Impact of Test Suite Coverage on Overfitting in Genetic Improvement of Software

Mingyi Lim, Giovani Guizzo, and Justyna Petke

Department of Computer Science, University College London, London, UK
{mingyi.lim.17, g.guizzo, j.petke}@ucl.ac.uk

Abstract. Genetic Improvement (GI) uses automated search to improve existing software. It can be used to improve runtime, energy consumption, fix bugs, and any other software property, provided that such property can be encoded into a fitness function. GI usually relies on testing to check whether the changes disrupt the intended functionality of the software, which makes test suites important artefacts for the overall success of GI. The objective of this work is to establish which characteristics of the test suites correlate with the effectiveness of GI. We hypothesise that different test suite properties may have different levels of correlation to the ratio between overfitting and non-overfitting patches generated by the GI algorithm. In order to test our hypothesis, we perform a set of experiments with automatically generated test suites using EvoSuite and 4 popular coverage criteria. We used these test suites as input to a GI process and collected the patches generated throughout such a process. We find that while test suite coverage has an impact on the ability of GI to produce correct patches, with branch coverage leading to least overfitting, the overfitting rate was still significant. We also compared automatically generated tests with manual, developer-written ones and found that while manual tests had lower coverage, the GI runs with manual tests led to less overfitting than in the case of automatically generated tests. Finally, we did not observe enough statistically significant correlations between the coverage metrics and overfitting ratios of patches, i.e., the coverage of test suites cannot be used as a linear predictor for the level of overfitting of the generated patches.

Keywords: Genetic Improvement · Search-Based Software Engineering · Overfitting

1 Introduction

Genetic Improvement uses automated search to improve existing software [19]. GI navigates the search space of mutated program variants in order to find one that improves the desired property. This technique has been successfully used to fix bugs [1, 14], add an additional feature [3, 20], improve runtime [12], energy [7], and reduce memory consumption [5, 24].

In the vast majority of GI work, each software variant is evaluated using a test suite, which is treated as a proxy for correctness. Although this assumption

cannot prove absence of bugs in the evolved software, it has been good enough to evolve useful patches that have been adopted into development [13]. On the other hand, the generated patches have been criticised for overfitting, i.e., passing the tests used during the GI search process, but not producing actual fixes that generalise to unseen scenarios [21]. This triggers the question about:

What feature should a given test suite have to aid the GI process in producing useful, correct patches?

The current state-of-the-art uses existing test suites, together with the given program, as input to the GI process. In the case of functional improvement, such as program repair, such a test suite would contain some failing tests that reveal a given property (such as a bug, or a feature not yet present in the software) that needs amending. The objective is to find a semantic change to the program so that the evolved software would pass all the provided test cases. In both functional and non-functional improvement branches of GI the test suite serves as an *oracle* for whether the evolved software has the desired semantics. Note, however, that in order to achieve improvement with respect to a functional property, its semantics needs to change, by definition. In contrast, in non-functional improvement we want to preserve the semantics of the original software, whilst improving a property of choice, such as running time. This has consequences in how test suites can be used in GI.

In order to improve a program using GI, we need a test suite that will faithfully capture the desired software behaviour. However, finding such test suites is a non-trivial task. Frequently in GI work the test suite needs to be usually manually improved before the GI process can begin [3]. If the test suite is too weak, GI will keep on deleting code (it can access) that contains uncovered functionality. Therefore, traditional software metrics, such as branch coverage, have been used to estimate how good a given test suite is before inputting them into the GI process [3]. This can be very costly, especially in the functional improvement case, where one has to devise test cases manually, as an automated approach treats the current implementation as the test case oracle [4]. However, in the non-functional improvement case, automated test case generation tools can be utilised.

Regardless of whether a given test suite has been generated manually or automatically, the question still remains: which features should it have that would lead to least overfitting when used within a GI process? Smith et al. [21] have made the first step by investigating the amount of overfitting by comparing manual vs. automated tests for the purpose of test-based automated program repair. Assiri and Bieman [2] sampled from existing test suites to show that statement-covering and random test suites tend to introduction of new faults in the automatically 'repaired' software. More recently, Yi et al. [25] tried to correlate various test metrics in existing test suites, with their ability to lead to a non-overfitting patch.

In this work we aim to measure the correlation between traditional test suite metrics and the given test suite's impact on overfitting in the GI process. In contrast to previous work, we focus on non-functional improvement (runtime, in

particular), and the Java program space. Hence, we use EvoSuite [11] to automatically generate test suites that achieve a given test suite coverage. We thus consider one test suite metric at a time, in order to provide a more systematic view of the metrics' impact. In this study we additionally measure the level of overfitting during the GI search process. We also re-run GI 20 times for each test suite – program pair (due to the non-deterministic nature of the GI search process we use). This way we can investigate a larger space of plausible software variants in order to establish a correlation between a given test suite coverage measure and the amount of overfitting. We also provide a replication package, available at: https://github.com/ssbseRENEsubmission/ssbseRENEsubmission.

2 Background

Genetic Improvement (GI) uses automated search to improve existing software [19]. In a typical scenario, the input to GI is a program and a test suite. GI then uses a set of mutation operators and a meta-heuristic, such as genetic programming, to evolve thousands of software variants, to be evaluated using a given fitness measure. In functional improvement fitness is based on the test suite alone, while in non-functional improvement an additional evaluation needs to be made against a property of choice, such as running time. The process runs until a given criterion is met. For the purpose of program repair, for instance, the search can be stopped when a program variant passes all the given tests, or only after a specified number of generations of the search algorithm of choice.

There have been several metrics presented in the literature to evaluate the strength of a test suite [8]. We focus on those that are implemented in the arguably most successful automated test case generation tool for Java, i.e., EvoSuite [11]. This tool implements a total of 8 coverage metrics. In this work we focus on 4, as the other ones are either not applicable to the benchmarks we use (for instance, our programs have a single method with no exceptions thrown, so there's no need to consider these) or is not fully supported for our purpose (output diversity measure cannot be automatically calculated for an existing test suite). Therefore, we consider line, branch, conditional branch, and weak mutation coverage.

Line and branch test suite coverage metrics are self-explanatory, i.e. the test suite aims to cover the largest number of lines or branches of the program, respectively. Conditional branch coverage aims to cover all branches with the right conditions, e.g., for an IF statement with an OR condition on 2 Boolean variables, 4 tests would have to be generated to cover all conditions, while branch coverage would only need 2 tests (for the *false* and any *true* evaluation of the *if* condition). Weak mutation refers to the case where the tests are run on multiple mutated source code variants. The coverage of the given test suite is equal to the proportion of mutants it manages to catch (i.e., 'kill') [17]. Strong mutation has also been proposed, which additionally requires that errors must propagate to the output of the program, imposing more stringent set of tests. However,

Offutt and Lee showed that weak mutation can nevertheless produce stronger test suites [16, 15].

3 Methodology

Our aim is to investigate the impact of various test suite coverage metrics on overfitting in the GI process. We also want to know if automatically generated tests, using such metrics, could yield to low overfitting rate and thus be used for the purpose of non-functional software improvement in GI.

In contrast to previous work, we do not analyse only the resulting patch of each GI run, but also all the valid patches generated during the GI search process. We analyse all of them because, even though they have been discarded during the search process (as not leading to better improvement than the final patch found), they are still valid and could still be used as feasible solutions. This gives us better statistical power during analysis, and thus more conclusive evidence from thousands of patches as opposed to a few hundreds.

Furthermore, we focus on Java programs, as they have not been investigated in this context before. We also use a non-functional property, namely, program's execution time as the goal for improvement.

3.1 Research Questions

In order to answer the question about which criteria should a given test suite satisfy in order to lead to least overfitting in the genetic improvement process, we generate test suites that achieve maximum coverage with respect to a given metric and compare them with respect to the amount of overfitting when input into the GI process. In particular, we pose the following research questions:

RQ1 (Validity) Given a particular test suite, can GI find a valid non-overfitting patch?

We want to know whether GI is able to find a non-empty, potentially runtime-improving patch[1] in the first place, given a particular input test suite. As done in related work [21], this question focuses on the final patch output by the GI process. This step validates whether GI can find a non-overfitting patch, regardless of how much overfitting might have occurred during the search process, and forms a baseline comparison with previous work (albeit in Java rather than C program space).

RQ2 (Overfitting) How does the overfitting rate vary with the input test suite during search?

[1] We use the word 'potentially' here, as although the patch might improve upon our training and test set, it does not mean the runtime improvement will generalise to all possible usages of software. Manual check is thus necessary.

We want to know how often produced patches overfit to the training test suite. Given that GI usually uses a heuristic approach, we conduct repeated runs and additionally report on the overfitting rates during the different runs. As mentioned at the beginning of this section, for this RQ (and RQs 3–4 too), we look at all valid patches generated during the whole GI search process.

RQ3 (Metric vs. Overfitting) How does the non-overfitting rate correlate with the changes in coverage?

This question is designed to answer how fragile is the GI process to the change in the coverage for a given test suite. Although we do not test for causation, we are interested to discover if any of the coverage measures can be used as a reliable predictor for the overall ratio of non-overfitting patches. If so, one can aim at improving their test suite with regards to that specific measure, in order to reduce the amount of overfitting.

RQ4 (Automated vs. Manual) How often do the automatically generated test suites overfit with respect to the manually generated ones?

The same question was asked by Smith et al. [21], though in the C domain and in the automated program repair context. We want to check if the same conclusions hold in our scenario. In order to answer this question, we perform a cross-validation using the automatically generated test suites using a set of coverage criteria, against manually curated test suites.

With the above research questions in mind we set up our experiments. The next subsections describe the datatset, tools, and experimental procedure in more detail.

3.2 Dataset & Tools

We used the genetic improvement toolbox Gin v2.0 [6] in our experiments. Gin fulfills all our requirements: it is open-source, targets Java programs, uses runtime improvement as fitness by default, and its second release provides integration with EvoSuite [11].

Unlike in the automated program repair field, there is no standard benchmark for runtime improvement using GI. We also require the programs to be relatively small, so we could run thousands of experiments in reasonable time and avoid the, often very costly, profiling stage of the GI process [6], targeting the whole software instead. With those restrictions in mind, we chose to use the set of 9 sort algorithms and the triangle example provided with the first release of Gin [23] in our study, for which improvements have previously been found using GI [9].

All experiments were run on a Mac Mini with a 3.2GHz 6-core Intel Core i7.

3.3 Experimental Procedure

We will now outline the details of the empirical study aimed at answering our research questions.

For each program and for each test suite coverage criterion (i.e., branch, line, conditional branch, and weak mutation) we generate a test suite using EvoSuite. Additionally, we generate a test suite that aims to cover all 4 coverage criteria at once. Next, we input the program and the given test suite to the GI process.

Gin uses a simple hill climber by default. It first generates a random mutation (which could be a delete, copy or replace operation), applies it to the code and evaluates it. If the change is beneficial in terms of runtime, it is retained, otherwise it is retained with 50% probability. The process continues for 100 iterations. Since this is a heuristic approach we repeat the GI cycle 20 times. We extended the algorithm to make changes at the statement rather than the default line-level (to allow for known improvements from previous work to be found). Moreover, we used Gin's PatchAnalyser to evaluate generalisability and runtime improvement of each generated non-empty patch on a held-out test suite.

In order to get variation in the coverage percentage for the various metrics, from each automatically generated test suite we sample 25%, 50%, and 75% of its tests, creating new test suites of varying coverage. We repeat the GI process with these as well. In order to check for overfitting we use the manual test suite, provided with the programs, as an oracle.

Altogether, we ran 20 rounds of GI on each of the 10 programs and 20 generated test suites (5 coverage criteria [4 single + 1 combined] × 4 samples), for a total of 400 GI runs per program. Finally, we also generated a test suite with the 4 coverage criteria as goals to treat as an oracle for GI runs on the manual test suite, to compare the impact of manual vs automated test suites on overfitting in GI. Therefore, a total of 4 200 GI runs was conducted, with 420 000 patches generated (4 200 × 100 steps of each local search run).

Pseudo-code for our experimental procedure is found in Algorithm 1. Note that whenever a patch passed the training suite during a GI run, it was evaluated against a test suite.

4 Results

In this section we present the results of our experiments and provide answers to research questions posed in Section 3.1. We deem a resultant patch as overfitting if it fails on the held-out test suite. For the runs where the manual test set was used as input, we generated tests using EvoSuite with the four coverage goals previously considered, i.e., branch, line, conditional branch, and weak mutation. For all the other test suites we used the manual test suite as the test set, to check for overfitting.

The experiments took a total of 16 hours to complete. All the data (and the modified Gin code to facilitate the experiments) is available as a replication package on GitHub: https://github.com/ssbseRENEsubmission/ssbseRENEsubmission.

Algorithm 1: Pseudo-code for the experimental procedure. Each GI run consists of 100 steps of local search.

```
for each Program P do
    for Coverage metric C from [line, branch, conditional branch, weak
      mutation, all4] do
        Generate test suite T using C as coverage goal
        for perc in [100%, 75%, 50%, 25%] do
            T' = Select perc of tests from T
            for i = 1; i <= 20; i++ do
                Run GI with P and T' as input
            end
        end
        for i = 1; i <= 20; i++ do
            Run GI with P and T_manual as input
        end
    end
end
```

4.1 RQ1 – Validity

To recap, for each of the 10 subject programs we generated 21 test suites: 4 satisfying 100% coverage of the test suite criterion; one that aimed to satisfy all 4 goals at once; and the manual one; the 5 automatically generated test suites were sampled at 100% 75%, 50% and 25%. This yielded 210 experimental scenarios. In answer to **RQ1**, a patch was found in all scenarios. That is, for each (test suite, program) pair GI found a non-empty patch in at least one run. Moreover, in 203 scenarios a patch was found in at least one of the GI runs that generalised to the held-out test suite.

Table 1. Number of all and non-overfitting patches found in at least one of the 20 repeated GI runs for each program.

Program	LoC	Test Sizes	Test Suites	Patch Found	Non-Overfitting
SortMerge	52	1–8	21	21	19
Triangle	40	1–10	21	21	21
SortQuick	32	1–9	21	21	21
SortBubbleDouble	24	1–7	21	21	21
SortRadix	24	1–7	21	21	21
SortSelection	19	1–7	21	21	21
SortBubbleLoops	17	1–7	21	21	21
SortSelection2	17	1–7	21	21	19
SortBubble	15	1–7	21	21	19
SortInsertion	14	1–7	21	21	20
Total	254	1–10	210	210	203

With these results, we can positively answer **RQ1** and state that, within our experimental setup, GI can indeed find valid and non-overfitting patches.

4.2 RQ2 – Overfitting

Table 2. Number of all intermediate and non-overfitting patches found during improvement for all 4 200 GI runs, aggregated by each test suite type. Summative results shown for the 10 programs investigated.

Criterion	Sample%	Patch Found	Non-Overfitting	Ratio
Branch	100	4 407	1 481	0.34
	75	3 907	1 443	0.37
	50	4 160	1 338	0.32
	25	4 651	1 486	0.32
Line	100	4 989	1 366	0.27
	75	5 009	1 412	0.28
	50	4 907	1 141	0.23
	25	5 122	1 374	0.27
W. Mutation	100	4 983	1 196	0.24
	75	4 455	1 356	0.30
	50	4 498	1 447	0.32
	25	4 902	1 138	0.23
C-Branch	100	4 719	1 253	0.27
	75	4 666	1 526	0.33
	50	4 826	1 167	0.24
	25	4 995	1 257	0.25
All 4 Criteria	100	5 360	1 413	0.26
	75	4 620	1 625	0.35
	50	4 670	1 234	0.26
	25	5 098	1 312	0.26
Manual	100	2 491	2 410	0.97

We provide more detailed results on the rate of overfitting throughout the search process (i.e., including intermediate solutions) in Tables 2 and 3, and Figure 1. Here we report on all intermediate patches found during all 4 200 GI runs.

The ratios yielded by the generation criteria average from approximately 0.23 to 0.37, with the branch criterion yielding a better ratio overall. Moreover, there is no apparent trend in the changes in sample percentages to the ratio, i.e., in some cases even sampling as few as 25% of the test cases yielded a non-overfitting ratio similar to that of using the whole test suite. However, when we group the data by program (Table 3), a higher variation in ratios becomes apparent. This may be an indication that the overfitting ratio is more dependent on the program being improved, rather than on the criteria used to create the test suites.

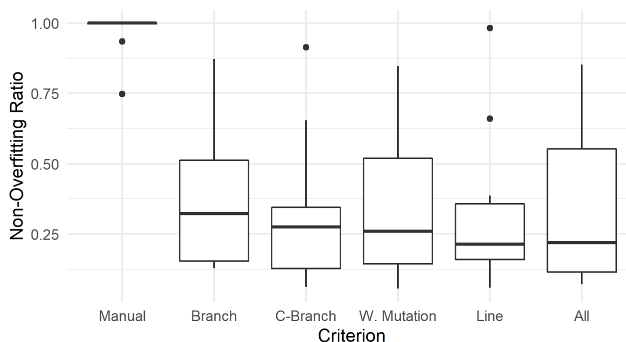

Fig. 1. Boxplot with the ratio between non-overfitting patches/total patches found (y-axis) per generation criterion (x-axis). Each box presents the ratios for the 10 programs.

Table 3. Number of all intermediate and non-overfitting patches found during improvement for all GI runs for each program. Data presented for all test suites investigated.

Program	Patch Found	Non-Overfitting	Ratio
SortBubble	12 131	3 213	0.26
SortBubbleDouble	12 632	8 537	0.68
SortBubbleLoops	12 466	3 366	0.27
SortInsertion	10 129	2 048	0.20
SortMerge	5 228	2 588	0.50
SortQuick	3 005	2 690	0.90
SortRadix	9 622	1 090	0.11
SortSelection	11 548	1 472	0.13
SortSelection2	12 081	1 187	0.10
Triangle	8 593	3 184	0.37

We applied the Fisher's exact test [10] on the data in order to determine if the differences in proportion of overfitting and non-overfitting patches obtained by the test generation criteria are statistically significant. Figure 2 presents the ranks of each generation criterion over the 10 programs. The results of the Fisher's exact test were used to perform the rank computation, such that two criteria are considered "statistically tied" if $p \geq 0.05$ for their pair comparison (i.e, the difference in their proportions of non- and overfitting patches is not statistically significant). In such case, the rank of a criterion is given by the average of the ranks of all criteria to which it ties (including its own). We adopted this analysis because it would be infeasible to report all the 150 p-values (10 programs $\times$ 15 pairwise combinations), and because it can easily depict rank superiority with statistical significance. We refer to this method herein as "statistical rank".

The branch criterion presented the best results among all criteria, with a mean rank of 3.35 (median 2.75). Line coverage presented the worst results, with an average rank of 4.20 (median 4.50). Surprisingly, using all available criteria

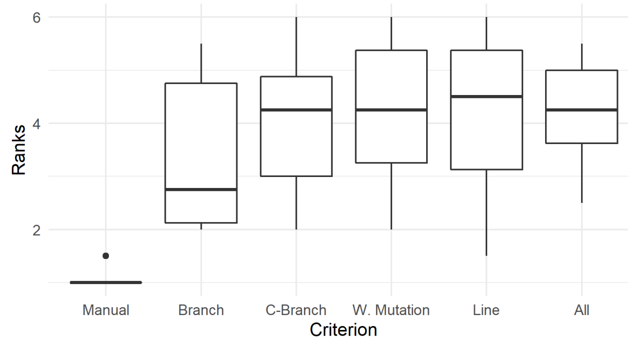

Fig. 2. Boxplot with the statistical ranks of non-overfitting ratio (y-axis) per generation criterion (x-axis). Each box presents the ranks for the 10 programs. Statistical ties are averaged.

to guide the creation of test suites yields the second worst ratio and rank (mean and median 4.20 and 4.25), even though it provides the best coverage.

Considering the analysis of this section, the answer to **RQ2** is somewhat mixed. First, the differences in non-overfitting ratio are not that striking as shown in Figure 1 and Table 2, but the results by program in Table 3 vary quite widely. However, when considering the statistical ranks of the criteria, test suites generated with branch coverage are slightly (but significantly) better than the others in the proportions. Hence, using only branch coverage as opposed to the other options can give the engineer a slight advantage on the result of GI in regards to overfitting, although it depends on the program being improved.

4.3 RQ3 – Metrics vs. Overfitting

In order to answer this question, we collected all the results from all GI executions and applied the Spearman's rank correlation coefficient test (Spearman's ρ) [22]. We use this correlation test because we cannot assume the normal distribution of data (the Pearson correlation coefficient test [18] would not be suitable). In fact, we checked for normality and could not reject the hypothesis that the data does not follow a normal distribution.

We aim at assessing if there is any correlation between the non-overfitting ratio and the coverage metrics of the test suites. In order to cater for the different confounding variables, we have also applied the correlation test on the data by sampling them based on their program, criteria, and sampling percentages. By isolating these variables and making them constant in each sample, we can unveil whether any of them can have some influence on the outcome of the correlation. Table 4 presents the correlation results.

From all the 100 correlations tested, only 24 showed statistical significance (p-value < 0.05), and only 4 of those showed large correlation (Spearman's $\rho > 0.5$). For instance, when GI is applied to SortRadix, the levels of Branch coverage and

the C-Branch coverage of the test suites showed large correlation to the ratio of non-overfitting patches generated using the test suites. This also happened to the levels of Line coverage and Weak Mutation coverage of the test suites when applied to SortBubbleLoops.

However, due to the low frequency of significant results and stronger correlations, our answer to **RQ3** is rather negative. We could not find any consistency from a specific coverage metric in linearly predicting the level of non-overfitting ratio yielded by the test suites during improvement. Furthermore, the correlation is significant only for a few programs, which may indicate that the properties of the programs are more important variables for the overfitting of GI.

4.4 RQ4 – Automated vs. Manual

Table 5 presents the non- and overfitting results for the manual created test suites and to the test suites created with all 4 criteria (test suites that yielded the best coverage overall). To recap, we perform a cross-validation, where the patches generated with the aid of one type of test suite (automatically generated/manual) are validated with the other type of test suites.

The first observation is that automatically generated test suites always have the best coverage, for all coverage metrics and in all programs, with only two ties. Moreover, automatically generated test suites always lead to more valid patches during GI optimisation. However, manually created test suites always produce more non-overfitting patches, despite generating fewer patches overall. Consequently, the proportion of non-overfitting patches is considerably and significantly better for manual test suites (Fisher's exact test [10], p-value < 0.05).

These results indicate that in fact, better coverage does not translate to better patches regarding overfitting. This analysis serves as further evidence for the lack of correlation between coverage and non-overfitting.

Finally, answering **RQ4**, manually created test suites generate patches that overfit significantly less than patches generated with automatically generated test suites. The non-overfitting ratio of manual test suites is almost always 1.0 of the valid patches, whereas the ratio for automatically generated test suites varies from 0.04 to 0.86 but never greater than their counterpart.

5 Threats to Validity

In this section we discuss threats to validity of the presented work.

Firstly, the programs investigated are quite small thus the results might not generalise. There were several reasons for choosing this small set. There's no standard benchmark for GI for runtime, yet improvements for the programs we use have been found in previous work. We also investigated the rate of overfitting during the search process, yielding 420 000 patches generated, roughly a quarter of those being re-run to check for overfitting. With this small sample the experiments took a non-substantial amount of time (16 hours). Moreover, similar

Table 4. Spearman's ρ correlation coefficient results. The table is subdivided into 4 sub-tables, where each sub-table shows the results for a different grouping. Each row represents a group of test suites and each column represents a measure of coverage (plus a column for the size of the test suites). A given cell $[R, C]$ shows the correlation result for all test suites in the group of row R, between their ratio of non-overfitting and their measure of column C. For instance, the first cell [SortBubble, Size] indicates a Spearman's correlation $\rho = 0.29$ between the ratio of non-overfitting and the size of test suites applied to program SortBubble. The * symbol highlights correlations for which p-value < 0.05.

Grouped by Program					
Groups	Size	Branch Cov.	Line Cov.	Weak Mut. Cov.	C-Branch Cov.
SortBubble	0.29	0.40	*0.45	0.29	0.40
SortBubbleDouble	0.42	-0.11	-0.14	-0.31	-0.11
SortBubbleLoops	0.27	0.37	*0.53	*0.54	0.37
SortInsertion	0.07	0.27	0.17	0.44	0.27
SortMerge	-0.18	0.19	0.10	0.15	-0.15
SortQuick	-0.23	-0.24	-0.22	-0.27	-0.22
SortRadix	0.42	*0.62	*0.49	*0.46	*0.62
SortSelection	-0.08	-0.14	-0.31	-0.31	-0.14
SortSelection2	0.10	0.15	0.09	0.09	0.15
Triangle	0.05	0.11	0.04	0.14	0.11

Grouped by Test Suite Generation Criterion					
Groups	Size	Branch Cov.	Line Cov.	Weak Mut. Cov.	C-Branch Cov.
All 4 Criteria	0.22	0.15	0.07	0.07	-0.03
Branch	0.24	-0.04	-0.04	0.02	-0.05
C-Branch	*0.37	-0.01	-0.23	0.05	-0.02
Line	0.21	*0.35	0.22	*0.41	0.19
W. Mutation	0.21	*0.40	*0.46	0.21	-0.05

Grouped by Test Suite Sample %					
Groups	Size	Branch Cov.	Line Cov.	Weak Mut. Cov.	C-Branch Cov.
25%	*0.28	0.21	0.17	0.25	-0.04
50%	*0.29	*0.35	*0.28	*0.36	0.12
75%	*0.41	*0.30	0.19	0.25	0.10
100%	*0.28	-0.17	*-0.30	-0.22	-0.21

No Data Grouping					
Groups	Size	Branch Cov.	Line Cov.	Weak Mut. Cov.	C-Branch Cov.
Whole Data	*0.24	*0.18	0.10	*0.17	0.03

size programs with similar size test suites were used in previous work [21]. We believe that for a preliminary study these were good enough.

Table 5. Coverage measures and cross-validation overfitting results for Manually vs. Automatically (abbreviated as "Aut.") generated test suites. T. Suite – type of test suite. Branch, Line, Weak Mutation (abbreviated "W. Mut."), and C-Branch – coverage obtained by the test suites. Patches – number of valid patches found during improvement. Non-Overf. – number of non-overfitting patches found during improvement. Ratio – ratio of number of non-overfitting by number of patches. Best values are highlighted in bold.

Program	T. Suite	Branch	Line	W. Mut.	C-Branch	Patches	Non-Overf.	Ratio
SortBubble	Manual	86%	88%	97%	86%	276	**276**	**1.00**
	Aut.	**100%**	**100%**	**99%**	**100%**	**537**	212	0.39
SortBubbleDouble	Manual	85%	71%	81%	85%	618	**618**	**1.00**
	Aut.	**92%**	**79%**	**83%**	**92%**	**643**	497	0.77
SortBubbleLoops	Manual	89%	89%	97%	89%	345	**344**	**1.00**
	Aut.	**100%**	**100%**	**98%**	**100%**	**523**	192	0.37
SortInsertion	Manual	86%	86%	97%	86%	218	**218**	**1.00**
	Aut.	**100%**	**100%**	**99%**	**100%**	**660**	89	0.13
SortMerge	Manual	94%	97%	96%	6%	118	**118**	**1.00**
	Aut.	**100%**	**100%**	**97%**	**67%**	**439**	29	0.07
SortQuick	Manual	93%	95%	**97%**	7%	152	**152**	**1.00**
	Aut.	**100%**	**100%**	**97%**	**100%**	**153**	132	0.86
SortRadix	Manual	93%	93%	97%	93%	122	**114**	**0.93**
	Aut.	**100%**	**100%**	**98%**	**100%**	**428**	50	0.12
SortSelection	Manual	86%	92%	97%	86%	178	**178**	**1.00**
	Aut.	**100%**	**100%**	**98%**	**100%**	**788**	35	0.04
SortSelection2	Manual	86%	91%	96%	86%	178	**178**	**1.00**
	Aut.	**100%**	**100%**	**97%**	**100%**	**704**	49	0.07
Triangle	Manual	89%	92%	**98%**	89%	286	**214**	**0.75**
	Aut.	**100%**	**96%**	**98%**	**100%**	**485**	128	0.26

Next, we used EvoSuite and Gin, thus inherited any limitations of the tools. For instance, EvoSuites generation of test suites is non-deterministic, thus multiple runs might yield different results. However, we have re-done the experiments with a few different seeds for EvoSuite and found the results consistent with the ones reported in this paper. We did not conduct enough to report on statistical significance of those though. Furthermore, different results might be obtained with other mutation strategies than the default in Gin's local search implementation. However, these mutation operators are currently standard ones in test-based GI.

6 Conclusions

In this paper we evaluate the levels of non- and overfitting patches obtained by different test suites during the GI optimisation process. The goal of our experiments is to show the differences between manually and automatically generated test suites, and the correlations between coverage and the ratios of non-overfitting patches.

Our results unveiled that, regardless of the criterion used to guide the automatic generation of test suites, the ratios of overfitting differ slightly between each other, with branch coverage being significantly better than the other criteria, but only by a small margin. Moreover, we could only find 4 significant and large correlations amongst a set of 100 tested correlations, which is not enough scientific evidence to consider any of the tested coverage measures as accurate predictors for the ratio of non-overfitting patches. Finally, our results showed that even though automatically generated test suites cover significantly more the programs under test and generate more valid patches throughout the search process, manually curated test suites yield a significantly better proportion of non-overfitting patches with almost no overfitting at all. This shows that classical automated test suite measures seem to have no bearing on how good a test suite is for the purpose of applying genetic improvement. Thus, a question of which characteristics a test suite should have so that useful, non-overfitting variants are produced during the GI search process, remains unanswered.

As future work, we intend to extend the study with larger programs as well as evaluate whether the properties of the programs under improvement play a bigger role in the overfitting of patches than the properties of tests suites.

Acknowlegements This work was funded by the EPSRC grant EP/P023991/1 and the ERC grant 741278 Evolving Program Improvement Collaborators (EPIC). The authors would also like to thank Prof. Gordon Fraser from University of Passau for consultation on the output diversity metric.

References

1. An, G., Kim, J., Yoo, S.: Comparing line and AST granularity level for program repair using pyggi. In: Petke, J., Stolee, K.T., Langdon, W.B., Weimer, W. (eds.) Proceedings of the 4th International Genetic Improvement Workshop, GI@ICSE 2018. pp. 19–26. ACM (2018). https://doi.org/10.1145/3194810.3194814
2. Assiri, F.Y., Bieman, J.M.: An assessment of the quality of automated program operator repair. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014. pp. 273–282. IEEE Computer Society (2014). https://doi.org/10.1109/ICST.2014.40
3. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: Young, M., Xie, T. (eds.) Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15). pp. 257–269. ACM (2015). https://doi.org/10.1145/2771783.2771796
4. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015). https://doi.org/10.1109/TSE.2014.2372785
5. Basios, M., Li, L., Wu, F., Kanthan, L., Barr, E.T.: Darwinian data structure selection. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT 2018. pp. 118–128. ACM (2018). https://doi.org/10.1145/3236024.3236043

6. Brownlee, A.E.I., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: genetic improvement research made easy. In: Auger, A., Stützle, T. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019. pp. 985–993. ACM (2019). https://doi.org/10.1145/3321707.3321841
7. Bruce, B.R., Petke, J., Harman, M., Barr, E.T.: Approximate oracles and synergy in software energy search spaces. IEEE Trans. Software Eng. **45**(11), 1150–1169 (2019). https://doi.org/10.1109/TSE.2018.2827066
8. Chekam, T.T., Papadakis, M., Le Traon, Y., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 597–608 (2017)
9. Cody-Kenny, B., Lopez, E.G., Barrett, S.: locoGP: improving performance by genetic programming Java source code. In: Langdon, W.B., Petke, J., White, D.R. (eds.) Genetic Improvement 2015 Workshop. pp. 811–818. ACM (2015). https://doi.org/doi:10.1145/2739482.2768419
10. Fisher, R.A.: On the interpretation of chi-squared from contingency tables, and the calculation of P. Journal of the Royal Statistical Society **85**(1), 87–94 (1922). https://doi.org/10.2307/2340521
11. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: Núñez, M., Hierons, R.M., Merayo, M.G. (eds.) Proceedings of the 11th International Conference on Quality Software, QSIC 2011. pp. 31–40. IEEE Computer Society (2011). https://doi.org/10.1109/QSIC.2011.19
12. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Trans. Evolutionary Computation **19**(1), 118–135 (2015). https://doi.org/10.1109/TEVC.2013.2281544
13. Langdon, W.B., Lam, B.Y.H., Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In: Silva, S., Esparcia-Alcázar, A.I. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015. pp. 1063–1070. ACM (2015). https://doi.org/10.1145/2739480.2754652
14. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Trans. Software Eng. **38**(1), 54–72 (2012). https://doi.org/10.1109/TSE.2011.104
15. Offutt, A.J., Lee, S.D.: How strong is weak mutation? In: Howden, W.E. (ed.) Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991. pp. 200–213. ACM (1991). https://doi.org/10.1145/120807.120826
16. Offutt, A.J., Lee, S.D.: An empirical evaluation of weak mutation. IEEE Trans. Software Eng. **20**(5), 337–344 (1994). https://doi.org/10.1109/32.286422
17. Offutt, A.J., Untch, R.H.: Mutation 2000: Uniting the Orthogonal, pp. 34–44. Springer US (2001). https://doi.org/10.1007/978-1-4757-5939-6-7
18. Pearson, K.: Vii. note on regression and inheritance in the case of two parents. proceedings of the royal society of London **58**(347-352), 240–242 (1895)
19. Petke, J., Haraldsson, S.O., Harman, M., White, D.R., Woodward, Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolutionary Computation (2017). https://doi.org/10.1109/TEVC.2017.2693219
20. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Proceedings of the 17th European Conference on Genetic Programming. vol. 8599, pp. 137–149. Springer (2014). https://doi.org/10.1007/978-3-662-44303-3-12

21. Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. p. 532–543. ESEC/FSE 2015 (2015). https://doi.org/10.1145/2786805.2786825
22. Spearman, C.: The proof and measurement of association between two things. American journal of Psychology **15**(1), 72–101 (1904). https://doi.org/10.2307/1422689
23. White, D.R.: GI in no time. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17). pp. 1549–1550. ACM (2017). https://doi.org/10.1145/3067695.3082515
24. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In: Silva, S., Esparcia-Alcázar, A.I. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015. pp. 1375–1382. ACM (2015). https://doi.org/10.1145/2739480.2754648
25. Yi, J., Tan, S.H., Mechtaev, S., Böhme, M., Roychoudhury, A.: A correlation study between automated program repair and test-suite metrics. Empirical Software Engineering **23**(5), 2948–2979 (2018). https://doi.org/10.1007/s10664-017-9552-y