Scientific Research Publishing

Journal of Software Engineering and Applications, 2021, 14, 95-109
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

# A Comparative Evaluation of Test Coverage Techniques Effectiveness

**Xaveria Youh Djam*[1], Nachamada Vachaku Blamah[2], Modesta Ero Ezema[3]**

[1]Department of Computer Science, University of Yaounde I, Yaounde, Cameroon
[2]Department of Computer Science, University of Jos, Jos, Nigeria
[3]Computer Science Department Faculty of Physical Sciences, University of Nigeria Nsukka, Nsukka, Nigeria
Email: kdxaveria@gmail.com

**How to cite this paper:** Djam, X.Y., Blamah, N.V. and Ezema, M.E. (2021) A Comparative Evaluation of Test Coverage Techniques Effectiveness. *Journal of Software Engineering and Applications*, **14**, 95-109.
https://doi.org/10.4236/jsea.2021.144007

**Received:** December 1, 2020
**Accepted:** April 19, 2021
**Published:** April 22, 2021



## Abstract

Software systems have become complex and challenging to develop and maintain because of the large size of test cases with increased scalability issues. Test case prioritization methods have been successfully utilized in test case management. However, the prohibitively exorbitant cost of large test cases is now the mainstream in the software industry. The growth of agile test-driven development has increased the expectations for software quality. Yet, our knowledge of when to use various path testing criteria for cost-effectiveness is inadequate due to the inherent complexity in software testing. Existing researches attempted to address the issue without effectively tackling the scalability of large test suites to reduce time in regression testing. In order to provide a more accurate way of fault detection in software projects, we introduced novel coverage criteria, called Incremental Cluster-based test case Prioritization (ICP), and investigated its potentials by making a comparative evaluation with three un-clustered traditional coverage-based criteria: Prime-Path Coverage (PPC), Edge-Pair Coverage (EPC) and Edge Coverage (EC) based on mutation analysis. By clustering test suites, based on their dynamic run-time behavior, the number of pair-wise comparisons is reduced significantly. To compare, we analyzed 20 functions from 25 C programs, instrumented faults into the programs, and used the Mull mutation tool to generate mutants and perform a statistical analysis of the results. The experimental results show that ICP can lead to cost-effective improvements in fault detection.

## Keywords

Software Testing, Fault Detection, Mutation Analysis, Test Case Prioritization, Control Flow Coverage

## 1. Introduction

Despite the huge advancement in agile test-driven development, the problem of inexhaustible testing continues to pose a major challenge in software quality assurance. The inherent complexity in software testing makes traditional unclustered path testing criteria inadequate. Software testing is an important verification and validation activity to reveal program failures in order to improve the quality of software [1]. Unfortunately, the problem of finding all faults in a program (or proving their absence), for any meaningful program, is inexhaustible. Owing to the complexities in software testing, therefore, developers and testers need ways to evaluate their testing efforts in terms of their ability to detect faults in order to make intelligent decisions about testing. The ability, given a test suite, to predict whether it is effective at finding faults is essential to rational testing efforts. In retrospect, using the set of defects discovered during a software product's lifetime, the quality of a test suite could be evaluated by measuring its ability to detect those faults. Faults directly jeopardize software by decreasing its performance and overall software quality.

Software systems have become complex and challenging to develop and maintain because of scalability issues. Software testing is a demanding task and the challenges of testing large-scale software cannot be overemphasized due to the large test suite size.

However, the question of real concern to researchers and potential users of test adequacy criteria is it possible to reduce the test suite size without compromising quality? Prioritization techniques involving humans present a lot of scalability issues because the maximum number of comparisons a human can make consistently is approximately 100, above this threshold, inconsistency grows significantly, leading to reduced effectiveness. Unfortunately, large-scale systems often contain many test cases potentially requiring more than 100 comparisons.

Furthermore, Developers and Tester would like to know whether the investment in systems to monitor code coverage is worthwhile and whether the effort to cluster test cases that increase coverage is important. They would like to know the additional cost of achieving adequate coverage through the use of incremental clustering using the Analytical Hierarchy Process (AHP), the payback for that cost, and in particular, whether fault detection increases significantly if test sets are adequate or close to adequate according to the criteria.

To address this problem, this paper uses incremental clustering-based prioritization (ICP) approach to reduce the cost of human-interactive prioritization. In our approach, the human tester prioritizes not the individual test cases but clusters of similar test cases.

Globally, this paper seeks to answer the following main research question:

**RQ:** How cost-effective is Incremental Clustering-based Prioritization (ICP) compared to PPC, EPC, and EC in fault detection?

This question was addressed by experimentation on 25 well-known C programs of various sizes. To make our results as relevant as possible to professional

software developers and testers, we searched available public archives for specifications and C programs that would be suitable for the study. We used mutants as proxies for faults. Experiments were performed by comparing the control flow coverage on four testing criteria: ICP, PPC, EPC, and EC using subject programs developed at Siemens Corporate Research.

The primary contributions of this paper are as follows:

1) The paper presents a novel use of clustering in test case prioritization using AHP for fault detection in software projects. A novel Comparison Coverage Matrix Generator was developed, in order to evaluate the effectiveness of a test suite for revealing faults.

2) This paper introduces novel coverage criteria, called Incremental Cluster-based test case Prioritization (ICP), and investigates its potentials by making a comparative evaluation with three unclustered traditional coverage-based criteria: Prime-Path Coverage (PPC), Edge-Pair Coverage (EPC), and Edge Coverage (EC) based on mutation analysis. The results of the empirical study show that APH-based prioritization can lead to cost-effective improvements over un-clustered coverage-based prioritization.

The rest of this paper is organized as follows: Section 2 summarizes relevant prior work on test criteria. Section 3 presents the background of this work. Section 4 introduces the incremental cluster-based test case prioritization technique. Section 5 describes our experimental comparison in detail including the subject pool and the method used results are discussed in Section 6. Treat to validity is discussed in section 7, with a conclusion and future work in Section 8.

## 2. Background

Structural coverage measures have long been studied as a means for evaluating the effectiveness of a test suite.

Test criteria can be compared both theoretically and experimentally. The two most common theoretical comparison techniques are traditional subsumption and the number of test requirements. A test criterion C1 subsumes another test criterion C2 if and only if every set of test cases that satisfies criterion C1 is guaranteed to satisfy C2. For example, if a test set takes every branch in a control flow graph (CFG), that test set is guaranteed to cover every node, thus edge coverage subsumes node coverage (also called statement coverage).

A lot of researches have been done on coverage criteria to determine their effectiveness, yet there is still a dual need for a more robust comparative technique to improve the strength of existing researches. Test case prioritization seeks to find an efficient ordering of test case execution to reduce time in regression testing. In recent times, many efforts have been dedicated to code coverage criteria to monitor the thoroughness of software tests [2] [3] [4] [5]. More recently, data flow-based methods have been defined and been implemented in several tools [6] [7]. Various comparisons have been made of the theoretical relations between coverage methods [8] [9] [10] [11] [12].

## 3. Related Work

Test adequacy criteria based on data flow were proposed Hutchins, *et al.* [12]. The first data flow adequacy tool was implemented by Frankl, *et al.* [13] [14], who built the ASSET system that operated on Pascal code in accordance with the definitions of Rapps and Weyuker. Test adequacy criteria based on data flow have been proposed in literature but yet the gap still exists to choose the most appropriate criteria in complex structures in large-scale systems. Many previous studies have attempted to evaluate the cost and effectiveness of test criteria. Frankl, *et al.* [14] [15] [16] carried out several studies comparing data flow criteria with EC, mutation coverage, and manual testing approaches. Ammann and Offutt [17] also compared data flow (all-uses) with mutation testing. The only study that involved edge-pair or prime paths was by Li, *et al.* [18], who compared mutation, EPC, all-uses, and PPC without clustering the test cases. The study found little difference between EPC and PPC, both were stronger than all-uses, and mutation testing was stronger than the other three. Structural coverage is an often used surrogate for fault detection capabilities [19] [20] [21] [22], above studies had a lot of gaps in prioritizing test cases. Our study is a bit larger in terms of subjects and number of test set pool than these older studies with novel coverage criteria, called Incremental Cluster-based test case Prioritization (ICP) with the use of Analytic Hierarchy Process (AHP). AHP algorithm [23] has been used in various software Engineering fields to help decision makers to prioritize tasks.

## 4. Incremental Cluster-Based Test Case Prioritization Technique

We wish to find an approach to reduce delay in testing (reducing testing efforts and cost) without compromising quality. To this end, we employed test case prioritization technique. This paper aims to reduce the number of comparisons required for the pair-wise comparison approach through the use of incremental clustering-based prioritization (ICP) using AHP (Analytical Hierarchical Process) technique, which has been studied in the field of Requirement Engineering. Instead of prioritizing individual test cases, clusters of test cases are prioritized using AHP technique and compare with three unclustered traditional coverage-based criteria: Prime-Path Coverage (PPC), Edge-Pair Coverage (EPC) and Edge Coverage (EC) based on mutation analysis. By clustering test suites, based on their dynamic run-time behaviour, the number of pair-wise comparisons is reduced significantly.

A pair-wise comparison approach for prioritization requires O(n2). The maximum number of comparisons a human can make consistently is approximately 100 [23], above this threshold, inconsistency grows significantly, leading to reduced effectiveness. In order to require less than 100 pair-wise comparisons, the test suite should contain no more than 14 test cases. Considering the scale of real world testing projects, the scalability issues present a significant challenge. For

example, suppose there 1000 test cases to prioritize, the total number of pair-wise comparisons would be 499,500. It is unrealistic to expect a human tester to provide reliable responses for such a large number of comparisons. This research aims to reduce the number of pair-wise comparisons through the use of incremental cluster-based test case prioritization technique using AHP.

The clustering process partitions objects into different subsets so that objects in each group share common properties. The clustering criterion determines which properties are used to measure the commonality. When considering test case prioritisation, the ideal clustering criterion would be the similarity between the faults detected by each test case. However, this information is inherently unavailable before the testing task is finished. Therefore, it is necessary to find a surrogate for this, in the same way as existing coverage-based prioritisation techniques turn to surrogates for fault-detection capabilities [23].

In this paper we utilize dynamic execution traces of each test case as a surrogate for the similarity between features tested. Execution of each test case is represented by a binary string. Each bit corresponds to a statement in the source code. If the statement has been executed by the test case, the digit is 1; otherwise it is 0. The similarity between two test cases is measured by the distance between two binary strings using Hamming distance.

A pair-wise comparison approach for prioritization requires O(n2) comparisons. While redundancy may make pair-wise comparison very robust, the high cost has prevented it from being applied to test case prioritization. The test cases are grouped into clusters: out-of-range, within-range (considering boundary value analysis and equivalent partitioning). It would be more advantageous to execute test suites in incremental clusters than an entire cluster. The latter approach would result in repeating similar parts of SUT before the prioritization technique chooses the next clusters.

In ICP, intra-cluster prioritization is performed first. Based on the results of intra-cluster prioritization, each cluster is assigned a test case that represents the cluster. Using this representative, ICP performs incremental cluster prioritization.

AHP allows the tester to compare two entities with degrees of preference rather than simply binary relations and in this research we call it prioritization score (PS). Previous work using human input for test case prioritization only required binary relations, which were obtained by checking which test case detects more faults than the other. We derived varying degrees of relative importance by checking how much difference there is between the numbers of faults detected by two test cases in an incremental preference.

Suppose two test cases $X_a$ and $Y_b$ are being compared. Let $f_a$ be the number of faults detected by $X_a$ and $f_b$ by $Y_b$, this paper sets the prioritization score (PS) between $X_a$ and $Y_b$ as shown in **Table 1**.

An incremental hierarchical clustering technique was employed and a comparative coverage matrix ($M'$) was derived as shown in **Algorithm 1**.

**Table 1.** Prioritization Score (PS) for the Tester (User).

| Condition | Prioritization Score (*PS*) | Intensity of Importance |
|---|---|---|
| $X_a = Y_b$ | 1 | Equal |
| $X_a > Y_b$ *and* $X_a = 0$ | 2 | Very Strongly Prefer |
| $X_a > 0,\ Y_b > 0,\ X_a \geq 4\,Y_b$ | 3 | Extremely Prefer |
| $X_a > 0,\ Y_b > 0,\ X_a \geq 3\,Y_b$ | 4 | Very Strongly Prefer |
| $X_a > 0,\ Y_b > 0,\ X_a \geq Y_b$ | 5 | Strongly Prefer |

**Algorithm 1.** Comparison coverage matrix generator.

Input: A set of n Test Cases, $T_p$ an ordered set of Clustered, $C_K$

Output: Clusters of Test Cases, $C$, Comparison Coverage Matrix Generator, $M'$

let $T_i \varepsilon T$ be the $i^{th}$ test cases

Form $n$ clusters, each with one test case $T_i$

C ← {}

Add clusters to $C$

Identify clusters $C_K$ with minimum distances from parent node to child node

**For** $i = 1$ *to* $i \leq n$

$M[i, i] = 1\ (1 \leq i \leq n)$

$M[j, j] = 1\ (1 \leq j \leq n)$

$M'(i, j) = (M(i, j)/\Sigma_{i \leq k \leq n} M(i, k))\ ^*PS$

By comparing a test case $T_1$ with other test cases, $T_j$ in pairs so that the value of the level of importance of all the test cases in the form of qualitative opinions is obtained based on the level of fault detection. To change these results into a form of quantitative opinion, the rating scale ratio is used (Table 1). Comparisons are made based on decision-making policies by assessing the importance of one test case to another in revealing fault.

When using multiple criteria, AHP requires the human user to determine the relative importance not only between test cases that are being prioritized (*i.e.* $T_i \varepsilon T$) but also between criteria themselves (*i.e.* expert knowledge and statement-based prioritization). Using Table 1, this paper applies a set of 5 different human-to-coverage preference values called prioritization score, *PS*. The process starts from the hierarchy level which is intended to identify clusters with minimum distances from parent node to child node. Then the arrangement of all ordered clusters, $C_K$ multiply with their corresponding prioritization score (*PS*) will form a Coverage Cluster Matrix, *M'*. The results of comparisons are combined in an $n$ by $n$ matrix $M'$ as shown in the Comparison Coverage Matrix Generator algorithm. Data normalization is done by dividing the elements in each column with the total number per column in question, then the normalized relative weight values are obtained by dividing each element. After that, proceed with calculating the eigenvalue of the vector and testing its consistency, if it is

not consistent then the data retrieval needs to be repeated. The eigenvalue of the vector in question is the maximum eigenvector value obtained from each eigenvector per line. Eigenvector values per line are obtained by dividing the total score on the line of each criterion by the number of columns. The maximum eigenvector value ($E_i$) is obtained by summing the multiplication of the total score in each criterion column with eigenvector per line. That is, the priority weighting vector $E$ is the eigenvector of a matrix $M'$, which is calculated from $M$, by normalizing the columns. $E$ is calculated by taking average across the rows of $M'$:

$$\text{Priority Weighting } Vector\ E_i = (\Sigma_{i \le k \le n} M(i, k))/n$$

## 5. Experimental Design

This section, describes the subject programs, the faulty programs, test cases and the experiments performed. The major results of this paper are gotten from an experiment to compare the cost-effectiveness of ICP, EC, EPC and PPC based on mutation analysis. This paper seeks to answer the following research question:

**RQ:** How cost-effective is Incremental Clustering-based Prioritization (ICP) compared to PPC, EPC and EC in fault detection?

In response to the research question, we designed an experiment with three (3) independent variables and two (2) dependent variables: The first independent variable is the Test Criterion, and it has four (4) values (ICP, EC, EPC, and PPC). The second is the identification of clusters. The third independent variable is the set of mutants. As stated previously, recent researches have questioned whether using all mutants is valid for experimental comparisons of test criteria. This study used the traditional approach of using all mutants as proxies for faults, as well as finding the minimal set (as defined below), and using them as proxies for faults. This not only allows us to have two views of the differences among the criteria we evaluate, but also provides evidence about the claim that experimental comparisons should use minimal sets of mutants. The experiment has two dependent variables: effectiveness and cost. Effectiveness is measured by the number of faults the criterion is able to reveal.

### 5.1. Experimental Subject Program

We used a set of 25 well-known subject programs written (classes) in C as experimental subject programs. These subject programs are the so-called Siemens Suite of Program, which is a well-known open-source program and one of the programs—Space, was developed at the European Space Agency. **Table 2** summaries the subject programs. We chose these programs as they are used in similar researches and accepted as standard. For each program, the table shows the name, number of lines of code, number of mutants generated by all operators, number of equivalent mutants, and number of tests in the mutation adequate test set. Equivalent mutants were determined by hand analysis. The number of mutants yielded from each subject ranged from 18 in TrashAndTakeout to 3987 in PrintTokens 2. The subject programs (**Table 2**) were chosen to meet special

criteria: To allow creation of a reasonable test pool, they must have an understandable specification. Because each program must be understood by several people (to seed faults, and to examine test cases in clusters), they must not be overly complex. But they also have to be large and complex enough to be considered realistic, and to permit the seeding of many hard-to-find errors. Each program must be able to compile and execute as a stand-alone unit. We chose these programs because of the maturity of the associated artifacts and because of their historical significance.

**Table 2.** Description of subject programs.

| Subject Program | LOC | Mutants | Equivalent Mutants | Test pool Size (#Test Cases) |
|---|---|---|---|---|
| Check Palindrome | 110 | 157 | 24 | 9 |
| Digital Reverser | 117 | 389 | 53 | 23 |
| Guassian | 1253 | 19 | 16 | 22 |
| Heap | 1041 | 78 | 67 | 9 |
| Inverse Permutation | 115 | 565 | 47 | 12 |
| Merge Sort | 132 | 991 | 69 | 16 |
| Num Zero | 110 | 187 | 18 | 6 |
| Power | 211 | 278 | 17 | 9 |
| PrintTokens | 726 | 3454 | 137 | 24 |
| PrintTokens 2 | 570 | 3987 | 145 | 26 |
| PrintPrime | 95 | 756 | 25 | 6 |
| Queue | 164 | 467 | 31 | 11 |
| Quicksort | 123 | 1034 | 26 | 12 |
| Recursive Sort | 117 | 546 | 12 | 9 |
| Repace | 564 | 1023 | 178 | 48 |
| Schedule | 412 | 294 | 35 | 35 |
| Schedule2 | 374 | 305 | 45 | 31 |
| Space | 8905 | 38 | 297 | 50 |
| Stack | 156 | 56 | 12 | 12 |
| Tcas | 173 | 29 | 37 | 32 |
| Totinfo | 281 | 245 | 89 | 21 |
| TestPad | 124 | 24 | 7 | 12 |
| TrashAndTakeout | 159 | 18 | 14 | 10 |
| TwoPred | 113 | 23 | 19 | 16 |
| UniCal | 119 | 319 | 3456 | 21 |

### 5.2. Mutation Operators

To generate mutants of the subject programs, we used Mull mutation tool to generate mutants for code written in *C*. To generate mutants from a source file, each line of code was considered in sequence and each of four classes of "mutation operators" was applied (whenever possible). In other words, every valid application of a mutation operator to a line of code resulted in another mutant being generated. The four classes of mutation operators were:

- Replace an integer constant *C* by 0, 1, −1, ((*C*) + 1), or ((*C*) − 1).
- Replace an arithmetic, relational, logical, bitwise logical, increment/decrement, or arithmetic-assignment operator with another operator from the same class.
- Negate the decision in an if or while statement.
- Delete a statement.

The first three operator classes were used to identify a set of "sufficient" mutation operators, *i.e.*, a set S of operators such that test suites that kill mutants formed by S tend to kill mutants formed by a very broad class of operators. They were adapted so that they would work on *C* programs rather than the Fortran of the original research. The fourth operator, was added because some of the subject programs contained a large number of pointer-manipulation and field-assignment statements that would not be vulnerable to any of the sufficient mutation operators. About 12.0% of the resulting mutants did not compile. The numbers of mutants of each subject program that compiled appear in Table 2. For the Space program, there were so many mutants generated that it was infeasible to run them all on the test suite. Therefore we ran the test suite on every 10th mutant generated. Because the number of mutants generated per line did not follow any pattern that would interact with the selection of every $10^{th}$ mutant (it depends on the constructs on the line only), this amounted to a random selection of 10% of the mutants, taken from a uniform distribution over all the possible mutants. Additionally, this ensured that the whole source code was seeded with faults (and not simply a few functions/procedures).

### 5.3. Effectiveness Analysis

The effectiveness of a test case can be measured by its ability to detect faults for both instrumented and none instrumented programs. Test cases of various sizes were used. For each subject program, we created a set of adequacy test sets by hand. The number of test cases for each program is shown in the last column of Table 2: Mull (an open-source mutation tool) was used to generate all mutants. After generating mutants, we designed adequate test sets and identified equivalent mutants by hand. To evaluate the effectiveness of a criterion (ICP, PPC, EPC, EC) for any subject program, we used all the adequate test sets. The effectiveness of any program, is the average of the mutation score. Effectiveness is calculated twice, once for all the mutants generated from each subject program and again for just the weak set of mutants.

### 5.4. Cost Analysis

The cost analysis for each criterion is based on two values:

1) The number of test cases needed;

2) The number of faults detected for clustered and unclustered test cases.

The two measures are dynamic in the sense that they are computed based only on the source code of the system under test (SUT). ICP can detect more faults (Table), especially in programs that have complicated control flows, but at a higher cost. Thus, a practical tester can make an informed cost versus benefit decision. A better understanding of which structures in the programs contribute to the expense might help to choose when to use PPC. This led to an argument that the expense of ICP is worthwhile because it will help the software testers find more faults. In general, human intervention is needed to determine pair-wise comparison and the choice of dissimilarity metric for clustering.

## 6. Results and Discussion

Table 3 shows data regarding the effectiveness of the test sets selected for each criterion. The first column gives the names of the 25 subject programs, as in Table 2. The next four columns, grouped under Mutation Score (full), show the mutation scores on the full set of mutants obtained by the test sets that satisfy ICP, PPC, EPC, and EC. For these programs, ICP-adequate test sets performed better than PPC, EPC- and EC-adequate test sets across the board. Across the programs, none achieved a 100% full mutation score. The next four columns, grouped under Weak Mutation Score, show the mutation score on the minimal set of mutants by the four criteria. As shown in Table 3, the minimal sets have significantly fewer mutants. For instance, the ICP test sets selected for the first subject program (Check Palindrom) killed on average 78% of the mutants.

Considering all subject programs, the mean mutation scores (full) achieved by the test sets for ICP, PPC, EPC, and EC were 98%, 97%, 95% and 95% respectively. The mean mutation scores (weak) achieved by the test sets for ICP, PPC, EPC, and EC were 79%, 78%, 71% and 68% respectively.

Table 4 gives an approximation of the cost-effectiveness ratio of the 4 criteria on the experimental subjects. It is important to note that this table only counts the number of tests, not the cost of creating those tests. The metric cost would vary dramatically by the amount of automation used, particularly if automatic test data generation was available. As in Table 4, these data indicate that PPC is the most efficient in detecting faults and EC the least. Anecdotally, we found out that generating and satisfying the test requirements for ICP was more difficult than PPC, EPC and EC. Finding values to kill the last few mutants was quite time consuming as well as intellectually challenging.

## 7. Treat to Validity

A common threat in software engineering experiments is the representative nature of the programs. No matter how many programs are used, it will never be

**Table 3.** Effectiveness of complete mutant sets and weak mutant set considering reduced requirement.

| Subject Program | Mutation Score (Ful) | | | | Weak Mutation Score | | | |
|---|---|---|---|---|---|---|---|---|
| | PPC | EPC | EC | ICP | PPC | EPC | EC | ICP |
| Check Palindrom | 0.93 | 0.91 | 0.67 | 0.93 | 0.68 | 0.46 | 0.37 | 0.78 |
| Digital Reverser | 0.93 | 0.95 | 0.88 | 0.96 | 0.51 | 0.49 | 0.61 | 0.61 |
| Guassian | 0.93 | 0.77 | 0.94 | 0.94 | 0.62 | 0.57 | 0.44 | 0.72 |
| Heap | 0.96 | 0.93 | 0.74 | 0.97 | 0.63 | 0.54 | 0.24 | 0.73 |
| Inverse Permutation | 0.94 | 0.96 | 0.65 | 0.90 | 0.73 | 0.65 | 0.62 | 0.83 |
| Merge Sort | 0.93 | 0.87 | 0.93 | 0.94 | 0.53 | 0.47 | 0.53 | 0.63 |
| Num Zero | 0.97 | 0.92 | 0.84 | 0.98 | 0.57 | 0.52 | 0.74 | 0.67 |
| Power | 0.87 | 0.91 | 0.74 | 0.88 | 0.77 | 0.54 | 0.71 | 0.87 |
| PrintTokens | 0.98 | 0.93 | 0.83 | 0.99 | 0.58 | 0.43 | 0.52 | 0.68 |
| PrintTokens 2 | 0.94 | 0.97 | 0.86 | 0.95 | 0.64 | 0.47 | 0.46 | 0.74 |
| PrintPrime | 0.96 | 0.98 | 0.87 | 0.94 | 0.58 | 0.58 | 0.47 | 0.68 |
| Queue | 0.98 | 0.89 | 0.97 | 0.99 | 0.68 | 0.69 | 0.37 | 0.78 |
| Quicksort | 0.95 | 0.88 | 0.98 | 0.96 | 0.55 | 0.68 | 0.58 | 0.65 |
| Recursive Sort | 0.97 | 0.92 | 0.91 | 0.98 | 0.57 | 0.51 | 0.51 | 0.67 |
| Repace | 0.89 | 0.86 | 0.76 | 0.87 | 0.81 | 0.46 | 0.36 | 0.83 |
| Schedule | 0.89 | 0.94 | 0.92 | 0.99 | 0.98 | 0.51 | 0.41 | 0.97 |
| Schedule2 | 0.99 | 0.93 | 0.73 | 0.98 | 0.79 | 0.43 | 0.52 | 0.69 |
| SPace | 0.97 | 0.98 | 0.91 | 0.97 | 0.77 | 0.58 | 0.71 | 0.77 |
| Stack | 0.98 | 0.89 | 0.92 | 0.99 | 0.58 | 0.69 | 0.92 | 0.68 |
| Tcas | 0.91 | 0.88 | 0.83 | 0.92 | 0.61 | 0.58 | 0.64 | 0.71 |
| Totinfo | 0.98 | 0.95 | 0.89 | 0.99 | 0.78 | 0.67 | 0.21 | 0.79 |
| TestPad | 0.93 | 0.86 | 0.79 | 0.94 | 0.99 | 0.56 | 0.49 | 0.98 |
| TrashAndTakeout | 0.96 | 0.89 | 0.88 | 0.97 | 0.81 | 0.79 | 0.43 | 0.91 |
| TwoPred | 0.97 | 0.93 | 0.95 | 0.98 | 0.79 | 0.63 | 0.51 | 0.89 |
| UniCal | 0.96 | 0.84 | 0.84 | 0.97 | 0.56 | 0.44 | 0.75 | 0.66 |
| Min(%) | 91 | 84 | 67 | 91 | 0.51 | 0.43 | 0.21 | 0.61 |
| Max(%) | 97 | 0.95 | 0.98 | 97 | 0.79 | 0.79 | 0.75 | 0.89 |
| SD | 0.04 | 0.03 | 0.05 | 0.04 | 0.11 | 0.23 | 0.22 | 0.21 |
| **Mean (Average)** | **0.97** | **0.96** | **0.95** | **0.98** | **0.78** | **0.71** | **0.68** | **0.79** |

**Table 4.** Cost-effectiveness ratio of full mutant sets.

| | Test Cases | Faults | Cost/Effectiveness (%) |
|---|---|---|---|
| PPC | 17,985 | 67 | 0.97 |
| EPC | 1456 | 34 | 0.96 |
| EC | 894 | 25 | 0.95 |
| ICP | 19,972 | 78 | 0.98 |

possible to ensure the results can generalize to all programs. The subjects were from Simen programs and open sources programs, rather than part of thousands of industrial software products. Therefore, the results may not generalize to industrial settings. In principle, given that the programs used in our experimental study are smaller than programs that solve industrial-scale, real-world problems, we cannot conclusively rule out the possibility that the results may not be generalized to larger and more complex programs. Our experimental design required completely adequate test suites, which had to be created by hand, thus limiting the size of the programs. This makes our conclusions more definitive at the potential cost of generalization.

Another threat is the selection of test cases. Given that we adopted the notion of building a minimal set of mutants, we needed a minimal test set to compute it. First, we generate for each program, by hand, a universe of mutation-adequate test cases. This potentially is a threat because the results could be diverse with different test sets. We opted for a reduced test pool because the larger the number of test cases in a test suite, the more likely it is that some test cases would be redundant. This is a somewhat non-intuitive consequence of the notion of minimal sets of mutants; testers really do not need nearly as many tests as we have always thought. We just need the right tests. Unfortunately, creating such test sets for 25 programs is a very demanding task and to create multiple sets for each program would be impractical. The second point is that the selection of test cases to cover structural criteria is also restricted to this universe of test cases and so could have the same threat. Nevertheless, in terms of code coverage, what matters is the sequence in which the test cases are applied since we know that all structural requirements are covered after applying all test cases.

## 8. Conclusion and Future Work

A testing criterion is good if and only if it is capable of revealing faults during the testing process in the system under test. We introduced novel coverage criteria, called Incremental Cluster-based test case Prioritization (ICP) and investigated it potentials by making a comparative evaluation with three unclustered traditional coverage-based criteria: Prime-Path Coverage (PPC), Edge-Pair Coverage (EPC) and Edge Coverage (EC) based on mutation analysis. We experimentally compared ICP, EC, EPC, and PPC in terms of cost and effectiveness using 25 C programs along with their test suits. As expected, the results indicated that ICP depicted more faults than other criteria. Based on the efficiency ratio in Table 4, ICP is the most efficient criterion.

ICP can detect more faults, especially in programs that have complicated control flows, but at a higher cost. Thus, a practical tester can make an informed cost versus benefit decision. A better understanding of which structures in the programs contribute to the expense might help to choose when to use PPC. This led to an argument that the expense of ICP is worthwhile because it will help the software testers find more faults.

Our experimental results open considerably a number of research issues.

With the current development in software testing, the combination of coverage criteria with symbolic execution technology in a large project such as kernel suites is our next line of action. Furthermore, further exploration of the weaknesses/strength of coverage criteria in order to move students from trial-and-error testing to evidence-based testing is equally an important part of future research. Future work equally includes improving the efficiency of our experiments to other problems in software testing, such as software fault prediction, software fault localization, test suite prioritization and test suit minimization.

## Acknowledgements

Many thanks to Siemens Corporate Research Team for making the Siemens and Space programs available for this research. Many thanks to Anne Sah for sharing and discussing her Java DU-coverage tools, which helped me to choose the appropriate mutant tool for C programs. Thanks also to all the researchers who worked on and improved these subject programs and artifacts over the years. We would like to also thank Victor Phaho Blaise for reviewing drafts of this paper.

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] Andrews, J.H. and Zhang, Y. (2003) General Test Result Checking with Log File Analysis. *IEEE Transactions on Software Engineering*, **29**, 634-648. https://doi.org/10.1109/TSE.2003.1214327

[2] Ma, Y.-S., Offutt, J. and Kwon, Y.-R. (2005) Mujava: An Automated Class Mutation System. *Software Testing, Verification, and Reliability*, **15**, 97-133. https://doi.org/10.1002/stvr.308

[3] Hemmati, H. (2015) How Effective Are Code Coverage Criteria? 2015 *IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, 3-5 August 2015, 151-156. https://doi.org/10.1109/QRS.2015.30

[4] Schwartz, A. and Hetzel, M. (2016) The Impact of Fault Type on the Relationship between Code Coverage and Fault Detection. *IEEE/ACM International Workshop in Automation of Software Test*, Austin, 2016, 29-35. https://doi.org/10.1145/2896921.2896926

[5] Papadakis, M., Henard, C., Harman, M., Jia, Y. and Le Traon, Y. (2016) Threats to the Validity of Mutation-Based Test Assessment. *Proceedings of the International Symposium on Software Testing and Analysis*, Saarbrücken, Germany, 2016, 354-365. https://doi.org/10.1145/2931037.2931040

[6] Chekam, T.T., Papadakis, M., Traon, Y.L. and Harman, M. (2017) An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption. *Proceedings of the International Conference on Software Engineering* (*ICSE*), Buenos Aires, Argentina, 20-28 May 2017, 597-608. https://doi.org/10.1109/ICSE.2017.61

[7] Kurtz, B., Ammann, P., Offutt, J. and Kurtz, M. (2016) Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. 2016 *IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops*, Chicago, 11-15 April 2016, 142-151. https://doi.org/10.1109/ICSTW.2016.41

[8] Yan, Z. and Zhang, L.J. (2019) A Data Dependent Parallel Computing Method Based on LLVM Intermediate Representation. *Computer Application Research*, **37**, 437-442.

[9] Ammann, P., Offutt, J. and Xu, W.Z. (2008) Coverage Computation Web Applications. http://cs.gmu.edu:8080/offutt/coverage/

[10] Frankl, P.G., Weiss, S.N. and Hu, C. (1997) All-Uses Versus Mutation Testing: An Experimental Comparison of Effectiveness. *Journal of System Software*, **38**, 235-253. https://doi.org/10.1016/S0164-1212(96)00154-9

[11] Ammann, P. and Offutt, J. (2017) Introduction to Software Testing. 2nd Edition, Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/9781316771273

[12] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T. (1994) Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. *Proceedings of the* 16*th International Conference on Software Engineering*, Sorrento, Italy, 16-21 May 1994, 191-200.

[13] Rothermel, G. and Harrold, M.J. (1998) Empirical Studies of a Safe Regression Test Selection Technique. *IEEE Transsction on Software Engineering*, **24**, 401-419. https://doi.org/10.1109/32.689399

[14] Frankl, P.G. and Iakounenko, O. (1998) Further Empirical Studies of Test Effectiveness. *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, **23**, 153-162. https://doi.org/10.1145/291252.288298

[15] Vokolos, F.I. and Frankl, P.G. (1998) Empirical Evaluation of the Textual Differencing Regression Testing Technique. *Proceedings of International Conference on Software Maintenance*, Bethesda, MD, USA, 20-20 November 1998, 44-53.

[16] Harder, M., Mellen, J. and Ernst, M.D. (2003) Improving Test Suites via Operational Abstraction. *Proceedings of* 25*th International Conference on Software Engineering*, Portland, OR, USA, 3-10 May 2003, 60-71. https://doi.org/10.1109/ICSE.2003.1201188

[17] Ammann, P., Delamaro, M.E. and Offutt, J. (2014) Establishing Theoretical Minimal Sets of Mutants. 2014 *IEEE* 7*th International Conference on Software Testing, Verification and Validation*, Cleveland, 31 March-4 April 2014, 21-30. https://doi.org/10.1109/ICST.2014.13

[18] Li, N., Praphamontripong, U. and Offutt, J. (2009) An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. *Fifth Workshop on Mutation Analysis*, *IEEE Mutation*, Denver, 1-4 April 2009, 220-229. https://doi.org/10.1109/ICSTW.2009.30

[19] Offutt, A.J., Lee, A., Rothermel, G., Untch R.H. and Zapf, C. (1996) An Experimental Determination of Sufficient Mutation Operators. *ACM Transactions on Software Engineering and Methodology*, **5**, 99-118. https://doi.org/10.1145/227607.227610

[20] Lee, J., Kang, S. and Jung, P. (2020) Test Coverage Criteria for Software Product Line Testing: Systematic Literature Review. *Information and Software Technology*, **122**, 301-329. https://doi.org/10.1016/j.infsof.2020.106272

[21] Grano, G., Titov, T.V. and Sebastiano, P.H.C. (2019) Branch Coverage Prediction in

Automated Testing. *Journal of Software Evolution and Process*, **31**, 1-18. https://doi.org/10.1002/smr.2158

[22] Yoo, S., Harman, M., Tonella, P. and Susi, A. (2009) Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, Chicago, 19-23 July 2009, 201-212. https://doi.org/10.1145/1572272.1572296

[23] Saaty, T. (1980) The Analytic Hierarchy Process, Planning, Priority Setting, Resource Allocation. McGraw-Hill, New York, USA.