-0.2	-0.1
	20m	-	0.3	0.8	0.9	0.9	0.0	0.1	0.4	0.1	0.0	0.3
After branch reduction												
Sed	10m	-	1.6	0.7	0.5	0.3	0.5	0.8	0.6	-3.6	-3.5	-3.7
	20m	-	1.4	1.0	0.9	0.5	0.6	0.8	0.7	-3.9	-4.2	-4.1
Space	10m	-	1.3	0.8	0.7	0.1	0.2	0.1	-0.0	-1.7	-1.7	-1.7
	20m	-	1.2	0.6	0.5	0.3	0.1	-0.0	-0.2	-1.7	-1.8	-1.8
Grep	10m	-	0.6	1.8	1.4	1.4	1.6	1.4	1.6	-2.4	-2.8	-2.5
	20m	-	0.9	1.9	1.3	1.6	1.7	1.7	2.0	-2.7	-2.8	-2.8
Gzip	10m	-	1.7	1.4	1.3	0.9	1.6	1.6	1.4	-4.9	-4.8	-4.9
	20m	-	2.1	2.6	2.5	2.3	2.4	2.5	2.4	-5.3	-5.3	-5.2
Vim	10m	-	0.6	2.0	2.0	1.9	1.2	1.2	1.4	1.3	1.1	1.0
	20m	-	0.1	2.1	2.1	2.0	1.5	1.5	1.5	1.7	1.2	1.0
YAFFS2	10m	-	0.3	0.7	0.6	0.7	0.0	-0.2	0.3	-0.3	-0.2	-0.1
	20m	-	0.6	0.9	0.9	0.9	0.1	0.2	0.6	0.3	0.2	0.5
After AIMP reduction												
Sed	10m	-	1.4	0.7	0.3	0.4	0.8	0.5	0.5	-3.8	-3.6	-3.9
	20m	-	1.6	0.8	0.7	0.6	0.6	0.8	0.7	-3.6	-3.6	-3.6
Space	10m	-	1.3	0.9	0.8	0.4	0.3	0.2	0.1	-1.5	-1.5	-1.5
	20m	-	1.2	0.6	0.5	0.3	0.1	-0.0	-0.2	-1.7	-1.8	-1.8
Grep	10m	-	-0.8	1.5	1.6	1.4	1.6	1.5	1.9	-3.4	-3.5	-3.8
	20m	-	-0.5	1.3	1.5	1.3	1.4	1.4	1.7	-2.5	-3.2	-3.1
Gzip	10m	-	1.6	1.6	1.7	1.0	1.6	1.6	1.4	-5.1	-5.1	-5.0
	20m	-	1.9	2.4	2.4	2.2	2.3	2.4	2.3	-5.2	-5.2	-5.2
Vim	10m	-	1.1	1.8	1.8	1.6	1.0	1.0	1.1	1.0	0.8	0.7
	20m	-	0.2	2.2	2.1	1.9	1.6	1.4	1.5	1.5	1.1	0.8
YAFFS2	10m	-	0.5	0.8	0.7	0.8	-0.0	-0.1	0.4	-0.2	-0.2	-0.0
	20m	-	0.2	0.7	0.7	0.7	-0.1	-0.1	0.1	-0.2	-0.2	-0.0
The best results		0	16	14	9	2	0	1	6	0	0	0

Table A.8: Cohen's *d* effect sizes of APBD values of different prioritization methods and random ordering on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Min Distance to Uncovered)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	0.9	1.8	1.0	1.5	1.8	1.3	1.0	-3.2	-2.7	-3.1
	20m	-	1.1	1.3	1.0	1.0	1.1	0.8	0.7	-1.4	-1.6	-1.6
Space	10m	-	0.8	0.7	0.7	0.7	0.2	0.3	0.1	-0.3	-0.3	-0.4
	20m	-	0.8	0.7	0.7	0.7	0.1	0.2	-0.1	-0.4	-0.4	-0.5
Grep	10m	-	0.0	1.4	1.4	1.5	1.4	1.2	1.6	-2.5	-3.1	-4.2
	20m	-	-0.3	1.7	1.7	1.6	1.4	1.2	1.5	-3.0	-3.4	-4.1
Gzip	10m	-	1.9	2.1	2.1	1.8	2.1	2.2	2.0	-5.3	-5.3	-5.1
	20m	-	2.0	2.2	2.2	2.0	2.2	2.2	2.2	-5.7	-5.5	-5.6
Vim	10m	-	1.0	2.0	1.8	1.6	1.1	0.9	1.0	0.9	0.7	0.4
	20m	-	0.4	2.2	2.0	1.9	1.4	1.3	1.4	1.5	1.0	0.9
YAFFS2	10m	-	0.3	0.8	1.0	0.8	0.2	0.3	0.4	-0.4	-0.0	-0.4
	20m	-	0.3	0.9	0.9	0.8	0.3	0.4	0.5	-0.1	-0.1	-0.2
After statement reduction												
Sed	10m	-	1.3	0.6	0.5	0.3	0.6	1.1	0.4	-2.7	-2.4	-2.9
	20m	-	1.3	1.0	1.0	0.6	0.6	0.9	0.8	-3.6	-4.2	-4.1
Space	10m	-	0.6	0.4	0.3	-0.1	0.7	0.6	0.5	-2.0	-2.1	-2.1
	20m	-	0.5	0.2	0.1	-0.2	0.7	0.6	0.4	-2.3	-2.3	-2.3
Grep	10m	-	0.7	1.7	1.4	1.4	1.6	1.5	1.7	-2.9	-3.3	-3.1
	20m	-	0.9	2.0	1.4	1.7	1.8	1.8	2.0	-2.9	-3.0	-3.0
Gzip	10m	-	2.1	2.4	2.4	2.1	2.4	2.4	2.3	-5.5	-5.5	-5.4
	20m	-	2.0	2.1	2.0	1.8	2.1	2.0	2.1	-4.6	-4.6	-4.4
Vim	10m	-	0.6	1.9	2.0	2.0	1.2	1.3	1.6	1.3	1.3	1.1
	20m	-	0.2	2.0	2.1	2.0	1.6	1.6	1.5	1.6	1.3	1.1
YAFFS2	10m	-	0.5	0.8	0.9	1.0	-0.0	-0.0	0.6	-0.0	0.1	0.3
	20m	-	0.2	0.7	1.0	1.0	0.0	0.2	0.5	0.1	0.2	0.4
After branch reduction												
Sed	10m	-	1.4	0.7	0.6	0.3	0.5	1.0	0.5	-2.9	-2.5	-3.0
	20m	-	1.2	0.9	0.9	0.4	0.5	0.9	0.7	-3.5	-4.0	-3.8
Space	10m	-	1.4	0.7	0.6	0.2	0.3	0.2	0.1	-1.4	-1.5	-1.5
	20m	-	1.2	0.7	0.6	0.3	0.2	0.0	-0.2	-1.7	-1.7	-1.8
Grep	10m	-	0.6	1.7	1.3	1.3	1.6	1.4	1.5	-2.5	-2.8	-2.6
	20m	-	0.9	1.8	1.2	1.7	1.6	1.6	1.8	-2.3	-2.4	-2.5
Gzip	10m	-	2.2	1.6	1.6	1.4	1.7	1.8	1.8	-5.1	-4.9	-4.9
	20m	-	2.2	2.1	1.8	1.8	2.2	2.0	2.1	-4.8	-4.7	-4.6
Vim	10m	-	0.9	1.9	1.8	1.8	1.3	1.2	1.3	1.3	1.1	1.0
	20m	-	0.1	2.1	2.1	1.9	1.5	1.5	1.5	1.7	1.3	1.1
YAFFS2	10m	-	0.6	0.8	0.8	0.9	-0.0	0.1	0.6	0.1	0.3	0.3
	20m	-	0.4	0.8	1.1	1.0	0.2	0.4	0.7	0.4	0.5	0.4
After AIMP reduction												
Sed	10m	-	1.0	0.7	0.6	0.4	0.6	0.7	0.8	-2.5	-2.2	-2.5
	20m	-	1.3	0.7	0.7	0.5	0.6	0.8	0.7	-3.1	-3.2	-3.2
Space	10m	-	1.2	0.6	0.8	0.4	0.4	0.3	0.2	-1.3	-1.3	-1.4
	20m	-	1.2	0.6	0.6	0.4	0.1	-0.0	-0.2	-1.6	-1.7	-1.8
Grep	10m	-	-0.1	1.5	1.6	1.5	1.4	1.2	1.4	-2.5	-2.9	-3.8
	20m	-	-0.2	1.5	1.5	1.4	1.5	1.0	1.4	-2.8	-3.3	-4.0
Gzip	10m	-	1.7	1.3	1.4	1.0	1.4	1.5	1.3	-5.0	-4.9	-4.9
	20m	-	2.1	2.4	2.4	2.1	2.3	2.4	2.3	-5.4	-5.2	-5.4
Vim	10m	-	1.0	1.9	1.9	1.7	1.1	0.9	1.1	1.0	0.8	0.5
	20m	-	0.3	2.0	2.0	1.8	1.4	1.3	1.4	1.5	1.1	0.8
YAFFS2	10m	-	0.6	0.8	0.9	0.9	-0.1	-0.1	0.5	-0.1	0.1	0.1
	20m	-	0.3	0.7	0.9	0.8	-0.0	0.0	0.2	-0.1	-0.0	0.1
The best results		0	14	11	10	4	4	3	2	0	0	0

Table A.9: Average Branch Discovered Increment values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Depth First Search)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	48.4	27.3	12.3	24.4	29.8	19.8	17.0	-76.4	-77.7	-80.0
	20m	-	46.4	25.7	16.4	24.0	24.8	19.6	13.4	-61.1	-65.8	-60.4
Space	10m	-	0.2	0.2	0.2	0.2	0.1	0.1	0.1	-0.1	-0.1	-0.1
	20m	-	0.2	0.2	0.2	0.2	0.1	0.1	0.0	-0.2	-0.2	-0.2
Grep	10m	-	3.2	3.1	2.3	2.0	2.6	1.9	2.0	-5.5	-5.8	-6.0
	20m	-	10.8	8.0	7.0	6.4	9.1	6.7	4.2	-16.0	-16.5	-16.2
Gzip	10m	-	34.2	36.2	31.0	25.7	37.5	31.9	29.7	-48.9	-54.8	-48.8
	20m	-	36.1	38.6	32.0	27.5	39.9	32.8	31.5	-51.2	-52.7	-51.9
Vim	10m	-	*9.7*	47.8	36.3	32.1	28.9	19.3	20.6	28.0	13.6	15.9
	20m	-	*11.5*	72.6	74.2	58.4	47.6	53.0	44.4	63.5	62.6	46.3
YAFFS2	10m	-	*0.2*	-3.3	-5.3	-6.8	*-0.9*	*-0.4*	-2.0	-13.1	-12.1	-13.7
	20m	-	*0.5*	-5.0	-5.7	-6.8	-1.2	-1.3	-2.9	-15.4	-14.5	-14.9
After statement reduction												
Sed	10m	-	43.0	7.7	*2.7*	*4.9*	10.3	8.4	*7.0*	-127.3	-126.2	-129.8
	20m	-	39.1	20.6	21.2	8.7	8.1	13.2	*3.0*	-95.8	-108.3	-106.6
Space	10m	-	0.9	0.3	0.1	-0.5	1.5	1.2	1.1	-3.4	-3.2	-3.2
	20m	-	0.6	0.1	0.2	-0.4	1.7	1.4	1.3	-4.1	-3.8	-3.8
Grep	10m	-	29.7	43.1	36.3	34.7	37.1	34.4	36.0	-76.1	-79.6	-80.7
	20m	-	29.9	40.8	30.6	34.2	32.8	28.6	32.8	-77.4	-82.4	-90.1
Gzip	10m	-	13.2	16.6	15.3	13.8	17.1	15.6	15.1	-28.0	-28.9	-26.3
	20m	-	8.6	12.5	12.0	11.0	13.0	12.0	12.0	-24.3	-25.2	-25.0
Vim	10m	-	*8.6*	39.6	33.6	23.9	34.1	23.5	20.1	33.2	19.2	23.7
	20m	-	*-15.5*	89.0	84.2	81.7	70.6	61.3	62.7	91.5	82.1	78.5
YAFFS2	10m	-	*0.6*	-4.0	-5.9	-8.3	*-1.1*	*-0.5*	-2.5	-15.0	-14.2	-15.8
	20m	-	*0.6*	-4.5	-5.5	-7.7	*-0.9*	*-1.2*	-2.9	-13.8	-13.2	-14.0
After branch reduction												
Sed	10m	-	46.0	9.4	*2.2*	*4.4*	11.4	11.6	8.6	-121.6	-122.3	-125.9
	20m	-	46.7	21.7	21.9	8.3	5.7	12.2	*4.0*	-111.2	-118.9	-112.7
Space	10m	-	0.6	0.2	0.2	-0.0	0.2	0.1	0.1	-1.3	-1.1	-1.1
	20m	-	0.4	0.1	0.3	0.1	0.2	0.1	0.1	-1.5	-1.5	-1.5
Grep	10m	-	32.3	43.8	36.4	35.7	39.8	35.6	37.6	-78.3	-83.6	-82.1
	20m	-	38.8	43.3	32.7	37.5	35.4	31.8	37.5	-82.5	-82.6	-87.6
Gzip	10m	-	13.8	17.1	15.8	14.1	17.6	16.1	15.7	-27.8	-28.2	-25.4
	20m	-	32.5	35.2	30.0	26.0	36.4	30.6	29.5	-47.8	-48.4	-47.6
Vim	10m	-	*-4.5*	41.5	35.8	32.2	28.4	28.9	27.0	22.2	16.6	17.9
	20m	-	*-2.9*	93.6	97.5	82.4	84.4	80.1	63.3	91.3	82.2	67.6
YAFFS2	10m	-	*-0.3*	*-0.6*	-2.9	-4.7	*-1.0*	*-0.6*	-1.6	-13.6	-12.7	-14.3
	20m	-	1.8	-3.7	-2.9	-5.7	*0.8*	*-0.1*	-1.4	-10.2	-9.6	-11.8
After AIMP reduction												
Sed	10m	-	43.9	13.9	5.7	8.2	15.7	7.3	11.4	-107.1	-111.0	-112.8
	20m	-	44.0	14.5	11.9	14.1	13.2	16.7	13.7	-108.0	-111.1	-110.0
Space	10m	-	0.6	0.2	0.2	-0.0	0.2	0.1	0.1	-1.3	-1.1	-1.1
	20m	-	0.5	0.0	0.2	0.1	0.1	0.1	0.1	-1.4	-1.4	-1.4
Grep	10m	-	3.3	3.1	2.7	1.6	3.0	2.2	1.5	-5.0	-5.5	-6.1
	20m	-	2.9	3.0	2.8	2.5	2.8	2.2	2.2	-5.4	-6.6	-6.7
Gzip	10m	-	25.2	27.1	24.3	19.9	27.9	25.0	22.9	-38.4	-41.3	-35.9
	20m	-	32.3	34.8	29.5	25.3	36.2	30.1	28.7	-45.7	-45.9	-44.4
Vim	10m	-	8.3	32.4	26.9	27.8	20.6	17.5	11.5	13.3	3.2	4.8
	20m	-	*9.9*	92.8	98.3	89.8	74.3	76.9	68.3	84.5	84.4	62.8
YAFFS2	10m	-	-1.9	*-0.9*	-3.8	-4.7	-1.2	-1.5	-2.0	-13.1	-12.7	-13.3
	20m	-	2.1	-3.3	-2.5	-5.1	*0.5*	*-0.3*	-1.0	-9.1	-8.7	-10.8
The best results		0	17	9	3	2	10	0	0	1	0	0
Win random		-	35	40	38	34	40	40	37	8	8	8
Lost to random		-	1	6	8	12	2	2	8	40	40	40

Table A.10: Average Branch Discovered Increment values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Random Path)

Subject	Time	Random	Shortest	FPF			CA			CT		
			path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	-1.8	*0.8*	*-0.5*	*-1.0*	*0.3*	-1.7	-3.3	-21.8	-21.0	-22.6
	20m	-	-4.4	2.3	*1.5*	*0.6*	*0.5*	*0.4*	*1.3*	-17.8	-17.5	-18.9
Space	10m	-	-0.0	-0.0	0.0	-0.0	0.0	0.0	0.0	0.0	0.0	0.0
	20m	-	-0.0	-0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Grep	10m	-	-1.0	1.0	1.1	0.9	1.0	0.9	0.9	-2.6	-2.6	-2.5
	20m	-	*0.8*	2.1	1.9	1.7	1.4	0.5	1.3	-4.0	-4.6	-4.2
Gzip	10m	-	31.0	29.5	29.3	25.7	30.1	30.3	30.1	-108.1	-110.8	-110.3
	20m	-	32.7	30.8	31.8	29.5	31.1	32.6	32.8	-114.9	-111.4	-110.6
Vim	10m	-	-1.3	11.8	10.0	9.7	10.6	9.0	8.9	10.6	9.0	8.8
	20m	-	-3.6	9.5	8.7	9.8	8.5	6.8	8.1	9.9	8.4	8.3
YAFFS2	10m	-	0.4	0.8	1.1	1.5	0.8	1.0	1.2	0.4	1.1	*0.7*
	20m	-	1.0	1.2	1.2	1.4	0.9	1.0	1.4	0.7	0.9	0.8
After statement reduction												
Sed	10m	-	-4.0	1.2	*0.4*	-1.0	*-0.3*	*0.9*	1.3	-37.3	-35.7	-36.4
	20m	-	-3.9	1.8	1.7	*0.9*	1.8	2.2	2.1	-24.3	-24.5	-26.8
Space	10m	-	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	0.0	-0.0	0.0
	20m	-	0.2	-0.0	0.0	-0.2	0.2	0.2	0.1	-0.3	-0.2	-0.3
Grep	10m	-	3.0	5.4	3.5	5.5	6.0	5.2	5.6	-11.5	-13.2	-10.9
	20m	-	7.0	16.6	6.6	13.3	18.0	13.5	15.9	-23.0	-25.1	-26.5
Gzip	10m	-	28.7	23.7	23.0	21.5	23.8	24.0	25.0	-99.1	-101.6	-101.4
	20m	-	29.4	25.5	27.1	25.8	25.0	27.0	27.9	-101.3	-98.7	-97.8
Vim	10m	-	*-3.0*	16.4	13.0	12.4	15.3	12.3	11.8	14.7	11.9	11.3
	20m	-	-10.7	12.9	11.3	12.9	10.4	7.7	8.8	13.2	7.5	3.2
YAFFS2	10m	-	0.5	0.9	1.0	1.3	0.5	0.8	1.0	0.4	0.8	*0.5*
	20m	-	1.3	0.9	1.0	1.1	0.7	1.1	1.0	*0.2*	*0.3*	*0.1*
After branch reduction												
Sed	10m	-	-2.4	1.3	*0.3*	-1.0	*-0.0*	*0.8*	*0.9*	-38.4	-36.8	-37.7
	20m	-	-4.4	2.2	2.4	1.5	2.4	3.3	3.4	-24.5	-24.6	-26.8
Space	10m	-	-0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	20m	-	-0.0	-0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Grep	10m	-	3.1	5.3	3.3	5.4	6.2	5.5	5.9	-11.3	-12.9	-10.3
	20m	-	6.9	16.6	6.4	13.5	18.0	13.8	16.3	-22.7	-24.8	-25.7
Gzip	10m	-	29.3	23.7	23.0	21.2	23.6	24.0	24.9	-98.7	-100.9	-100.8
	20m	-	33.6	31.7	32.6	30.7	32.3	34.4	34.2	-105.9	-102.4	-101.5
Vim	10m	-	*-1.4*	14.4	11.6	11.1	13.3	10.6	10.4	12.8	10.2	9.9
	20m	-	-5.0	12.6	9.3	10.6	10.9	6.6	7.4	12.0	6.6	2.0
YAFFS2	10m	-	0.6	0.9	0.9	1.2	0.6	0.9	1.2	0.4	0.9	*0.5*
	20m	-	1.2	1.0	1.3	1.3	0.8	1.5	1.2	0.6	1.1	*0.5*
After AIMP reduction												
Sed	10m	-	*-1.9*	*1.0*	*-0.8*	-2.2	*0.6*	-1.1	-3.0	-24.6	-23.9	-25.4
	20m	-	-3.0	2.2	1.9	*0.0*	*-0.3*	*0.1*	*0.4*	-24.7	-24.3	-24.8
Space	10m	-	-0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	20m	-	-0.0	-0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Grep	10m	-	-0.9	1.1	1.1	1.0	1.3	1.0	1.1	-2.5	-2.5	-2.3
	20m	-	*0.0*	1.7	1.5	1.4	1.4	1.0	1.0	-3.6	-4.1	-4.1
Gzip	10m	-	36.1	19.2	18.2	11.9	16.3	19.0	14.6	-88.9	-90.1	-90.8
	20m	-	31.8	30.6	31.5	29.4	31.0	32.2	32.5	-109.2	-106.2	-105.1
Vim	10m	-	*-0.9*	11.7	9.9	9.9	10.3	8.9	8.7	10.2	8.6	8.4
	20m	-	-4.9	10.5	7.8	8.7	8.9	4.9	5.3	10.5	5.6	0.5
YAFFS2	10m	-	*0.4*	0.8	1.0	1.3	*0.3*	0.6	1.1	*0.3*	0.9	*0.5*
	20m	-	0.9	1.1	1.4	1.3	0.8	1.4	1.3	0.7	1.3	0.6
The best results		0	6	11	1	5	6	7	4	2	0	1
Win random		-	20	40	39	35	37	38	39	18	18	14
Lost to random		-	21	6	1	6	1	3	3	25	26	25

Table A.11: Average Branch Discovered Increment values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Random State)

Subject	Time	Random	Shortest	FPF			CA			CT		
			path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	9.7	13.7	9.1	12.5	11.4	7.9	5.6	-27.8	-26.4	-27.6
	20m	-	18.2	16.5	11.6	12.1	14.1	9.8	6.8	-35.0	-37.8	-37.0
Space	10m	-	0.2	0.2	0.3	0.3	0.1	0.2	0.2	-0.2	-0.0	-0.0
	20m	-	0.2	0.2	0.4	0.3	0.0	0.3	0.1	-0.2	0.0	-0.1
Grep	10m	-	-2.2	3.7	3.8	3.3	4.0	3.6	4.1	-11.3	-11.8	-13.2
	20m	-	-1.5	4.0	4.3	3.8	4.4	4.0	4.6	-10.8	-11.8	-13.1
Gzip	10m	-	21.3	25.6	24.7	22.7	24.5	24.6	24.4	-68.9	-68.7	-68.9
	20m	-	24.2	28.6	27.9	25.6	27.3	27.6	27.2	-73.1	-71.7	-71.3
Vim	10m	-	21.0	37.4	35.1	31.3	21.3	18.4	22.6	20.1	14.4	8.2
	20m	-	*2.2*	35.6	35.1	32.6	26.6	26.1	26.2	28.2	22.2	17.1
YAFFS2	10m	-	*-0.0*	0.5	0.8	0.7	0.4	0.6	0.5	*-0.2*	*0.3*	-0.5
	20m	-	0.7	0.7	0.9	0.9	0.6	0.8	0.8	*0.0*	*0.0*	*-0.0*
After statement reduction												
Sed	10m	-	20.6	11.5	8.7	6.6	9.6	13.2	9.1	-54.3	-53.0	-56.4
	20m	-	14.6	13.1	12.8	7.9	8.3	11.2	9.3	-53.5	-56.4	-55.6
Space	10m	-	0.5	0.1	0.2	-0.3	0.4	0.3	0.2	-2.0	-1.9	-1.9
	20m	-	0.5	0.1	0.2	-0.2	0.6	0.4	0.3	-2.1	-2.0	-2.1
Grep	10m	-	11.4	24.5	19.3	18.8	22.8	20.1	22.2	-46.5	-50.2	-50.5
	20m	-	14.7	28.9	21.5	26.1	27.7	25.6	28.8	-53.3	-56.3	-58.0
Gzip	10m	-	21.6	25.2	24.8	22.6	25.0	25.0	24.5	-68.3	-68.7	-68.8
	20m	-	24.9	29.7	28.0	25.9	28.4	28.0	27.4	-71.6	-70.2	-70.4
Vim	10m	-	15.5	40.3	38.9	36.6	27.4	25.3	27.9	28.5	23.4	20.6
	20m	-	*-0.6*	41.4	40.4	39.2	32.8	31.5	33.2	34.1	27.4	24.5
YAFFS2	10m	-	0.9	2.3	2.1	2.3	*0.0*	*-0.4*	1.3	-0.9	*-0.6*	*-0.2*
	20m	-	0.8	2.3	2.5	2.5	*0.2*	*0.5*	1.1	*0.2*	*0.1*	*0.9*
After branch reduction												
Sed	10m	-	21.3	10.3	7.2	4.1	8.0	12.0	7.9	-56.0	-54.6	-58.4
	20m	-	16.1	13.3	12.8	6.3	7.1	10.9	8.3	-52.4	-54.7	-54.7
Space	10m	-	0.4	0.2	0.3	0.1	0.1	0.1	0.1	-1.1	-1.0	-1.1
	20m	-	0.3	0.2	0.2	0.1	0.0	0.1	0.0	-1.1	-1.1	-1.1
Grep	10m	-	11.2	24.1	18.0	18.0	22.8	19.4	21.0	-44.6	-49.4	-49.5
	20m	-	14.4	26.1	18.8	22.7	25.8	23.9	27.1	-48.7	-51.3	-53.1
Gzip	10m	-	15.6	15.5	14.7	10.6	16.4	16.3	14.1	-53.4	-53.4	-53.0
	20m	-	25.2	30.2	29.0	26.6	28.8	28.8	28.0	-71.5	-69.7	-69.6
Vim	10m	-	14.4	37.1	36.1	33.7	25.8	23.9	26.2	27.2	22.6	19.9
	20m	-	*2.7*	37.7	36.8	35.5	28.9	28.0	28.8	31.0	24.0	19.7
YAFFS2	10m	-	0.9	2.1	2.0	2.3	*-0.0*	*-0.4*	1.2	-0.9	-0.7	*-0.2*
	20m	-	1.7	2.4	2.6	2.5	*0.4*	*0.9*	1.9	0.9	*0.8*	1.4
After AIMP reduction												
Sed	10m	-	15.0	8.3	3.7	3.8	9.0	6.2	5.7	-49.0	-49.5	-53.0
	20m	-	18.6	10.4	9.0	7.2	8.2	10.5	8.4	-52.1	-54.0	-53.8
Space	10m	-	0.4	0.3	0.4	0.2	0.1	0.2	0.1	-1.0	-0.9	-0.9
	20m	-	0.3	0.2	0.2	0.1	0.0	0.1	0.0	-1.1	-1.1	-1.1
Grep	10m	-	-2.4	3.1	3.4	3.1	3.4	3.2	3.7	-10.7	-11.0	-12.1
	20m	-	-1.6	3.6	3.9	3.6	4.0	3.7	4.5	-10.4	-11.4	-12.4
Gzip	10m	-	11.6	13.3	13.6	9.5	13.8	13.9	12.0	-41.2	-40.8	-39.3
	20m	-	22.7	28.4	27.7	25.5	27.0	27.1	26.5	-70.7	-69.3	-69.0
Vim	10m	-	24.2	35.1	34.3	31.2	22.1	20.4	20.9	21.8	17.9	13.0
	20m	-	4.4	34.6	33.3	31.1	27.1	25.6	24.7	26.5	20.6	15.7
YAFFS2	10m	-	1.4	2.3	2.4	2.6	*-0.1*	*-0.4*	1.6	-0.8	*-0.6*	*0.1*
	20m	-	0.7	2.0	2.0	1.8	*-0.3*	*-0.1*	*0.4*	-0.6	-0.6	*-0.0*
The best results		0	12	18	7	3	2	1	5	0	0	0
Win random		-	40	48	48	46	42	42	47	9	9	9
Lost to random		-	4	0	0	2	0	0	0	36	33	33

Table A.12: Average Branch Discovered Increment values with different prioritization methods on 100 random selected test cases (FPF: Furthest Point First, CA: Coverage Additional, CT: Coverage Total, Stmt: Statement) (Search strategy: Min Distance to Uncovered)

Subject	Time	Random	Shortest path	FPF			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	-	5.7	11.2	6.9	10.1	10.7	7.9	5.7	-23.0	-21.6	-24.1
	20m	-	16.5	18.9	14.0	14.2	15.6	11.3	9.0	-27.3	-30.9	-30.2
Space	10m	-	0.2	0.2	0.3	0.3	0.1	0.2	0.2	-0.2	-0.0	-0.0
	20m	-	0.2	0.2	0.4	0.3	0.0	0.3	0.1	-0.2	0.0	-0.1
Grep	10m	-	*0.1*	3.5	3.5	3.5	3.3	2.8	3.4	-8.9	-9.5	-12.2
	20m	-	-1.0	3.5	3.5	3.3	3.0	2.6	3.0	-9.1	-10.1	-12.3
Gzip	10m	-	26.4	29.2	28.2	24.4	29.6	29.0	27.7	-79.1	-80.7	-77.8
	20m	-	32.1	35.9	35.2	31.7	35.5	34.9	34.1	-96.7	-93.4	-93.1
Vim	10m	-	23.2	40.5	37.4	33.6	24.6	20.4	22.6	23.1	15.1	8.6
	20m	-	9.5	39.2	36.4	34.5	27.3	25.1	27.0	29.4	21.0	17.7
YAFFS2	10m	-	0.4	0.9	1.4	1.1	0.3	0.5	0.6	-0.5	*0.1*	-0.5
	20m	-	0.4	1.1	1.2	1.2	0.4	0.6	0.8	*-0.1*	*-0.0*	*-0.1*
After statement reduction												
Sed	10m	-	16.8	9.1	6.6	3.8	9.0	13.5	5.1	-43.2	-41.6	-48.0
	20m	-	16.0	14.4	13.9	8.5	8.7	12.0	9.9	-53.1	-57.2	-55.5
Space	10m	-	0.5	0.2	0.2	-0.2	0.6	0.5	0.3	-2.0	-1.9	-1.9
	20m	-	0.3	0.1	0.0	-0.3	0.7	0.5	0.3	-2.4	-2.3	-2.3
Grep	10m	-	12.8	25.6	21.1	20.1	24.4	22.4	23.7	-54.0	-58.1	-59.8
	20m	-	16.3	30.3	22.1	27.0	29.3	27.1	30.2	-58.7	-61.1	-63.0
Gzip	10m	-	26.2	30.3	29.4	25.9	30.0	29.6	28.3	-78.9	-79.6	-78.3
	20m	-	36.4	40.0	37.3	32.8	38.7	36.9	36.1	-92.1	-90.0	-89.2
Vim	10m	-	16.7	43.0	43.9	42.0	29.2	30.3	33.6	33.1	30.2	25.5
	20m	-	*4.0*	46.2	45.5	44.2	37.0	35.4	36.0	37.9	30.1	26.9
YAFFS2	10m	-	1.5	2.6	2.9	3.2	*-0.1*	*0.1*	2.2	*-0.1*	*0.6*	*1.0*
	20m	-	0.8	2.6	3.2	3.3	*0.2*	*0.9*	2.1	*0.5*	*0.9*	1.4
After branch reduction												
Sed	10m	-	19.0	10.4	8.7	4.5	8.0	13.7	7.6	-48.5	-45.3	-51.6
	20m	-	15.6	13.1	12.7	6.0	7.5	12.0	7.9	-52.3	-56.7	-54.9
Space	10m	-	0.5	0.2	0.3	0.1	0.1	0.2	0.1	-1.0	-0.9	-0.9
	20m	-	0.3	0.2	0.2	0.1	0.1	0.1	0.1	-1.1	-1.1	-1.1
Grep	10m	-	11.1	23.8	18.1	18.1	23.3	20.3	20.9	-47.2	-51.6	-52.6
	20m	-	15.9	29.3	19.8	25.6	28.1	25.2	28.1	-51.1	-54.3	-57.3
Gzip	10m	-	29.2	25.6	24.5	21.7	26.5	26.5	26.1	-76.5	-76.9	-76.8
	20m	-	41.1	43.8	38.0	36.5	42.7	39.2	39.9	-97.6	-95.8	-95.6
Vim	10m	-	23.3	42.0	39.1	37.5	30.3	26.6	28.5	32.5	26.4	22.8
	20m	-	*2.1*	43.9	42.2	41.0	33.5	32.1	33.5	38.0	29.1	26.5
YAFFS2	10m	-	1.7	2.2	2.6	2.5	*-0.2*	*0.5*	2.2	*0.5*	*1.0*	1.1
	20m	-	1.4	2.7	3.4	3.3	0.8	1.6	2.6	1.3	1.6	1.7
After AIMP reduction												
Sed	10m	-	11.0	8.5	6.6	4.2	8.3	8.6	7.6	-37.4	-34.2	-40.9
	20m	-	16.4	10.7	9.4	7.1	9.0	11.0	8.1	-48.1	-50.4	-50.1
Space	10m	-	0.4	0.2	0.4	0.2	0.2	0.2	0.2	-0.9	-0.8	-0.8
	20m	-	0.3	0.2	0.3	0.2	0.0	0.1	0.0	-1.1	-1.0	-1.1
Grep	10m	-	*-0.4*	3.7	3.8	3.6	3.2	2.8	3.1	-8.8	-9.1	-11.3
	20m	-	*-0.6*	3.3	3.3	3.1	3.1	2.2	2.8	-8.9	-9.8	-11.9
Gzip	10m	-	17.2	15.7	16.5	13.2	16.3	17.1	15.4	-55.5	-54.0	-53.4
	20m	-	33.7	39.1	37.9	34.3	37.8	37.6	36.5	-96.3	-93.6	-94.3
Vim	10m	-	23.5	40.0	37.8	34.5	24.0	20.1	21.9	24.1	18.4	9.7
	20m	-	7.3	37.6	37.8	34.7	28.6	26.7	26.8	29.8	22.4	17.3
YAFFS2	10m	-	1.8	2.6	2.8	3.0	*-0.4*	*-0.1*	2.0	*-0.4*	*0.3*	*0.5*
	20m	-	1.0	2.2	2.5	2.3	*-0.1*	*0.1*	*0.7*	*-0.2*	*-0.1*	*0.2*
The best results		0	12	19	11	3	3	0	0	0	0	0
Win random		-	42	48	48	46	43	43	47	9	10	11
Lost to random		-	1	0	0	2	0	0	0	33	31	33

Appendix B: Tables of Test Case Selection Effects

Table B.1: Test case selection values (#/%) with different prioritization methods on 100 random selected test cases (Search strategy: DFS)

Subject	Time	Random	Shortest	FPF (slope: -1)			CA			CT		
			Path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/41	20/98	14/61	17/52	23/70	22/90	26/86	46/93	20/10	20/11	20/9
	20m	20/43	20/95	14/62	17/55	23/73	22/84	27/86	45/92	20/15	20/10	20/12
Space	10m	20/82	20/97	20/96	21/97	18/100	46/100	48/100	53/100	20/79	20/80	20/82
	20m	20/83	20/97	19/96	21/97	19/99	47/100	48/100	53/100	20/76	20/75	20/75
Grep	10m	20/58	20/83	12/81	17/82	15/77	16/83	22/85	39/95	20/43	20/39	20/38
	20m	20/46	20/82	12/81	16/82	14/82	16/91	22/91	39/90	20/26	20/24	20/24
Gzip	10m	20/61	20/100	11/100	12/100	14/94	13/100	14/100	26/100	20/43	20/44	20/46
	20m	20/61	20/100	11/100	12/100	14/98	13/100	14/100	26/100	20/42	20/44	20/43
Vim	10m	20/51	20/64	19/70	21/73	23/72	65/91	70/94	75/94	20/66	20/61	20/62
	20m	20/45	20/53	19/64	20/65	22/62	65/93	69/93	74/93	20/63	20/60	20/56
YAFFS2	10m	20/57	20/56	10/48	10/46	3/29	9/40	12/44	25/55	20/44	20/46	20/45
	20m	20/60	20/58	11/48	10/46	3/29	9/38	12/43	25/53	20/42	20/42	20/42
After statement reduction												
Sed	10m	20/59	20/99	14/49	17/52	23/62	22/78	26/84	46/93	20/8	20/8	20/7
	20m	20/62	20/98	14/70	17/71	23/71	22/78	27/88	45/92	20/24	20/11	20/11
Space	10m	20/47	20/56	20/47	21/49	18/40	46/98	48/98	53/99	20/18	20/19	20/18
	20m	20/51	20/55	19/47	21/52	19/43	47/98	48/98	53/99	20/15	20/15	20/15
Grep	10m	20/42	20/69	12/78	17/67	15/57	16/81	22/84	39/93	20/6	20/5	20/5
	20m	20/44	20/74	12/76	16/65	14/67	16/80	22/82	39/94	20/7	20/7	20/6
Gzip	10m	20/74	20/98	11/96	12/96	14/94	13/97	14/97	26/99	20/57	20/57	20/59
	20m	20/79	20/98	11/95	12/96	14/96	13/98	14/97	26/99	20/63	20/62	20/63
Vim	10m	20/53	20/63	19/67	21/68	23/70	65/91	70/93	75/93	20/69	20/63	20/65
	20m	20/49	20/52	19/67	20/69	22/67	65/92	69/93	74/95	20/68	20/66	20/65
YAFFS2	10m	20/59	20/58	10/48	10/45	3/29	9/41	12/45	25/55	20/44	20/44	20/45
	20m	20/59	20/59	11/48	10/46	3/29	9/39	12/43	25/53	20/45	20/48	20/46
After branch reduction												
Sed	10m	20/59	20/99	14/48	17/47	23/56	22/74	26/84	46/95	20/8	20/8	20/6
	20m	20/56	20/98	14/62	17/64	23/62	22/72	27/82	45/92	20/7	20/7	20/6
Space	10m	20/67	20/95	20/82	21/87	18/77	46/100	48/100	53/100	20/47	20/49	20/52
	20m	20/78	20/98	19/87	21/94	19/86	47/100	48/100	53/100	20/42	20/42	20/43
Grep	10m	20/41	20/71	12/79	17/68	15/59	16/84	22/86	39/95	20/6	20/4	20/5
	20m	20/38	20/77	12/71	16/59	14/67	16/75	22/81	39/97	20/4	20/4	20/5
Gzip	10m	20/74	20/98	11/96	12/96	14/93	13/96	14/97	26/99	20/58	20/59	20/61
	20m	20/64	20/98	11/97	12/98	14/96	13/98	14/98	26/99	20/47	20/48	20/48
Vim	10m	20/53	20/62	19/69	21/71	23/71	65/92	70/96	75/95	20/67	20/63	20/63
	20m	20/47	20/53	19/67	20/72	22/68	65/93	69/94	74/95	20/69	20/66	20/64
YAFFS2	10m	20/58	20/58	10/51	10/48	3/30	9/39	12/44	25/56	20/41	20/42	20/42
	20m	20/60	20/62	11/46	10/46	3/27	9/44	12/47	25/60	20/49	20/49	20/47
After AIMP reduction												
Sed	10m	20/54	20/98	14/52	17/49	23/60	22/81	26/79	46/95	20/8	20/8	20/6
	20m	20/52	20/97	14/54	17/49	23/65	22/79	27/83	45/96	20/7	20/7	20/6
Space	10m	20/67	20/95	20/82	21/87	18/77	46/100	48/100	53/100	20/47	20/49	20/52
	20m	20/74	20/98	19/82	21/89	19/84	47/100	48/100	53/100	20/43	20/43	20/44
Grep	10m	20/57	20/82	12/82	17/86	15/77	16/86	22/88	39/91	20/44	20/40	20/38
	20m	20/60	20/79	12/76	16/80	14/81	16/86	22/89	39/94	20/42	20/37	20/36
Gzip	10m	20/65	20/99	11/96	12/97	14/93	13/96	14/97	26/98	20/49	20/51	20/54
	20m	20/64	20/99	11/98	12/98	14/95	13/98	14/98	26/99	20/48	20/50	20/50
Vim	10m	20/53	20/65	19/67	21/66	23/72	65/91	70/95	75/93	20/62	20/55	20/58
	20m	20/49	20/59	19/71	20/72	22/71	65/93	69/93	74/95	20/67	20/67	20/65
YAFFS2	10m	20/58	20/54	10/50	10/47	3/29	9/38	12/41	25/54	20/40	20/41	20/42
	20m	20/60	20/64	11/46	10/47	3/27	9/44	12/48	25/62	20/50	20/50	20/48
Average results		20/58	20/80	14/71	16/71	16/67	29/82	32/84	44/89	20/40	20/39	20/39

Table B.2: Test case selection values (#/%) with different prioritization methods on 100 random selected test cases (Search strategy: RP)

Subject	Time	Random	Shortest	FPF (slope: -1)			CA			CT		
			Path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/90	20/87	14/86	17/87	23/89	22/93	26/94	46/97	20/82	20/82	20/82
	20m	20/92	20/89	14/89	17/89	23/92	22/94	27/96	45/99	20/88	20/88	20/87
Space	10m	20/100	20/100	20/100	21/100	18/100	46/100	48/100	53/100	20/100	20/100	20/100
	20m	20/100	20/100	19/100	21/100	19/100	47/100	48/100	53/100	20/100	20/100	20/100
Grep	10m	20/96	20/96	12/96	17/97	15/97	16/98	22/98	39/99	20/94	20/94	20/94
	20m	20/92	20/94	12/92	16/93	14/93	16/94	22/94	39/97	20/90	20/89	20/90
Gzip	10m	20/67	20/100	11/95	12/95	14/91	13/94	14/94	26/94	20/26	20/25	20/25
	20m	20/66	20/100	11/94	12/95	14/93	13/92	14/94	26/95	20/25	20/25	20/26
Vim	10m	20/80	20/84	19/96	21/96	23/96	65/99	70/99	75/100	20/94	20/94	20/93
	20m	20/81	20/82	19/94	20/94	22/95	65/98	69/98	74/99	20/93	20/92	20/91
YAFFS2	10m	20/89	20/89	10/83	10/82	3/61	9/83	12/87	25/94	20/88	20/90	20/89
	20m	20/89	20/91	11/85	10/82	3/61	9/84	12/88	25/94	20/89	20/90	20/90
After statement reduction												
Sed	10m	20/93	20/89	14/90	17/91	23/93	22/95	26/97	46/99	20/79	20/81	20/80
	20m	20/93	20/91	14/90	17/91	23/93	22/95	27/96	45/98	20/84	20/84	20/83
Space	10m	20/98	20/97	20/98	21/97	18/97	46/99	48/99	53/99	20/98	20/98	20/98
	20m	20/89	20/92	19/89	21/91	19/88	47/99	48/100	53/99	20/86	20/88	20/86
Grep	10m	20/88	20/89	12/92	17/90	15/92	16/95	22/96	39/98	20/82	20/81	20/83
	20m	20/77	20/79	12/87	16/83	14/85	16/95	22/95	39/98	20/70	20/71	20/70
Gzip	10m	20/68	20/99	11/88	12/89	14/87	13/88	14/87	26/90	20/31	20/30	20/30
	20m	20/67	20/99	11/89	12/91	14/89	13/88	14/90	26/92	20/31	20/31	20/31
Vim	10m	20/76	20/80	19/95	21/95	23/95	65/98	70/99	75/99	20/92	20/92	20/92
	20m	20/75	20/75	19/90	20/91	22/90	65/94	69/96	74/96	20/87	20/85	20/82
YAFFS2	10m	20/90	20/91	10/84	10/83	3/63	9/84	12/87	25/93	20/90	20/91	20/90
	20m	20/89	20/93	11/86	10/83	3/63	9/84	12/88	25/93	20/90	20/90	20/89
After branch reduction												
Sed	10m	20/92	20/90	14/89	17/90	23/91	22/94	26/96	46/99	20/78	20/79	20/79
	20m	20/92	20/90	14/90	17/92	23/93	22/96	27/97	45/98	20/84	20/85	20/83
Space	10m	20/100	20/100	20/100	21/100	18/100	46/100	48/100	53/100	20/100	20/100	20/100
	20m	20/100	20/100	19/100	21/100	19/100	47/100	48/100	53/100	20/100	20/100	20/100
Grep	10m	20/88	20/90	12/92	17/90	15/92	16/96	22/97	39/99	20/82	20/82	20/84
	20m	20/77	20/79	12/87	16/83	14/86	16/94	22/95	39/98	20/71	20/71	20/72
Gzip	10m	20/67	20/99	11/88	12/89	14/86	13/87	14/87	26/90	20/31	20/31	20/31
	20m	20/66	20/99	11/93	12/94	14/93	13/92	14/94	26/95	20/30	20/30	20/31
Vim	10m	20/78	20/83	19/96	21/96	23/95	65/99	70/99	75/99	20/93	20/93	20/92
	20m	20/78	20/81	19/94	20/92	22/93	65/97	69/97	74/97	20/91	20/88	20/85
YAFFS2	10m	20/90	20/91	10/84	10/83	3/63	9/85	12/88	25/94	20/91	20/91	20/91
	20m	20/89	20/92	11/85	10/82	3/62	9/84	12/89	25/93	20/91	20/91	20/90
After AIMP reduction												
Sed	10m	20/91	20/90	14/87	17/88	23/89	22/93	26/94	46/97	20/80	20/80	20/81
	20m	20/93	20/90	14/89	17/90	23/92	22/95	27/96	45/99	20/85	20/85	20/85
Space	10m	20/100	20/100	20/100	21/100	18/100	46/100	48/100	53/100	20/100	20/100	20/100
	20m	20/100	20/100	19/100	21/100	19/100	47/100	48/100	53/100	20/100	20/100	20/100
Grep	10m	20/96	20/96	12/96	17/97	15/97	16/98	22/98	39/99	20/94	20/94	20/94
	20m	20/94	20/95	12/94	16/96	14/95	16/96	22/97	39/98	20/92	20/91	20/91
Gzip	10m	20/62	20/99	11/78	12/80	14/73	13/74	14/77	26/78	20/31	20/31	20/32
	20m	20/67	20/99	11/93	12/95	14/93	13/92	14/93	26/95	20/30	20/30	20/30
Vim	10m	20/79	20/84	19/96	21/96	23/95	65/99	70/99	75/99	20/93	20/94	20/93
	20m	20/79	20/81	19/94	20/92	22/93	65/97	69/97	74/97	20/92	20/88	20/86
YAFFS2	10m	20/90	20/91	10/84	10/83	3/62	9/84	12/87	25/94	20/90	20/91	20/91
	20m	20/89	20/91	11/85	10/83	3/62	9/84	12/88	25/94	20/90	20/91	20/91
Average results		20/85	20/92	14/91	16/91	16/88	29/93	32/95	44/97	20/79	20/79	20/79

Table B.3: Test case selection values (#/%) with different prioritization methods on 100 random selected test cases (Search strategy: RS)

Subject	Time	Random	Shortest	FPF (slope: -1)			CA			CT		
			Path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/79	20/82	14/85	17/84	23/90	22/93	26/93	46/97	20/64	20/63	20/62
	20m	20/76	20/87	14/83	17/82	23/87	22/93	27/92	45/96	20/61	20/60	20/59
Space	10m	20/79	20/100	20/98	21/100	18/100	46/100	48/100	53/100	20/76	20/76	20/74
	20m	20/78	20/100	19/99	21/99	19/99	47/100	48/100	53/100	20/71	20/70	20/67
Grep	10m	20/89	20/86	12/92	17/94	15/93	16/96	22/95	39/99	20/75	20/75	20/75
	20m	20/88	20/87	12/92	16/94	14/93	16/95	22/96	39/99	20/76	20/76	20/75
Gzip	10m	20/70	20/100	11/98	12/98	14/96	13/98	14/98	26/98	20/34	20/35	20/36
	20m	20/68	20/100	11/99	12/98	14/97	13/98	14/98	26/99	20/34	20/34	20/34
Vim	10m	20/62	20/83	19/86	21/84	23/83	65/94	70/95	75/98	20/76	20/74	20/72
	20m	20/71	20/81	19/90	20/91	22/88	65/98	69/98	74/99	20/86	20/82	20/81
YAFFS2	10m	20/92	20/92	10/86	10/86	3/64	9/86	12/88	25/94	20/90	20/91	20/90
	20m	20/92	20/93	11/89	10/85	3/64	9/87	12/90	25/96	20/91	20/91	20/91
After statement reduction												
Sed	10m	20/77	20/91	14/78	17/79	23/81	22/87	26/94	46/96	20/58	20/57	20/55
	20m	20/79	20/91	14/82	17/84	23/86	22/91	27/95	45/98	20/61	20/59	20/59
Space	10m	20/72	20/85	20/76	21/80	18/65	46/99	48/99	53/99	20/37	20/37	20/35
	20m	20/71	20/80	19/72	21/77	19/65	47/99	48/100	53/99	20/32	20/33	20/31
Grep	10m	20/68	20/76	12/85	17/81	15/79	16/91	22/91	39/97	20/49	20/47	20/47
	20m	20/65	20/76	12/83	16/77	14/81	16/91	22/93	39/98	20/45	20/45	20/45
Gzip	10m	20/71	20/98	11/97	12/97	14/95	13/97	14/97	26/98	20/39	20/39	20/39
	20m	20/69	20/99	11/97	12/97	14/95	13/98	14/97	26/99	20/38	20/39	20/39
Vim	10m	20/64	20/81	19/83	21/84	23/85	65/94	70/96	75/97	20/80	20/80	20/79
	20m	20/69	20/79	19/90	20/90	22/88	65/97	69/98	74/99	20/86	20/83	20/81
YAFFS2	10m	20/87	20/88	10/83	10/81	3/61	9/80	12/83	25/91	20/85	20/86	20/85
	20m	20/87	20/87	11/84	10/80	3/59	9/80	12/83	25/91	20/87	20/86	20/86
After branch reduction												
Sed	10m	20/77	20/91	14/77	17/77	23/77	22/84	26/92	46/96	20/57	20/57	20/54
	20m	20/79	20/91	14/82	17/83	23/82	22/88	27/93	45/98	20/61	20/60	20/59
Space	10m	20/77	20/97	20/89	21/91	18/76	46/100	48/100	53/100	20/51	20/50	20/49
	20m	20/78	20/98	19/88	21/90	19/80	47/100	48/100	53/100	20/48	20/46	20/44
Grep	10m	20/69	20/77	12/86	17/82	15/80	16/92	22/93	39/97	20/49	20/47	20/48
	20m	20/66	20/77	12/83	16/77	14/81	16/91	22/93	39/98	20/47	20/46	20/47
Gzip	10m	20/74	20/99	11/94	12/94	14/91	13/94	14/94	26/94	20/46	20/46	20/47
	20m	20/69	20/99	11/98	12/98	14/96	13/97	14/98	26/99	20/38	20/39	20/39
Vim	10m	20/65	20/82	19/85	21/86	23/87	65/95	70/96	75/97	20/81	20/81	20/79
	20m	20/70	20/81	19/89	20/89	22/87	65/97	69/98	74/99	20/86	20/82	20/80
YAFFS2	10m	20/88	20/88	10/83	10/82	3/61	9/80	12/83	25/91	20/85	20/86	20/86
	20m	20/87	20/89	11/83	10/80	3/58	9/79	12/84	25/92	20/88	20/87	20/87
After AIMP reduction												
Sed	10m	20/77	20/89	14/76	17/76	23/79	22/89	26/90	46/97	20/56	20/55	20/54
	20m	20/77	20/90	14/78	17/77	23/81	22/88	27/91	45/98	20/56	20/55	20/54
Space	10m	20/75	20/98	20/90	21/92	18/78	46/100	48/100	53/100	20/52	20/51	20/50
	20m	20/78	20/98	19/88	21/90	19/80	47/100	48/100	53/100	20/48	20/46	20/44
Grep	10m	20/89	20/85	12/91	17/93	15/93	16/94	22/95	39/98	20/76	20/76	20/76
	20m	20/88	20/86	12/92	16/93	14/93	16/94	22/95	39/99	20/76	20/76	20/76
Gzip	10m	20/75	20/99	11/94	12/94	14/92	13/93	14/93	26/94	20/49	20/50	20/52
	20m	20/70	20/99	11/98	12/98	14/97	13/97	14/97	26/99	20/39	20/40	20/40
Vim	10m	20/62	20/84	19/83	21/84	23/84	65/94	70/96	75/97	20/78	20/77	20/76
	20m	20/72	20/83	19/90	20/90	22/89	65/98	69/98	74/99	20/86	20/83	20/81
YAFFS2	10m	20/88	20/89	10/84	10/82	3/62	9/80	12/83	25/93	20/85	20/86	20/86
	20m	20/88	20/88	11/85	10/81	3/60	9/81	12/84	25/92	20/86	20/86	20/85
Average results		20/76	20/89	14/87	16/87	16/82	29/93	32/94	44/97	20/64	20/63	20/62

Table B.4: Test case selection values (#/%) with different prioritization methods on 100 random selected test cases (Search strategy: MD2U)

Subject	Time	Random	Shortest Path	FPF (slope: -1)			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/83	20/84	14/88	17/87	23/92	22/96	26/96	46/98	20/71	20/71	20/69
	20m	20/77	20/86	14/87	17/85	23/90	22/94	27/93	45/96	20/69	20/68	20/67
Space	10m	20/79	20/100	20/98	21/100	18/100	46/100	48/100	53/100	20/76	20/76	20/74
	20m	20/78	20/100	19/99	21/99	19/99	47/100	48/100	53/100	20/71	20/70	20/67
Grep	10m	20/89	20/90	12/93	17/94	15/94	16/95	22/95	39/99	20/78	20/77	20/75
	20m	20/90	20/90	12/94	16/94	14/93	16/95	22/95	39/99	20/78	20/78	20/76
Gzip	10m	20/68	20/100	11/98	12/97	14/95	13/97	14/97	26/97	20/31	20/32	20/33
	20m	20/65	20/100	11/98	12/98	14/97	13/98	14/97	26/98	20/28	20/29	20/29
Vim	10m	20/63	20/83	19/84	21/84	23/83	65/94	70/94	75/97	20/78	20/75	20/73
	20m	20/70	20/82	19/87	20/86	22/85	65/96	69/97	74/99	20/83	20/79	20/78
YAFFS2	10m	20/92	20/91	10/88	10/87	3/65	9/86	12/89	25/94	20/90	20/92	20/91
	20m	20/92	20/91	11/89	10/86	3/65	9/86	12/89	25/95	20/91	20/91	20/91
After statement reduction												
Sed	10m	20/78	20/90	14/81	17/82	23/81	22/86	26/95	46/95	20/65	20/65	20/62
	20m	20/79	20/91	14/83	17/84	23/85	22/92	27/95	45/98	20/63	20/61	20/61
Space	10m	20/65	20/74	20/70	21/72	18/59	46/99	48/99	53/99	20/34	20/33	20/33
	20m	20/65	20/69	19/63	21/67	19/56	47/98	48/97	53/97	20/28	20/28	20/27
Grep	10m	20/68	20/78	12/86	17/83	15/80	16/93	22/93	39/97	20/44	20/42	20/42
	20m	20/65	20/77	12/84	16/78	14/82	16/93	22/94	39/98	20/43	20/43	20/42
Gzip	10m	20/70	20/99	11/97	12/97	14/94	13/97	14/97	26/98	20/36	20/36	20/36
	20m	20/65	20/99	11/96	12/97	14/92	13/97	14/96	26/97	20/33	20/33	20/34
Vim	10m	20/64	20/81	19/84	21/86	23/86	65/95	70/96	75/98	20/82	20/81	20/79
	20m	20/68	20/79	19/86	20/87	22/85	65/97	69/97	74/98	20/84	20/80	20/80
YAFFS2	10m	20/87	20/87	10/83	10/81	3/62	9/79	12/83	25/93	20/86	20/87	20/86
	20m	20/86	20/85	11/82	10/80	3/58	9/78	12/82	25/92	20/87	20/86	20/86
After branch reduction												
Sed	10m	20/77	20/91	14/80	17/80	23/79	22/84	26/93	46/96	20/63	20/64	20/60
	20m	20/79	20/91	14/83	17/83	23/83	22/89	27/93	45/98	20/64	20/62	20/61
Space	10m	20/75	20/99	20/88	21/90	18/77	46/99	48/99	53/100	20/52	20/52	20/50
	20m	20/77	20/98	19/88	21/90	19/80	47/100	48/100	53/100	20/48	20/46	20/44
Grep	10m	20/70	20/78	12/88	17/83	15/82	16/94	22/95	39/98	20/49	20/48	20/47
	20m	20/67	20/77	12/86	16/79	14/83	16/93	22/95	39/98	20/47	20/47	20/46
Gzip	10m	20/68	20/99	11/93	12/93	14/91	13/93	14/93	26/95	20/37	20/37	20/37
	20m	20/63	20/99	11/97	12/95	14/93	13/96	14/95	26/97	20/31	20/31	20/32
Vim	10m	20/65	20/84	19/84	21/85	23/86	65/95	70/96	75/96	20/82	20/81	20/80
	20m	20/69	20/79	19/87	20/87	22/86	65/96	69/97	74/98	20/85	20/81	20/81
YAFFS2	10m	20/88	20/89	10/83	10/82	3/61	9/80	12/84	25/93	20/88	20/89	20/88
	20m	20/85	20/86	11/82	10/81	3/58	9/79	12/83	25/93	20/88	20/88	20/87
After AIMP reduction												
Sed	10m	20/79	20/87	14/80	17/80	23/82	22/89	26/91	46/98	20/65	20/65	20/63
	20m	20/78	20/90	14/81	17/79	23/83	22/90	27/92	45/98	20/60	20/59	20/58
Space	10m	20/70	20/97	20/85	21/92	18/77	46/100	48/100	53/100	20/52	20/52	20/50
	20m	20/77	20/99	19/88	21/91	19/81	47/99	48/99	53/100	20/49	20/47	20/45
Grep	10m	20/89	20/89	12/93	17/94	15/94	16/95	22/95	39/99	20/77	20/77	20/76
	20m	20/90	20/89	12/92	16/94	14/93	16/94	22/94	39/98	20/78	20/77	20/75
Gzip	10m	20/72	20/99	11/92	12/93	14/91	13/90	14/92	26/93	20/43	20/44	20/45
	20m	20/65	20/99	11/98	12/98	14/96	13/97	14/97	26/98	20/32	20/33	20/32
Vim	10m	20/63	20/83	19/84	21/83	23/83	65/94	70/95	75/97	20/79	20/77	20/74
	20m	20/71	20/81	19/86	20/87	22/85	65/97	69/97	74/99	20/83	20/80	20/79
YAFFS2	10m	20/87	20/87	10/83	10/82	3/62	9/79	12/82	25/92	20/85	20/87	20/85
	20m	20/88	20/87	11/84	10/81	3/59	9/80	12/83	25/92	20/86	20/87	20/85
Average results		20/75	20/89	14/87	16/87	16/82	29/93	32/94	44/97	20/64	20/63	20/62

Table B.5: Test case selection values (#/%) with different prioritization methods on 200 random selected test cases (Search strategy: DFS)

Subject	Time	Random	Shortest	FPF (slope: -1)			CA			CT		
			Path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/33	20/97	24/56	30/54	37/72	28/84	32/82	63/83	20/7	20/7	20/7
	20m	20/39	20/95	25/62	31/60	35/70	28/82	32/86	62/82	20/6	20/7	20/6
Space	10m	20/66	20/98	34/97	31/98	27/99	66/100	68/100	77/100	20/58	20/53	20/49
	20m	20/51	20/99	34/96	31/98	27/99	66/100	69/100	76/100	20/46	20/49	20/58
Grep	10m	20/45	20/88	16/63	21/65	19/62	19/67	26/72	50/78	20/36	20/32	20/26
	20m	20/29	20/92	16/72	21/77	20/73	19/87	26/88	50/72	20/15	20/13	20/10
Gzip	10m	20/40	20/91	19/100	20/100	21/100	19/100	20/100	34/100	20/27	20/27	20/27
	20m	20/40	20/91	19/100	20/100	21/100	19/100	20/100	34/100	20/27	20/27	20/27
Vim	10m	20/38	20/62	29/69	32/68	32/61	104/85	112/86	125/88	20/54	20/53	20/49
	20m	20/36	20/49	29/65	32/69	30/61	104/90	112/93	124/93	20/56	20/53	20/52
YAFFS2	10m	20/44	20/40	15/39	14/36	3/18	9/26	12/31	29/43	20/33	20/36	20/34
	20m	20/47	20/43	16/43	14/38	4/21	9/26	12/31	29/40	20/31	20/32	20/31
After statement reduction												
Sed	10m	20/56	20/93	24/53	30/67	37/75	28/86	32/91	63/95	20/6	20/6	20/6
	20m	20/65	20/95	25/88	31/88	35/85	28/86	32/90	62/92	20/6	20/6	20/6
Space	10m	20/36	20/50	34/42	31/43	27/35	66/97	68/96	77/97	20/11	20/11	20/10
	20m	20/36	20/43	34/42	31/43	27/39	66/96	69/95	76/95	20/9	20/9	20/10
Grep	10m	20/30	20/74	16/85	21/70	19/66	19/88	26/91	50/95	20/4	20/4	20/3
	20m	20/39	20/76	16/86	21/74	20/67	19/90	26/87	50/93	20/7	20/5	20/5
Gzip	10m	20/57	20/82	19/98	20/99	21/99	19/96	20/96	34/99	20/44	20/44	20/45
	20m	20/69	20/79	19/96	20/96	21/97	19/96	20/94	34/97	20/54	20/54	20/54
Vim	10m	20/43	20/54	29/63	32/64	32/58	104/83	112/86	125/87	20/53	20/51	20/47
	20m	20/40	20/48	29/69	32/71	30/65	104/90	112/91	124/93	20/60	20/55	20/56
YAFFS2	10m	20/44	20/43	15/39	14/35	3/19	9/27	12/32	29/43	20/33	20/36	20/34
	20m	20/48	20/41	16/40	14/37	4/20	9/27	12/31	29/41	20/35	20/36	20/36
After branch reduction												
Sed	10m	20/54	20/94	24/51	30/57	37/61	28/76	32/88	63/96	20/6	20/6	20/5
	20m	20/57	20/95	25/77	31/77	35/63	28/73	32/85	62/90	20/5	20/5	20/5
Space	10m	20/60	20/98	34/86	31/94	27/91	66/100	68/100	77/100	20/34	20/35	20/33
	20m	20/71	20/100	34/96	31/99	27/94	66/100	69/100	76/100	20/32	20/33	20/34
Grep	10m	20/28	20/75	16/86	21/70	19/68	19/91	26/94	50/98	20/4	20/3	20/2
	20m	20/35	20/79	16/86	21/71	20/65	19/90	26/88	50/96	20/3	20/3	20/2
Gzip	10m	20/57	20/82	19/98	20/99	21/99	19/96	20/98	34/98	20/44	20/44	20/45
	20m	20/46	20/88	19/99	20/99	21/99	19/98	20/98	34/99	20/32	20/32	20/33
Vim	10m	20/40	20/55	29/67	32/71	32/66	104/87	112/91	125/91	20/54	20/53	20/52
	20m	20/39	20/47	29/73	32/74	30/72	104/91	112/93	124/94	20/62	20/61	20/60
YAFFS2	10m	20/44	20/42	15/44	14/41	3/18	9/26	12/30	29/43	20/30	20/32	20/31
	20m	20/51	20/54	16/43	14/40	4/19	9/39	12/40	29/53	20/42	20/41	20/41
After AIMP reduction												
Sed	10m	20/44	20/84	24/50	30/55	37/62	28/76	32/78	63/94	20/6	20/6	20/6
	20m	20/48	20/85	25/56	31/58	35/65	28/76	32/81	62/94	20/6	20/6	20/6
Space	10m	20/60	20/98	34/86	31/94	27/91	66/100	68/100	77/100	20/34	20/35	20/33
	20m	20/66	20/100	34/94	31/97	27/91	66/100	69/100	76/100	20/32	20/33	20/34
Grep	10m	20/42	20/88	16/65	21/74	19/60	19/73	26/74	50/66	20/35	20/29	20/24
	20m	20/46	20/86	16/67	21/71	20/66	19/76	26/77	50/79	20/34	20/30	20/23
Gzip	10m	20/45	20/87	19/98	20/99	21/99	19/96	20/99	34/99	20/33	20/33	20/34
	20m	20/45	20/88	19/99	20/100	21/99	19/98	20/97	34/99	20/33	20/32	20/33
Vim	10m	20/38	20/54	29/59	32/61	32/62	104/86	112/89	125/88	20/45	20/46	20/48
	20m	20/42	20/54	29/74	32/75	30/74	104/91	112/92	124/93	20/63	20/60	20/61
YAFFS2	10m	20/44	20/40	15/44	14/40	3/18	9/24	12/29	29/41	20/30	20/33	20/32
	20m	20/50	20/58	16/42	14/41	4/20	9/39	12/41	29/55	20/44	20/44	20/43
Average results		20/46	20/74	23/71	25/71	23/66	41/79	45/82	63/85	20/30	20/30	20/29

Table B.6: Test case selection values (#/%) with different prioritization methods on 200 random selected test cases (Search strategy: RP)

Subject	Time	Random	Shortest	FPF (slope: -1)			CA			CT		
			Path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/89	20/83	24/89	30/91	37/92	28/93	32/94	63/96	20/76	20/76	20/74
	20m	20/88	20/80	25/90	31/93	35/93	28/92	32/93	62/97	20/81	20/83	20/83
Space	10m	20/100	20/100	34/100	31/100	27/100	66/100	68/100	77/100	20/100	20/100	20/100
	20m	20/100	20/100	34/100	31/100	27/100	66/100	69/100	76/100	20/100	20/100	20/100
Grep	10m	20/94	20/96	16/95	21/97	19/96	19/96	26/97	50/98	20/92	20/92	20/92
	20m	20/88	20/95	16/90	21/91	20/91	19/90	26/92	50/94	20/86	20/87	20/85
Gzip	10m	20/59	20/98	19/96	20/96	21/96	19/96	20/96	34/96	20/25	20/25	20/25
	20m	20/60	20/96	19/96	20/96	21/96	19/96	20/96	34/96	20/25	20/25	20/25
Vim	10m	20/77	20/83	29/94	32/95	32/93	104/98	112/99	125/99	20/88	20/89	20/87
	20m	20/74	20/81	29/92	32/94	30/91	104/96	112/97	124/96	20/85	20/86	20/85
YAFFS2	10m	20/85	20/87	15/85	14/80	3/53	9/78	12/84	29/91	20/85	20/90	20/88
	20m	20/86	20/87	16/85	14/82	4/59	9/81	12/84	29/92	20/86	20/90	20/89
After statement reduction												
Sed	10m	20/91	20/83	24/93	30/95	37/95	28/96	32/97	63/99	20/63	20/63	20/62
	20m	20/91	20/84	25/93	31/95	35/96	28/97	32/97	62/99	20/80	20/80	20/80
Space	10m	20/99	20/99	34/99	31/100	27/99	66/99	68/100	77/100	20/99	20/100	20/99
	20m	20/77	20/92	34/76	31/79	27/82	66/100	69/100	76/99	20/74	20/75	20/80
Grep	10m	20/81	20/88	16/91	21/86	19/87	19/94	26/95	50/97	20/76	20/76	20/75
	20m	20/67	20/71	16/88	21/78	20/79	19/94	26/95	50/98	20/59	20/60	20/57
Gzip	10m	20/59	20/95	19/92	20/93	21/94	19/90	20/90	34/94	20/30	20/30	20/30
	20m	20/63	20/94	19/93	20/93	21/94	19/93	20/92	34/94	20/32	20/32	20/32
Vim	10m	20/72	20/78	29/94	32/94	32/92	104/98	112/98	125/99	20/85	20/86	20/84
	20m	20/65	20/71	29/88	32/87	30/83	104/92	112/91	124/93	20/74	20/72	20/72
YAFFS2	10m	20/86	20/90	15/87	14/83	3/56	9/80	12/85	29/91	20/86	20/92	20/89
	20m	20/87	20/90	16/86	14/83	4/61	9/81	12/85	29/91	20/87	20/90	20/87
After branch reduction												
Sed	10m	20/90	20/83	24/92	30/94	37/94	28/95	32/97	63/99	20/62	20/62	20/62
	20m	20/90	20/84	25/93	31/95	35/95	28/97	32/97	62/99	20/79	20/80	20/80
Space	10m	20/100	20/100	34/100	31/100	27/100	66/100	68/100	77/100	20/100	20/100	20/100
	20m	20/100	20/100	34/100	31/100	27/100	66/100	69/100	76/100	20/100	20/100	20/100
Grep	10m	20/81	20/88	16/92	21/86	19/87	19/94	26/96	50/97	20/77	20/76	20/75
	20m	20/67	20/71	16/88	21/77	20/78	19/94	26/95	50/98	20/61	20/60	20/58
Gzip	10m	20/59	20/95	19/92	20/92	21/92	19/90	20/90	34/91	20/30	20/30	20/30
	20m	20/60	20/94	19/95	20/96	21/96	19/94	20/94	34/96	20/29	20/29	20/30
Vim	10m	20/74	20/81	29/95	32/96	32/93	104/98	112/99	125/99	20/86	20/87	20/85
	20m	20/67	20/74	29/90	32/91	30/87	104/94	112/95	124/95	20/77	20/78	20/78
YAFFS2	10m	20/87	20/91	15/87	14/82	3/55	9/80	12/86	29/91	20/86	20/91	20/89
	20m	20/87	20/88	16/86	14/82	4/59	9/81	12/86	29/91	20/88	20/91	20/87
After AIMP reduction												
Sed	10m	20/88	20/86	24/90	30/90	37/90	28/93	32/94	63/97	20/76	20/76	20/76
	20m	20/89	20/83	25/91	31/93	35/94	28/93	32/95	62/98	20/76	20/78	20/78
Space	10m	20/100	20/100	34/100	31/100	27/100	66/100	68/100	77/100	20/100	20/100	20/100
	20m	20/100	20/100	34/100	31/100	27/100	66/100	69/100	76/100	20/100	20/100	20/100
Grep	10m	20/94	20/97	16/96	21/97	19/97	19/96	26/98	50/98	20/92	20/92	20/92
	20m	20/91	20/96	16/94	21/95	20/95	19/95	26/96	50/97	20/90	20/89	20/89
Gzip	10m	20/55	20/95	19/73	20/73	21/73	19/70	20/74	34/74	20/30	20/30	20/31
	20m	20/62	20/94	19/95	20/96	21/96	19/94	20/94	34/96	20/29	20/29	20/29
Vim	10m	20/76	20/83	29/94	32/95	32/93	104/98	112/99	125/98	20/87	20/88	20/86
	20m	20/70	20/76	29/90	32/91	30/88	104/93	112/95	124/94	20/80	20/81	20/80
YAFFS2	10m	20/87	20/91	15/87	14/82	3/55	9/79	12/86	29/91	20/86	20/91	20/88
	20m	20/87	20/89	16/86	14/83	4/60	9/82	12/87	29/93	20/88	20/92	20/89
Average results		20/81	20/89	23/92	25/91	23/87	41/92	45/94	63/96	20/75	20/76	20/75

Table B.7: Test case selection values (#/%) with different prioritization methods on 200 random selected test cases (Search strategy: RS)

Subject	Time	Random	Shortest	FPF (slope: -1)			CA			CT		
			Path	Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/74	20/75	24/84	30/85	37/91	28/92	32/92	63/97	20/57	20/57	20/56
	20m	20/68	20/85	25/84	31/84	35/86	28/89	32/91	62/91	20/53	20/53	20/52
Space	10m	20/55	20/100	34/98	31/99	27/98	66/100	68/100	77/100	20/46	20/35	20/30
	20m	20/34	20/100	34/97	31/98	27/99	66/100	69/100	76/100	20/27	20/30	20/45
Grep	10m	20/85	20/89	16/92	21/93	19/92	19/92	26/93	50/97	20/72	20/72	20/71
	20m	20/86	20/90	16/91	21/93	20/92	19/92	26/94	50/98	20/73	20/73	20/72
Gzip	10m	20/62	20/88	19/97	20/97	21/97	19/97	20/97	34/97	20/34	20/35	20/35
	20m	20/63	20/89	19/99	20/99	21/99	19/99	20/99	34/99	20/33	20/33	20/33
Vim	10m	20/55	20/82	29/81	32/81	32/75	104/90	112/93	125/97	20/63	20/64	20/62
	20m	20/59	20/79	29/89	32/90	30/84	104/97	112/97	124/99	20/72	20/74	20/69
YAFFS2	10m	20/89	20/88	15/88	14/84	3/56	9/81	12/85	29/91	20/89	20/89	20/87
	20m	20/90	20/89	16/89	14/86	4/62	9/84	12/87	29/94	20/90	20/91	20/89
After statement reduction												
Sed	10m	20/71	20/84	24/82	30/84	37/84	28/89	32/94	63/94	20/47	20/47	20/47
	20m	20/79	20/86	25/90	31/92	35/92	28/94	32/97	62/99	20/52	20/53	20/52
Space	10m	20/63	20/87	34/77	31/78	27/67	66/99	68/99	77/99	20/20	20/15	20/14
	20m	20/57	20/80	34/75	31/76	27/68	66/100	69/100	76/99	20/10	20/12	20/19
Grep	10m	20/57	20/80	16/85	21/79	19/79	19/91	26/93	50/98	20/42	20/41	20/40
	20m	20/59	20/78	16/87	21/81	20/79	19/93	26/93	50/97	20/40	20/38	20/38
Gzip	10m	20/61	20/87	19/94	20/95	21/96	19/93	20/94	34/96	20/38	20/38	20/38
	20m	20/64	20/87	19/92	20/92	21/92	19/92	20/91	34/93	20/37	20/37	20/37
Vim	10m	20/55	20/79	29/80	32/80	32/76	104/91	112/92	125/95	20/67	20/67	20/65
	20m	20/57	20/76	29/89	32/91	30/84	104/96	112/97	124/98	20/71	20/73	20/69
YAFFS2	10m	20/82	20/82	15/83	14/78	3/51	9/73	12/77	29/86	20/81	20/80	20/81
	20m	20/82	20/79	16/83	14/79	4/55	9/74	12/78	29/87	20/80	20/81	20/82
After branch reduction												
Sed	10m	20/69	20/83	24/80	30/81	37/79	28/84	32/93	63/97	20/45	20/45	20/45
	20m	20/77	20/85	25/90	31/90	35/86	28/90	32/95	62/99	20/51	20/51	20/51
Space	10m	20/62	20/99	34/88	31/89	27/83	66/100	68/100	77/100	20/29	20/21	20/20
	20m	20/58	20/99	34/92	31/95	27/83	66/100	69/100	76/100	20/16	20/19	20/28
Grep	10m	20/58	20/80	16/86	21/80	19/79	19/93	26/94	50/98	20/43	20/41	20/40
	20m	20/59	20/78	16/87	21/80	20/77	19/94	26/94	50/98	20/41	20/41	20/39
Gzip	10m	20/64	20/85	19/90	20/90	21/90	19/89	20/89	34/89	20/43	20/43	20/43
	20m	20/63	20/87	19/96	20/96	21/96	19/95	20/95	34/97	20/37	20/37	20/37
Vim	10m	20/56	20/79	29/82	32/82	32/78	104/92	112/93	125/95	20/67	20/68	20/65
	20m	20/58	20/76	29/88	32/89	30/82	104/96	112/96	124/98	20/71	20/72	20/69
YAFFS2	10m	20/82	20/82	15/83	14/78	3/51	9/73	12/77	29/86	20/81	20/81	20/80
	20m	20/82	20/83	16/83	14/80	4/54	9/74	12/78	29/91	20/86	20/86	20/85
After AIMP reduction												
Sed	10m	20/68	20/80	24/74	30/75	37/80	28/86	32/88	63/96	20/50	20/50	20/50
	20m	20/71	20/83	25/80	31/81	35/82	28/87	32/90	62/97	20/50	20/51	20/50
Space	10m	20/60	20/99	34/88	31/90	27/83	66/100	68/100	77/100	20/29	20/22	20/20
	20m	20/58	20/99	34/92	31/95	27/83	66/100	69/100	76/100	20/16	20/19	20/28
Grep	10m	20/86	20/88	16/90	21/92	19/91	19/91	26/92	50/97	20/72	20/71	20/73
	20m	20/85	20/90	16/90	21/91	20/91	19/91	26/93	50/97	20/73	20/73	20/73
Gzip	10m	20/69	20/81	19/98	20/98	21/98	19/95	20/99	34/98	20/49	20/49	20/50
	20m	20/65	20/87	19/97	20/98	21/97	19/96	20/95	34/98	20/39	20/38	20/39
Vim	10m	20/54	20/83	29/79	32/79	32/76	104/90	112/93	125/95	20/66	20/65	20/64
	20m	20/59	20/79	29/90	32/91	30/85	104/97	112/98	124/99	20/72	20/74	20/69
YAFFS2	10m	20/83	20/83	15/84	14/80	3/52	9/74	12/78	29/87	20/81	20/81	20/80
	20m	20/83	20/81	16/84	14/80	4/56	9/74	12/77	29/87	20/81	20/82	20/79
Average results		20/67	20/85	23/87	25/87	23/81	41/91	45/93	63/96	20/54	20/54	20/53

Table B.8: Test case selection values (#/%) with different prioritization methods on 200 random selected test cases (Search strategy: MD2U)

Subject	Time	Random	Shortest Path	FPF (slope: -1)			CA			CT		
				Stmt	Branch	AIMP	Stmt	Branch	AIMP	Stmt	Branch	AIMP
Before reduction												
Sed	10m	20/79	20/76	24/87	30/87	37/94	28/94	32/95	63/96	20/64	20/64	20/64
	20m	20/67	20/81	25/88	31/88	35/90	28/89	32/90	62/91	20/55	20/55	20/55
Space	10m	20/55	20/100	34/98	31/99	27/98	66/100	68/100	77/100	20/46	20/35	20/30
	20m	20/34	20/100	34/97	31/98	27/99	66/100	69/100	76/100	20/27	20/30	20/45
Grep	10m	20/87	20/93	16/94	21/95	19/94	19/94	26/95	50/98	20/72	20/74	20/68
	20m	20/90	20/93	16/94	21/95	20/94	19/94	26/95	50/98	20/75	20/75	20/72
Gzip	10m	20/55	20/90	19/98	20/98	21/98	19/98	20/98	34/98	20/28	20/29	20/29
	20m	20/57	20/91	19/99	20/100	21/99	19/99	20/99	34/99	20/26	20/26	20/26
Vim	10m	20/54	20/82	29/81	32/81	32/76	104/89	112/92	125/96	20/66	20/67	20/64
	20m	20/58	20/79	29/86	32/87	30/81	104/96	112/97	124/98	20/69	20/72	20/68
YAFFS2	10m	20/89	20/88	15/90	14/86	3/58	9/82	12/85	29/92	20/89	20/89	20/87
	20m	20/90	20/88	16/90	14/86	4/63	9/84	12/86	29/93	20/90	20/90	20/88
After statement reduction												
Sed	10m	20/70	20/83	24/79	30/80	37/81	28/83	32/92	63/91	20/52	20/52	20/52
	20m	20/78	20/86	25/91	31/93	35/92	28/95	32/97	62/99	20/55	20/55	20/55
Space	10m	20/53	20/71	34/67	31/64	27/56	66/98	68/97	77/99	20/18	20/13	20/12
	20m	20/45	20/62	34/59	31/60	27/51	66/94	69/93	76/90	20/9	20/10	20/15
Grep	10m	20/56	20/81	16/86	21/79	19/79	19/93	26/94	50/97	20/36	20/36	20/36
	20m	20/58	20/79	16/88	21/82	20/78	19/94	26/94	50/97	20/38	20/38	20/35
Gzip	10m	20/58	20/88	19/98	20/99	21/100	19/97	20/98	34/99	20/33	20/33	20/33
	20m	20/61	20/89	19/98	20/98	21/99	19/98	20/97	34/99	20/32	20/32	20/32
Vim	10m	20/55	20/78	29/81	32/82	32/78	104/92	112/93	125/95	20/69	20/69	20/67
	20m	20/55	20/75	29/86	32/87	30/81	104/95	112/96	124/97	20/70	20/71	20/68
YAFFS2	10m	20/82	20/82	15/83	14/80	3/51	9/73	12/77	29/88	20/80	20/80	20/82
	20m	20/81	20/77	16/83	14/80	4/55	9/73	12/77	29/89	20/78	20/78	20/83
After branch reduction												
Sed	10m	20/70	20/82	24/81	30/82	37/80	28/82	32/92	63/97	20/52	20/52	20/51
	20m	20/76	20/85	25/91	31/91	35/86	28/90	32/96	62/98	20/56	20/56	20/56
Space	10m	20/59	20/100	34/88	31/87	27/83	66/100	68/99	77/99	20/29	20/22	20/20
	20m	20/56	20/99	34/92	31/95	27/83	66/100	69/100	76/100	20/16	20/19	20/28
Grep	10m	20/58	20/80	16/87	21/81	19/80	19/95	26/95	50/98	20/42	20/42	20/38
	20m	20/59	20/77	16/91	21/82	20/79	19/97	26/96	50/98	20/40	20/40	20/37
Gzip	10m	20/57	20/89	19/98	20/98	21/98	19/97	20/98	34/98	20/33	20/33	20/33
	20m	20/60	20/90	19/98	20/98	21/98	19/97	20/97	34/98	20/29	20/29	20/30
Vim	10m	20/55	20/80	29/82	32/83	32/78	104/92	112/93	125/95	20/69	20/69	20/67
	20m	20/57	20/75	29/86	32/87	30/82	104/96	112/96	124/98	20/72	20/74	20/69
YAFFS2	10m	20/83	20/84	15/83	14/81	3/52	9/74	12/79	29/92	20/86	20/86	20/86
	20m	20/81	20/79	16/82	14/80	4/54	9/75	12/79	29/92	20/84	20/84	20/86
After AIMP reduction												
Sed	10m	20/69	20/67	24/74	30/75	37/79	28/82	32/85	63/98	20/59	20/60	20/58
	20m	20/73	20/83	25/84	31/83	35/85	28/89	32/91	62/98	20/54	20/54	20/54
Space	10m	20/57	20/99	34/87	31/86	27/81	66/100	68/100	77/100	20/29	20/22	20/20
	20m	20/56	20/99	34/92	31/94	27/83	66/100	69/100	76/100	20/16	20/19	20/28
Grep	10m	20/87	20/93	16/94	21/95	19/95	19/95	26/95	50/98	20/76	20/76	20/74
	20m	20/88	20/91	16/93	21/93	20/92	19/94	26/94	50/97	20/75	20/74	20/72
Gzip	10m	20/68	20/85	19/98	20/98	21/98	19/96	20/99	34/99	20/44	20/44	20/45
	20m	20/60	20/89	19/98	20/99	21/98	19/97	20/97	34/99	20/31	20/30	20/31
Vim	10m	20/54	20/82	29/79	32/80	32/77	104/91	112/93	125/96	20/65	20/67	20/63
	20m	20/58	20/78	29/87	32/87	30/82	104/96	112/97	124/98	20/69	20/72	20/69
YAFFS2	10m	20/82	20/81	15/83	14/80	3/52	9/73	12/77	29/87	20/80	20/80	20/78
	20m	20/83	20/78	16/84	14/81	4/56	9/74	12/77	29/89	20/80	20/80	20/79
Average results		20/66	20/85	23/88	25/87	23/81	41/91	45/93	63/96	20/53	20/53	20/53