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

As a response to ever-growing test suites with long runtimes, 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. This paper attempts to replicate the findings in [Cru+19], which borrows techniques from big data to handle very large test suites. We use independently generated testing data from open source projects, and find that TODO.

Contents

1 Introduction

3 pages

Software often has has faults: ways in which the actual behavior of the software diverges from the intended or specifie 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), integration testing, which shows how well the software works in the environment it is used in, and regression testing, which shows developers whether they have introduced new bugs after the last change to the software.

Regression testing takes up a significant portion of development cost, and the resulting test suites can grow quite significantly in size and execution time, which hinders development speed and increases costs.

To mitigate the costs and runtimes of regression 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 algorithms (first developed in [Mir+18]), and compares them to the algorithms presented in [Che+10], as well as the greedy algorithm presented in [Rot+01]. They implement 4 algorithms from the FAST family, 2 from the ART family and the greedy algorithm, and compare the different algorithms on 10 different test programs and their test suites. They compare the performance of the test suite reduction algorithms on 3 different variables: test suite reduction, fault detection loss, and runtime.

This work attempts to replicate their findings using the data from 6 additional 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]).

This work then attempts to determine how different test suite reduction strategies compare to each other in terms of time performance, fault detection loss and magnitude of reduction.

2 Terms and Definitions

$\frac{1}{2}$ a page

Candidates:

SUT (System under Test) TSR (Test Suite Reduction) FDL (Fault Detection Loss)

3 Related Work

5 pages

The literature on handling large test suites is comprehensive, 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 TSR frameworks/tools and specifically their implementions. [Kha+18] focuses on the algorithmic approach (i.e. greedy/clustering/searching/hybrid) of the TSR system and gives recommendations for future the evaluation of TSR 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 $m : T \to \mathcal{P}(R)$ 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

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

Test case priorization subsumes test suite reduction: Given an ordering T' of test cases, it is easy to select the first o test cases that together fulfill all requirements.

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.

One can now distinguish two cases:

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).

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

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

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 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.

[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 a coverage criterion $C = \{c_1, \ldots, c_{\text{ncov}}\}$.

[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

3.3.3 Searching

3.3.4 Hybrids

4 Approach

8 pages

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.

1 page

4.2 Implemented Algorithms

[Cru+19] compares 7 different methods of test-suite reduction. 4 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. 2 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].

7 pages

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 TSR algorithms that work both with adequate and inadequate TSR problems, especially for large test suites.

4 pages

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).

The second phase of FAST++ is executing the k-means++ algorithm 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. Example:

$$\begin{array}{cccc}
s_1 & s_2 & s_3 & s_4 \\
9 & 4 & 1 & 1 \\
3 & 8 & 0 & 5 \\
0 & 2 & 4 & 5 \\
0 & 4 & 2 & 9 \\
2 & 7 & 6 & 3 \\
1 & 4 & 0 & 6
\end{array}$$

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 below 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:

$$\text{arg max}_{c \in T \backslash C_s} \text{EstimateJD}(M(S), M(c))$$

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 $T \backslash C_s$, 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

$\frac{1}{2}$ a page

[Kha+18] gives 12 recommendations on how to execute and evaluate test suite reduction experiments. [Cru+19] follows 11 out of these 12 analyses, but does not include a simple alternative baseline, such as random selection from the set of tests.

This analysis adds an algorithm that randomly selects tests from the test set. In the inadequate case, it simply chooses B random tests from the test set:

```
def random_selection(input_file, B=0):
       TS = loadTestSuite(input_file)
       sel=random.sample(list(range(1,len(TS)+1)), min(B, len(TS)))
       return sel
```

Figure 4.1: The algorithm for selecting B tests from the test suite randomly

In the adequate case, random selection removes a random test from the set of tests and adds it to the selected tests until the coverage is adequate.

```
def random_selection_adequate(input_file, B=0):
       TS = loadTestSuite(input_file)

       selcov=set()
       allcov=set()
       for k in list(TS.keys()):
              allcov=allcov.union(TS[k])

       tests=set(range(1, len(TS)+1))
       sel=list()

       while len(tests)>0 and len(allcov.difference(selcov))>0:
              selected=random.sample(tests, 1)[0]
              sel.append(selected)
              tests.remove(selected)
              selcov=selcov.union(TS[selected])

       return sel
```

Figure 4.2: The algorithm for selecting tests from the test suite randomly, adequate case

4.2.3 Adaptive Random Testing

2 pages

ART-D

ART-F

4.2.4 Greedy Algorithm

$\frac{1}{2}$ a page

5 Evaluation

17 pages

5.1 Research Questions

Research Question 1: Does the Relative Effectiveness of the Different Algorithms Replicate the Results in [Cru+19]?

Research Question 1.1: Does Their Relative Effectiveness in TSR Replicate the Findings in [Cru+19]?

Research Question 1.2: Does Their Relative Effectiveness in FDL Replicate the Findings in [Cru+19]?

Research Question 2: Does the Relative Runtime Performance of the Different Algorithms Replicate the Results in [Cru+19]?

Research Question 3: How Much Better Than Random Selection are Specialized Algorithms?

(Possibly) Research Question 4: Do the Results in [Cru+19] Replicate with the Original Test Data?

5.2 Study Design

5.3 Study Objects

5.4 Selecting Projects

The code bases and test suites selected for this study were small to medium sized open-source Java projects. They 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.

The projects used, their versions and size are presented in table CANDO (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

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

5.5 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.

5.5.1 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:

5.5.2 Generating Coverage Information

Coverage information was generated using jacoco with the testwise mode of the 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

Table 5.2

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	/

the numbers $n_1, \dots n_i$ 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**.

5.5.3 Collecting Fault Coverage Information

Fault detection information was not available for the projects used, and was therefore generated by mutation testing using the mutation testing framework pitest.

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.

5.6 Results

5.6.1 Research Questions

Research Question 1.1

Research Question 1.2

Research Question 2

Research Question 3

(Possibly) Research Question 4

5.6.2 Running Time

5.7 Discussion

Comparison to ″Scalable Approaches to Test Suite Reduction″

5.8 Threats to Validity

5.8.1 Conclusion Validity

5.8.2 Internal Validity

5.8.3 Construct Validity

5.8.4 External Validity

6 Future Work

1 page

7 Summary

2 pages

List of Figures

List of Tables

Bibliography

[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.

[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.

[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.

[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.

[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.