Comparing Non-adequate Test Suites using Coverage Criteria

Milos Gligoric[1], Alex Groce[2], Chaoqiang Zhang[2],
Rohan Sharma[1], Mohammad Amin Alipour[2], and Darko Marinov[1]
[1] University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
[2] Oregon State University
Corvallis, OR 97331, USA
{gliga,sharma27,marinov}@illinois.edu, agroce@gmail.com, {zhangch,alipour}@onid.orst.edu

ABSTRACT

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. A suite that satisfies 100% of the (feasible) requirements is C*-adequate*. Previous rigorous evaluations of coverage criteria mostly focused on such *adequate* test suites: given criteria C and C', are C-adequate suites (on average) more effective than C'-adequate suites? However, in many realistic cases producing adequate suites is impractical or even impossible.

We present the first extensive study that evaluates coverage criteria 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 intra-procedural acyclic path coverage.

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

General Terms: Experimentation

Keywords: Coverage criteria, non-adequate test suites

1. INTRODUCTION

Software developers run test suites and inspect failures to identify faults. A fundamental task in software testing research is evaluating (and improving) test suites. For example, evaluating suites is central to the development of automated test-generation techniques whose goal is to generate high-quality suites. The primary quality measure for a suite is the number of real faults it can find in the code under test.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '13, July 15-20, 2013, Lugano, Switzerland
Copyright 13 ACM 978-1-4503-2159-4/13/07 ...$15.00.

To compare suites, researchers typically use real faults, seeded faults, and/or coverage criteria. For real faults, researchers measure how many faults (previously known or newly found) the suites find. However, collecting code with real faults and analyzing failures takes substantial effort. Thus, experiments often use a relatively small set of real faults, preventing rigorous statistical analysis [5].

Researchers also use mutation testing [19, 32, 38] to seed a large number of artificial faults and measure the mutation score, i.e., how many mutants a suite kills. Several studies [3,4,58] show that the results obtained on mutants predict detection of real faults, i.e., suites that kill more mutants are *likely, on average,* to find more real faults. While mutation testing can provide a good basis for statistical analysis [5], it can also be prohibitively expensive to perform. Even a small program with only a few hundred lines of code may have thousands of mutants, and determining killed mutants may require running a suite on each mutant.

Researchers therefore most often use *coverage* to compare suites. A traditional coverage criterion provides a finite set of test requirements for the code under test, and one measures how many requirements a given suite satisfies. For example, statement and branch coverage are well-known structural criteria [2]. A suite that satisfies (close to) 100% of the (feasible) requirements for a criterion C is called C-adequate. Measuring test coverage is almost always much cheaper than performing mutation testing; even if the criterion has a high runtime overhead, it only requires running tests once per program, not once per mutant. Coverage criteria are widely used in testing research and practice, e.g., papers on automated testing techniques often report that one technique is better than another because it generates, say, "suites with 10% more branch coverage on average."

This paper addresses the following question: What coverage criteria should researchers use to evaluate suites? Research comparing[1] coverage criteria dates back at least 20 years [21, 22, 36] but has largely *focused on adequate test suites*: given two criteria C and C', do C-adequate suites (on average) find more faults than C'-adequate suites? However, testing practice and research widely use non-adequate test suites because determining which test requirements are feasible is hard, generating suites for all feasible requirements is often impractical, and some recently used criteria [6, 12, 13, 26, 27, 44, 47, 51, 54] even have an infinite (or astronomically large) set of requirements.

[1] Note that we use the term "comparison" to refer to both comparisons of suites and comparisons of coverage criteria, but the intended use should be clear from the context.

We present the first *extensive* study that evaluates coverage criteria over *non-adequate suites*. This paper focuses on two critical questions. First, are *any* coverage criteria able to predict mutation scores for non-adequate suites, and thus suitable for use in evaluations? Second, given two criteria C and C', is it better to use C or C' 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 $c'_1, c'_2 \ldots c'_n$?

To illustrate the key difference in comparisons with adequate and non-adequate suites, consider a comparison of statement coverage (SC) and branch coverage (BC). For adequate suites, it is well known that BC subsumes SC: a suite with 100% BC would have 100% SC and should, on average, be likely to find more faults than another suite with 100% SC but less than 100% BC. For non-adequate suites, however, the situation is less clear. For instance, suppose a suite T_1 has 50% BC and 75% SC, and a suite T_2 has 60% BC and 65% SC. (Our experiments show that up to 8% of test-suite pairs have such discordant values for BC and SC.) Should we use BC and declare T_2 better (60%>50%), or should we use SC and declare T_1 better (75%>65%)?

The major contribution of this paper is an evaluation of multiple criteria, both traditional (statement and branch) and recently used (based on program paths and predicates). We evaluated criteria on a large set of Java and C programs with both manually written and automatically generated tests. We measured the effectiveness of criteria (using two statistical correlation coefficients) in terms of how well they predicted the mutation scores of suites (and thus, arguably, the real-fault detection of suites [3,4,58]). We designed our experiments to have a direct application to the evaluation of suites (and thus testing techniques) in testing research, and propose that our experimental approach would easily extend to other criteria and subjects. A minor contribution of this paper is the first implementation and evaluation of Ball's predicate-complete test coverage criterion [6,7].

Our results show that a variety of criteria are able to effectively predict mutation scores. This provides support for previous research studies that used these criteria to compare test suites. Moreover, for future studies, we propose two guidelines for researchers using coverage criteria to evaluate suites. First, branch coverage performs as well as or better than all other criteria studied, in terms of ability to predict mutation scores, and has a very low measurement overhead and implementation complexity. However, in some settings, branch coverage provides values that do not distinguish between test suites. Second, if researchers want a stronger criterion that can distinguish more test suites, but comes at a price in increased measurement overhead and implementation complexity, our results show that an acyclic intra-procedural variation of path coverage is about as effective as branch coverage. Our results also demonstrate that for *non-adequate suites*, criteria that are stronger (in terms of subsumption for *adequate suites*) do *not necessarily* have better ability to predict mutation scores. All tools and more results are available at: `http://mir.cs.illinois.edu/coco/`.

2. COVERAGE CRITERIA

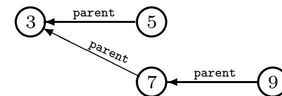
Our comparison of criteria includes SC and BC, which are standard, and a set of coverages based on program paths and predicates, which we define and illustrate in this section using a simple Java data structure. Figure 1(a) shows the relevant part of a class implementing the binomial heap data structure [16,51]. Each `BinomialHeap` object has a pointer to the root of the heap (`nodes`) and the number of nodes in the heap (`size`). Every node keeps a value (`key`) and pointers to parent, sibling, and child. The `decreaseKey` method decreases the value of a node, which may affect the heap invariant that each parent should not have a higher value than its children, so the value is propagated to ancestors until the appropriate position is found.

2.1 Intra-Method Path Coverages

We next describe two forms of path-based coverage (IMP and AIMP) used in our evaluation. Whole-program path coverage was proposed over 20 years ago [41] 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 [13,25,27,54] 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 [24]). 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 BC (and thus SC) but faces the problem that loops introduce an unbounded number of test requirements.

Our second variant of path coverage, *acyclic intra-method paths (AIMP)*, retains subsumption of BC but bounds the total number of requirements by considering only acyclic paths in intra-method control-flow graphs [8]. 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[2]. Ball and Larus present an efficient approach to compute AIMP coverage [8].

Figure 1(b) shows an instrumented version of `decreaseKey` that can be used to collect IMP and AIMP coverages. (The `p$` methods will be discussed in the next section.) `Coverage.beginMethod` and `Coverage.endMethod` are invoked at the beginning and end of the method, respectively, and they are used to begin and end a path. `Coverage.cover` is invoked at each basic block and is used to collect the block IDs in a path. In addition, for AIMP, the `Coverage.cover` method may end the current path and start a new path if the block ID is repeated on the current path. For example, consider the following instance of `BinomialHeap`:



Invoking `decreaseKey` on that heap with arguments `(9, 8)` executes the IMP $0 \rightarrow 2 \rightarrow 4$ and covers the same path for AIMP. (Note that 0, 2, and 4 refer to ids of basic blocks). Invoking `decreaseKey` on that heap with `(9, 2)` instead executes the IMP $0 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4$ but covers two paths for AIMP: $0 \rightarrow 2 \rightarrow 3$ and $3 \rightarrow 4$. Note that IMP and AIMP

[2]Our AIMP uses the notion of simple path common in graph theory, where no vertex is repeated, rather than definition of prime path found in some testing literature [2].

```
1 // public class BinomialHeap { ...
2 static class Node {
3   int key;
4   Node parent;
5   // ...
6 }
7 Node nodes;
8 int size;
9
10 void decreaseKey(int oldValue, int newValue) {
11   Node tmp = nodes.findANodeWithKey(oldValue);
12   if (tmp == null) return;
13   tmp.key = newValue;
14   Node tmpParent = tmp.parent;
15   while ((tmpParent != null)
16          && (tmp.key < tmpParent.key)) {
17     int z = tmp.key;
18     tmp.key = tmpParent.key;
19     tmpParent.key = z;
20     tmp = tmpParent;
21     tmpParent = tmpParent.parent;
22   }
23 }
```

(a) Code snippet from BinomialHeap [51]

```
1 void decreaseKey(int oldValue, int newValue) {
2   try {
3     Coverage.beginMethod(0);
4     Node tmp = nodes.findANodeWithKey(oldValue);
5     if (tmp == null) {
6       Coverage.cover(1, p$10(nodes), p$20(tmp));
7       return;
8     }
9     Coverage.cover(2, p$10(nodes), p$20(tmp));
10
11    tmp.key = newValue;
12    Node tmpParent = tmp.parent;
13    while ((tmpParent != null)
14           && (tmp.key < tmpParent.key)) {
15      Coverage.cover(3, p$10(nodes), p$20(tmp),
16        p$21(tmpParent), p$49(tmp, tmpParent));
17      int z = tmp.key;
18      tmp.key = tmpParent.key;
19      tmpParent.key = z;
20      tmp = tmpParent;
21      tmpParent = tmpParent.parent;
22    }
23    Coverage.cover(4, p$10(nodes), p$20(tmp),
24      p$21(tmpParent), p$49(tmp, tmpParent));
25  } catch (Exception e) {
26    Coverage.endMethod();
27  }
28 }
```

(b) Instrumented method under test

```
1 // tmp.key < tmpParent.key
2 boolean p$49(Node tmp, Node tmpParent) {
3   try {
4     if (PCT.testAndSetInPredicate()) return false;
5     if (tmpParent == null) return false;
6     if (tmp == null) return false;
7     return tmp.key < tmpParent.key;
8   } catch (Exception _) { return false;
9   } finally { PCT.resetInPredicate(); }
10 }
```

(c) An example method for predicate

Figure 1: BinomialHeap as running example

collect paths for *every method* run: e.g., each invocation of `decreaseKey` calls `findANodeWithKey` (which may invoke other methods), so for each invocation, IMP has one path (and AIMP at least one path) for both methods.

2.2 Predicate-Complete Test Coverage (PCT)

Predicate-complete test coverage (PCT) [6, 7] was introduced by Ball as a finite-state alternative to path coverage, inspired by predicate abstraction in model checking [9]. Like path coverage, PCT subsumes both BC and SC, 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 [26, 27, 44, 47, 51] 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. In fact, evaluating predicates both *near and far* from where they appear is what makes PCT even stronger than MC/DC or other related criteria sometimes called "predicate coverage" [2] that evaluate predicates only near where they appear. 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.

We next illustrate PCT using the `BinomialHeap` example. The first step is to extract a set of Boolean predicates from the code under test. Our example code has two conditional statements at lines 12 and 15 (Figure 1(a)), which lead to three predicates: `tmp == null`, `tmpParent != null`, and `tmp.key < tmpParent.key`. Note that we take as a predicate each atomic condition rather than the complex expression. The implicit run-time checks in our example guard against dereferencing null: `nodes != null`, `tmp != null`, and `tmpParent != null`. A key goal for PCT is to extract *all* predicates, as otherwise PCT may not subsume BC or MC/DC.

The second step is to insert evaluation of predicates at *all* appropriate program points. Our tool first generates a method for evaluating each predicate and then inserts calls to these methods. Note that one cannot simply evaluate the predicate as it could lead to problems, e.g., raise an exception if certain variables are `null`. The method for each predicate performs the necessary checks. Figure 1(c) shows the method for `tmp.key < tmpParent.key`. The `Coverage.testAndSetInPredicate` and `Coverage.resetInPredicate` guard against infinite recursion. The `catch` clause handles exceptions in predicate evaluations.

For program points, our PCT tools for Java and C allow instrumenting all statements, *PCT(ST)*, or all beginnings of basic blocks, *PCT(BB)*. Figure 1(b) shows an example instrumentation at the basic-block level. Each `Coverage.cover` call informs the tool that a certain program point (identified with an integer ID) is being executed with a specific combination of predicate values. Note that predicates cannot be evaluated at points where their variables are not in scope, e.g., the predicates for `tmpParent` cannot be evaluated before line 12. Our tools insert evaluation for *all* predicates that can be evaluated. Some predicates can be evaluated far from where they are extracted, e.g., `nodes != null` is evaluated on line 15 (Figure 1(b)), although it is extracted based on line 4 (Figure 1(b)). Some predicates (on instance fields,

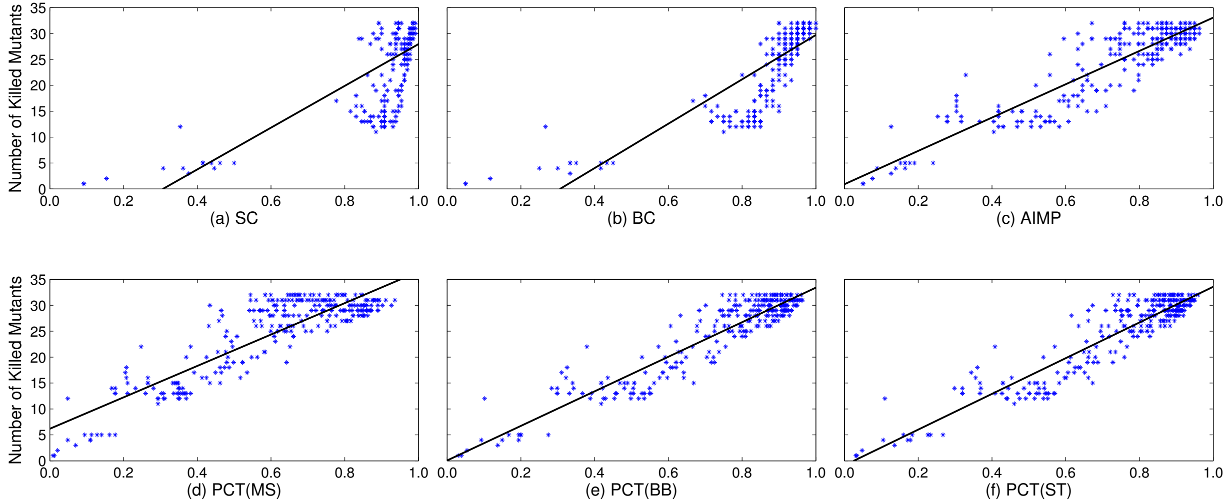


Figure 2: Coverage criteria values and mutation scores correlation for BinomialHeap

rather than on method local variables) can even be extracted in one method and evaluated in another method.

While PCT(BB) maintains the key subsumption properties of PCT over BC, it is only an approximation of PCT(ST) because statements within a block can change predicate values. The example shows that this is not unusual: `tmp.key`, `tmpParent.key`, and `tmp` are all modified inside the block beginning at line 13 (Figure 1(b)) in ways that may introduce combinations of predicate values that will never be seen at basic block entries.

3. EXPERIMENTAL METHODOLOGY

To compare coverage criteria, we examine how well the coverage values predict test suite quality in terms of mutation scores, and we also consider the cost of measuring coverage. We compare two traditional criteria (SC and BC) and two sets of recently used criteria based on paths (IMP and AIMP) and predicates (various PCT coverages).

Specifically we examine the ability of coverage values to *predict (the relative ordering or absolute values of) mutation scores*. To visualize this concept, Figure 2 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 3.2). The X-axis shows coverage, normalized between 0.0 and 1.0, and the Y-axis shows mutation score[3]. 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 *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 3.3. 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.

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

3.1 Experimental Subjects

Programs: Table 1 summarizes the programs used in our experiments, showing the name and number of NBNC (non-blank, non-comment) lines of code (measured by CLOC [15]) 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 [23,26,27,47,48,51]. `JFreeChart` [37] is an open-source library for both interactive and non-interactive manipulation of charts; `JodaTime` [39] is an open-source library for manipulating date and time. For C, seven programs are from the Siemens suite from the SIR repository [20, 36], `Space` [20,53] is a bigger program from the same repository, `SglibRbtree` [52] is the red-black tree implementation from the Sglib library, `YAFFS2` [57] is a widely used open-source flash file system for embedded devices (the default image format for older versions of Android), and `SQLite` [50] is a widely deployed database engine.

Tests: Table 1 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 [26, 27, 47] using three test-generation techniques: random (*Random*), shape abstraction (*ShapeAbs*) [51], and adaptation-based programming (*ABP*) [26,27]. Table 1 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 [28] for `YAFFS2`). For `SQLite` we use manually written tests available from the SQLite repository [50].

Mutants: Table 1 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 exe-

Table 1: Subject programs used in the evaluation

Subject	NBNC	Size of test pool	Mutation killed/mutants	BC branches exe/static	PCT predicates MS	PCT predicates BB,ST	PCT points MS	PCT points BB	PCT points ST	PCT states BB	PCT states 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

cute only *some* core methods for the smaller subjects [47]. 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 [46] to create mutants. Because the number of mutants may be lower than one would expect, it should be noted that Javalanche uses selective mutation [43] 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. [3], which produces mutants based on a set of operators selected through an empirical study on selective mutation [49].

Branch Coverage Information: The BC column in Table 1 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 1 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. [51] and then similarly selected for the remaining structures by Sharma et al. [47]. These programs, manually instrumented for PCT coverage are publicly available [18]. *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. Due to lack of space, we do not show here the detailed numbers for statements, IMP, and AIMP, but these numbers can be found at `http://mir.cs.illinois.edu/coco`.

3.2 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 [34,42]. 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[4], 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.

3.3 Correlation Analysis

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

Kendall τ_b: One core question of this paper 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" [27,51]. The most robust and usefully interpreted statistical measure for this question is the *Kendall τ rank correlation coefficient* [14,40].

Consider the coverage and mutation score data as a set of pairs (C, M), where 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 [17]. 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 [17]. Despite these desirable features of τ_b, our study is among the first to use τ_b in comparison of multiple criteria[5]. (A few studies [42, 55, 56] 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 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.

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 [4, 10, 34, 42], 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 2 shows lines that best fit the observed data.

4. EXPERIMENTAL RESULTS

4.1 Rank Correlation

Table 2 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 [30], we would say 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 [6, 12, 13, 26, 27, 44, 47, 51,

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

[5]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.

Table 2: 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

54]: 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.

4.2 Linear Regression

Table 3 shows R^2 values for our subjects and criteria. For the primary research question of this paper (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 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.

4.3 Test Suite Size

We also examined the importance of suite size as a criterion, because previous work has considered the possibility that coverage criteria are primarily valuable because they force the production of large suites. This is not a major concern for us, because we minimize size as a confounding factor by using a wide range of sizes with numerous suites of each size, and computing τ_b over *all* pairs (including many tied in size). We also note that a trend towards comparing only suites that require the same *computational effort* further reduces the importance of size [27, 29, 33]. For our subjects, using size alone to predict mutation score is an extremely ineffective predictor, with values of τ_b and R^2 much worse than for other criteria (typically < 0.25). Further, using size as an additional variable in regressions [42] did not change our general results: adding either *size* or ln(*size*) to coverage values improved R^2 for PCT criteria most, but BC and AIMP still had higher correlations overall.

4.4 Combining Criteria

After observing the high effectiveness of BC, we attempted to exploit it by using BC as a base criterion and breaking ties with stronger criteria. Specifically, we lexicographically compared pairs, e.g., $\langle BC, AIMP \rangle$, for each suite such that BC is the primary criterion to compare suites, and iff two suites have the same BC, then the second criterion (AIMP in the example) is used to predict the mutation score ranking. However, the correlations were almost uniformly worse than for either criterion alone. It is possible that some other weighting of multiple criteria would perform better than any of the studied approaches; however, the complexity of devising such a scheme and measuring multiple criteria does not make this an immediately attractive approach, given that studied criteria are already effective.

Table 3: 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

4.5 Cost of Measurement

While our key questions are about the predictive power of coverage criteria, we are also interested in the cost of measuring coverage. Table 4 (left side) shows the average overhead of measuring various criteria using our prototype tools. Our implementation of IMP/AIMP is extremely simple; Ball and Larus [8] provide a much faster precise approach, and the hash-based imprecise approach of Hassan and Andrews would also apply [34]. The key point is that our *worst* slowdown was slightly over 108X, and computing mutation score can take over 1000X. In some cases, the instrumented code is faster, due to very low overhead and experimental noise.

4.6 Quality of Mutants

Our results depend on the quality of the mutants, i.e., the difficulty of killing them. If all the mutants are easy to kill, a simple coverage criterion may perform unrealistically well. We therefore compare the percentage of tests that kill specific mutants to execution rates for branches. Table 4 (middle and right side) shows the results; we can see that some mutants, especially for large programs, can be killed by only a small fraction of tests, e.g., only 0.34% of all tests kill the least killed mutant for JFreeChart. It is clear that on average mutants are "harder" than branches for most subjects, with a lower minimum and mean kill/execute rate as well as a higher standard deviation.

5. DISCUSSION

The most surprising result in our study is that BC performs so well. A second somewhat surprising result is that, of non-BC criteria, AIMP performs best and performs *much* better than the more frequently used IMP, despite the fact that IMP subsumes AIMP. We believe that these two results are related. The ranking of criteria (to predict mutation scores) does *not* follow the subsumption hierarchy, although one might expect stronger criteria to predict mutation scores better than weaker criteria do. In fact, in many cases, exactly the opposite is true. Our belief is that there is a fundamental tension between strength and predictive power. Consider a criterion C that is weaker than another criterion C'; C' is most likely a better predictor than C for C-*adequate* suites (e.g., if we have many suites with 100% BC, then we cannot predict varying mutation scores among those suites using BC itself, but we can still use AIMP), but C' is less likely a better predictor than C for C-*non-adequate* suites (e.g., IMP is a worse predictor than AIMP, but BC is a better predictor than SC).

Viewed differently, we can consider the question: how much *information* does the coverage value for one criterion provide about the coverage value for another criterion? We realize that a subsumed criterion often (but not always) provides *more* information about the criterion that subsumes it than the reverse. For example, if a suite has an absolute BC value of k (with each test contributing at least one unique branch), we know that the suite has absolute AIMP, IMP, and PCT values of at least k. However, if we know that a suite has absolute AIMP, IMP, or PCT coverage of k, with each test contributing at least one path or PCT state, the absolute BC may be arbitrarily lower than k. In a sense, the weaker criteria in these cases provide "more" informa-

Table 4: (left) Overhead measured as ratio of execution time of all tests on instrumented to original code; (middle and right) Statistics about percentage of tests that kill a mutant and execute a branch

Subject	Overhead/Slowdown						Tests killing mutant [%]				Tests executing branch [%]			
	SC	BC	IMP	PCT MS	PCT BB	PCT ST	Min	Max	Mean	SD	Min	Max	Mean	SD
language: Java														
JFreeChart	4.21	3.71	3.84	-	4.30	4.79	0.05	26.79	0.34	1.00	0.05	29.72	0.44	1.41
JodaTime	55.38	63.50	92.31	-	67.50	61.88	0.03	75.10	0.61	2.65	0.03	82.42	1.35	5.29
AvlTree	3.73	2.07	39.87	4.14	22.59	21.92	0.01	100.00	41.94	38.69	45.39	100.00	77.05	17.12
BinomialHeap	2.48	2.14	13.01	4.96	11.58	12.27	0.07	98.72	41.86	28.09	2.48	98.72	67.52	24.19
BinTree	2.13	1.63	4.91	2.22	3.65	3.74	1.40	99.23	33.31	32.53	9.77	99.23	74.16	19.13
FibHeap	2.38	1.86	7.65	3.13	5.63	7.54	0.02	100.00	38.45	42.80	2.16	100.00	64.05	39.45
FibonacciHeap	2.05	1.31	5.95	3.00	4.17	5.48	0.02	99.98	32.91	37.54	4.89	99.98	69.60	27.84
HeapArray	1.79	2.00	6.41	2.34	6.62	6.70	1.33	100.00	49.87	37.24	1.48	100.00	59.33	33.26
IntAVLTreeMap	2.29	1.59	15.75	2.48	7.56	7.70	0.04	100.00	61.74	31.46	5.73	100.00	58.89	30.25
IntRedBlackTree	2.13	1.41	10.88	2.65	5.10	6.19	0.00	99.51	17.97	29.87	4.77	99.51	51.75	27.80
LinkedList	1.63	0.94	4.28	1.64	3.15	3.57	69.01	100.00	91.80	13.15	63.43	92.35	76.63	10.49
NodeCachLList	1.56	1.09	6.01	1.74	5.07	5.68	22.52	100.00	69.31	25.38	3.21	94.37	63.14	25.26
SinglyLList	1.97	1.86	5.85	3.22	4.80	5.14	7.15	94.32	41.90	29.95	24.80	94.32	47.70	22.85
TreeMap	2.25	1.62	15.33	3.45	11.41	10.19	0.04	99.29	20.11	26.44	2.29	99.29	40.67	26.34
TreeSet	2.02	1.66	14.11	4.59	10.98	9.24	0.03	99.42	26.96	29.95	3.33	99.42	49.57	27.15
language: C														
Space	0.87	0.87	1.33	-	0.86	1.02	0.07	100.00	17.22	27.41	0.07	100.00	24.67	33.16
SQLite	1.40	1.40	31.83	-	15.87	58.43	0.17	100.00	26.85	38.77	0.21	100.00	26.73	37.33
YAFFS2	1.96	1.96	108.25	-	9.82	28.58	0.02	100.00	32.83	42.23	0.02	100.00	77.61	33.90
Printtokens	1.88	1.88	1.85	-	1.75	1.81	0.17	100.00	38.86	34.60	0.29	99.27	57.95	39.36
Printtokens2	2.29	2.29	2.85	-	2.35	2.86	0.73	99.27	39.17	36.89	0.73	98.54	52.55	36.29
Replace	2.30	2.30	2.68	-	2.17	2.59	0.02	89.32	24.09	24.57	0.40	99.60	39.02	31.53
Schedule	1.33	1.33	1.63	-	1.42	1.57	0.04	100.00	45.61	29.06	0.45	98.87	64.28	30.86
Schedule2	1.82	1.82	2.62	-	1.85	1.99	0.04	85.28	60.40	28.82	0.33	98.86	69.92	36.61
SglibRbtree	0.99	0.99	4.71	-	1.98	2.69	0.70	100.00	81.24	32.05	0.02	100.00	62.60	37.91
Totinfo	1.66	1.66	2.13	-	1.77	1.90	9.16	100.00	44.04	30.50	8.29	99.89	61.26	29.63
Tcas	1.99	1.99	2.01	-	2.27	2.65	0.06	100.00	19.35	32.37	1.87	98.13	24.54	20.49
Geometric mean	2.20	1.90	6.96	2.88	4.75	5.66								

tion about a suite, so we can expect them to better predict mutation score. For example, a suite may obtain very high AIMP coverage without executing most code in the program, if the suite takes a huge number of paths through a single loop with many internal branches; similarly, absolute PCT coverage cannot distinguish between a suite that covers many (irrelevant) states of a small portion of a program and a suite that covers fewer states but executes most of the program. Given the nearly uniform distribution of mutants across a program, suites that do not execute most of the code are likely to have poor mutation scores. In contrast, a high BC value indicates that many easy-to-kill mutants are almost certainly killed. BC thus "warns" if a suite misses many "easy" faults; IMP/AIMP and PCT may not "warn".

The predictive power of BC weakens considerably, however, as suites approach adequacy, when more ties are seen in BC, but mutation scores continue to diverge. The best predictive coverage may be the criterion that minimizes potentially meaningless information without converging too rapidly on 100% coverage. Among our evaluated criteria, AIMP seems to balance information content and avoidance of ties best: it always has a percentage of tied values for suites that is between the very high percentage of ties for BC and the very low percentages for PCT and IMP criteria. IMP had the lowest percentage of ties of all criteria but also proved the least useful for predicting mutation score.

The usefulness of AIMP is encouraging. Hassan and Andrews have suggested that one reason def-use and other dataflow coverages have been little used in practice, despite encouraging results in some studies, is the difficulty of implementing the required static analyses [34]. AIMP is usually trivial to add to instrumentation for collecting BC, if a fairly high overhead is acceptable (as done in this paper), and can be much more efficient if needed [8]. Moreover, loop-free paths within a single function are intuitively easy to interpret, and Godefroid's compositional approach to dynamic symbolic execution essentially maximizes AIMP [24]. In future studies evaluating test suites, our results suggest that IMP should be replaced with AIMP.

We believe PCT coverage may be less effective than AIMP because it uses *too many* predicates. PCT is inspired by abstraction in software model checking, which does not use all in-scope predicates at all points (which leads to a state-space explosion) but instead only uses those relevant to a specification [11,35]. Investigating whether the superior performance of AIMP truly indicates that path-sensitivity is more important than logical-state-space coverage would require a similar selectivity. Unfortunately, the methods used in model checking are impractical for testing large programs.

5.1 Threats to Validity

The primary threat is to external validity: our set of programs and suites, while fairly large by the standards of previous literature, may not be representative of general results. In particular, we examined a larger number of data structures and a smaller number of real-world programs, and our examples were chosen in a partly opportunistic, rather than random, way: we needed subjects with many tests available or easily produced. Our selection of Java data structures, however, at minimum sheds light on the validity of several previous evaluations of testing techniques over these subjects. Construct validity is primarily threatened by ignoring

some predicates for PCT because of technical constraints (e.g., we were not able to generate predicates in a class where instrumented methods would exceed the 64KB limit set by the Java classfile specification).

6. RELATED WORK

Many previous studies have investigated the effectiveness of coverage criteria. The contribution of this paper is to perform a large-scale study to address the specific needs of researchers now investigating automated testing techniques: given two test suites, likely non-adequate, what criteria are best for predicting the ability of those suites to kill mutants (and thus, arguably, detect faults)? Are criteria more recently adopted by researchers effective for this purpose?

Frankl and Weiss [22] performed an experimental comparison of branch coverage and def-use coverage, showing that def-use is more effective than branch coverage and that there is stronger correlation between def-use and fault detection than branch coverage and fault detection; their primary conclusions concerned *adequate* suites, but some experiments included *non-adequate* suites. Our work targets similar questions but differs in that we compare SC, BC, IMP, AIMP, and PCT coverages, use larger applications, use a much larger set of tests produced by various testing techniques, use (many) mutants as opposed to (few) real bugs, and extensively explore non-adequate test suites.

Cai and Lyu [10] also investigated the correlation between different coverage criteria—branch coverage, decision coverage, P-use, and C-use—and fault detection, using a *linear* regression model. Their conclusions are drawn based on experiments on one (large) example, with 426 mutants and 1,200 tests. Different test suites were formed: all tests, tests from a specification, randomly generated tests, tests that cause exceptions, and tests that do not cause exceptions. Their results showed that coverage criteria were only a moderate indicator for fault detection, with large variance for different test suites. Some other studies [21, 36] also showed small or inconsistent correlation between coverage criteria and fault detection. Namin and Andrews [42] investigated the correlation between coverage criteria, effectiveness, and size of a test suite. The study showed that both coverage and size are *non-linearly* correlated with effectiveness. An additional conclusion was that the best result is achieved if both size and coverage are taken into account. Gupta and Jalote examined the *efficiency* of coverage criteria using minimal *adequate* test suites for statement, branch, and predicate coverage (the latter simply being coverage of all atomic predicates from conditionals measured only at the conditionals, not to be confused with PCT) [31]. In their results, while predicate coverage was the most effective (correlated to mutation score), branch coverage was the most efficient when suite size was considered. Others (e.g., [1]) used smaller programs and suites than the listed studies, and/or only examined small sets of (seeded) faults.

Studies investigating related questions (e.g., which criteria are best for prioritizing/minimizing regression suites) are numerous, with results that also vary, though branch coverage has arguably performed fairly well [45]. Harder et al. examined the power of various adequacy criteria, noting the possibility of size as a confounding factor [33]. Another related work is that of Hassan and Andrews [34], which extends previous work [42] to a comparison of branch coverage, def-use coverage, and a novel coverage, called Multi-point Stride Coverage (MPSC), that has resemblances to a generalized version of AIMP. Their results showed that def-use coverage was highly correlated with branch coverage in practice, branch coverage was more correlated with fault detection than other criteria, and MPSC was fairly well correlated with fault detection. Since some MPSC coverages subsume AIMP, we would like to compare the two methods using rank correlation to see if our findings with respect to strength and predictive power hold here as well. Of all previous studies, we find that only a few [42, 55, 56] mention Kendall τ correlation, and those do not provide a comparison of multiple criteria as candidates for use in evaluating suites. In contrast, we use τ_b to compare criteria.

Ball [6] introduced the theory behind PCT coverage and showed that PCT subsumes branch coverage and various decision coverages, and is incomparable to path coverage. Although PCT was introduced in 2004 and used to compare test-generation techniques, it was not extensively evaluated empirically. Our study is the first that implements PCT and empirically investigates the PCT criteria.

The second category of related work includes studies that used some of our criteria for measuring the quality of test suites, which inspired our efforts. Visser et al. [51] were the first to instrument code for measuring an approximation of PCT coverage and compared a number of advanced test generation techniques against random testing using PCT. Because of the lack of tools that can perform instrumentation for PCT, predicates were inserted manually. Pacheco et al. [44] used the same approach to PCT to demonstrate the effectiveness of feedback in random test generation. Later, Sharma et al. [47] compared random testing and shape abstraction on the same set of predicates as previous studies, but predicates were instantiated systematically at all basic blocks. An extended version of that instrumentation was used recently [26, 27] to evaluate the effectiveness of a new test generation technique based on reinforcement learning.

7. CONCLUSIONS

This paper considers these questions: (1) for researchers wishing to compare test suites but lacking a statistically significant number of real faults and lacking the computational resources to perform mutation testing, is it useful to compare suites using coverage criteria; if so, (2) which criteria are best at predicting mutation scores? Recent literature has shown that these are critical questions to answer, because publications are increasingly using coverage criteria to compare test suites and techniques. Our results suggest that due to high effectiveness and low overhead, researchers should use *branch coverage* to compare suites whenever possible, but all evaluated criteria performed well in terms of predicting mutation score for our subjects. A variation of intra-procedural acyclic path coverage performed best of all non-branch coverage criteria, and has desirable simplicity, ease of implementation, and reasonable overhead.

Acknowledgments. We thank Yu Lin, Qingzhou Luo, and Shalini Shamasunder for discussions about this work, Mladen Laudanovic and Douglas Simpson for help with statistical analysis, Lingming Zhang for help with Javalanche, Jamie Andrews for valuable comments and providing the C mutation tool, and Fredrik Kjolstad for help with WALA. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1054876, CNS-0958199, and CCF-0746856.

8. REFERENCES

[1] M. Adolfsen. Industrial validation of test coverage quality. Master's thesis, University of Twente, 2011.

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

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

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

[5] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering*, pages 1–10, 2011.

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

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

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

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

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

[11] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Correct Hardware Design and Verification Methods*, pages 19–34, 2003.

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

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

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

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

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

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

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

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, 1978.

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

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

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

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

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

[25] A. Groce. (Quickly) testing the tester via path coverage. In *Workshop on Dynamic Analysis*, pages 22–28, 2009.

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

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

[28] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[29] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.

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

[31] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *Softw. Tools Technol. Transf.*, 10:145–160, 2008.

[32] R. G. Hamlet. Testing programs with the aid of a compiler. *Trans. Softw. Eng.*, 3:279–290, 1977.

[33] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *International Conference on Software Engineering*, pages 60–71, 2003.

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

[35] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

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

[37] JFreeChart Home Page. `http://www.jfree.org/`

jfreechart/.

[38] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Trans. Soft. Eng.*, 37:649–678, 2011.

[39] JodaTime Home Page. http://joda-time.sourceforge.net/.

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

[41] J. R. Larus. Whole program paths. In *Programming Language Design and Implementation*, pages 259–269, 1999.

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

[43] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, pages 100–107, 1993.

[44] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

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

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

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

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

[49] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, pages 351–360, 2008.

[50] SQLite Home Page. http://www.sqlite.org/.

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

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

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

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

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

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

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

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