An Evaluation of Test Suite Minimization Techniques *

Raphael Noemmer[1] and Roman Haas[2]

[1] Technical University of Munich, Munich, Germany,
[2] CQSE GmbH, Munich, Germany

Summary. As a software project evolves over time, the associated test suite usually grows with it. If test suites are not carefully maintained, this can easily result in massive test execution duration, reducing the benefits of regression testing because faults are found later in development or even after release. Test suite minimization aims to combat long running test suites by removing redundant test cases. Previous work mainly evaluates test suite minimization techniques based on comparably small projects, which are less practically relevant. In this paper, we compare four test suite minimization techniques by applying them to several open source software projects and evaluate the results. We find that the size and execution time of all the test suites can be reduced by over 70% on average. However, there is a substantial loss in fault detection capability of, on average, around 12.5%, restricting the applicability of this form of test suite minimization.

1 Introduction

The size of test suites tends to grow over time, which leads to an increasing amount of time used for each test run [5]. Test suites of large projects may run for days or even weeks. This is problematic for continuous integration where tests are ideally executed after every commit to give feedback to the developers as early as possible. The delay of feedback makes it harder to fix failures found by the tests because the changes might have been made several days ago, requiring the developer to refamiliarize himself with the changed code. Besides, more changes might have been made to the same code since the tests have started running. Additionally, there is a lot more changed code at once, making it harder to identify the fault that is the root cause of test failures. In the literature, three research areas, coping with long running test suites, can be found. The first approach is *Test case selection* where test cases to be executed are chosen depending on the changes made since the last test run. Because unit tests cover

* This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SOFIE, 01IS18012A". The responsibility for this article lies with the authors.

The final authenticated version is available online at `https://doi.org/10.1007/978-3-030-35510-4_4`.

specific areas of a system and, in general, changes over a limited time span are limited to parts of a system, it is usually not necessary to execute all test cases every time. The main difficulty of this approach is to identify, which test case runs through which code after changes have been made. To get this information precisely, the whole test suite would need to run again, rendering the test selection approach useless. Usually, heuristics, based on test coverage data from earlier test runs, are used to resolve this issue. The second option is *test case prioritization*. In contrast to the other approaches, it does not aim for run time reduction of test suites but instead for a faster fault detection. With this method, test cases are executed in order of their relevance for the changes made. This can be accomplished by executing fast test cases that cover changes first. So, with this approach, the whole test suite is still executed, but the tests that are executed first have the highest likelihood of finding faults. This allows for the developers to get quick feedback without any loss in the overall fault detection. The last approach, and the one we are using in this paper, is *test suite minimization*. Test suite minimization attempts to find redundant tests that have little to no impact on fault detection capabilities of a test suite and remove them permanently. There are several common ways of determining whether a test is redundant, that is, has a low likelihood of detecting faults which are not found by the remaining tests. For determining the redundancy of a test, one or multiple criteria can be used, for example, statement coverage, execution cost, mutation coverage, mc/dc coverage etc. The tests that satisfy the chosen criteria are then selected and the rest is removed, ideally leading to a permanent reduction in the runtime of a test suite.

Problem Statement. In many software projects, regression testing takes up large amounts of time which can slow down development. Test suite minimization can be used to reduce the time each test run takes by removing redundant test cases. However, test suite minimization is rarely used in practice. We identified two core reasons for this, the first of which is that it is not an easy task to perform, especially with complex builds. The second reason is that removing test cases always carries the risk of reducing the effectiveness of the test suite. Due to the nature of test suite minimization, tests are usually removed permanently, which is a risk that has to be taken compared to test case selection or prioritization.

Contribution. In this paper, we evaluate different algorithms for test suite minimization with seven open source projects and make the following contributions:

- *Random Mutation Testing for Evaluation*
  A lot of the papers on test suite reduction utilize manually introduced faults in their underlying research for the evaluation of the loss in fault detection capability. By using mutation testing instead, we can generate a higher number of faults, equally spread over the whole project, which allows us to assess fault detection capability at a larger scale.
- *Real Open Source Projects*
  For the evaluation of test suite minimization, small test projects that were published for research purpose only, are used [7, 16]. We chose to use actively

maintained open source projects which are developed by a variety of organizations to investigate the applicability of test suite minimization techniques in practice.
- *Time Measurements of Minimized Test Suites*
  Test suite minimization approaches are often evaluated, based on the number of tests that could be removed from the original test suite. Although, this is a relevant statistic which we report as well, the biggest benefit of test suite minimization in practice is to save execution time. Since execution times for tests can vary a lot in practice, the time savings need to be considered separately from the number of tests. To find out whether the practical benefits of test suite reduction are proportional to the number of removed tests, we analyze execution times of test suite before and after minimization is applied.

2 Fundamentals

In the following, we describe two basic concepts in the field software testing that we need for our study. The first is a formal description of test suite minimization, a technique that aims to reduce the runtime of test suites by removing redundant tests, the second concept is mutation testing which can be used evaluating the fault detection capability of a test suite.

2.1 Test Suite Minimization

In their 2002 paper on test suite reduction, Rothermel et al. define the minimization problem as follows: Given a test suite T that contains test cases $t_1, t_2 \dots t_n$ and requirements $r_1, r_2 \dots r_n$ which can be satisfied by the test cases in T, find a minimal subset of T that satisfies the same requirements as T itself [13]. When all r_i need to be satisfied, the technique is called *adequate*. An *inadequate* approach means that some of the requirements may be left unsatisfied. The r_i can be different functional or structural requirements, for example line coverage or mutation coverage.

Selecting a minimal set of tests that satisfies the requirements means finding a minimal hitting subset of T over the r_i, which is an NP-complete problem. Due to this difficulty, heuristics are a compelling option. There are many different heuristics that have been used and analyzed for the purpose of test suite minimization [19] some of which we investigate in Section 3.

2.2 Mutation Testing

Mutation testing is a method of assessing a test suite where mutations, a set of simple changes, supposed to represent typical faults, are introduced into a system. The test suite is then rated based on how many of the introduced faults are detected [9]. These faults are called mutants, and finding one of them is called 'killing a mutant'. The score of a test suite is calculated as follows:

$$MutationScore = \frac{numberOfFoundMutants}{numberOfIntroducedMutants}$$

It has been shown that mutants are similar enough to real faults to allow the mutation score to give a good indication of the real-world fault detection capability of a test suite [1]. There are, however, some inherent flaws of mutation testing. It is, for example, possible for mutants to cancel each other out which leads to undetectable mutants. Mutations can also cause infinite loops which makes it hard to tell whether a test takes a long time or is stuck in an infinite loop.

3 Related Work

The test suite minimization algorithms most commonly found in research are variations of the greedy algorithm [3,10] which has been shown to be an effective heuristic for the minimal hitting set problem [11]. Two well-known extensions of the greedy algorithm are the GE (Greedy Essential) and GRE (Greedy Redundant Essential) algorithms. Chen and Lau compared these two greedy variants to another heuristic called HGS (Harrold-Gupta-Soffa) [3, 6]. Their results suggested that, though there are differences, neither technique is better than the others in all cases. Tallam and Gupta invented another version of the greedy heuristic, the delayed greedy algorithm [15]. It avoids selecting test cases that may later be rendered redundant by other selected test cases. This can happen when large tests are selected early on but then the subsequently selected tests, together, cover the same requirements. To avoid it, they removed the test cases, whose coverage is either a subset of another test or is completely covered by multiple other tests. After the tests are removed, the normal greedy heuristic is applied.

Offut et al. used mutation testing but instead of evaluating the quality of the minimized test suite on the basis of the resulting mutation score [10], they used the score as a criterion for minimization, that is, the mutants are the requirements that need to be satisfied by the algorithm. They compared statement coverage to mutation score as testing requirements for minimization. Their mutation score was based on manually created mutants. In our study, we used automated mutation testing which allows for about two orders of magnitude more mutants. The automation also allows us to use larger projects.

There are also approaches that use more than one objective for test suite minimization. Selective redundancy is the approach used by Jeffrey and Gupta [7,8] in their multi objective approach. Selective redundancy means that, if a test is marked redundant by the first set of testing requirements, it is not removed until it is also redundant with respect to the second set of requirements. Only if a test is redundant for both sets of requirements, it is omitted. Gupta et al. used branch-coverage and all-uses coverage as their criteria. Their results showed less omitted tests but also an improvement to fault detection capability compared to the HGS heuristic with only one requirement.

Wei et al. also utilized mutation testing, but instead of evaluating their results with the mutation score, they used it as a goal for several different many-objective evolutionary algorithms [16]. These algorithms can be used to optimize many-objective problems with four or more conflicthing criterions. While this provides a good approach for selecting tests, the resulting mutation score is not comparable since tests are selected based on their mutation killing capability. They also employed smaller test suites compared to our study subjects.

Regarding the fault detection loss of test suites through minimization, there are conflicting results. While Rothermel et al. found significant losses in fault detection effectiveness of test suites through the use of minimization [12, 13], Zhang et al. found only small losses in fault detection when using test suite reduction on the same projects from the Software-artifact Infrastructure Repositository[2] [20]. Wong et al. also found that the impact of test suite minimization on a test suite's ability to detect faults is negligible [17, 18].

Shi et al. have taken a very similar approach to test suite reduction as we do in this paper [14]. They used mutation testing to evaluate 18 open source projects from GitHub. Their focus was on using the mutation score instead of line coverage as testing requirement for minimization. They also evaluated adequate and inadequate approaches and looked at different versions of the projects they investigated. They found that mutant-based minimization is better with regard to the fault detection loss while the statement-based approach delivers slightly better minimization results.

4 Implementation

Our goal in this paper is to investigate the applicability of test suite minimization techniques in practice. To achieve this, we implemented test suite minimization and a way to run mutation testing, as an indicator of fault detection capability, on the reduced test suites.

In Figure 1, we provide a structural overview of how we evaluate our chosen test suite minimization algorithms. First, we recorded testwise coverage for the tests of a project, using a modified version of JaCoCo[3]. We need this testwise coverage to apply the coverage-based minimization algorithms we have chosen. For our evaluation, we used two algorithms, a greedy algorithm, and the HGS heuristic. The goal of both of these algorithms is to select a subset of tests that covers the same as the original test suite but with different approaches. For both of them, we used statement coverage as the testing requirements for minimization. We used both, an adequate and an inadequate approach for each of the two algorithms. The adequate approach selects test cases until all lines are covered while, for the inadequate approach, new tests are selected until they no longer contribute at least five lines of additional coverage to the chosen subset of tests.

[2] https://sir.csc.ncsu.edu/php/previewfiles.php
[3] https://github.com/cqse/teamscale-jacoco-agent

4.1 The Greedy Algorithm

The greedy algorithm selects test cases by iteratively choosing the test case with the most additional statement coverage. First, the test with the most overall coverage is selected, that is, the test case t_k that satisfies the most testing requirements r_i. The requirements $r_n \dots r_m$ covered by this first test are then removed from the coverage of all other test cases. This means that for each $t_j, j \neq k$, the operation $\{r_y \dots r_z\} \setminus \{r_n \dots r_m\}$ where $r_y \dots r_z$ are the requirements satisfied by t_j is performed. The result of the set-theoretic difference is the new set of requirements satisfied by t_j. With this, we optimize for additional coverage and ignore what has already been covered. New tests are selected according to this rule until no additional coverage can be achieved by selecting more tests. In case of the inadequate approach, the heuristics stops earlier, in our case when no more than five lines can be added by selecting an additional test case.

4.2 The HGS Algorithm

The HGS heuristic works by adding test cases based on their cardinality, starting with the tests that have the lowest cardinality. To determine the cardinality of test case t_j take all testing requirements $r_n \dots r_m$ covered by test case t_j. For $r_n \dots r_m$, check by how many test cases each r_i is covered. The requirement r_i with the lowest number of test cases covering it, is the cardinality of test case t_j. For the algorithm this means, we start with the lines that are only covered by one test case. This gives us a set of test cases. All these test cases need to be added since they are essential, that is, they are the only tests that cover some lines. We then proceed with requirements that are covered by two test cases and iteratively add the test case with the highest additional coverage. We add test cases one by one and go up in cardinality until all requirements are met.

4.3 Mutation Testing

After we have the full and minimized test suites, we use the mutation testing tool Pitest[4] on the resulting test suites to get an approximation of how well the fault detection capability is maintained after the minimization is applied. Pitest provides a list of mutators[5], most of which are active by default. Their goal is to emulate real faults as realistically as possible. The advantage of mutation testing is the number of faults we can introduce and the randomness of them. By using real-world projects instead of the fairly small projects from the Software-artifact Infrastructure Repository, which are often used in research on test suite reduction, we want to evaluate how well test suite minimization works in practice.

[4] http://pitest.org
[5] http://pitest.org/quickstart/mutators/

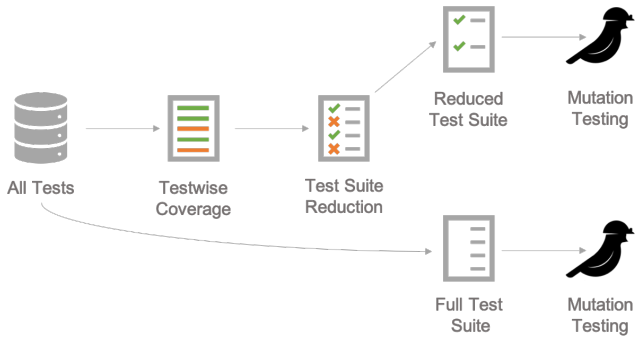

Fig. 1. Approach

5 Empirical Assessment

In this section, we want to examine the performance of statement coverage based test suite minimization. First, we describe our research questions followed by the subjects we chose to examine to answer said questions. We then explain, how we investigated each question and finally answer the questions according to the results we obtained.

5.1 Research Questions

The goal of our research is to find out how much our chosen test suite minimization techniques influence the fault revelation capability of test suites. Since the mutation score of a test suite is linked to its fault detection capability the resulting mutation score will give us an indication on how much the quality of a test suite suffers when minimization is used. With our experiments, we answer the following research questions.

RQ1 – How well is a test suite's capability to kill mutants preserved after test suite minimization is applied? Test suite minimization is only useful if a test suite preserves its fault detection capability through the process. We want to find out, if and how much worse a test suite gets at detecting faults, represented by mutants, when test suite minimization is used.

RQ2 – How much does the mutant killing capability vary between different test suite minimization techniques? We look at two different techniques to find out whether there is a considerable difference. We chose a simple greedy algorithm and compare it with a more complex algorithm, the HGS heuristic, to investigate how much of a difference using a more sophisticated algorithm, like the HGS, makes, compared to a simple greedy heuristic.

RQ3 – How does adequate test minimization perform compared to inadequate test minimization with a lower limit of five lines per test? Using adequate test suite minimization techniques means that even if it covers only one additional line, a test case has to be included in the set of tests. With an inadequate approach, we can fix a number of minimum required newly

covered lines and only include a test if it exceeds the lower limit. We expect this to reduce the number of tests considerably, improving the minimization result, but with five lines as the minimum number of newly covered lines, it might also have a substantial negative impact on the fault detection capability.

RQ4 – How big are the time savings and are they proportional to the number of omitted tests? The goal of test suite reduction is to save test execution time. To find out whether this goal is achieved, we analyze how much time is actually saved in a test run after applying test suite minimization. Since test cases can have vastly different runtime, we want to find out how much the runtime reduction is connected to the reduced number of tests and whether this behaves similarly in all study subjects.

5.2 Study Subjects

For our study, we examined seven systems, all of which are open source projects hosted on GitHub[6] and implemented in Java. We decided on Java because it is the only language supported by Pitest which is one of the most comprehensive and well maintained mutation testing tools we could find. The systems we chose vary in size from around 1k SLOC (Source Lines of Code) to 170k SLOC. We chose three projects from the Apache Software Foundation. They are well maintained, have a solid number of tests and are among our larger study subjects. With Ebean and Spoon, we included two other fairly large projects. To cover a wider set of different characteristics, we chose two smaller projects, JSoup, and Faux-pas, as well. All of our subject projects use Apache Maven as their build tool and use either JUnit version 4 or 5 for unit testing, which allows us to apply mutation testing (using Pitest) to their tests.

Table 1 shows a detailed overview of our subject projects. The LLOC (Logical Lines of Code) and coverage thereof are the numbers relevant for Pitest. For this value, only lines that can actually be executed are counted. For example, function headers and class declarations are not included in this metric. For mutation testing, only these lines are relevant since they are the ones that can potentially be mutated. For the SLOC metric, all lines that contain source code are counted, so only empty lines and comments are excluded. The number of mutants that are introduced is determined by Pitest according to the number of possible mutations.

Also note that the number of tests in Table 1 are the test cases that we executed. This number may be lower than the total number of tests in some instances because we removed or ignored tests that caused problems. These are, for the most part, tests that failed when running the test suite and some parametrized tests which cannot properly undergo test suite minimization.

5.3 Study Design

In this section, we describe, how we approached answering each of our research questions.

[6] https://github.com

Table 1. Study Subject Details.

Study Subjects	SLOC	SLOC Project	SLOC Test	LLOC Cov	#Tests	# Mutants
Commons Collection	62,897	28,708	34,189	46%	14,770	8,253
Commons Lang	75,408	27,825	47,583	95%	3,252	13,088
Commons Math	171,060	82,706	88,354	90%	4,825	37,674
Ebean	170,619	99,317	71,302	64%	2,598	25,056
Fauxpas	1,141	315	826	96%	81	50
JSoup	20,099	12,037	8,062	83%	666	4,711
Spoon	112,614	60,619	51,995	83%	1,608	15,887

RQ1 – How well is a test suite's capability to kill mutants preserved after test suite minimization is applied? To answer this question, we ran mutation testing on the original test suite and on the test suite minimized by the greedy algorithm for all our study subjects. We compare the mutation scores and calculate the relative mutation score loss from the minimization. Besides the timeout factor and constant, we used the default settings of Pitest. These factors were increased to reduce the number of false positive timeouts, a timeout is reported despite no infinite loop present. This increases the runtime but also increases the accuracy of the results we get.

RQ2 – How much does the mutant killing capability vary between different test suite minimization techniques? For our second research question, we ran the greedy and the HGS algorithm on our study subjects. We used mutation testing to find out how well the different minimization algorithms work with our study subjects and whether there is a significant difference between the techniques.

RQ3 – How does adequate test minimization compare to inadequate test minimization with a lower limit of five lines per test? For this, we use the same set-up as for RQ2 and additionally use inadequate versions of our algorithms. We compare the different approaches using mutation testing and their respective reduction in test suite size.

RQ4 – How much time can be saved per test run and are the time savings proportional to the number of omitted tests? To analyze, how much time is saved per run, we measure the run time of each test suite. We investigate both our algorithms in their adequate and inadequate forms. To minimize the possibility of background tasks influencing our results, we take five measurements for each project and minimization algorithm. We report the average reduction in execution time for each project's test suite.

6 Results and Discussion

In this section, we present and discuss the results of our experiments.

RQ1 – How well is a test suite's capability to kill mutants preserved after test suite minimization is applied?

In Table 2, we display the number of killed mutants as well as the number of tests before and after minimization. Our results for the reduction are closely related to the results in the 2014 paper on test suite minimization by Shi et al. [14] who applied a similar approach. Most of the projects have a reduction in test suite size from roughly 60% to 75% with the median at 67%. The difference between our results and the results in the other paper can most likely be attributed to the difference in project selection. We consider a reduction of more than 50% in all projects very high and it was particularly impressive to us that even for the small projects, we got a reduction in test suite size of more than half.

The number that sticks out the most in terms of the test reduction is the 93% of the Apache Commons Collections library. On closer inspection we found that a lot of the test cases of that project are focused on very few classes which leads to an extreme effectiveness of the minimization as well as a low overall mutations score. A hint to this can be found in the comparably low LLOC coverage in Table 1 even though the number of tests is very high. In cases like this, we ignore that the developer might have a reason for having many tests for a small section of code. Even though this usually means that the minimization is very effective, we have no way of knowing whether that code is particularly important or complex and requires more testing.

The most important column of the table, however, is the relative MS (mutation score) loss. Due to the fact that it ranges from 3.5% to 21% in our study, the effectiveness appears to be dependent on the project in question and is not strongly correlated with the size reduction percentage. Our results also show that the Apache Foundation projects facilitate test suite minimization a lot better than the other projects we tested. They have similar reduction rates to the other projects but at a substantially lower loss in mutation score.

Overall, while the reduction figures are promising, the loss in mutation coverage for some of the projects is quite high. Potentially missing 21% of faults is unacceptable in a lot of cases. These results suggest that a stricter set of testing requirements for minimization instead of only statement coverage could make sense. This would likely limit the number of missed mutants but also reduce the effectiveness of the minimization. Our results also show that there is potential for test suite minimization in big software projects that are actively maintained.

Table 2. Comparison Full test suite and minimized Greedy.

	Full Test Suite			Minimized Test Suite				
Study Subjects	# Tests	# MK	MS	# Tests	# MK	MS	Reduct	Rel MS Loss
Commons Collection	14,770	3,459	42%	960	3,145	38%	93%	9.5%
Commons Lang	3,252	11,285	86%	1,638	10,873	83%	50%	3.5%
Commons Math	4,825	29,721	79%	1,574	27,893	74%	67%	6,3%
Ebean	2,598	10,971	44%	811	9,565	38%	69%	21%
Fauxpas	81	47	94%	23	39	78%	72%	17%
JSoup	666	3,167	67%	240	2,660	56%	64%	16%
Spoon	1,608	11,229	71%	482	9,683	61%	70%	14%

RQ2 – How much does the mutant killing capability vary between different test suite minimization techniques?

In Table 3, we have listed the number of tests after minimization as well as the number of killed mutants for both algorithms. We also display the relative difference in the number of selected tests and mutant killing capability. Our main finding here is that the differences between the greedy and the HGS algorithm are very minute in terms of the number of retained tests as well their mutant killing capability.

We observe that the HGS algorithm retains slightly fewer tests than the greedy algorithm but the difference is at most around 3% and with a p value $\gg$ 0.05, the HGS algorithm is not significantly better than the greedy algorithm. In terms of their mutation score, neither algorithm is superior as they are very close for all study subjects and neither of the two consistently outperforms the other.

These results confirm the results that Shi et al. found [14] which indicate that the differences between the simple greedy algorithm and more sophisticated algorithms is minute. Though we have only tried two algorithms, we observe that the more expensive HGS heuristic does not result in a palpable benefit for any of our study subjects. Though, because of the nature of test suite minimization, the algorithm is only applied rarely, so a more time consuming, but slightly more effective algorithm might still be worth it.

Table 3. Comparison Greedy HGS.

	Greedy		HGS			
Study Subjects	# Tests	# MK	# Tests	# MK	Rel Test Diff	Rel MS Diff
Commons Collections	960	3,145	943	3,116	1.77%	0.92%
Commons Lang	1,638	10,873	1,632	10,883	0.37%	0.092%
Commons Math	1,574	27,893	1,544	27,574	1.91%	1.14%
Ebean	811	9,565	799	9,450	1.48%	1.20%
Fauxpas	23	39	23	41	0.0%	4.88%
JSoup	240	2,660	233	2,656	2.92%	0.15%
Spoon	482	9,683	477	10,123	1.04%	4.35%

RQ3 – How does adequate test minimization compare to inadequate test minimization with a lower limit of five lines per test?

We display our results for this question in Table 4. In the table, we can see that the benefits of using the inadequate technique are quite substantial. Compared to the adequate variants, the number of remaining tests is more than halfed for most of our study subjects. Of course, the overall impact is significantly lower with an absolute average decrease in the number of tests of 86.7% for the inadequate greedy algorithm compared to 69.2% for the adequate greedy algorithm. The HGS algorithm behaves very similar.

However, there is also a substantial drop in mutation score across most of our study subjects. For most of our projects, the drop in fault detection capability compared to the adequate version is considerably larger than the drop from the

full test suite to the adequately minimized version. Compared to the adequate version, the inadequate minimization is not worthwile due to the lower absolute gain and the higher loss in fault detection capability.

There are different versions of inadequacy, for example we could also limit the overall coverage we want to achieve instead of introducing a lower limit per test.

Table 4. Comparison Adequate Inadequate

	Greedy						Hgs					
	Adequate		Inadequate		Difference		Adequate		Inadequate		Difference	
Study Subjects	# Tests	# MK	# Tests	# MK	Tests	MS	# Tests	MS	# Tests	# MK	Tests	MS
Commons Collections	960	3,145	401	2,522	58%	19.8%	943	3,116	434	2,598	54%	16.6%
Commons Lang	1,638	10,873	662	9,201	60%	15.4%	1,632	10,883	691	9,280	58%	14.7%
Commons Maths	1,574	27,893	725	26,053	54%	6.6%	1,544	27,574	793	26,089	49%	5.4%
Ebean	811	9,565	363	8,182	55%	14.5%	799	9,450	377	8,424	53%	10.9%
Fauxpas	23	39	6	27	74%	30.7%	23	41	6	28	74%	31.7%
JSoup	240	2,660	127	2,394	47%	10.0%	233	2,656	131	2,436	44%	8.3%
Spoon	482	9,683	233	9,447	52%	2.4%	477	10,123	239	9,563	50%	5.5%

RQ4 – How much time can be saved per test run and are the time savings proportional to the number of omitted tests?

In Figure 2 we give an overview of the time savings of the different algorithms applied to all of our study subjects. The y-axis shows the time savings in percent of the runtime of the full test suite. First, we can observe that the results vary a lot between the different projects. The time savings range from 4.5% to 68.6%. However, for most of our study subjects, the savings of all minimization techniques exceed 35%, making the benefits of test suite minimization quite attractive.

A, to us, surprising result is that the project with the smallest time savings, the Apache Commons Collections library, also has the highest relative reduction in its test suite size. This suggests that execution time of the test suites is not equally distributed. The removal of few, long running tests has more impact than omitting as many tests as possible. A good indicator for this is the difference between the inadequate versions of the algorithms for the Apache Commons Collections library and the adequate versions. The difference in the number of selected tests is rather small but the savings increase a lot more than they did with the first ~93% of removed tests.

Regarding the difference between the greedy and HGS algorithms, we can, once again, not determine a consistently superior algorithm. However, the inadequate versions of both algorithms show clear improvement over their adequate counterparts reaching from 4.4% to 24.3%.

By including the execution time of the individual test cases in the minimization, the variation in the effectiveness of test suite minimization could most likely be reduced considerably.

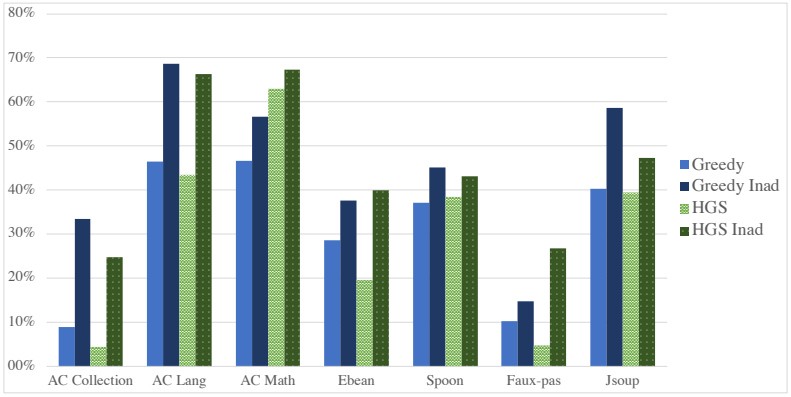

Fig. 2. Time Savings of Test Suite Minimization

7 Threats to Validity

In this section, to understand the limitations of our evaluation of test suite minimization, we discuss some possible threats that could affect the validity of our results.

Internal Threats The first possible criticism of our method is the use of mutation testing as a replacement for real faults. Though, it has been shown that the ability of a test suite to kill mutants is highly correlated with its ability to detect real faults [1,2,4], mutants are not the same thing as real faults. Another problem of mutation testing are equivalent mutants which cannot be detected and endless loops which can be caused by mutants. However, since our results involve mainly comparisons of the loss of mutation score from test suite minimization, the overall mutation score is not critical for the validity of our results.

Another issue of mutation testing is that timeouts can vary between test runs. Since mutation testing can cause infinite loops, there is a timeout value necessary to keep the mutation testing going when an infinite loop has been created. However, this can also happen by accident if a test runs longer than it should for some reason. We found a variation in timeouts between runs caused by false positives in the determination of timeouts. However, the margin between our runs did not exceed 1% in our results.

External Threats There is no guarantee that these results are representative beyond the scope of our study subjects. Even though, we used actively developed open source projects from different developers, commercial, closed source projects and other open source projects with very different characteristics may behave different with regard to test suite minimization. The projects we chose were limited in size due to the cost of mutation testing being fairly high and it's compatibility, especially with complicated builds being fairly low. Furthermore, we investigated only a small set of proposed minimization techniques to evaluate applicability of test suite minimization techniques in practice. We have

implemented two common test suite minimization algorithms but there are a lot more, also incorporating different testing requirements instead of only using statement coverage.

8 Conclusion

We evaluated the benefits and drawbacks of two test suite minimization algorithms over a range of seven open source software projects. We used two different statement coverage-based algorithms, the basic greedy algorithm and the HGS algorithm. For both, we applied an adequate and an inadequate variant to all of our study subjects. To find out, how well the fault detection capability of a test suite is maintained after test suite minimization, we compared the results of mutation testing of the full test suites and the minimized ones. We found that with the algorithms we used, there is a considerable trade-off between the reduction in test suite size and the loss in fault detection capability. The number of test cases was reduced by at least 50% for all the study subjects; the average reduction of our adequate algorithms being around 69% of tests removed.

To get an insight into the practical benefits of test suite minimization, we measured the execution time of test suites of our study subjects before and after the test suites were minimized. We found that, even though there were substantial reductions in all projects, there is a huge range between the execution time reduction of the individual projects (5% to 69%). The reduction in number of tests appears to be a bad indicator for the reduction in execution time.

Overall, test suite minimization shows great potential in terms of test suite execution time reduction. However, the implementation we chose in this paper does not provide a great trade-off between runtime improvements and loss in fault detection.

9 Future Work

We found that using statement coverage as the only criterion for minimization, while producing great results in terms of test suite size, leads to a substantial loss in fault detection capability which, in practice, will not be acceptable in a lot of cases. That is why we plan on investigating multiple objective-based algorithms, which could improve the outcome of test suite minimization in terms of the maintained fault detection capability. Another factor in a multi objective approach could be execution time, which shows substantial reductions in our experiments but could most likely be improved by introducing it as a criterion to be optimized for.

Another interesting possibility, which we want to pursue, is be to increase the variety of study subjects by shifting the focus from only open source projects to include closed source projects as well. Covering as many different ways of software development as possible can increase the viability of test suite minimization. So far it is rarely used in practice. Proving that commercial projects

can gain from test suite minimization could benefit its propagation from the domain of research into widespread use.

Future research could also evaluate test suite minimization techniques on the basis of real faults from the past according to how many of them are found. This would deliver even more relevant results than using mutation testing but finding and extracting data of sufficient volume for this kind of evaluation is much more difficult and laborious than using generated mutants.

References

1. James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.
2. James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.
3. Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.
4. Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.
5. Mark Harman. Making the case for morto: Multi objective regression test optimization. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 111–114. IEEE, 2011.
6. M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.
7. Dennis Jeffrey and Neelam Gupta. Test suite reduction with selective redundancy. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 549–558. IEEE, 2005.
8. Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering*, 33(2):108–123, 2007.
9. Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.
10. Jie Pan and Loudon Tech Center. Procedures for reducing the size of coverage-based test sets. In *Proceedings of International Conference on Testing Computer Software*, pages 111–123. Citeseer, 1995.
11. Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.
12. Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43. IEEE, 1998.
13. Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

14. August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–256. ACM, 2014.
15. Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2006.
16. Zheng Wei, Wu Xiaoxue, Yang Xibing, Cao Shichao, Liu Wenxin, and Lin Jun. Test suite minimization with mutation testing-based many-objective evolutionary optimization. In *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 30–36. IEEE, 2017.
17. W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.
18. W Eric Wong, Joseph Robert Horgan, Aditya P Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 522–528. IEEE, 1997.
19. Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.
20. Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An empirical study of junit test-suite reduction. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 170–179. IEEE, 2011.