DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Empirical Evaluation of Test Suite Reduction

Adrian Regenfuß

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Empirical Evaluation of Test Suite Reduction

Empirische Evaluation von Test-Suiten Reduktion

Author:	Adrian Regenfuß
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens, Raphael Nömmer, Roman Haas
Submission Date:	Submission date

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Submission date Adrian Regenfuß

Acknowledgments

Abstract

Test suites have been growing along with codebases, which has resulted in them having long run-times (slowing down the development process) and higher computational needs (costing electricity and hardware). Several different approaches have been developed to reduce the time for test suite execution to give useful results. One of these approaches is test suite reduction: selecting a sample of tests that maximizes coverage on the tested source code while still reducing runtimes. This Bachelor's thesis attempts to replicate the findings in [Cru+19], which borrows techniques from big data to handle very large test suites. From 6 open-source projects, we collect test suites, generate coverage information and fault detection information and perform test suite reductions on on these test suites using the algorithms from [Cru+19]. We find that our experiments largely replicate the rankings on the metrics of fault detection loss and test suite reduction, but disagree on the runtime performance of the different algorithms. We discuss possible reasons for this discrepancy.

Contents

1 Introduction

Software often has faults: ways in which the actual behavior of the software diverges from the intended or specified behavior. Computer scientists have devised different strategies for finding and removing bugs: Formal software verification, code reviews, and different types of testing: unit testing, which tests the behavior of small software modules (such as classes), model-based testing, which automatically generates test suites from the specification of a software system, and regression testing, which re-validates a software after changes have been made, aiming to show developers whether they have introduced new bugs after the last change to the software.

Regression testing takes up a significant portion of development cost (up to 50%, according to [Wol06]), and the resulting test suites can grow quite significantly in size and execution time, which hinders development speed and increases costs.

Similarly, model-based testing suffers from issues of combinatorial explosion, and the resulting test suites tend to be very big.

To mitigate the costs and runtimes of regression and model-based test suites, different strategies have been devised: test case selection, which selects a subset of tests for the current execution of tests, test suite reduction, which permanently deletes a subset of tests from the test suite, and test case priorization, which changes the order of test execution to maximize the amount of faults that is found early in the test suite execution.

[Cru+19] presents a new family of test suite reduction algorithms, called the FAST ("FAST Approaches to Similarity-based Testing") algorithms (first developed in [Mir+18]), and compares them to the algorithms presented in [Che+10], as well as the greedy additional algorithm presented in [Rot+01]. They implement 4 algorithms from the FAST family, 2 from the ART ("Adaptive Random Testing") family and the greedy additional algorithm, and run the different algorithms on 10 different test programs and their test suites. They then compare the performance of the test suite reduction algorithms on 3 different variables: test suite reduction, fault detection loss, and runtime.

This work then attempts to replicate their findings using the data from 6 additional open-source projects, as well as adding random test case selection as a baseline test suite reduction method to compare the other methods to (following Recommendation 8 from [Kha+18]). Specifically, we rank the algorithms with respect to their performance in TSR, FDL and runtime, and compare our rankings to the results from [Cru+19]. The

fact that our findings mostly replicate the findings in [Cru+19] can give researchers and test system designers more reliable information about which test suite reduction approaches to use and research further.

We first explore the field of test suite reduction, touching on other methods for dealing with large test suites. We then explain in detail the algorithms used in [Cru+19].

Finally, we attempt to determine how different test suite reduction strategies compare to each other in terms of time performance, fault detection loss and magnitude of reduction, and close with thoughts on further possible approaches to test suite reduction, as well as thoughts on other issues in the area.

2 Terms and Definitions

This paper uses some terms and abbreviations that are common in the literature on test suite reduction, but not used much elsewhere.

The **System Under Test (SUT)** is a software system being tested using a test suite.

The **Test Suite Reduction (TSR)** is a metric that evaluates the magnitude of the reduction of a test suite. Given a test suite $T = \{t_1, \dots t_n\}$ and a reduced test suite $T' \subset T$, the TSR of the test suite is determined using the formula

$$\text{TSR}(T, T') = 100 * \frac{|T| - |T'|}{|T|}$$

which gives the percentage of tests removed from the test suite.

Everything else equal, higher values for TSR are better.

The abbreviation TSR also sometimes stands for the technique of test suite reduction, i.e. the permanent removal of redundant test cases from a test suite. In this text, "TSR" will only stand for the metric, and the technique will be written out as "test suite reduction".

The **Fault Detection Loss (FDL)** is a metric that evaluates the amount of faults detected by a reduced test suite. Let F be the faults detected by T, and F' be the faults detected by T', then the fault detection loss is calculated as such:

$$\text{FDL}(F, F') = 100 * \frac{|F| - |F'|}{|F|}$$

which returns the percentage of faults not detected anymore.

In general, lower FDL values are better. A related metric (e.g. used in [Kha+16] or [YH12]) is the Fault Detection Effectiveness (FDE), defined as $\text{FDE} = 1 - FDL(F, F')$.

Mutation testing (first described by [Bud80]) is the practice of intentionally introducing faults (e.g. changing comparisons, moving lines or deleting function bodies) into a SUT in order to discover which tests are activated by the faults introduced.

Mutation testing is often often automated.

3 Related Work

The literature on handling large test suites is large, and this summarization will only cover small parts of it. For an older, but comprehensive overview dividing the field into test-suite reduction, test case selection and test case priorization, see [YH12]. A newer overview focused exclusively on test suite reduction is [Kha+16], which attempts to create a full taxonomy of test suite reduction frameworks/tools and specifically their implementions. [Kha+18] focuses on the algorithmic approach (i.e. greedy/clustering/searching/hybrid) of the test suite reduction system and gives recommendations for future the evaluation of test suite reduction methods.

The methods for improving the runtimes of large test suites discussed in the literature can be divided into three related approaches ([YH12]): Test suite minimization (here called test suite reduction), test case priorization, and test case selection.

Let $T = \{t_1, t_2, \ldots, t_n\}$ be a set of n tests, and a set of requirements $R = \{r_1, \ldots, r_m\}$. These requirements can take very different forms: they can correspond to coverage of lines/branches/functions, parts of an explicit specification that must be tested, runtimes of individual tests, or faults that were discovered in the past.

Let $m : T \to \mathcal{P}(R)$ ($\mathcal{P}(R)$ denoting the powerset of R) a function that maps test cases to the requirements they test (one test can test multiple requirements, but any requirement needs only one test case to be fulfilled, i.e. there can be no requirements that need 2 or more tests to be fulfilled).

3.1 Test Case Priorization

Test case priorization attempts to find a permutation $T' \in S_T$ (S_T being the set of all permutations of T) of test cases that cover as many requirements as early as possible.

More formally, the test case priorization problem is to find a permutation $T' \in S_T$ so that

$$\forall T'' \in S_T : T' \neq T'' \wedge \forall i \in 1..n : |\bigcup_{j=1}^{i} m(T'(j))| \geq |\bigcup_{j=1}^{i} m(T''(j))|$$

i.e finding the optimal ordering of test cases, so that every requirement is met as soon as possible.

3.2 Test Case Selection

Test case selection attempts to temporarily find a subset of tests that maximize the requirements for a set of of recent changes to the software.

[YH12] formulate the problem of test case selection as

"*Given*: The program, P, the modified version of P, P′ and a test suite, T.
Problem: Find a subset of T, T′, with which to test P′."

Test case selection usually assumes a high degree of transparency and the availability of a lot of information about the SUT, such as execution graphs, coverage information, and sometimes information about the running times of individual tests.

3.3 Test Suite Reduction

Test suite reduction, on the other hand, attempts to permanently reduce the size of the test suite by removing redundant tests. The distinguishing feature from test case selection is that it doesn't take into account recent changes.

Generally, in test case reduction two different cases are distinguished: adequate and inadequate reduction.

In adequate test suite reduction, the goal is to find the smallest subset of T so that all R are satisfied:

$$T_r = \text{argmin}_{T_r} | T_r \subset T : R = \bigcup_{t_r \in T_r} m(t_r) |$$

This goal, however, is rarely fulfilled. Often, the objective is to keep the resulting test suite as small as possible, since reaching this optimum is computationally hard (as researchers have remarked ([Kha+16]), it is equivalent to the NP-complete set cover problem ([GJ79])).

Instead, researchers attempt to reduce the size of test suites further, while still keeping running times small.

Adequate reduction needs the requirements R to be present at the time of reduction to judge whether all requirements have been fulfilled.

The second case of test suite reduction is the inadequate/budget one. An inadequate test suite reduction fixes the size of the reduce test suite to a budget $B = |T_r|$. This cannot guarantee that all requirements are completely fulfilled, but often reduces runtimes significantly.

Inadequate reduction can work without access to the requirements. For example, the FAST family needs only the content of the testcases to perform a reduction. However,

inadequate reduction gives no guarantees about the performance of the resulting reduced test suite.

Other possible goals of test suite reduction include reducing the total runtime of the reduced test suite given past runtime information, but due to the absence of information about test runtimes, this goal has been studied less.

Test case priorization subsumes test suite reduction: Given an ordering T' of test cases, it is easy to select the first B test cases in the inadequate scenario or the first i test cases that together fulfill all requirements in the adequate scenario whilst keeping the number of selected test cases at a minimum.

[Kha+18] distinguishes 4 different types of test suite reduction approaches: Greedy selection, Clustering, Searching, and Hybrid methods.

3.3.1 Greedy Selection

Greedy approaches, first introduced by [Rot+01] (in the context of test case priorization), work by selecting test cases based on test requirements $R = \{r_1, \ldots, r_m\}$ (mostly statement/branch/function coverage).

[Kha+18] describe the fully general case of greedy-based test suite reduction, in which a test case is first selected using the coverage criterion and a heuristic, and then removed from the test set and added to the set of the reduced test suite, until the coverage criterion is completely fulfilled by the reduced test suite.

Two simple heuristics, described more in detail later, are the total heuristic (choosing the test with the highest amount of coverage from the test set) and the additional heuristic (choosing the test with the highest amount of coverage yet not covered by the reduced test suite).

3.3.2 Clustering

Clustering approaches to test suite reduction attempt to maximize a distance metric on the resulting test suite. This is achieved by separating T into disjunctive subsets using a clustering algorithm so that the difference between two subsets doesn't fall under a given threshold, and then sampling test cases from the subsets (in the optimal case one representative test case per subset).

This method doesn't need the test requirements to be available: many approaches (for example from the FAST family examined in this paper [Mir+18]) calculate the similarity of two test cases using the string of the test case's source code. However, those approaches can only be inadequate: The test case strings give no information about when the requirements are completely fulfilled.

Examples of clustering-based approaches to test suite reduction are of course the FAST algorithms from [Cru+19] and [Mir+18], but also the ART family from [Che+10] and the algorithms tested in [H H10].

3.3.3 Searching

Search-based methods for test suite reduction evaluate subsets $S = \{s_1, \dots s_m\} \subset \mathcal{P}(T)$ of T using a cost and/or effectiveness measure $F : S \to \mathbb{R}$, and then select the subset of tests that scores best on these measures. Because setting $S := \mathcal{P}(T)$ results in an exponential search space, search-based approaches are often combined with other methods – for example generating the subsets by applying clustering methods using a similarity measure.

The reduced test suite is then searched using the criterion

$$T_s = \arg\max_{s_i \in S} F(s_i)$$

Methods for test suite reduction based on searching are presented in [Cou+13], and [S W14].

3.3.4 Hybrids and Other Methods

Hybrid methods for test suite reduction combine different approaches to test suite reduction to achieve greater performance or speed up the reduction process. For example, search-based test suite reduction approaches can be enhanced by creating the subsets S of T by applying different clustering algorithms.

Other methods that can not be easily classified into any of those three categories have been proposed in the literature, for example Integer Linear Programming-based solutions, which are used for multi-objective optimization (e.g. optimizing for high coverage, low running times and low memory usage of tests). An example for for an ILP-based test suite reduction system is FLOWER [GM14], as well as an approach based on reducing the test suite minimization problem to a satisfiability problem [ACA12] (solving instances of test suite minimization problems on SIR [Do 05] in most cases in less than 2 seconds).

4 Approach

4.1 Replicating ”Scalable Approaches for Test Suite Reduction”

This paper attempts to replicate the findings in [Cru+19], using different test data and adding a further algorithm as a baseline. [Cru+19] builds on [Mir+18] (which introduced the FAST family), and adds two new algorithms to the FAST family.

We use the code from the original paper, published online at `https://github.com/ICSE19-FAST/FAST-R`, with some slight modifications to fix faults discovered in the original code.

4.2 Implemented Algorithms

[Cru+19] compares seven different methods of test-suite reduction. Four of those (**FAST++**, **FAST-all**, **FAST-CS** and **FAST-pw**) are in the FAST family (first introduced in [Mir+18]), which uses clustering techniques to find representative test cases. Two other algorithms are taken from the similarly clustering-based ART family, first developed in [Che+10]. The last reduction method examined in [Cru+19] is the Greedy Additional (**GA**) algorithm developed in [Rot+01], which is included because "for its simplicity and effectiveness [it] is often considered as a baseline."

This paper also includes a random selection algorithm as a baseline, as recommended by [Kha+18].

4.2.1 FAST

The FAST family is a clustering-based family of algorithms for test suite reduction ([Mir+18], [Cru+19]).

The FAST family is a collection of 4 clustering based test suite reduction algorithms that work both with adequate and inadequate test suite reduction problems, especially for large test suites.

FAST++

The FAST++ algorithm starts with a preparation phase: the tests from T are transformed into points in a vector space by treating each token (e.g. character) of the test case as a dimension, with the value of that dimension being the value of the token at the position (e.g. the value of the nth character), with the components being "weighted according to [a] term-frequency scheme, i.e., the weights are equal to the frequency of the corresponding terms" ([Cru+19]).

Since dealing with high-dimensional vector spaces is computationally costly (e.g. when computing the Euclidean distance), the algorithm performs a random projection into a lower-dimensional vector-space that nonetheless still mostly preserves the pairwise distances of the vectors (in this case a sparse random projection ([Ach03])).

The second phase of FAST++ is executing the k-means++ algorithm [AV06] on the resulting vectors, with k in the inadequate case being the budget of the reduction, and in the adequate case being a variable that is incremented until the requirements are met. The k-means algorithm is a clustering algorithm that finds k clusters of vectors in a high-dimensional space, i.e. minimizing the distance between points within a cluster, by iteratively assigning points to the nearest mean (distance being measured in squared Euclidean distances) and recalculating means until a fixed point is reached. The k-means++ algorithm only differs from k-means in the method for choosing initial centers of clusters: While k-means selects values at random, k-means++ selects the initial by computing the center of all points, and then selecting k other centers by sampling points using a distribution proportional to the distance of the points to the global mean. This both increases speed and gives guarantees that the solution is $\mathcal{O}(\log k)$ competitive to the optimal solution.

After computing the k clusters, FAST++ returns the vectors closest to the centers of the k clusters (i.e. the tests most representatitve for those clusters).

FAST-CS

The FAST-CS algorithm has the same preparation phase as FAST++: test cases are vectorized, and the resulting vectors are projected into a lower-dimensional vectorspace.

The second phase is different, as it attempts to cluster the set of points by finding a coreset, a set of points that approximate the shape of the set of vectors.

This is achieved by importance sampling: "All points have nonzero probability of being sampled, but points that are far from the center of the dataset (potentially good centers for clustering) are sampled with higher probability."

The metric used for importance sampling is as follows:

$$Q(t) \leftarrow \frac{1}{2|T|} + \frac{d(P(t), \mu)^2}{\sum_{t' \in P} d(P(t'), \mu)^2}$$

where μ is the mean of the whole dataset, T is the test suite, d is a distance metric (in this case the Euclidean metric), and P is the random projection of the test suite.

The points are sampled without replacement until either the coverage is adequate or the budget has been reached.

FAST-pw

FAST-pw was first introduced in [Mir+18] as a test suite priorization algorithm. It attempts to maximize the Jaccard distance between early test cases (the Jaccard distance between two sets A and B being $JD(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$).

The algorithm uses three different concepts: *shingling*, *minhashes* (and *minhash signatures*), and *locality-sensitive hashing*.

Shingling is used as a method for transforming a test case into a set, and is therefore only used in the case where coverage information is not available. Given a string of S, a k-shingle is the set of all substrings of S of length k. For example, the 7-shingle of "bananas" is just the set $\{"bananas"\}$, the 5-shingle of "bananas" is $\{"banan", "anana", "nanas"\}$. [Mir+18] states that "if two documents are similar they will have many shingles in common".

However, as [Mir+18] observes, sets of shingles (or of coverage information) can be very large.

Minhashing is a method for deriving compact representations of sets. This is achieved by creating a list of hash functions $H = \{h_1, \ldots h_k\}$, and for each test case T (a set of shingles or of coverage information) calculating the list $[\arg\min_{t \in T} h_1(t), \ldots \arg\min_{t \in T} h_k(t)]$ (which is called the signature of the set). Less formally, the minhash at position i is the minimal value of the hash function for any element of the set.

The Jaccard distance of two sets can be estimated by calculating the number of positions on which the minhash signatures s_1, s_2 of the two sets agree:

$$\text{EstimateJD}(s_1, s_2) = 1 - \frac{|\{i | i \in 1..k, s_1(i) = s_2(i)\}|}{|s_1|}$$

Locality-sensitive hashing is a further technique to make runtimes shorter. Let $S = \{s_1, \ldots s_n\}$ be a set of minhash signatures of all tests. Now a matrix $M \in \mathbb{R}^{n \times k}$ is created, by using the minhash signatures as columns. The rows of this matrix are now divided into bands of length r (in the example 2):

b_n is the nth band, r_n is the nth row, and s_n is the nth signature.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	
r_1	1	16	9	5	5	1	2	6	b_1
r_2	2	19	16	18	0	9	5	5	
	8	10	9	14	1	3	11	2	lsh_1
r_3	10	8	4	11	18	9	16	7	b_2
r_4	12	14	3	5	8	3	1	16	
	3	5	1	6	17	13	12	7	lsh_2
r_5	10	11	5	18	14	6	14	7	b_3
r_6	18	5	2	13	7	19	9	11	
	7	6	6	13	6	14	12	16	lsh_3

Figure 4.1: Locality-Sensitive Signature Hash Matrix

The columns within the bands are hashed into lsh values. If two lsh values correspond (e.g. in the case where the columns with values 11 and 5 are both hashed into 6), they are put into the same candidate set.

When the hash of the values in different columns of two bands is the same (the two minhashes are in the same bucket), the two test cases the minhashes belong to are added to the confusingly named candidate set (tests in the candidate set are excluded from selection). The candidate set contains tests that have a Jaccard distance over a threshold s, where $s \approx 1 - (r/n)^{1/r}$.

The algorithm follows different steps:

1. Calculate the minhashes

2. Calculate the candidate set C_s

3. Select a test from the set of remaining tests $T \backslash C_s$

4. Remove that test from the remaining tests T and add it to the set of selected tests S

5. If the condition is not fulfilled (budget not fulfilled in the inadequate case, coverage not achieved in the adequate case), go to step 2

6. Return S

The function that selects the test case from the remaining test cases maximizes the Jaccard distance of the returned test and the so far selected test cases S (where M is a function that produces minhashes, and Estimate:

$$\arg\max_{c \in T \setminus C_s} \text{EstimateJD}(M(S), M(c))$$

FAST-all

FAST-all is very similar to FAST-pw. The only difference lies in the selection function: While FAST-pw chooses based on Jaccard distance, FAST-all selects all of $T \setminus C_s$, simply choosing all non-candidate tests. This is a special case for the originally described algorithm, which allowed to select a random sample from the non-candidate tests with a given size (*one*, *log*, *sqrt*, *all*).

4.2.2 Random Selection

[Kha+18] gives 12 recommendations on how to execute and evaluate test suite reduction experiments. One of them is to include a baseline algorithm to compare the other algorithms to, such as random selection.

In the inadequate case, random selection simply chooses B random tests from the test set (Algorithm 1).

Algorithm 1 Random selection, budget

```
function RSBUDGET(TS, B)
    return randomsubset(TS, B)
end function
```

In the adquate case, a random element is removed from the original test suite and added to the selected tests until the set of selected tests fulfills all requirements (in this case, covers all coverable lines). See Algorithm 2.

4.2.3 Adaptive Random Testing

The second family of algorithms compared to in this paper is the ART (Adaptive Random Testing) family introduced in [Jia+09] and elaborated on in [Che+10]. These

Algorithm 2 Random selection, adequate

function RSADEQUATE(TS, cov, B)
  $RS \leftarrow \emptyset$
  $allcov \leftarrow \bigcup_{t \in TS} \text{cov}(t)$
  while $\bigcup_{t \in RS} \text{cov}(t) \neq allcov$ **do**
    $s \leftarrow selectrandom(TS)$
    $TS \leftarrow TS \backslash \{s\}$
    $RS \leftarrow RS \cup \{s\}$
  end while
  return RS
end function

algorithms were originally developed for test case priorization (as an alternative to random selection), but can be trivially converted into test suite reduction algorithms by applying the priorization procedure and then either selecting the first B test cases or so many that they fulfill all requirements.

ART-D

The ART-D algorithm was presented in [Jia+09] (although there only called ART). It starts with a set of tests T and an empty sequence of prioritized tests P. The algorithm works in two stages:

First, tests are selected from T and added to a candidate set C as long as new tests add new coverage information to C (this will not result in the joint coverage of tests in C being the joint coverage of tests in T). This results in the size of candidate sets being variable during exection.

Second, a test c is selected from C and appended to P. The criterion for selecting c is to maximize the distance between all $p \in P$ and c; [Jia+09] use the Jaccard distance in coverage and examine different methods for comparing the sequence of priorizited test cases to the tests in the candidate set (maximizing the maximum/average/minimum distance between the selected test case and the already priorized cases), and conclude that maximizing the minimal distance might be the best strategy.

The selected test case is appended to the sequence of priorizited test cases and removed from T. Unless T is now empty, the first stage is executed again.

ART-F

The ART-F algorithm was introduced in [ZSS12]. It is similar in structure to ART-D, but uses the Manhattan distance instead of the Jaccard distance, and frequency information

instead of coverage information, as well as having a fixed-size candidate set. Frequency information is information about how often a part of the SUT is executed by a test. However, in the presented experiment, since frequency information is not available, coverage information is used as a place-holder, where a test covering a statement is treated as the test executing tha statement with frequency 1. The Manhattan distance of two sequences of frequency information is calculated as such:

$$\mathrm{MD}(u, v) = \sum_{i=1}^{n} |u_i - v_v|$$

The stages in ART-F are the same as in ART-D, and it also returns a sequence of prioritized test cases.

4.2.4 Greedy Algorithm

The greedy algorithm used is the greedy additional algorithm from [Rot+01]: In each iteration, it selects the test case with the highest number of yet uncovered program entities.

Algorithm 3 Greedy selection next test case

function GANEXT(RS, TS, cov)
  already cov $\leftarrow \bigcup_{t \in RS} \mathrm{cov}(t)$
  return arg $\max_{t \in TS} |cov(t) \backslash$ already cov$|$
end function

In the adequate case, we select greedily until we achieve complete coverage:

Algorithm 4 Greedy selection, adequate

function GAADEQUATE(TS, cov)
  $RS \leftarrow \emptyset$
  $allcov \leftarrow \bigcup_{t \in TS} \mathrm{cov}(t)$
  while $\bigcup_{t \in RS} \mathrm{cov}(t) \neq allcov$ **do**:
    $s \leftarrow ganext(RS, TS, cov)$
    $TS \leftarrow TS \backslash \{s\}$
    $RS \leftarrow RS \cup \{s\}$
  end while
  return RS
end function

In the budget case, we instead select greedily B times.

Algorithm 5 Greedy selection, budget

function GABUDGET(TS, cov, B)
  $RS \leftarrow \emptyset$
  $allcov \leftarrow \bigcup_{t \in TS} \text{cov}(t)$
  for do$B$ times
    $s \leftarrow ganext(RS, TS, cov)$
    $TS \leftarrow TS \backslash \{s\}$
    $RS \leftarrow RS \cup \{s\}$
  end for
  return RS
end function

5 Replication Study

In this chapter, we first present our research questions. We then describe our study design and setup, and both describe and evaluate the results we obtained. We close by discussing the results in the broader context of test suite reduction, and enumerate possible threats to validity.

5.1 Research Questions

We define four research questions, adapting three from [Cru+19] and adding one additional question on performance in relation to a baseline algorithm.

Research Question 1.1: Can the relative effectiveness in TSR be replicated on other study objects? We care about the size of the reduced test suite, therefore it is valuable by how much test suites are reduced. We use the TSR metric (defined in 2) to evaluate the magnitude of the reduction. Using the TSR is mostly useful in adequate scenarios, since in inadequate scenarios, the size of the reduced test suite relative to the original test suite is fixed.

Research Question 1.2: Can the relative effectiveness in FDL be replicated on other study objects? [Rot+02] finds that test suite reduction can severely compromise fault detection capability. To test whether this is the case, the FDL metric is used to examine the fault detection capabilities for the algorithms given new data.

Research Question 2: Can the relative runtime performance of the different algorithms be replicated? Due to using different datasets (of both larger and smaller size), the absolute runtimes of the different test suite reduction methods can be expected to be different from the original runs. However, the ordering of the methods according to runtime is expected to be the same (that is, if algorithm X was faster than algorithm Y on the old data, we can expect algorithm X to also be faster than algorithm Y on the new data).

Research Question 3: How much better than random selection are specialized algorithms? [Kha+18] recommends a comparison of the examined technique to a simple baseline. We use their recommendation (random selection) and compare the other approaches to how well they fare against random selection on runtime, TSR and FDL.

5.2 Study Design

We approached the research questions by collecting test case source code, converage information and fault-detection data from projects that had not been used in [Cru+19], and running the code from that paper on the new data.

Research Question 1 In order to answer Research Question 1, we then generated descriptive statistics (the mean, median, and standard deviation) on the performance measures (i.e. FDL and TSR) for the different algorithms. We then, just as the approach in [Cru+19], first performed the Kruskal-Wallis test for all pairs of algorithms on the same metric (with a significance level of 5%, as in the original paper). Algorithms for which no statistically significant difference could be found were ranked equally high, for algorithms with significantly different performance we used the median FDL/TSR to determine the ranking.

We used this approach to ranking the different algorithms to ensure our results were comparable to the results in the original paper.

The result was operationalized by assigning the algorithms letters, with "(a)" being the signifier for the best performing algorithms, "(b)" for the second-best, and so on (several algorithms could receive the same ranking).

We then compared the rankings according to TSR and FDL to the ones in [Cru+19].

Research Question 2 We also computed the mean, median and standard deviation for the runtime of the different approaches, and ranked them according to the method described for Research Question 2.

We then compared our rankings for runtime to the ones in [Cru+19].

Research Question 3 For Research Question 3, we used the previously collected rankings for different algorithms, and evaluated the ranking of random selection in comparison to the other approaches presented.

5.3 Study Objects

The code bases and test suites selected for this study were small to medium sized open-source Java projects, which we selected because tools for generating coverage information and mutation-test data using Maven are available and comparatively easy to use. The selected projects all used either JUnit 4 or JUnit 5 as their testing framework, and the tool Maven for building and testing. We used the latest version as of November 2020.

We decided to use open-source projects to make checking our generated data easier, as well as for availability reasons.

The selected projects are both under development and in use, which makes our results relevant to real-life systems.

The projects used, their versions and size are presented in Table 5.1 (project and test suite size are in lines of code, the version is the 6-digit prefix of the git commit used).

Table 5.1: Basic Information about the selected open-source Java Projects

Project name	Version	Project size	Test-suite size	Number of tests
assertj-core	cb2829	135k	241k	3578
commons-collections	242918	59k	45k	238
commons-lang	6b3f25	51k	40k	181
commons-math	649b13	148k	98k	438
jopt-simple	5a1d72	1.7k	2.7k	145
jsoup	89580c	18k	12k	52

5.4 Study Setup

Testing the performance of the algorithms required three different kinds of information: the contents of the test suites, coverage information and fault information.

5.4.1 Combining Tests Suites

For every examined project, the test suite was converted into a format suitable for the code from [Cru+19] by replacing the newlines from each test with spaces and concatenating the tests into one file (the black-box file), such that every line contained one test case.

For some of the projects, test cases were excluded since they failed during the default test run (see Table 5.2), with several tests being excluded from assertj-core, one or two

tests each being excluded from the Apache commons libraries, and none from the remaining two projects.

Table 5.2: Tests excluded from data gathering

Project name	Test classes excluded
assertj-core	BDDSoftAssertionsTest, SoftAssertionsTest, SoftAssertions_overriding_afterAssertionErrorCollected_Test, SoftAssertionsErrorsCollectedTest, SoftAssertionsMultipleProjectsTest, SoftAssertions_setAfterAssertionErrorCollected_Test, AssertJMultipleFailuresError_getMessage_Test
commons-collections	BulkTest
commons-lang	FieldUtilsTest
commons-math	FastMathTest, EvaluationTestValidation
jopt-simple	/
jsoup	/

5.4.2 Generating Coverage Information

Since we needed information about which test covers which parts of the program, and not just information about which parts of the program were being covered, we used an extension to the widely used coverage program JaCoCo, the Teamscale JaCoCo Agent (available at `https://github.com/cqse/teamscale-jacoco-agent`).

Since the Teamscale JaCoCo Agent only supports line coverage, other types of coverage had to be eschewed.

The json files created by the teamscale jacoco agent were converted from JSON into the format used by [Cru+19]: a file containing a list of numbers in each line, with the numbers $n_1, \ldots n_i$ in line n corresponding to the source lines of code in the tested project covered by the test case at line n in the black-box file.

5.4.3 Collecting Fault Coverage Information

Fault detection information was not available for the projects used, and was therefore generated using the mutation testing framework Pitest (first described in [Col+16], available at `http://pitest.org/`).

The generated mutation test data was converted from XML to the format used in the code of [Cru+19]: a text file containing a list of number per line, the numbers at line n

corresponding to different classes for which the test case at line n in the black-box file did find faults.

5.5 Results

To obtain the results, we used the code from [Cru+19], and applied the test suite reduction algorithms to new test data (i.e. black-box test suite files and coverage information of 6 open source projects) after minimal modification. These modifications were as follows:

1. We added a further algorithm for the random selection of test cases.
2. We fixed three bugs we found in the code:
   a) an off-by-one error in loading coverage information
   b) a bug where in small test suites sometimes the candidate set would be empty
   c) a mistake where a a measured variable was not reported correctly for FAST-all

In the budget scenario, we considered budgets between 1% and 30%, with a step increase of 1%. We considered only line coverage, and performed 50 measurements of FDL, TSR, preparation and reduction time.

In the adequate scenario, we also considered only line coverage, and made 50 measurements of preparation time, coverage preparation time, and reduction time.

We did not replicate the large-scale scenario.

All measurements were performed under a Ubuntu 20.04 64-bit system, using 6 AMD® Ryzen 5 4500u with radeon graphics processors, and 7.2 GiB of RAM available.

5.5.1 Research Questions

Research Question 1.1 (Can TSR findings be replicated?) Table 5.3 shows the descriptive statistical results (median, standard deviation and ordering for the Kruskal-Wallis test) in the adequate case, and Figure 5.1 shows boxplots for TSR for the different algorithms.

For Java programs with statement coverage in the adequate case, [Cru+19] order the different methods of test suite reduction according to TSR. Their results mostly agree with ours: They rank GA highest, then FAST++ and FAST-pw, then FAST-CS, FAST-all below that, and the ART family lowest; we also rank GA highest, followed by FAST++, then FAST-CS and FAST-pw at the same level, then FAST-all, and the ART family last (on par with random selection). However, the median test suite reduction they report

for these methods is much lower than what we observe (at least for the java programs): even GA only achives a TSR of 12.30, while in our experiment GA achieves a TSR of over 40. However, their results with C programs are more stark than ours, achieving a TSR of more than 90 for GA or FAST++.

Our data shows higher variance in TSR performance on all algorithms.

Table 5.3: Different variables relating to TSR (mdn is the median, σ being the standard deviation, and δ being the ranking in the Kruskal-Wallis test)

Approach	μ	mdn	σ	δ
ART-D	18.421	17.351	8.997	(e)
ART-F	0.945	0.228	1.654	(f)
FAST++	36.116	33.811	13.581	(b)
FAST-all	21.414	18.264	8.536	(d)
FAST-CS	33.683	31.893	12.064	(c)
FAST-pw	33.258	32.718	11.458	(c)
GA	40.599	39.124	14.922	(a)
RS	0.947	0.420	1.491	(f)

Research Question 1.2 (CAN FDL findings be replicated?) Table 5.4 shows the descriptive statistics (mean, median, standard deviation, and ordering according to the Kruskal-Wallis test) for the FDL of different algorithms in both the budget and adequate case, and Figure 5.2 shows boxplots for FDL for different algorithms.

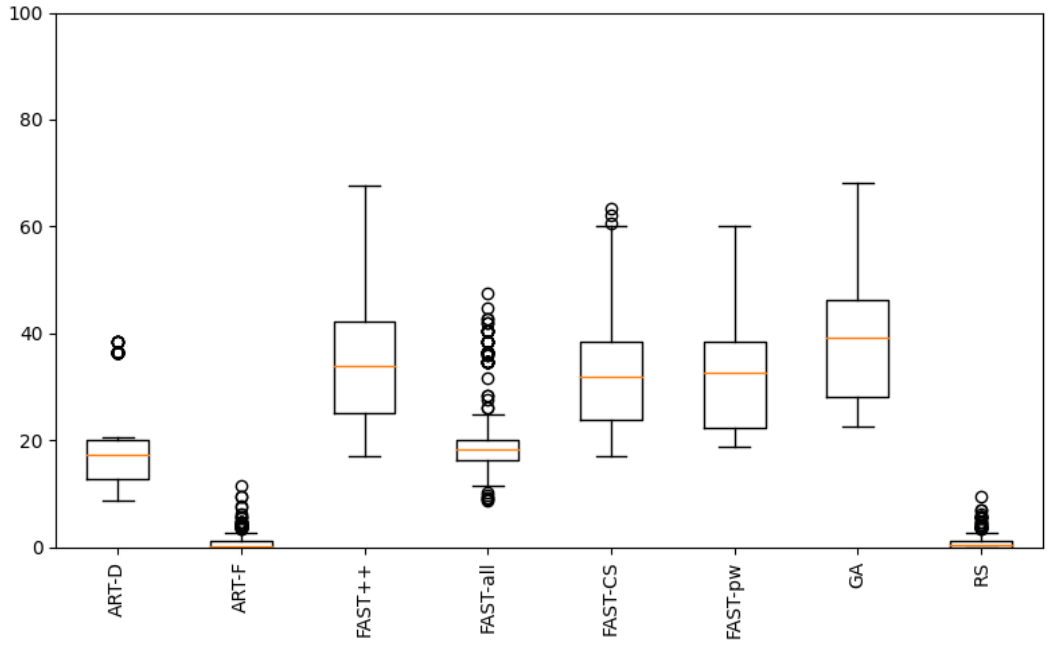
We compare our findings to FDL performance on statement-covered C program data, since the fault information for the java programs used in [Cru+19] is very sparse (1-3 faults per program), while fault information is much richer for the C programs.

In the budget scenario, we could not detect strong differences in performance in FDL. All algorithms score equally, with the exception of GA dominating every other algorithm and RS being dominated by every other algorithm. Here, again, [Cru+19] come to very similar conclusions: They find that GA is statistically significantly better than all other algorithms, which are in turn approximately equally effective at detecting faults.

Here, we also find that the standard deviations in FDL for our data is higher than the ones in [Cru+19]. Similar to their findings, we see no strong difference in variance for different algorithms.

In the adequate case, our data ranks the algorithms into just two categories: The better-performing category contains algorithms from the ART family, FAST++ and

Figure 5.1: Boxplots for TSR for different algorithms in the adequate case



FAST-CS, while the rest of the approaches is sorted into the bucket of worse-performing approaches. [Cru+19] agrees that the ART family is best-performing, but find that FAST-all is second-best, GA being third-best, and FAST-CS, FAST-pw and FAST++ coming last. At the moment, we don't have a hypothesis about why this might be the case. It seems relevant, however, that the median fault detection loss for data was 0 for all algorithms (the mean fault detection loss being non-zero, however), just as the ART family and FAST-all in [Cru+19]. Perhaps this is caused by single tests both having large coverage and detecting a majority of faults.

This hypothesis is further supported by the observation that the variance in performance on all algorithms is much lower than in the original paper, for all algorithms.

We include the mean in the descriptive statistics to illustrate that while the median adequate reduction has no fault detection loss, it can nonetheless still occur.

Research Question 2 (Can total runtime performance findings be replicated?) Table 5.5 lists statistics for the total reduction times in seconds ("total" including the time for loading the coverage files, the time for processing the black-box test files, and the reduction time). The table lists both the median and the mean of the total time, since especially in the adequate case there are big differences in these measurements

Table 5.4: Different variables relating to FDL (μ being the mean, mdn the median, σ the standard deviation, and δ the ranking in the Kruskal-Wallis test)

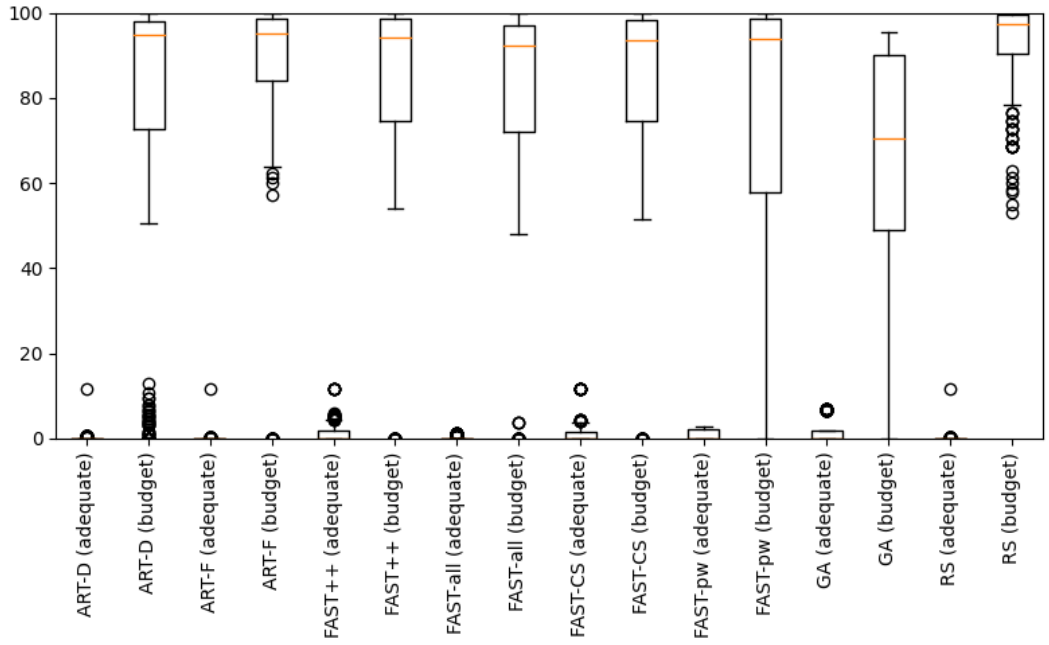
Budget					Adequate				
Approach	μ	mdn	σ	δ	Approach	μ	mdn	σ	δ
ART-D	76.414	94.75	34.502	(b)	ART-D	0.129	0.0	0.702	(a)
ART-F	77.72	95.04	35.684	(b)	ART-F	0.057	0.0	0.679	(a)
FAST++	76.49	94.05	35.662	(b)	FAST++	1.122	0.0	2.172	(a)
FAST-all	74.78	92.19	35.269	(b)	FAST-all	0.155	0.0	0.331	(b)
FAST-CS	76.19	93.39	35.493	(b)	FAST-CS	1.144	0.0	2.367	(a)
FAST-pw	71.63	93.82	35.665	(b)	FAST-pw	0.792	0.0	1.131	(b)
GA	62.53	70.45	33.367	(a)	GA	1.432	0.0	2.444	(b)
RS	93.43	97.46	9.305	(c)	RS	0.058	0.0	0.679	(b)

(e.g. in the case of ART-D, where the mean reduction time is more than two orders of magnitude greater than the median reduction time).

Comparing the budget scenario to the statement-coverage based budget (we remove function and branch-coverage based approaches) scenario in [Cru+19], one can observe striking differences. While GA performs best in our measurements, the original paper ranks it in the middle (although, in the function-based coverage test, their measurements agree with ours). The approaches from the FAST family perform rather mediocre, receiving ranks (d) (FAST++ and and FAST-CS) and (e) (FAST-all and FAST-pw), while in the original measurements, FAST++ is ranked second-best, followed by FAST-CS, and FAST-all and FAST-pw being ranked lowest. Algorithms from the ART family receive relatively high rankings in our measurements ((b) and (c)), while in the original measurements, they also perform mediocre (around as well as GA).

In the adequate scenario, we also only examine the statement-coverage based measurements from the original paper. Our results rank RS highest, followed by GA, while GA receives only rank (c) in the original results. From the FAST family, FAST++ and FAST-CS rank just below GA, while in the original results, they come out as clear winners. FAST-pw and FAST-all rank are ranked lowest in our results, even below the ART family, while in the original results, ART algorithms are clearly ranked lowest. We suspect that this result is due to the difference in medians between the datasets we used and the original dataset: The dataset we used contained several relatively small test suites, and a relatively big outlier, while the test suites in the original paper were more similar in size. That leads to comparatively small medians (and comparatively large means) in total runtime for our data (as can be seen in the greater standard

Figure 5.2: Boxplots for FDL for different algorithms



deviations) on the ART algorithms. Since the Kruskal-Wallis test we and [Cru+19] use only considers medians for ordering, the outliers do not influence the final ordering.

Research Question 3 (How well do algorithms perform against a random baseline?) In short, the answer is that they perform better on performance metrics, while being slower. Table 5.3 shows that RS ranks lowest on TSR (together with ART-F). Similarly, Table 5.4 ranks RS lowest on FDL in the budget and adequate case (although in the latter case, RS can not be shown to be statistically significantly worse than three other algorithms). We also found that RS performs surprisingly badly in terms of runtime performance (see Table 5.5) in the budget case, ranking third, despite being the conceptually simplest algorithm we examined. However, in the adequate case, it ranks best.

5.6 Discussion

In general, we find that our results largely replicate the results in [Cru+19]. On TSR and FDL, we (and [Cru+19]) rank GA very high, often first, usually followed by FAST++ and

Table 5.5: Different variables relating to total runtime in seconds (μ being the mean, mdn the median, σ the standard deviation, and δ the ranking in the Kruskal-Wallis test)

Budget					Adequate				
Approach	μ	mdn	σ	δ	Approach	μ	mdn	σ	δ
ART-D	0.81	0.975	0.318	(b)	ART-D	58.232	0.475	127.311	(e)
ART-F	0.919	0.982	0.133	(c)	ART-F	158.892	1.926	341.625	(f)
FAST++	1.064	1.156	0.545	(d)	FAST++	1.792	0.329	3.254	(c)
FAST-all	7.005	3.567	8.425	(e)	FAST-all	6.345	2.622	8.494	(f)
FAST-CS	1.034	1.152	0.506	(d)	FAST-CS	2.232	0.334	4.053	(d)
FAST-pw	7.025	3.582	8.473	(e)	FAST-pw	8.224	2.736	12.206	(g)
GA	0.974	0.968	0.632	(a)	GA	4.286	0.147	8.725	(b)
RS	0.95	0.982	0.087	(c)	RS	0.965	0.143	1.535	(a)

FAST-CS, then FAST-pw and FAST-all, then the ART algorithms and finally RS. With regards to FDL, we often can't statistically significantly differentiate the algorithms in performance, similar to the original paper.

Our results in runtime performance differ from the original results, though, disagreeing both in the adequate and the budget case. We hypothesize that this is due to different distributions in the sizes of the test suites tested on, with the test suites we used being both smaller and larger than the test suites used in the original paper, which makes the median of runtimes less useful at determining the relative performance of algorithms.

We cautiously conclude that for test suites of the sizes we have examined, the GA algorithm makes a good trade-off between test suite reduction, fault detection loss and runtime performance, if coverage information is available.

5.7 Threats to Validity

Although we attempted to prevent it, the results we obtained might suffer from threats to validity.

5.7.1 Construct Validity

Construct validity considers whether the methods are adequate to answer the research questions posed. As in the original paper, we used FDL and TSR as standard metrics,

and reported the mean, median and standard deviation of the data as standard statistical properties of the generated data. We used faults generated by mutation testing software instead of real faults, which might be dissimilar from real faults. However, it has been argued that mutation faults are similar to real-world faults ([Bud80], and programmers of mutation-testing software try to make the faults introduced as realistic as possible.

To prevent bugs from our code and the code by [Cru+19] introducing incorrect results, we checked the code using a small, transparently understandable test project. We then manually verified that this project was handled correctly. This led to finding one of the bugs in the original code by [Cru+19].

Due to restrictions on availability, we used line coverage instead of statement coverage. Since the original study mostly examined statement coverage, there might be problems in comparing the results of the two studies (such as statements spanning multiple lines, or several statements in one line). However, we claim that in practice, statement coverage and line coverage are similar in terms of granularity. For example, both [AKY18] and [YL09] treat statement and line coverage as equivalent.

5.7.2 Internal Validity

The results we obtained might not be due to actual differences in the performance of the different methods, but due to other factors, such as the usage of the computer used for gathering data for other purposes during that time. To migitate these effects, we performed every measurement 50 times, and abstained from using the computer during the time of gathering data.

5.7.3 External Validity

The projects we selected are all small to medium-sized projects, and therefore the results we obtained might not generalize to large-scale software systems. While one of the projects we obtained test data from (assertj-core) was larger than the projects examined in [Cru+19], it was not analyzed separately from the other data. However, since the projects we used were of similar size as the ones in [Cru+19], and our interest was to replicate their findings, we regard this consideration as less relevant.

6 Future Work

The literature on test suite reduction is very comprehensive, [Kha+18] identify 4320 papers published in the field of test suite reduction between 1993 and 2016. This makes it difficult to assess which ideas have already been evaluated, and which haven't. Still, we attempt to identify some possible further lines of work that haven't been explored as much as desirable yet.

The first possible like of work we identify is further replication attempts of existing papers, and comparisons with random baseline algorithms. Replication attempts are useful because they can unveil methodological issues in existing papers (such as discovering bugs in the source code used, statistical mistakes and distorted data), but are usually neglected because of low academic payoff.

The second possible line of work is large-scale comparisons of many different test suite reduction algorithms on comprehensive datasets. A good example for this kind of work is [H H10], but it uses a relatively small dataset with 2 test suites (although the test suites themselves are very large), one containing 4 and the other 15 faults. They examine 300 different approaches to test suite reduction. It would be useful (though very expensive, both computationally and time-wise) to collect a large number of test suite reduction approaches presented in the literature, and compare them to each other on several metrics.

A possible line of work that is not directly a novel approach to test suite reduction is the collection of fault and coverage data for open-source project to create bigger datasets on which to test novel approaches to test suite reduction. [YH12] state that "subject programs from SIR account for almost 60% of the subjects of empirical studies observed in the regression testing techniques literature" (SIR being the software infrastructure repository [Do 05]). The programs and their test suites in SIR are relatively small, and include only one language. It might be useful to provide a combined and extended database of projects with test suites, coverage and fault information and faults ([Cru+19] uses SIR, as well as programs from Defects4j [RE14]).

Test suite reduction and similar methods are often motivated by stating that test suites in the industry are often very large and have long runtimes. It would be useful to have a collection of cases which show runtimes of very large test suites in the real world, to provide sources for these types of motivations.

7 Summary

In this paper, we motivated and formulated the problem of handling large test suites: As test suites for regression testing and model-based testing become more comprehensive, their running times are often prohibitive to effective software development. We examined three different approaches to this problem formulated in the literature: test case priorization, test case selection, and test suite reduction. For test suite reduction, we in turn presented and shortly discussed a categorization of different algorithms: greedy selection algorithms, clustering-based algorithms, search-based algorithms, and hybrids.

We then explained 7 algorithms examined in the paper [Cru+19]: four algorithms from the FAST family (FAST++, based on the k-means++ algorithm; FAST-CS, based on coreset discovery; FAST-pw, based on locality-sensitive signature hashing; and FAST-all, which is based on FAST-pw, but uses random sampling for selecting test cases instead of Jaccard distance); two algorithms from the ART family (ART-D, which maximizes Jaccard distance between prioritized test cases; and ART-F, which maximizes the Manhattan distance); and the greedy additional algorithm (which selects the test case with the highest amount of yet uncovered program components). We also described random selection of test cases, which was not described in the original paper.

We then formulated four research questions, which outlined our attempt to replicate the findings in [Cru+19], examining the metrics FDL, TSR, total runtime performance, and performance of the different algorithms against a random baseline. We described our process of gathering independent coverage and fault data and executing the algorithms on this data.

We then collected and presented both descriptive and inductive statistical measurements on the data collected. We showed the mean, median, and standard deviation of FDL, TSR, and total runtime performance for all algorithms, and ranked the algorithms on FDL, TSR and total runtime performance using the Kruskal-Wallis test. We found that our results largely replicated the original results from [Cru+19], although we observed differences especially in total runtime performance.

We described both threats to validity for our results, and presented ideas for future work in the area.

List of Figures

List of Tables

Bibliography

[ACA12] F. Arito, F. Chicano, and E. Alba. "On the application of sat solvers to the test suite minimization problem." In: *International Symposium on Search Based Software Engineering*. Springer. 2012, pp. 45–59.

[Ach03] D. Achlioptas. "Database-friendly random projections: Johnson-Lindenstrauss with binary coins." In: *Journal of computer and System Sciences* 66.4 (2003), pp. 671–687.

[AKY18] G. An, J. KHim, and S. Yoo. "Comparing line and AST granularity level for program repair using PyGGI." In: *PRoceedings of the 4th International Workshop of Genetic Improvement Workshop*. 2018, pp. 19–26.

[AV06] D. Arthur and S. Vassilvitskii. *k-means++: The advantages of careful seeding*. Tech. rep. Stanford, 2006.

[Bud80] T. A. Budd. "Mutation analysis of program test data[Ph. D. Thesis]." In: (1980).

[Che+10] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. "Adaptive random testing: The art of test case diversity." In: *Journal of Systems and Software* 83.1 (2010), pp. 60–66.

[Col+16] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. "Pit: a practical mutation testing tool for java." In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 449–452.

[Cou+13] A. Coutinho, E. G. Cartaxo, P. D. Machado, and C. G. SPLab-UFCG. "Test suite reduction based on similarity of test cases." In: *7st Brazilian workshop on systematic and automated software testing—CBSoft*. Vol. 2013. 2013.

[Cru+19] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino. "Scalable approaches for test suite reduction." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 419–429.

[Do 05] R. G. Do H Elbaum SG. "Suporting controlled experimentation with testing techniques: An infrastructure and its potential impact." In: *Empirical Software Engineering* (2005), pp. 405–435.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979.

[GM14] A. Gotlieb and D. Marijan. "FLOWER: Optimal test suite reduction as a network maximum flow." In: *Proc. ACM ISSTA* (2014), pp. 171–180.

[H H10] A. B. H. Hemmati A. Arcuri. "Achieving Scalable Model-Based Testing Through Test Cas Diversity." In: (2010).

[Jia+09] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. "Adaptive random test case prioritization." In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2009, pp. 233–244.

[Kha+16] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, and V. Chang. "A survey on Test Suite Reduction frameworks and tools." In: *International Journal of Information Management* 36.6 (2016), pp. 963–975.

[Kha+18] S. U. R. Khan, S. P. Lee, N. Javaid, and W. Abdul. "A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines." In: *IEEE Access* 6 (2018), pp. 11816–11841.

[Mir+18] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. "Fast approaches to scalable similarity-based test case prioritization." In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 222–232.

[RE14] D. J. R. Just and M. D. Ernst. "Defects4j: A database of existing faults to enable controlled testing studies for java programs." In: *ACM* (2014), pp. 437–440.

[Rot+01] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. "Prioritizing test cases for regression testing." In: *IEEE Transactions on software engineering* 27.10 (2001), pp. 929–948.

[Rot+02] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong. "Empirical studies of test-suite reduction." In: *Software Testing, Verification and Reliability* 12.4 (2002), pp. 219–249.

[S W14] A. G. S. Wang S. Ali. "Cost-effective test-suite minimization in product lines using search techniques." In: *J. Syst. Software* 103 (2014), pp. 370–391.

[Wol06] R. R.
bibinitperiod K. Wolfmaier. "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost." In: *PRoceedings of the 2006 International Workshop on Automation of Software Test* (2006), pp. 85–91.

[YH12] S. Yoo and M. Harman. "Regression testing minimization, selection and prioritization: a survey." In: *Software testing, verification and reliability* 22.2 (2012), pp. 67–120.

[YL09] Q. Yang and D. M. Li J Jenny an Weiss. "A survey of coverage-based testing tools." In: *The Computer Journal* 52.5 (2009), pp. 589–597.

[ZSS12] Z. Q. Zhou, A. Sinaga, and W. Susilo. "On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites." In: *2012 45th Hawaii International Conference on System Sciences*. IEEE. 2012, pp. 5584–5593.