DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Conception and Evaluation of Test Suite Minimization Techniques for Regression Testing in Practice

Raphael Nömmer

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Conception and Evaluation of Test Suite Minimization Techniques for Regression Testing in Practice

Konzeption und Evaluierung von Minimierungsverfahren für Regressions-Testsuites in der Praxis

Author:	Raphael Nömmer
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Juergens & Roman Haas
Submission Date:	15.10.2019

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

Munich, 15.10.2019 Raphael Nömmer

Acknowledgments

First, I want to thank Roman Haas for his guidance during the entire process of researching and writing this thesis, for the insightful discussions on the topic and for the tireless proofreading of the thesis. I also want to give a big thanks to Dr. Elmar Juergens for helping with finding an exciting topic as well as his helping me to find industry partner projects for the study. I would also like to thank Prof. Dr. Manfred Broy who allowed me to write this thesis under his supervision. Finally I want give a special thanks to Hans Noemmer, for helping me with the tedious task of proofreading the thesis and finding mistakes that I consistently overlooked.

Abstract

The increasing size and complexity of software systems leads to an increase in the use of regression testing. With large software systems, tests can give the developer confidence that the project still works after changes have been made, or new features have been introduced. However, when not carefully implemented, the cost of testing can get unreasonably high. The high cost of running tests can lead to test suites getting rarely executed because they are too expensive to run. Oversized test suites also make maintenance harder which can lead to a lot of failing tests and, again, rarely executed test suites with results that are difficult to evaluate due to the large amounts of changed code. Test suite minimization strives to reduce the cost of testing by selecting a subset of relevant tests and removing the remaining, ideally redundant, tests. In this thesis, we present a weighted-sum test suite minimization algorithm that aims to find the subset of most relevant tests by selecting test cases that find the most faults in the least amount of time and cover the most code. We evaluate this new algorithm based on open source projects as well as closed source industry projects to include as many scenarios as possible. By comparing our algorithm to a greedy minimization algorithm, we want to find out, how it matches with a well-established test suite minimization algorithm. We find that, with the new algorithm, we can reduce the runtime of the open source projects' test suites by 85% while maintaining, on average, 97% of the full test suite's line coverage as well as 95% of its fault detection capability. For the closed source projects, the algorithm maintained around 90% coverage on average, performing not quite as well as with the smaller open source projects. In both cases, the weighted-sum algorithm outperforms the greedy algorithm in terms of its time savings and the mutation coverage. Our experimental results show that the weighted-sum algorithm allows us to target a steep runtime reduction of 85% while maintaining good code and mutation coverage. Depending on the requirements, the weights of the individual criteria can be changed, allowing for a more customizable minimization compared to the greedy algorithm. The main hurdle of the algorithm for large projects are mutation score calculations which become infeasible for long-running test suites.

Contents

1 Introduction

Modern software projects increase in size and complexity. At the same time, the speed with which they evolve is increasing. In actively developed projects, changes are pushed to a project's repository up to multiple times a minute [AKM08; Shi+14; Bir+09; Bri+14]. These factors are the reason for the increasing use and importance of software testing.

However, as software evolves and gets bigger, the associated test suite usually grows alongside it. For large scale software projects, test suites can take days or even weeks to run [Rot+99; ST02]. This causes delayed feedback to the developers, which is compounded by the fact that a test suite that takes several days can not be executed as often. This delayed feedback causes difficulties for the developers. Less frequently executed tests lead to more changed code for each test run. It also means that the time since the developer has worked on the code is increased. Both of these issues make it harder for developers to find the fault from which a test failure originates.

The problems caused by over-sized test suites have given rise to three different approaches to coping with the problem. *Test Case Prioritization*, *Test Case Selection* and *Test Suite Minimization*. While test suite minimization and test case selection are concerned with reducing the overall runtime of a test suite, test case prioritization focuses on getting feedback to the developer as fast as possible. This is achieved by prioritizing tests based on criteria like coverage of changed code, their speed or overall coverage. The whole test suite is still executed, however, the order of execution changes in every run.

The second approach, test case selection, aims to pick the most relevant tests for each run and execute only those, while the rest of the test suite is skipped. To achieve this, tests are selected based on their coverage of changed code, that is, only tests that cover code that has been changed since the last test run, are executed. This means that, though tests are picked, the test suite as a whole stays unchanged and a new selection is made for the next test run based on the changes made in-between these two runs.

Test suite minimization, the final approach, is the only one that permanently changes test suites. The idea behind this method is, to permanently remove redundant tests, that is, tests that contribute nothing or very little to the overall quality of the test suite. The main difficulty of this approach is determining whether a test case is redundant. Most implementations of test suite minimization use one or more criteria, such as code coverage, mutation coverage, mc/dc coverage, or execution time. The tests are selected based on how much they contribute to satisfying the selected criteria. Tests that add nothing or very little to the satisfaction of the minimization criteria are omitted. Ideally, this leads to a test suite that can be executed quicker and is easier to maintain due to the reduced size, without drastically impacting its fault detection capability or structure. For this thesis, we only analyze test suite minimization, which means from here onward, only this method is relevant.

Despite its apparent benefits, test suite minimization is rarely used in practice. Though there might be more reasons why it is not used, we identified two central difficulties with test suite minimization.

The first is that the changes that are made with test suite minimization are permanent, that is, tests are usually permanently removed. Removing tests permanently that might at some point still find bugs might be a difficult thing to do. Though there are ways around this like smoke tests, the deletion of test cases can prevent people or companies from considering tests suite minimization and make them gravitate towards the other solutions like test case selection and prioritization. However, these other solutions are not widely implemented either.

The reason for this is most likely similar to the second reason test suite minimization is not often adopted: It is challenging to implement. The main issue is obtaining the data that is used for minimization. Depending on the size of the project, it can be challenging to get even code coverage data. The type of data plays a vital role in how difficult it is to get. While collecting coverage data is still possible with rather large projects, def-use coverage and mutation coverage get expensive quickly with increasing size.

Problem Statement In modern, large scale software projects, regression testing can slow down development with long runtimes and late feedback. One approach to combat this problem is test suite minimization, which aims to reduce the runtime of a test suite by removing redundant or useless tests from a test suite. The main problem to solve with test suite minimization is determining when a test case is redundant or useless. This can be done based on a variety of different criteria, but the selection is never perfect. Even though test suite minimization could provide valuable benefits in practice, it is not yet widely used in practice due to the problems mentioned above. With the following contributions, we aim to improve the applicability of test suite minimization in a realistic environment.

Contribution We provide two core contributions to increase the understanding and viability of test suite minimization.

- *Proposal of a test suite minimization algorithm with a new criterion*

Our first contribution is a new test suite minimization algorithm. It uses a criterion that we consider essential for the quality of a test suite, the locality. By including this, we want the resulting test suite to not only maintain its fault detection capability but also maintain or even improve its overall quality in the process. The overall quality here refers to a test suite's maintainability and its structure, that is, how well it conforms to the test pyramid. Contrary to most test suite minimization algorithm which optimizes for the lowest number of tests, our proposed algorithm aims to improve runtime while maintaining coverage, mutation coverage and quality of the test suite as much as possible.

- *Use of test suite minimization on large scale industry projects*
  The second core contribution of this thesis is the use of large software projects from industry. We could not find any prior research that has been conducted on similar projects. Since these are the projects, where test suite minimization would provide a real-world benefit, it is vital to test suite minimization on this kind of project. However, since it is a lot harder to get access to this type of project as opposed to open source projects.

Besides these major contributions, a minor contribution is the focus on runtime. As previously mentioned, most research focuses on the test suite size before and after minimization. However, in most cases, the problem is not the number of tests but rather the time the tests take to run and the quality of the test suite.

Overview In Chapter 2, we give an overview of some fundamentals of software testing, mutation testing, and test suite minimization, which are required for the rest of the thesis. We then take a look at some of the related research that has been done so far in Chapter 3. In Chapter 4, we explain the approach we took to test suite minimization and how we implemented it. We present and discuss our study at the hand of five research questions in Chapter 5. Finally, we present our conclusion in Chapter 6 and some interesting possibilities for future research of the topic in Chapter 7.

2 Terms and Definitions

In this chapter, we lay out some fundamentals, necessary to understand the rest of the thesis. We begin with an overview of testing terms and definitions. We describe test suite minimization, which aims to reduce the size and runtime of a test suite while keeping the impact on the overall quality of the test suite as low as possible. Finally, we look at the tools that we used for mutation testing and obtaining test coverage.

2.1 Testing

Software testing aims to improve the quality of published software. It consists of several different activities, which can be both manual and automated. However, the relevant kind for our research is automated testing. Automated testing can be used to achieve a variety of different goals, including testing the performance of an application or testing if an application still runs well under load, which evaluates non-functional aspects of a system. However, we only concern ourselves with functional tests, that is, tests that have the goal to assure that a system acts according to its specifications. In practice, this means they test for faults and software bugs in the system.

2.1.1 Test Cases and Test Suites

The definitions of a test case and a test suite vary in the literature. We define a test case as a single executable test function. A parametrized test is also only counted as a single test case, even though the test case is executed multiple times. A test suite, in our scenario, is defined as all the test cases of a single project, so every project has exactly one associated test suite.

The idea of the test pyramid is to give a guideline on how to structure a test suite.

2.1.2 Test Pyramid

The test pyramid is a model for how to structure a test suite, so it is efficient and scales well. Mike Cohn developed the concept of the test pyramid in 2004 and presented it later in his book *Succeeding with Agile* [Coh10]. The test pyramid divides tests into different categories according to the level at which they test. Though the naming can be inconsistent, most often varying names for the following three levels are used:

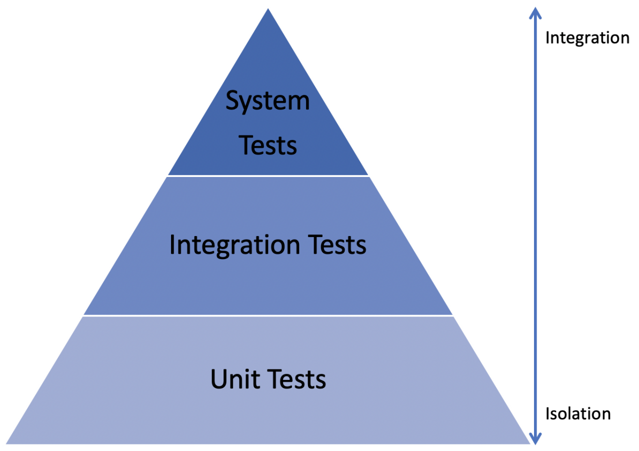

Figure 2.1: Test Pyramid

- *System Tests*
  System tests, depending on the application also called UI tests, cover the whole integrated system. These emulate user input or interact with a system's outward API. This is great for making sure that the actual interactions that a user has with the system work correctly. However, because the user interface can change independently from the underlying logic, these tests can break from non-functional changes to the system, making them more unstable than the other kinds of tests. In conclusion, even though system tests are important for testing a complete software system, they should not be used exhaustively for testing.
- *Integration Tests*
  Integration testing aims to verify the internal interfaces between components of a system against its design. In a web application for example, a system test might simulate user inputs to a web browser, while an integration test would directly make the HTTP call to the server. This means it would test the REST API of the server but exclude the UI from its test. Though similar in what they test, integration tests are faster than system tests but usually slower than unit tests and cover more code than a unit tests but less than a system test.
- *Unit Tests*
  Unit tests are concerned with a specific, usually short section of code like a single

function. Instead of going through the whole application stack, they directly check a part of the code. They are the lowest level tests and normally run a lot faster than the other levels since they do not need to wait for UI changes or database queries. Finding a software fault from a failed unit test is usually also quicker than it is for the other test levels because a unit test covers less code than the higher-level tests. Due to them being short and finely targeted tests, they are typically also easier to write, understand, and maintain.

The visualization of the test pyramid in Figure 2.1 shows the levels of test and the intended distribution of tests. Unit tests should make up the bulk of a test suite since they run quick and reliably. Integration tests should be used to cover areas that are not addressed by unit tests, that is, interfaces of components of a system. Even though they are still important to make sure everything works as intended, UI tests should be implemented sparingly, since they are slow and prone to problems like flickering tests where they occasionally fail unpredictably.

2.1.3 Mutation Testing

Mutation testing can be used to determine the quality of a test suite in terms of its ability to detect faults. To achieve this, artificial faults which are called mutants are introduced into a system. Mutants aim to emulate real fault as closely as possible. Detecting one of these mutants is called killing a mutant. When the test suite does not find a mutant, the mutant has survived. The `Mutation Score` of a test suite tells us how good the test suite is at detecting mutants [JH10]. It is determined by the ratio between mutants which are killed the number of overall mutants and can be calculated with the following formula:

$$\text{MutationScore} = \frac{\text{numberOfKilledMutants}}{\text{numberOfIntroducedMutants}}$$

In Figure 2.2, we show how mutation testing typically works in practice. First, we run a mutant generator on the source code. The mutant generator creates a list of mutants, each of which is essentially a copy of the source code which includes an artificially introduced fault. The selected mutation operators determine the types of introduced faults. The number of mutants can be manually limited. If the number is not limited, mutants are generated wherever possible. After the mutants have been generated, we run tests for each mutant. To increase performance, only tests that can potentially find the mutant, that is, tests that cover the part of the code in which the current mutant is located, are considered. The result of mutation testing is a report in which killed, and surviving mutants are listed.

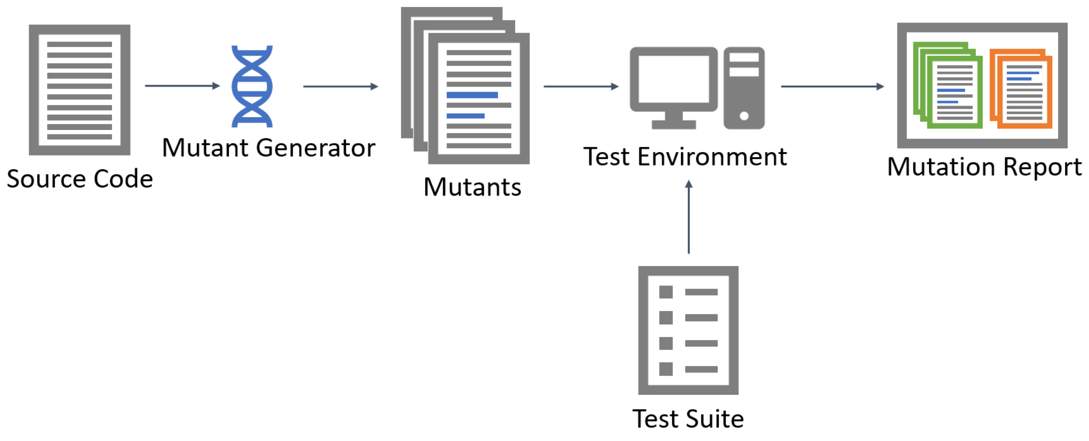

Figure 2.2: Mutation Testing

Since it is vital to know whether the detection of these artificial faults has any impact on a test suite's ability to find real faults, several studies have been conducted which investigate this issue. It has been found that there is a significant correlation between mutation detection and real fault detection [ABL05; Jus+14; DT96].

Mutation testing also has some inherent flaws caused by the mutators. Some of the mutants might be impossible to find because they cause an endless loop or sometimes, a mutant can be logically identical to the original source code. This means that, depending on the mutation operators, a perfect mutation score is often not possible.

2.2 Test Suite Minimization

The goal of test suite minimization is to find and remove redundant test cases from a test suite to decrease runtime, increase maintainability, and reduce the overall size of the test suite. Note that in the following chapters, when we mention testing requirements, we do not refer to traditional requirements in software engineering which tell us what a system should be able to do. Instead, a testing requirement denotes a single unit of a minimization criterion. This could, for example, be a single covered line, a single killed mutant or a def-use pair covered.

Test suite minimization is often treated as a variation of the minimal hitting set problem. Rothermel et al. defined the minimization problem as follows: Given a test suite T consisting of test cases $t_1, t_2, \ldots, t_m$, testing requirements $r_1, r_2, \ldots, r_n$ which can be fulfilled by T and subsets $T_1, T_2, \ldots, T_n$ of T where T_i contains all test cases that satisfy r_i, find a representative set of test cases from T that satisfies all testing requirements r_i. The representative set ideally is a minimal subset of T that satisfies the testing requirements. With this goal, test suite minimization can be described as finding a minimal hitting set of test cases that meet the testing requirements.

Because the minimal hitting set problem is NP-complete [GJ02], previous research on the test suite minimization was often concerned with finding better heuristics for the minimal hitting set problem [HGS93; CL96].

We divide test suite minimization techniques into categories. We make the fist differentiation between single and multi criteria test suite minimization algorithms. There are a number of different testing requirements that can be used for test suite minimization, like line coverage, data flow coverage, decision coverage, or mutation coverage. Additional factors like runtime or complexity can be used for preferential selection of tests. A single criterion test suite minimization algorithm optimizes the reduced test suite to fulfill one criterion, while a multi criteria algorithm takes multiple, sometimes competing criteria into consideration. We make the second differentiation between adequate and inadequate minimization techniques. An adequate technique requires all r_i to be satisfied while an inadequate method allows for some r_i to be left unsatisfied.

Especially multi criteria algorithms are often inadequate since, if we have multiple, sometimes conflicting criteria, satisfying 100% of all test requirements would mean reducing the size of the test suite by very little.

2.3 Tools

In this section, we present the tools that were used for the implementation of our minimization approach. We used third party tools for coverage collection and mutation testing.

2.3.1 Coverage Collection

We used coverage from two different sources for our research. The coverage that we collected ourselves was captured using JaCoCo, while we got the OpenCover results for one of our industry projects.

The first tool, JaCoCo, is an open source coverage collection tool for Java. Since Jacoco starts profiling when a JVM starts and ends when that JVM stops, we need a modified version of JaCoCo. That is why we used the Teamscale JaCoCo Agent[1]. It allows us to start and stop coverage collection for a test case by sending HTTP messages when a

[1]https://github.com/cqse/teamscale-jacoco-agent

test case starts and stops. With these signals, the JaCoCo agent can write coverage after every test case.

The second source of coverage is OpenCover[2] which provides testwise coverage for `.NET`. OpenCover provides testwise coverage by default, so no modifications were necessary here.

2.3.2 Mutation Testing Tool

For mutation testing, we used `Pitest (PIT)` [3], an open source tool for mutation testing in Java. Pitest provides a variety of mutation operators that aim to replicate real faults as closely as possible, which means, they aim to be neither too difficult nor too easy to find. Manually injected faults tend to be more difficult to find compared to real faults [ABL05] . Another goal of the mutation operators used by Pitest is to minimize the number of equivalent mutants.

[2]https://github.com/OpenCover/opencover
[3]http://pitest.org

3 Related Work

The most straightforward regression test suite minimization techniques use a single criterion and are based on finding a minimal hitting set for some type of coverage, like code coverage or all-uses coverage. Among these algorithms, the most prominent is the greedy algorithm which we explain in detail in Subsection 4.2.1. A simplified explanation of the greedy algorithm is, it always chooses the test case that hast the most additional coverage until 100% of the original coverage is achieved. Chen and Lau used two variations of the greedy algorithm in a simulation study, the GE(Greedy Extended) and the GRE(Greedy Redundant Extended) [CL96]. The GE algorithm starts by selecting essential test cases, that is, test cases that cover testing requirements that are not covered by any other tests. After these tests are selected, the standard greedy algorithm is applied, which means that iteratively, the test case which covers the most remaining requirements is selected. The GRE algorithm adds an additional step to the GE algorithm. Before applying the GE heuristic, the GRE algorithm removes all redundant test cases, that is test cases that are completely covered by one or more other test cases.

Chen and Lau compared the results of these two algorithms to the HGS (Harrold-Gupta-Soffa) algorithm, which was introduced by Harrold et al. in their paper 1993 paper "A Methodology for Controlling the Size of a Test Suite" [HGS93]. Their heuristic selects test cases based on the cardinality, which is defined by the requirement covered by a test that is covered by the least number of other tests. At the example of line coverage, this means, a test that covers a line that is only covered by this test has a cardinality of one. On the other hand, a test that only covers lines that are covered by at least two other tests has a cardinality of three. The tests are selected from lowest to highest cardinality until full coverage is achieved. For tests with the same cardinality, tests with higher overall coverage are preferred. The comparison between GE, GRE, and HGS by Chen and Lau showed that no heuristic is better than the others in all cases [CL96].

Tallam and Gupta introduced another variation on the greedy algorithm in 2006 [TG06]. They suggested a delayed greedy algorithm which, before applying the greedy approach, applies two improvements. First, if the coverage of a test case t_i is a subset of the coverage of another test case t_j, then t_i is removed. The second step is to remove redundant requirements. A requirement r_i is considered redundant if all tests that cover it, also cover another requirement r_j. After these redundancies have been removed, the normal greedy algorithm is applied. They found that their algorithm performed either equally as good or better than the HGS algorithm in their experiments.

A different approach to single criterion test suite minimization was taken by Horgan and London who implemented an implicit enumeration algorithm for test suite minimization with ILP (Integer Linear Programming) in their testing tool ATAC [HL92]. Wong et al. later used ATAC to minimize randomly generated test suites and found that they can significantly reduce the size while maintaining good fault detection capability [Won+98].

In a more recent study, Shi et al. conducted extensive experiments on 18 GitHub open source projects using several single criterion test suite minimization algorithms [Shi+14]. They also analyzed the differences between code coverage based minimization and mutation score based minimization. In terms of the minimization algorithms, they found that the differences between greedy, GE, GRE, HGS and ILP are marginal, which is why we decided to use the greedy algorithm as a baseline for the evaluation of our proposed weighted-sum algorithm. In terms of the different criteria, they concluded that, even though it is a bit worse in terms of the size of the resulting test suite, the minimization with mutation score as the basis is more beneficial since it retains better fault detection capability.

The second notable category of test suite minimization techniques are multi criteria based test suite minimization approaches. These consider more than one criterion for test suite minimization. The simplest type of multi criteria algorithms are ones like the extended HGS by Jeffrey and Gupta, which still use single criterion algorithms but check for redundancy with respect to two or more criteria [JG05]. This means that if a test is deemed redundant according to the first criterion, the second criterion is checked, and only if it is redundant for both, it is removed. Using edge coverage as their primary criterion and ATAC as their secondary criterion, they maintained better fault detection compared to a single criterion while still achieving a substantial size reduction.

In their 2007 paper, Yoo and Harman took a different approach to multi criteria test suite minimization [YH07]. They used a Pareto efficient approach using up to three criteria which they compared to the greedy algorithm. Wei et al. used the same approach and compared a number of different Pareto efficient algorithms [Wei+17]. Both of these papers used a criterion to represent the fault detection capability, and they found that including this as a minimization criterion improves the fault detection of the minimized test suites significantly.

Black et al. pursued a different approach to combining multiple criteria [BMK04]. They combined def-use coverage and historic fault data as a weighted-sum. For selecting the best tests according to the two criteria, they used ILP. Hsu and Orso also used ILP to perform multi criteria test suite minimization [HO09]. They extended the work of Black et al. by adding the execution time, using more and larger projects and by evaluating seven weighted-sum and one prioritized minimization policy.

In recent years, a new type of test suite minimization has appeared. These new algorithms don't use coverage or other previously obtained statistics about the code but instead operate solely on the source code. They target large scale projects for which performance is essential, and obtaining coverage or other metrics is too expensive.

Philip et al. used their machine learning based tool FastLane to select tests based on the testing history and the commit log of a project [Phi+19]. They managed to reduce the runtime of a test suite by roughly $1/5$ while maintaining 99.99% test accuracy.

A different approach to test suite minimization for massive systems was taken by Cruciani et al., who used a similarity based approach for finding a representative subset of test cases [Cru+19]. They found that their approach delivers results, similar to code coverage based state of the art techniques while running in a fraction of the time.

4 Implementation

In this thesis, we want to evaluate the effectiveness of a new, multi-criteria, weighted-sum minimization algorithm. The goal of this new algorithm is to reduce a test suite's runtime with as little reduction in the overall capabilities of the test suite as possible. To assess the performance of the proposed algorithm, we use a code coverage based greedy algorithm as a baseline. As described in Chapter 3, the greedy algorithm is a well-known minimization algorithm, and since the differences between the single criteria based minimization algorithm have been minor in previous studies, we decided to use the simple greedy algorithm. In Figure 4.1, we depict the procedure which we used for test suite minimization. We begin by recording the minimization data that we require, which means running the test suite while tracking coverage and runtime of the individual tests. The mutation report is also generated at this stage. The obtained data is then combined to one report which we use as input for minimization. With this report, we then use either greedy or weighted-sum algorithm to minimize the test suite.

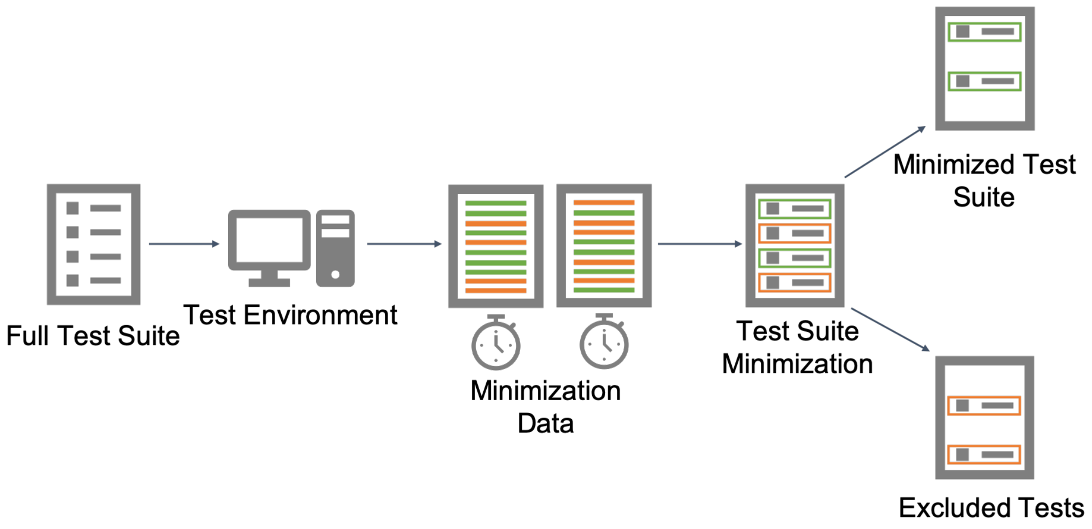

Figure 4.1: Implementation

4.1 Minimization Data

The greedy algorithm only requires testwise coverage as its input. However, our weighted sum algorithm additionally requires time measurements and, if it is feasible in regards to runtime, mutation coverage as minimization criteria. In this section, we describe how we obtained the data for minimization, and we discuss the usefulness of our selected minimization criteria.

4.1.1 Obtaining Minimization Data

To obtain testwise coverage, we use the modified JaCoCo agent, previously described in Subsection 2.3.1. By implementing the `RunListener` interface from JUnit4 or the `TestExecutionListener` interface from JUnit5, we can listen to test execution and notify the JaCoCo agent via HTTP when a test starts and ends as well as the tests name to assign coverage to each test case. The `TestListener` implementation also allows us to measure test execution times.

For the mutation coverage, we use Pitest, previously mentioned in Subsection 2.3.2. It allows us to generate a full mutation matrix. This means, Pitest doesn't stop running tests for a mutant as soon as it is killed but instead runs all tests that could possibly kill a mutant. From this, a report is generated that shows which mutant is killed by which tests.

We combine this matrix with the testwise coverage report and the test execution times to get the input data for our weighted sum test suite minimization algorithm. The greedy algorithm uses the same input data, however, it only considers code coverage when selecting tests.

4.1.2 Minimization Criteria

We have chosen four minimization criteria, which we found to be some of the most relevant criteria for choosing good tests and respectively rejecting redundant tests.

Code Coverage Code coverage is widely used for measuring a test suite's adequacy in software projects. Even though covering code does not tell us anything about whether faults in the covered code are found. However, with uncovered code, we can be sure that no faults are found and assuming that the developers intend to find faults when testing the code, we can assume that a test suite that covers more code is generally preferential to one that covers less. We count code as covered when it is covered by one test. This means that tests that subsequently cover the same code do not gain any additional score in our algorithm from this coverage.

Mutation Score Since, as previously mentioned, code coverage does not tell us anything about how well the tests detect faults in the covered code, we use the mutation score as our second criterion for test suite minimization. The goal of the mutation score is to indicate a test's fault detection capability. More details about mutation testing can be found in Subsection 2.1.3. For the mutation score, we want to find as many faults as possible. That is, kill as many mutants as possible. We want to find each fault only once, which means that a test does not gain from finding faults that are detected by another already selected test.

Execution Time One of the central goals of test suite minimization is to reduce the cost of regression testing. This means, we want to reduce the execution time of the minimized test suites. To achieve this, we prefer faster tests over slower ones.

Locality The fourth and last criterion we use for test suite minimization is what we call locality. It means that we prefer tests that cover fewer classes over tests that cover more. The reason for this criterion is that we want the minimized test suite to conform to the concept of the test pyramid, explained in Subsection 2.1.2. Because the lower levels of the test pyramid are easier to maintain and it is typically easier to find faults from their failures, we prefer these over system or UI tests.

4.2 Minimization Algorithms

For our evaluation, we implemented two test suite minimization algorithms. Our first algorithm is a coverage based implementation of the greedy algorithm. We use it as a baseline to compare our weighted-sum algorithm to. Since the greedy algorithm has been well researched and widely used in literature, it is well suited as a baseline to compare a more novel algorithm to. Our second algorithm follows an enhanced weighted-sum approach. It selects tests to be included based on a score calculated from multiple criteria as opposed to the greedy algorithm's single criterion.

4.2.1 Greedy

The code coverage based greedy algorithm is an adequate test suite minimization algorithm. This means, it chooses tests until all the requirements $r_1 \ldots r_n$ which are satisfied by the original test suite are fulfilled with the minimized test suite. Tests are chosen based on the coverage they add to the existing selection of tests. For the implementation of the algorithm this means that after we add the test case t_i which currently fulfils the most testing requirements $r_l \ldots r_m$, we subtract its coverage from all remaining test cases. By ignoring what has already been covered and iteratively adding the test with the most additional coverage, the greedy algorithm's goal is to satisfy all requirements $r_1 \ldots r_n$ with as few test cases as possible.

4.2.2 Weighted-Sum

Our weighted-sum algorithm selects test cases on the basis of their score which is derived from three or four minimization criteria. The three criteria which we use for all projects are execution time, code coverage and locality. For all projects where mutation testing is feasible in terms of runtime, we additionally use mutation coverage. In the following section, when coverage is mentioned, we refer to both, code coverage and mutation coverage.

Score Calculation The first step we take is to normalize all the values by bringing them into a range from zero to one. Since for time and locality low values are good and for coverage and mutation coverage high values are good, there are two cases for normalization, as shown in Equation 4.1. In this and the following equations in this chapter, the i denotes the index of the test case for which the score is currently calculated. Max and Min denote the highest and lowest value of the respective criterion among all test cases. The first equation shows the formula for values where higher is better and the second on where lower is better. We then take the square root to slightly reduce the impact that a single test with a large amount of coverage or a very fast tests case has.

$$\begin{aligned} \text{normValueHigh}_\text{i} &= \sqrt{\frac{\text{value}_\text{i}}{\text{value}_\text{max}}} \\ \text{normValueLow}_\text{i} &= \sqrt{\frac{\text{value}_\text{min}}{\text{value}_\text{i}}} \end{aligned} \tag{4.1}$$

We then add the normalized coverage, mutation and locality scores in a weighted-sum as shown in Equation 4.2. Since we regard mutation score and coverage as the more relevant criteria, we give them a higher score than the locality.

$$\text{weightedSum}_\text{i} = 2 * \sqrt{\frac{\text{coverage}_\text{i}}{\text{coverage}_\text{max}}} + 2 * \sqrt{\frac{\text{mutations}_\text{i}}{\text{mutations}_\text{max}}} + \sqrt{\frac{\text{locality}_\text{min}}{\text{locality}_\text{i}}} \tag{4.2}$$

By using the time score as a factor instead of an addend, we decrease the overall score if the test is slow. Time influencing the other factors better represents our goals since a fast test without coverage should not have a high value and a slow test that covers a lot can be worse than a fast test that covers little. On the other hand, using time alone can rate a test as useless even though it adds a lot of coverage. To consider this case, we add a base value of 0.2 to the time factor.

We also add a penalty factor for tests without any coverage at all. This has a value of 1 if the test adds any coverage and 0.5 if it does not. A test case that does not touch many classes and is very fast might not add much cost, but we still prefer tests that add additional coverage.

$$\begin{aligned}\mathrm{score_i} = \left(\sqrt{\frac{\mathrm{time_{min}}}{\mathrm{time_i}}} + 0.2\right) * \Bigg(2 * \sqrt{\frac{\mathrm{coverage_i}}{\mathrm{coverage_{max}}}} \\ + 2 * \sqrt{\frac{\mathrm{mutations_i}}{\mathrm{mutations_{max}}}} + \sqrt{\frac{\mathrm{locality_{min}}}{\mathrm{locality_i}}}\Bigg) * \mathrm{penalty} \quad (4.3)\end{aligned}$$

Algorithm The algorithm starts by calculating an initial score for each test case in the test suite, according to Equation 4.3. Then, same as the greedy algorithm, the best test is selected iteratively according to the score. After every selected test, the coverage values for the remaining tests are updated, that is, the coverage from the newly added test is subtracted from the remaining tests. This means the scores have to be updated after every round as well to represent the current additional coverages.

Since our weighted-sum implementation is an inadequate test suite minimization algorithm, it doesn't aim for 100% mutation or code coverage but instead selects additional tests until a time limit is reached.

4.3 Limitations of the Implementation

Though our implementation of test suite minimization itself is independent of the language of the project, the tools we have used to obtain our data, we are limited to Java. The only exception is the Siemens project which is a .NET project. For this project, we got testwise coverage from Siemens.

When combining testwise coverage with the mutation report, we need to match the test cases from both reports. For parameterized tests, this does not always work. Pitest, in some cases, reports distorted test names when handling parameterized tests. This means that for some parameterized tests in the testwise coverage report, we cannot assign the according entries in the mutation report. However, due to the low number of parameterized tests in the test suites of our study subjects, we decided to accept this limitation since fixing it would have required making changes to Pitest.

5 Empirical Assessment

Our research revolves around two central questions. First, we want to find out how well our proposed weighted-sum multi criteria test suite minimization algorithm performs when it is applied to real world projects. The second question is, how well test suite minimization performs when using it on large scale, closed source industry projects, and how they compare to open source projects. We begin this chapter with a description of our research questions followed by a section on the study subjects. We then explain our study design and present and discuss the results. Finally, we consider threats to the validity of our study.

5.1 Research Questions

We have posed five research questions that address different aspects of the above mentioned two central questions that this thesis aims to answer.

RQ1 – What is the sweet spot between execution time and coverage of our proposed weighted sum algorithm?

With our weighted sum approach, we don't aim for 100% coverage on any of the criteria. Instead, the algorithm aims to select the most relevant tests until a set time limit is reached. The goal of this research question is to find a sweet spot between runtime and coverage where a significant amount of time can be saved while keeping as much as possible of the original line coverage.

RQ2 – How well does our weighted-sum algorithm maintain the fault detection capability of a test suite?

Since a test suite's primary purpose usually is to find faults, test suite minimization is useless, if it does not maintain most of the fault detection capabilities of the full test suite. We want to find out how well the time target that was determined in RQ1 maintains a test suite's fault detection capability.

RQ3 – How does our proposed algorithm compare with the greedy algorithm

To get a reference on how well our weighted-sum algorithm performs in test suite minimization, we compare it to one of the most well-known minimization algorithms, the greedy algorithm. Our implementation of the greedy algorithm is a single criterion test suite minimization algorithm that is based on code coverage, making it a lot simpler than the weighted-sum approach. If the new algorithm does not perform considerably better than the greedy algorithm, the high cost of collecting the additional criteria might be a good reason to stick with the simpler greedy algorithm. To get an overview of the strengths and weaknesses of each algorithm, we compare them based on several important characteristics of a minimized test suite.

- **in regards to line coverage?** Line coverage is the primary goal of the greedy algorithm, and since we use the adequate variant of the greedy algorithm, it is better in terms of coverage. However, we want to know how much of a difference there is between the two algorithms. We also analyze how coverage behaves over time, that is, how much code coverage can be achieved with how much test suite runtime with each algorithm.

- **in regards to mutation coverage?** Since mutation coverage is not a criterion of the greedy algorithm, while it is one of the weighted-sum's criteria, we expect the weighted-sum algorithm to perform better than the greedy algorithm in terms of the mutation coverage of the minimized test suite. Depending on how much the mutation-score is correlated with coverage, the greedy algorithm might be similarly good or even better.

- **in regards to test execution time?** The test suite execution time reduction of the weighted-sum algorithm is what we set as goal for the minimization process. However, we want to find out whether there is a significant benefit compared to the greedy algorithm.

- **in regards to the files covered per test?** Ideally, we maintain the characteristics of a test pyramid when using test suite minimization, that is, keep mostly unit tests and fewer integration and system tests. By comparing the number of files a test covers, we want to find out, how test suite minimization affects the shape of the test pyramid, that is, how it changes the number of files touched per test. We expect the greedy algorithm to perform worse in this comparison, since this is not part of its minimization criteria and preferring tests that cover as much code as possible is a disadvantage in this category.

RQ4 – Do results from large scale, industry projects differ from those of open source projects?

A vital factor for test suite minimization is that it has to work with massive closed source projects as well as open source ones. This is vital since the larger a project is, the more likely it is that it actually has the problem of an oversized test suite. Because we did not find any papers evaluating test suite minimization with large scale industry projects, this is one central contribution of this thesis. We use open source projects as well as closed source, large scale projects which are developed in an industry environment for our analyses. Our goal is to determine whether test suite minimization algorithms perform comparatively well with closed source projects as they do with open source ones.

RQ5 – How well do the minimization algorithms scale with large projects?

The minimization algorithms are only useful if they can be calculated with reasonable effort. By testing the algorithm with projects of varying sizes, we want to find out whether the runtime of the test suite minimization algorithms scales well with project size.

5.2 Study Subjects

To answer the posed research questions, we analyzed ten software systems. We have divided the ten projects into two different types of projects. Eight of them are open source projects from GitHub while the remaining two are closed source industry projects. The open source projects are Java based and use Maven as their build system. They range from 9k SLOC (Source Lines of Code) to 175k SLOC. We chose these projects based on several limitations. We wanted all of the projects to be of a certain size and to have a comprehensive test suite, preferably implemented in JUnit. Another important criterion for the open source projects was that Pitest needs to run on them without major modifications to the test suite. Pitest can run into problems when tests need to be run in a specific order due to, for example, files being written to disk or when a project has a very complex build.

We did not require the closed source projects to be able to run Pitest. Due to these projects being considerably larger in scale and the test suites taking at least two orders of magnitude longer to execute, mutation testing was not viable for these projects in the time we had for our research. Including large projects with high test suite duration was an important aspect for us since large projects which have been in development for a long time are much more likely to have the problem of oversized test suites.

Table 5.1: Study Subjects

Open Source Projects	SLOC	SLOC Project	SLOC Test	Test Coverage	# Tests	# Mutants
AC Collections	62,934	28,708	34,226	86%	15,183	8,253
AC Lang	75,467	27,825	47,642	95%	3,484	13,088
AC Math	174,522	86,131	88,391	92%	4,828	37,674
AssertJ	161,306	35,650	125,656	91%	14,685	10,354
Ebean	170,656	99,317	71,339	77%	2,618	25,056
JOpt Simple	9,433	7,023	2,410	98%	838	678
JSoup	20,136	12,037	8,099	84%	671	4,711
Spoon	112,650	60,619	52,031	83%	1,619	15,877
Closed Source Projects						
Siemens	Unknown	203,964*	Unknown	75%	5,366	-
Teamscale	516,949	407,678	109,271	76%	2,979	-

In Table 5.1, we show a detailed overview of the ten selected systems. The SLOC count counts all lines in code files, excluding empty lines and source code comments. The SLOC and test coverage numbers are as reported by JaCoCo. The number of tests includes all tests which we could successfully track testwise coverage for.

The Siemens project is the only project for which we do not have the source code, which is the reason for the incomplete table entry. Contrary to all other projects, it is implemented in .Net, and the tool OpenCover was used to obtain coverage. We worked with the resulting coverage reports from a test run. That is the reason that some numbers for the Siemens project are unknown. We only know the code that was covered. The SLOC for the Siemens project are only coverable lines which means that import statements and class declarations are not included. However, since we do not know the actual number, we use the coverable lines as an indicator of the size of the project. The actual number can be expected to be considerably higher.

5.3 Study Design

Here, we give a short description of how we designed the experiments for each of our research questions.

RQ1 – What is the sweet spot between execution time and coverage of our proposed weighted sum algorithm?

To answer this question, we used the weighted-sum algorithm on the test suites of the projects with which mutation testing was possible and set the time goal to a specific percentage of the full time. We iterate in one percent steps from one percent to 100 percent. We then plot the results, combine them into a line graph, and try to find a time where all projects reach a sufficiently high coverage and achieve substantial time savings compared to the full test suite. Though an individual sweet spot could be found for every project, we try to find one sweet spot that delivers satisfying results for all projects.

RQ2 – How well does our weighted-sum algorithm maintain the fault detection capability of a test suite?

To check, whether our sweet spot from RQ1 maintains good fault detection capability, we take the tests that are selected with the sweet spot for each of our projects and evaluate the mutation coverage. We compare the fault detection capability, which we achieve with the tests selected at the sweet spot from RQ1 with the fault detection capability of the complete test suite.

RQ3 – How does our proposed algorithm compare with the greedy algorithm

We partitioned this question into several aspects that can be used to determine the performance of a test suite minimization algorithm. For each criterion, we determine the performance of both algorithms and compare them to each other.

- **in regards to line coverage?** To evaluate the algorithms' performance in terms of line coverage, we compare how well they perform when they are normally executed as well as how they perform over time, that is, at what time they achieve what coverage on average.
- **in regards to mutation coverage?** We compare the mutation score in a similar fashion to the code coverage. This means we compare the values when the algorithms are executed normally as well as how well they perform across test suite runtime limits.
- **in regards to test execution time?** For the third part of this comparison, we measure the speedup of each algorithm relative to the full test suite, that is, how many times longer does the full test suite need than the minimized test suites.
- **in regards to the files covered per test?** For the final point of comparison, we check, how many files a test covers on average with each algorithm. For this, we simply count the number of files that are part of a test's code coverage.

RQ4 – Do results from large scale, industry projects differ from those of open source projects?

For our second but last research question, we analyze the two closed source industry projects which we have shown in Section 5.2. We compare the coverage of the closed source projects to the open source ones for both algorithms.

RQ5 – How well do the minimization algorithms scale with large projects?

To evaluate our last research question, we look at how long the test suite minimization takes for all projects and, more importantly, how the runtime changes with the size of the project and the number of tests.

5.4 Results and Discussion

In this section, we present and discuss our obtained results and their potential implications.

Answering RQ1 – What is the sweet spot between execution time and coverage of our proposed weighted sum algorithm?

In Figure 5.1, we have plotted the relative runtime on the x-axis and coverage, relative to the full test suite, on the y-axis. Note that we omitted the lower 50% relative coverage of the y-axis for better visibility of the relevant chart area. We can see that most of the curves follow a similar path. The coverage of all projects is above 90% from 11% onward of the original runtime when tests are selected with the weighted sum algorithm. To determine a sweet spot between execution time and coverage, we picked a value where the average is significantly above 90% and, equally important, we don't lose too much coverage in the worst case.

We decided to set the target test suite duration to 15% of the original test suite's runtime. The red vertical line in Figure 5.1 marks the target of 15% relative runtime. In Table 5.2, we have listed the resulting coverage for each project at the determined sweet spot. With the average loss being only 3.07% coverage and the worst-case coverage retention with 93.61% still closer to 95% than to 90%, the loss in coverage is quite small. The average total coverage drops from 88.25% to 85.67% which, at under 3%, is satisfactory. We experience the highest loss in absolute coverage with the project Spoon at 5.11%, which is still a rather small loss in coverage.

We can observe clear differences in how well the minimization performs between the different projects. Our results seem to indicate that there is a correlation between the

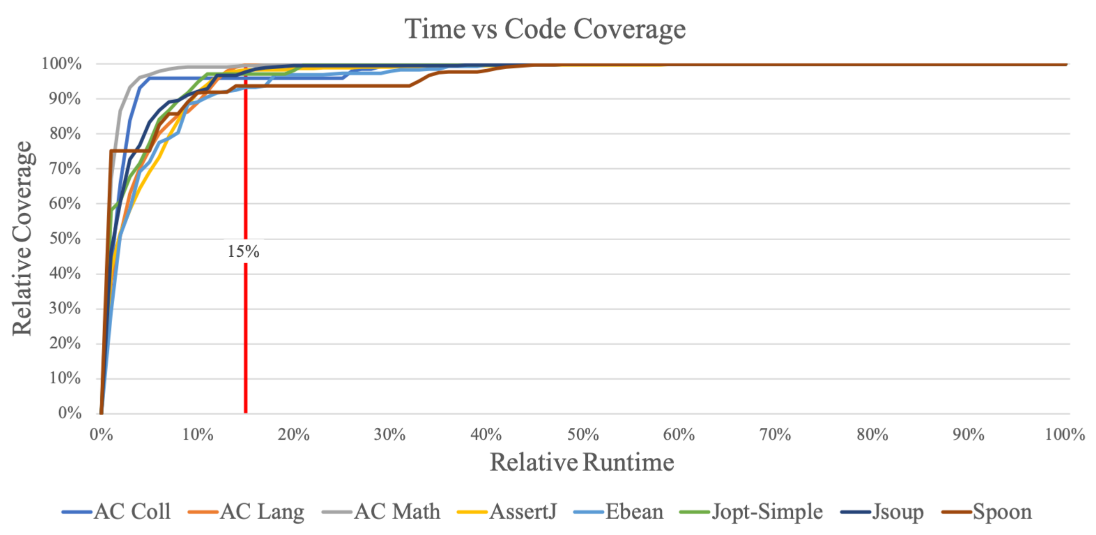

Figure 5.1: Coverage over Time of the Weighted-Sum Test Suite Minimization Algorithm

total coverage before minimization and how well the minimization algorithm performs. For the projects with higher test coverage, the results we get for our study subjects are better with the only two projects which are below 80% absolute coverage delivering the worst results and the projects with over 90% total coverage delivering some of the best results. This finding makes sense when considering that at these high coverages, increasing the coverage more easily leads to the introduction of redundant tests in terms of code coverage.

Overall, the loss of only 3.07% coverage with a run time reduction of 85%, which equates to an execution time of 1/6 of the original test suite, is very promising. We noticed, that for the different projects, the weighing factors of Equation 4.2 have different ideal values. An individualization of the formula we used for calculating the test score could improve upon the results that we obtained in our experiments.

Answering RQ2 – How well does our weighted-sum algorithm maintain the fault detection capability of a test suite?

To answer our second research question, we plotted a similar graph to the one from the first question. In Figure 5.2, we display the runtime, relative to the full test suite on the x-axis and the mutation score, also relative to the mutation score of the full test suite, on the y-axis. We once again cut off the lower 50% of relative mutation coverage to show more detail in the relevant area of the chart. At the goal of 15%, we have a very solid mutation score for five of the projects, however the projects Spoon, JOpt Simple

Table 5.2: Weighted-Sum Algorithm - Coverage

	Absolute Coverage	Absolute Coverage - W-S	Relative Coverage - W-S
AC Collections	86%	82.49%	95.91%
AC Lang	95%	94.67%	99.65%
AC Math	92%	91.57%	99.54%
AssertJ	91%	89.41%	98.25%
Ebean	77%	72.08%	93.61%
JOpt Simple	98%	95.17%	97.11%
JSoup	84%	82.09%	97.73%
Spoon	83%	77.89%	93.84%
Average	88.25%	85.67%	96.93%

and AssertJ are a bit lower at around 90%.

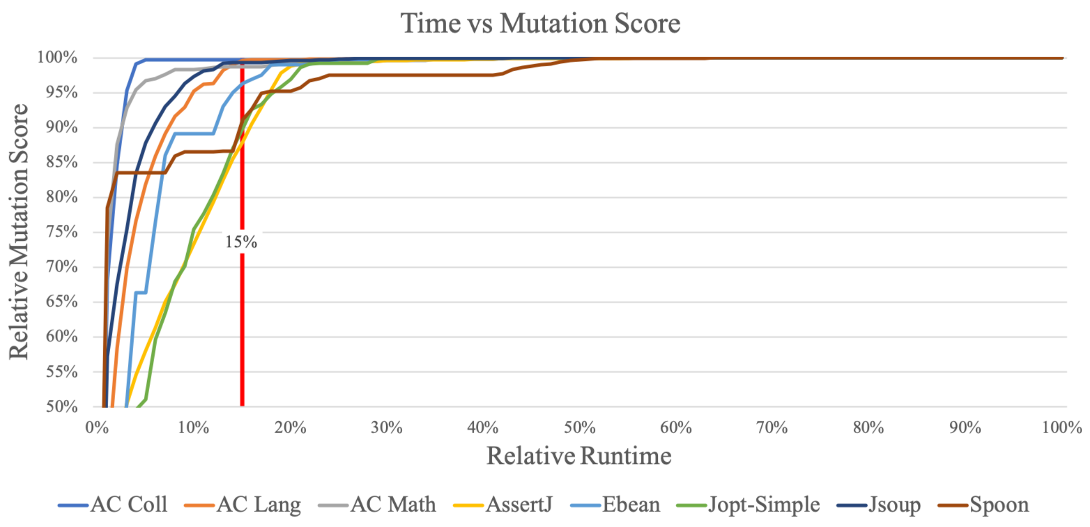

Figure 5.2: Mutation Score over Time of the Weighted-Sum Test Suite Minimization Algorithm

We show the exact mutation score values at the target of 15% relative runtime in Table 5.3. In the table, we can see that the average loss in mutation score is quite low at 4.79%. An interesting result here is that the mutation score appears to vary considerably stronger than the coverage. We can see that in Figure 5.2 , the lines are a lot farther apart than in Figure 5.1. Table 5.3 confirms this finding, the results in relative mutation coverage range from 88.12% to 99.71% which means, we have a 11.59% difference between the highest and the lowest value. For coverage, the maximum difference was only 6.04%. The mutation scores are also in groups, while the coverage is more evenly spread between best and worst results. In terms of the total mutation score, we have a loss of 3.69%, making the minimization technique quite promising.

Another interesting finding is that we see no obvious connection between the total mutation score and the mutation score retention with the weighted-sum algorithm. AC Collections and AC Lang both have very high mutation score retention, while AC Lang has a fairly high total mutation score and AC Collections a low one. The same goes for projects with lower mutation score retention like Spoon and JOpt Simple. They differ quite strong in their total mutation score but both have a relatively low mutation score retention compared to the rest.

Table 5.3: Weighted-Sum Algorithm - Mutation Score

	Mutation Score	Mutation Score - W-S	Relative Mutation Score - W-S
AC Collections	41.50%	41.38%	99.71%
AC Lang	83.56%	83.20%	99.58%
AC Math	74.73%	73.80%	98.75%
AssertJ	80.43%	70.87%	88.12%
Ebean	42.75%	40.74%	95.32%
JOpt Simple	94.81%	85.18%	89.84%
JSoup	64.35%	63.88%	99.27%
Spoon	71.62%	65.25%	91.11%
Average	69.23%	65.54%	95.21%

Answering RQ3 – How does our proposed algorithm compare with the greedy algorithm

We split the comparison of the greedy algorithm and the weighted-sum algorithm into multiple parts. In each part, we compare a key characteristic of the minimized test suites.

- **in regards to line coverage?** Since we have implemented a coverage based, adequate greedy algorithm, it has 100% relative line coverage per definition. In Figure 5.3, we have plotted the coverage relative to the coverage of the full test suite. We see that the full test suite and the greedy algorithm both have 100% as is to be expected. The weighted-sum algorithm trails behind with 96.93%, however, the difference is quite small. In Figure 5.4, we show the average relative coverage of all projects over time for greedy and weighted-sum algorithm. Note that for this graph, we did not cut off the lower 50% coverage as we did above. We can see that the coverage of the weighted-sum algorithm is superior to the greedy

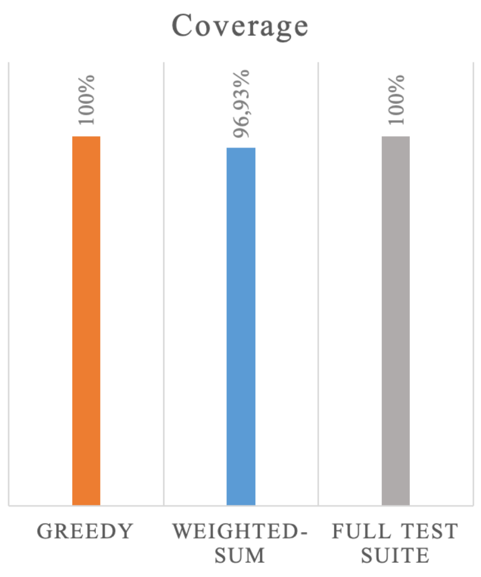

Figure 5.3: Coverage Comparison of the Two Minimization Algorithms

algorithm for most of the graph. However, the greedy algorithm actually reaches 100% coverage first at 72% runtime while the weighted-sum algorithm reaches 100% coverage at 83,% runtime. While this only tells us the worst case, it shows that running the weighted-sum algorithm in an adequate manner towards line coverage provides bad results.

The diagram also shows that the greedy algorithm is considerably worse when it comes to inadequate performance because, for the most part, the greedy curve is below the weighted-sum algorithm's curve. If, for example, we assumed a limit of 5% coverage loss, we could get 65% runtime savings with the greedy algorithm, while we could achieve 87% with the weighted-sum algorithm. While this difference might not seem that large, the resulting tests suite with the weighted-sum algorithm is 2.7 times faster compared to the greedy algorithm.

Also if we look at the 15% target of the weighted-sum algorithm which is indicated by a red vertical line in the diagram, we can see that the weighted-sum algorithm is almost 30% above the standard greedy algorithm.

- **in regards to mutation coverage?** While the greedy algorithm has a slight advantage in the code coverage of the minimized test suites, the resulting mutation coverage is significantly worse. In Figure 5.5, we show the mutation score retention of the different test suites. The weighted-sum algorithm loses 4.68% mutation

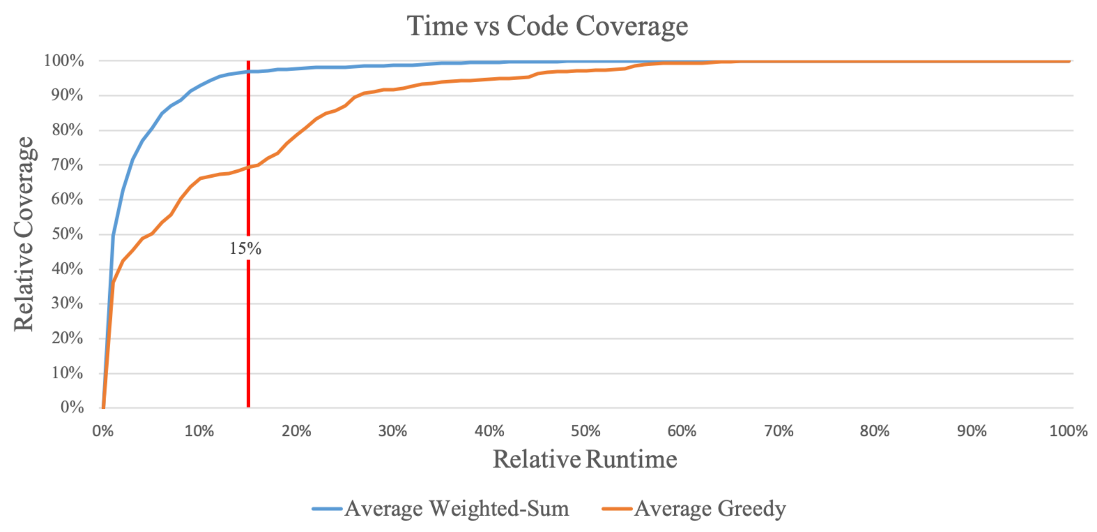

Figure 5.4: Average Coverage over time for the Two Minimization Algorithms

score. The greedy algorithm, on the other hand, loses over 22.22% mutation score which means that 1/5 of the faults that were previously found would slip through after the minimization, almost five times as many as with the weighted-sum algorithm.

This is a very important aspect if test suite minimization is to be used in practice. Loosing 4.68% means that less than every 20th fault that could be detected before goes unnoticed by the test suite after test suite minimization. However, 1/5 of the faults that were found before slipping through might very well turn off a lot of people who might otherwise have a need for test suite minimization.

This difference between the two algorithms is even more pronounced when we look at Figure 5.6, where we mapped the average mutation score of all projects to the runtime for both algorithms. Because the greedy algorithm's only criterion is code coverage, additional killed mutants are not considered when selecting tests. This results in rather bad behavior in terms of its mutation score which is visible in the first half of the chart where there is an almost 40% difference between the relative mutation coverage of the two algorithms.

- **in regards to test execution time?** In Figure 5.7, we display the resulting speedup of the minimized test suite as compared to the full test suites with the two algorithms as well as for the full test suite. We decided to show the speedup instead fo the percentage reduction because we think it pronounces the most

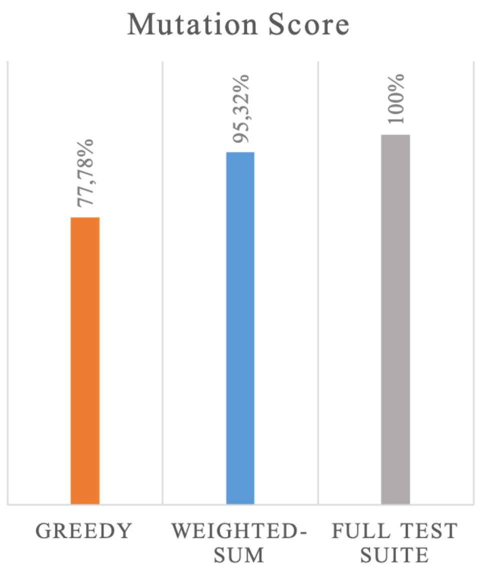

Figure 5.5: Mutation Score Comparison of the Two Minimization Algorithms

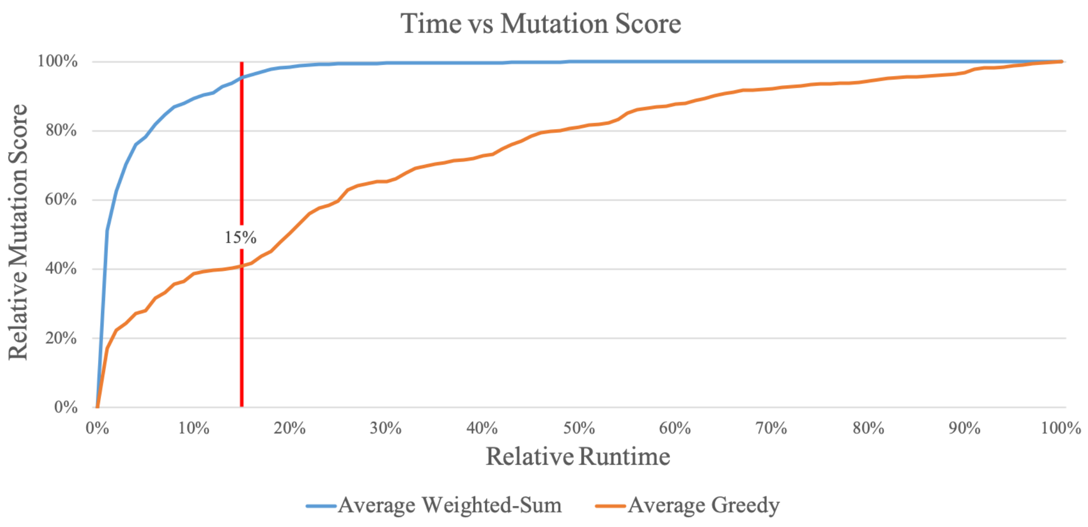

Figure 5.6: Average Mutation Score over Time for the Two Minimization Algorithms

relevant result here. How much faster is the resulting test suite than the full test suite.

Combined with the previous results, this metric shows one of the main benefits of the weighted-sum algorithm over the greedy algorithm. While the greedy algorithm more than halves the runtime, the weighted-sum algorithm leaves less than 1/6 of the original duration. This difference is not immediately visible when we only look at the time reduction percentages of 54.75% for the greedy algorithm and 85% for the weighted-sum algorithm.

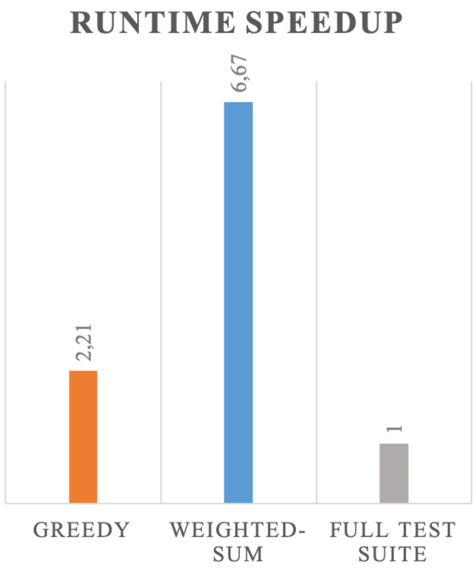

Figure 5.7: Speedup Comparison for the Two Minimization Algorithms

- **in regards to the files covered per test?** As a basis of comparison, we used the average number of files that is covered when running the full test suite. That is the reference value of 100% in Figure 5.8. Note that in this diagram, a lower number equates to fewer classes covered per test, which means lower percentages are better.

  Our first finding is that introducing locality as a factor had the intended result, as with the weighted-sum algorithm, we could reduce the number of files that a test runs through by around 33%. This suggests that we achieved the intended result of making a test suite conform to a test pyramid by preferably removing system and integration tests.

  Interestingly, the greedy algorithm has the opposite effect. It increases the average number of classes that a test from the test suite runs through. It does not shape the minimized test suite in the testing pyramid but rather into what is often referred to as an ice cream cone where system tests are preferred. The reason for this is how the greedy algorithm selects its tests. By purely prioritizing the tests with the most coverage, system tests are the preferable kind of tests since they run through the whole system and thus often cover lots of code with only one test. While this gives a good result for the remaining number of tests, which is often stated as the target of test suite minimization in literature, the overall quality of the test suite suffers significantly from this way of minimization.

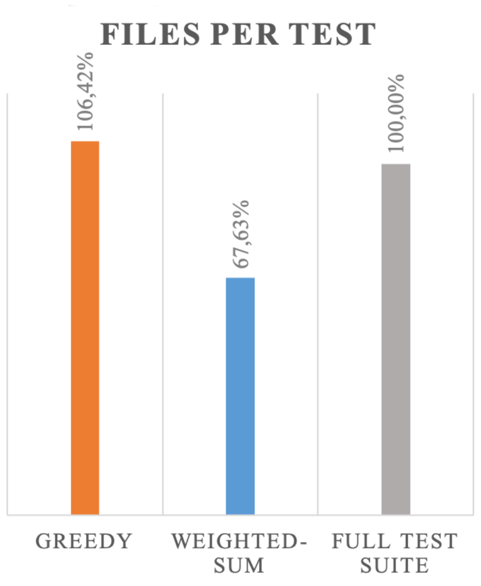

Figure 5.8: Files per Test Comparison of the Two Minimization Algorithms

Answering RQ4 – Do results from large scale, industry projects differ from those of open source projects?

We have split the results of the experiments we conducted for this question into two charts for better visibility. In Figure 5.9, we display the results for the greedy algorithm and in Figure 5.10, we have plotted the results for the weighted-sum algorithm. In each line-chart, we show the average of the open source projects and separate lines for Teamscale and the Siemens project.

Besides the number of SLOC, another important difference between the open source and the closed source projects is the duration of the test suites. While the longest running test suite of the open source projects takes roughly three and a half minutes, the test suite of the Siemens projects takes a bit over 85 hours to run. For projects like that, test suite minimization is way more relevant since a test suite that takes under five minutes to run won't pose a problem for many people while one that runs over three days can be a hindrance to development.

In Figure 5.9, we see that for the greedy algorithm, the average result of the open source projects lies between the results of the Siemens project and Teamscale. Test suite minimization works very well with the Siemens project but not well at all for Teamscale. When looking at the Teamscale curve, one can see that it reaches 100% coverage at around 90%. Full coverage is reached with over 99% of the original runtime, making it virtually useless in terms of reducing the duration of the minimized test suite. For the Siemens project, the greedy algorithm reduces the test suite's duration by around 55%, which puts it at the average for the greedy algorithm as displayed in Figure 5.9.

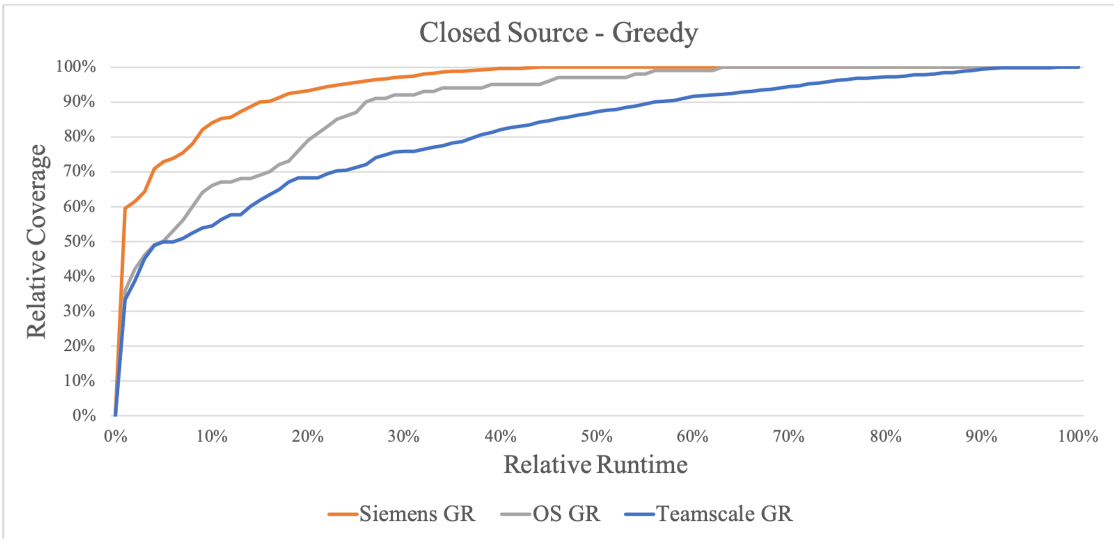

Figure 5.9: Coverage over Time of the Greedy Algorithm for the Closed Source Projects

For the weighted-sum algorithm in Figure 5.10, the Siemens project shows very similar results to the open source projects. Teamscale, again, behaves significantly worse than the open source projects. At the goal of 85% reduction in runtime, we have 97.4% of the full test suite's line coverage with for the Siemens project and 82.55% for Teamscale. With Teamscale, the reduction target of 85% reduction in runtime is too high. A target of around 50% which would result in around 95% coverage would be more sensible here.

From the bad results for Teamscale for both algorithms, we conclude that Teamscale is not well suited for minimization, meaning it has considerably fewer redundant tests in its test suite. That the greedy algorithm barely removes any tests means that almost every test adds additional coverage. Since the total coverage is more distributed over the test suite, minimization does not work to the extent that it does with the other projects.

For the Siemens project, which has the problem of an overly large test suite, test suite minimization works just as well as it did on the open source projects which shows us that test suite minimization is viable for large scale, closed source industry projects.

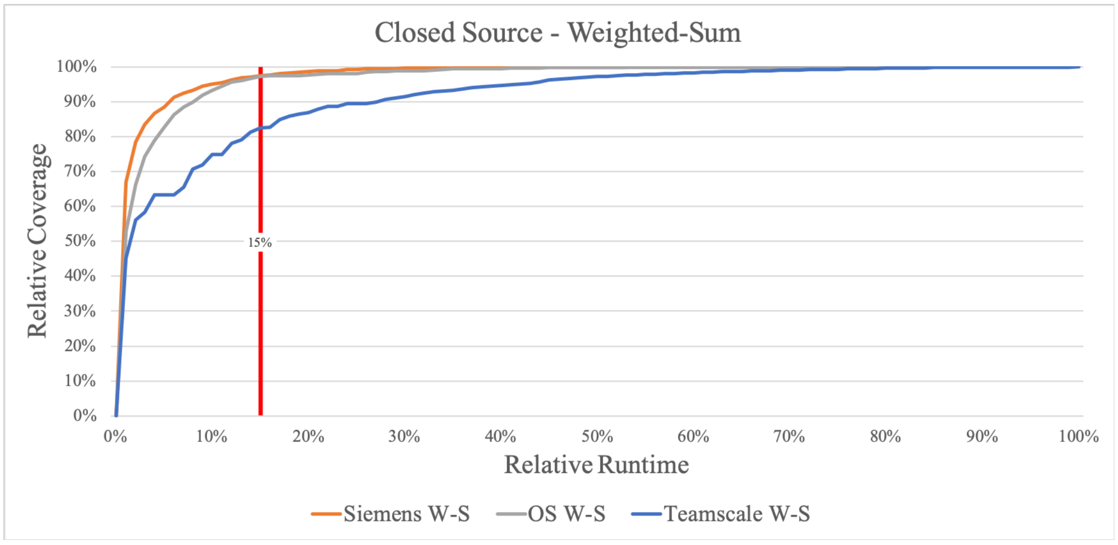

Figure 5.10: Coverage over Time of the Weighted-Sum Algorithm for the Closed Source Projects

Answering RQ5 – How well do the minimization algorithms scale with large projects?

With our final research question, we want to find out, whether the two test suite minimization algorithms scale well with project size. In Table 5.4, we have listed the runtime of the test suite minimization itself with both algorithms for each of the projects we analyzed. As expected, the weighted-sum algorithm is, for the most part, more expensive in its execution duration.

For most of the projects, the minimization runs for less than one minute. The only project which exceeds that is the Siemens project which is considerably larger than all the others. With the un-minimized test suite taking around 85 hours to run, a minimization duration of under four minutes is not an issue.

The most surprising result is that for JSoup the weighted-sum algorithm is faster than the greedy algorithm. One expensive part of the algorithm is the subtraction of the coverage of a newly selected test from the rest. However, the weighted-sum algorithm has additional steps that increase the runtime, like updating the scores and updating mutation scores after a test has been selected.

Table 5.4: Runtime Test Suite Minimization Algorithms in Seconds

	AC Coll	AC Lang	AC Math	AssertJ	Ebean	JOptSimple	JSoup	Spoon	Siemens	Teamscale
Greedy	11.09	3.08	15.64	22.74	2.48	0.087	0.587	8.32	19.09	7.11
Weighted-Sum	43.56	3.81	20.97	45.65	5.88	0.102	0.513	13.41	196.47	39.28

Another difference between the algorithms that affect the runtime is the number of selected tests. Since the operations all have to be performed once per test, an increased number of selected tests leads to a higher runtime. The greedy algorithm attempts to minimize the number of tests by selecting the tests that add the most coverage. Because the weighted-sum algorithm does not only aim for a reduction of the number of tests but also of runtime, for the most part, results in a minimized test suite with more tests remaining as can be seen in Table 5.5.

Table 5.5: Number of Remaining Tests

	AC Coll	AC Lang	AC Math	AssertJ	Ebean	JOptSimple	JSoup	Spoon	Siemens	Teamscale
Greedy	955	1635	1541	2318	812	99	238	431	2860	2829
Weighted-Sum	3877	1802	4604	3337	1725	149	328	605	4353	2507

5.5 Threats to Validity

In this section, we discuss possible threats to the validity of our study and what measures we took to minimize said threats.

External Threats The most relevant threat is that the results we obtained may not be universally applicable, that is, they may not be representative beyond the projects we analyzed. To reduce the risk of this, we chose projects from different developers and of different sizes. We also considered open source and closed source projects. Our study subjects were mostly arbitrarily chosen, so there is no inherent connection between them. The only restriction which limits the choice of our study subjects is that they had to be able to run Pitest.

For the closed source projects, the main issue is that our sample size was small. With only two projects, these results might not be universal. Though, the fact that the results of the closed source project that had the problem of an oversized test suite were very similar to the open source ones suggests that the behavior could be similar in general. However, projects that do not have that many redundancies appear to behave differently, both for open and closed source projects.

Internal Threats Another potential issue is the implementation of mutation testing, which we used for our research. While Pitest is one of the most comprehensive mutation testing tools, it cannot get around the issues that inherently plague mutation testing. The first issue is the selection of mutation operators. There is only a limited amount of mutation operators, and they typically represent certain kinds of faults. Even though mutation testing has been proven to be a reliable representation of real world faults, mutation operators are sill more predictable than real faults.

The second issue we see with mutation testing is the occurrence of infinite loops and equivalent mutants. While they are not an essential issue, they still make the results of mutation testing less accurate than it would ideally be. To get around the problem of false positives for infinite loops, we gave the tests as much time as we could afford. This reduces the number of Pitest timeout reports that are not caused by infinite loops. Equivalent are mutants that cannot be found. While they are a problem of mutation testing, the relative mutation score of the minimized test suites is not affected by them. This makes the overall mutation score less accurate, however, it is not a problem when looking at the relative mutation scores of the reduced test suites.

6 Conclusion

We proposed a multi criteria based weighted-sum test suite minimization algorithm that aims to balance the trade-offs between a high reduction in runtime and maintaining as much of the original fault detection capability, coverage and quality of the test suite as possible.

We evaluated this algorithm based on eight medium sized open source projects and two large scale closed source projects. We found that we can achieve an average line coverage of around 97% and mutation coverage of around 95% while reducing the runtime by 85%. When comparing the weighted-sum algorithm to a code coverage based greedy algorithm, it is superior in almost every way. While the relative coverage of the greedy algorithm is slightly higher, in terms of retained mutation coverage, runtime improvements and the average files touched per test, the weighted-sum algorithm outperforms the greedy algorithm by a lot. The speedup of the weighted-sum algorithm is around three times higher at a factor of around 6.7 versus 2.2 for the greedy algorithm. The loss in fault detection capability is less than 1/4 of the greedy algorithm's at 4.7% versus 22.2%. For the files touched per test, the greedy algorithm has a negative effect, increasing the amount by around 6.4% compared to the full test suite while the weighted-sum algorithm decreases the number by roughly 32.4%.

Our second central goal besides introducing a new algorithm and evaluating its performance was to find out how well test suite minimization performs with large scale closed-source projects. All of our open source projects have test suite runtimes where test suite minimization is not necessary. The closed-source test suites, on the other hand, have up to 85 hours of runtime. Our evaluation showed that both algorithms work similarly well with large scale, closed source projects as they do with medium sized open source ones. However, we also found that the performance highly depends on the degree of redundance in a test suite. Depending on how much overlapping coverage and similar tests there are, test suite minimization can deliver better or worse results.

Overall, we found that the proposed weighted-sum algorithm performs very well when compared to the well established greedy heuristic, outperforming it in almost every way. For the closed source projects, our results show that test suite minimization can work well, but, same as with open source projects, it depends on the degree of redundancy in a test suite. However, for the closed source projects, our sample size was small, which means that more research is necessary to confirm these results with a larger sample size. The main difficulty here is that it is considerably harder to investigate closed source projects since they are not as readily available and can be harder to set up since they often have more complicated builds. Open source projects are often easy to set up and run because they rely on contributions from outside developers.

7 Future Work

During the making of this thesis, we found three topics that seem particularly interesting for future research based on this study.

The first possible option to go from here is to perform a similar but expanded analysis of additional large scale projects. One of the limiting factors of our study was getting access to suitable, large scale projects. Since these are typically closed source industry projects, they are a lot harder to obtain than open source projects. Because large projects with long running test suites are the real target of test suite minimization, they are particularly relevant for the use in real world environments. Finding these kinds of projects is challenging but provides excellent insight into the capabilities of test suite minimization in practice.

The second topic is to perform mutation testing on larger projects. Due to time and performance constraints, we only used mutation testing for the small to medium sized projects. Since the primary purpose of test suites is to detect faults, not to cover as much code of a project as possible, mutation testing is a more meaningful metric to use for minimization, compared to line coverage. On the other hand, mutation testing increases the test execution duration by at least a factor of 100 from our experience, making it difficult for big projects. Depending on whether a test suite supports parallel execution, the speed of mutation testing could be increased a lot with more powerful hardware. Another way of reducing the cost of mutation testing would be to reduce the number of mutants and restrict the selection of mutants to the most relevant ones. For this, it should be analyzed how many mutants are needed to keep the significance of the mutation report at a high enough level and which mutants are more or less relevant.

The third and final topic to research from this thesis is adjusting the weighted-sum algorithm to the project that is currently analyzed. This means adjusting the constant factors in Equation 4.2 so they deliver the best results for the current project. Additionally, the goal should be individualized per project since the requirement is not always a reduction of 85% and, as we saw, the degree of redundancy and thus the performance of test suite minimization varies between projects. When evaluating the projects with the weighted-sum algorithm, we already noticed that the constant factors have a significant impact on the outcome and can behave very differently with the different projects. From these observations, we think that the results of the weighted-sum algorithm could be improved considerably by adjusting the constant factors and the reduction target.

Bibliography

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. "Is mutation an appropriate tool for testing experiments?" In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 402–411.

[AKM08] A. Alali, H. Kagdi, and J. I. Maletic. "What's a typical commit? a characterization of open source software repositories." In: *2008 16th IEEE International Conference on Program Comprehension*. IEEE. 2008, pp. 182–191.

[Bir+09] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. "The promises and perils of mining git." In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE. 2009, pp. 1–10.

[BMK04] J. Black, E. Melachrinoudis, and D. Kaeli. "Bi-criteria models for all-uses test suite reduction." In: *Proceedings. 26th International Conference on Software Engineering*. IEEE. 2004, pp. 106–115.

[Bri+14] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. "How do centralized and distributed version control systems impact software changes?" In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 322–333.

[CL96] T. Y. Chen and M. F. Lau. "Dividing strategies for the optimization of a test suite." In: *Information Processing Letters* 60.3 (1996), pp. 135–141.

[Coh10] M. Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

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

[DT96] M. Daran and P. Thévenod-Fosse. "Software error analysis: A real case study involving real faults and mutations." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 21. 3. ACM. 1996, pp. 158–171.

[GJ02] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 29. wh freeman New York, 2002.

[HGS93] M. J. Harrold, R. Gupta, and M. L. Soffa. "A methodology for controlling the size of a test suite." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.3 (1993), pp. 270–285.

[HL92] J. R. Horgan and S. London. "A data flow coverage testing tool for C." In: *[1992] Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*. IEEE. 1992, pp. 2–10.

[HO09] H.-Y. Hsu and A. Orso. "MINTS: A general framework and tool for supporting test-suite minimization." In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 419–429.

[JG05] D. Jeffrey and N. Gupta. "Test suite reduction with selective redundancy." In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE. 2005, pp. 549–558.

[JH10] Y. Jia and M. Harman. "An analysis and survey of the development of mutation testing." In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678.

[Jus+14] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. "Are mutants a valid substitute for real faults in software testing?" In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 654–665.

[Phi+19] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagappan. "FastLane: test minimization for rapidly deployed large-scale online services." In: *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press. 2019, pp. 408–418.

[Rot+99] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. "Test case prioritization: An empirical study." In: *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE. 1999, pp. 179–188.

[Shi+14] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. "Balancing trade-offs in test-suite reduction." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 246–256.

[ST02] A. Srivastava and J. Thiagarajan. "Effectively prioritizing tests in development environment." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 27. 4. ACM. 2002, pp. 97–106.

[TG06] S. Tallam and N. Gupta. "A concept analysis inspired greedy algorithm for test suite minimization." In: *ACM SIGSOFT Software Engineering Notes* 31.1 (2006), pp. 35–42.

[Wei+17] Z. Wei, W. Xiaoxue, Y. Xibing, C. Shichao, L. Wenxin, and L. Jun. "Test suite minimization with mutation testing-based many-objective evolutionary optimization." In: *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE. 2017, pp. 30–36.

[Won+98] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of test set minimization on fault detection effectiveness." In: *Software: Practice and Experience* 28.4 (1998), pp. 347–369.

[YH07] S. Yoo and M. Harman. "Pareto efficient multi-objective test case selection." In: *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM. 2007, pp. 140–150.