Enhancing Genetic Improvement of Software with Regression Test Selection

Giovani Guizzo, Justyna Petke, Federica Sarro and Mark Harman
Centre for Research on Evolution, Search and Testing (CREST)
Department of Computer Science
University College London (UCL)
London, United Kingdom
{g.guizzo, j.petke, f.sarro, mark.harman}@ucl.ac.uk

***Abstract*—Genetic improvement uses artificial intelligence to automatically improve software with respect to non-functional properties (AI for SE). In this paper, we propose the use of existing software engineering best practice to enhance Genetic Improvement (SE for AI).**

We conjecture that existing Regression Test Selection (RTS) techniques (which have been proven to be efficient and effective) can and should be used as a core component of the GI search process for maximising its effectiveness.

To assess our idea, we have carried out a thorough empirical study assessing the use of both dynamic and static RTS techniques with GI to improve seven real-world software programs.

The results of our empirical evaluation show that incorporation of RTS within GI significantly speeds up the whole GI process, making it up to 68% faster on our benchmark set, being still able to produce valid software improvements.

Our findings are significant in that they can save hours to days of computational time, and can facilitate the uptake of GI in an industrial setting, by significantly reducing the time for the developer to receive feedback from such an automated technique. Therefore, we recommend the use of RTS in future test-based automated software improvement work. Finally, we hope this successful application of SE for AI will encourage other researchers to investigate further applications in this area.

***Index Terms*—Genetic Improvement, Regression Test Selection, Search Based Software Engineering, Genetic Programming**

I. Introduction

Genetic Improvement (GI) is an Artificial Intelligence technique used to improve a given property of an existing software in an automated way [1]. The software property can be either functional (e.g., bug fixing [2]–[4]) or non-functional (e.g., runtime [5]–[7], memory usage [8], [9], energy consumption [10]–[12]), provided such a property can be formulated as a fitness function. The fitness function guides the iterative search process by quantitatively measuring the level of improvement of the software.

At each iteration, multiple variants of the software are produced and tested in order to assess whether the transformations preserve the overall software functionality, and then their level of improvement is measured. As one can infer, this process of constantly testing different versions of the software is time-consuming [5], [13]–[15], especially when the software is accompanied by a costly test suite. Even for toy programs with relatively small test suites, GI executions can take several hours or even days [16]. One possible way of solving this problem is to select only a subset of test cases to execute instead of using the whole test suite.

Regression Test Selection (RTS) [17] has been extensively studied in the Software Engineering literature with the main purpose of selecting subsets of test cases for newer versions of the software. RTS has been successfully used [18] to, following a set of code modifications, select only the subset of test cases that can test the changed code. Differently from Regression Test Minimisation and Regression Test Prioritisation that aim at permanently removing redundant/useless test cases and ordering test cases for execution, respectively, RTS temporarily selects subsets of test cases for the imminent testing task. The underlying objective is to reduce the overall cost of software testing by simply executing fewer test cases at each iteration.

We hypothesise that existing RTS techniques (which have been proven to be efficient and effective [17], [19], [20]) can and should be used as a core component of the GI search process for maximising its effectiveness. Prior work showed the effectiveness of RTS in various contexts, but never for non-functional GI. In this context, different/unseen challenges can arise when RTS is applied because not only it concerns whether the software variants pass all test cases (which Automated Program Repair — APR — is solely concerned with), but also whether the non-functional properties will be affected. In other words, speeding up the GI process impacts the search itself, because the software speed-up is one of the fitness functions that guides the search. Therefore, a thorough investigation is needed to discover and report the magnitude of such effects. Specifically, we present results to show how RTS can imbue GI fitness computation with a faster procedure, whilst also maintaining the correctness of the generated improved software.

The main objective of this paper is to investigate: i) how effective are the RTS techniques in the context of GI; ii) what is the efficiency gain provided by such techniques; and iii) what is the overall trade-off between efficacy and efficiency gain in various scenarios.

In order to answer these questions and evaluate the effectiveness of RTS in the context of GI, we carry out an extensive empirical evaluation with two state-of-the-art RTS techniques based on dynamic and static analysis [19] over seven real-world programs.

The main contributions of this work are:

1) Assessment of the effectiveness of RTS in the context of GI with extensive experimentation;
2) Report of results that can shape how GI is designed and applied in the future;
3) Integration of state-of-the-art RTS techniques into an open-source genetic improvement tool.

The results of our empirical evaluation show that incorporation of RTS within a GI framework can significantly speed up the whole GI process, making it up to 68% faster on our benchmark set, with minimal loss to generality to the whole test set. In fact, the combination of both dynamic and static RTS with GI led to improved program variants that passed all the tests for the given program. The results are significant in that they can save hours to days of computational time yet still be able to find valid software improvements. This should provide a faster uptake of the techniques in an industrial setting, by significantly reducing the time for the developer to receive feedback from such an automated technique. Therefore, we recommend the use of RTS in future test-based automated software improvement work. Finally, we hope this successful application of SE for AI will encourage other researchers to investigate further applications in this area.

To aid reproducibility we provide a replication package at https://figshare.com/s/440c2105bad3259bda6f.[1]

II. Background

This section presents the background on the two main topics of this paper: RTS and GI.

A. Regression Test Selection

Regression Testing is a task performed to assess whether changes to a given software harm any of its pre-existing functionalities [17]. The default strategy is *retest-all*, which basically re-runs all the available test cases against the modified program. However, the cost of this strategy becomes prohibitive as the software and its test suite grow in size and complexity. In order to speed-up the process and avoid the execution of the whole test suite, a set of regression test strategies have been proposed [17]. The most common ones are test case prioritisation, test suite minimisation, and test case selection.

Test case prioritisation aims at rearranging the test cases execution order to maximise fault detection or to reveal faults earlier. Test suite minimisation focuses on permanently removing obsolete or irrelevant test cases from the test suite. Finally, test case selection (also known as Regression Test Selection – RTS) techniques select test cases based on current changes between one software version to another. Its main objective is to avoid the execution of test cases that do not exercise the modified code. Note that RTS mostly depends on the differences between versions of the software, whereas test suite minimisation and test case prioritisation do not necessarily rely on such information. This work focuses on RTS, hence we present it in more detail.

According to Yoo and Harman [17], a subset of test cases T' from the test suite T should contain all available test cases that can reveal the fault in the modified version P' of the original program P. A test case t is fault revealing in relation to P and P' if the result of the execution of t is different in both versions. In order to check whether t is fault revealing, t must be executed against P and P'. The underlying and reasonable assumption is that $P(t)$ halted and produced the correct output. Hence, for a new version P', one should select all test cases that traverse the modified code in P', or that used to traverse a deleted piece of code in P', and then compare the results. If an RTS technique selects all modification-traversing test cases in relation to P and P', this technique is called *safe*.

In this work, we use two RTS techniques: a modification-based approach (dynamic analysis) and a firewall approach (static analysis) [17]. The modification-based approach implementation is based on the Ekstazi [18] tool. Ekstazi performs dynamic analysis over a given program in order to find dependency between Java files. It applies hashing functions over the content of the files, and when a mismatch between old hashes and new ones occur, all test cases that depend on such a file are selected. On the other hand, the firewall approach implementation is based on the STARTS [21] tool. STARTS collects dependency information between classes with static analysis before executing the tests. This technique then analyses the modifications and selects all test cases with dependencies to entities inside the "firewall".

Both tools have been evaluated in the literature [19], [20] and have been shown to be relatively inexpensive and safe. However, as shown by Chen and Zhang. [22], RTS can also be used in other testing contexts. The authors compared both tools for speeding up Mutation Testing, showing that both are able to select the adequate test cases to test a given mutant. We believe that RTS can also be used in the GI context with significant benefits.

B. Genetic Improvement and Efficiency

Genetic Improvement (GI) consists of the improvement of existing software through search based algorithms [1]. Search Based algorithms try to find an approximately optimum solution for a given problem for which the true optimum solution cannot be found in feasible time [23]. GI can be seen as part of the Search Based Software Engineering (SBSE) field [24], in which search based algorithms are used to solve hard Software Engineering problems.

During GI optimisation, software undergoes transformations in order to improve a set of properties, either functional or non-functional [1]. The most common type of functional improvement is APR [13], [14], [25], in which a faulty program is modified until the failing test suite passes. In non-functional GI however, the goal is to improve the software's memory usage [8], [9], execution time [5]–[7], energy consumption [10]–[12], and other non-functional properties, whilst maintaining

[1]For anonymity, we will only disclose the code of the tool and experiments in case the paper gets accepted. For now, the link contains only data analysis results.

the functional properties of the software, measured with the use of the program's test suite, i.e., test cases should pass after the program transformation. Either way, the GI process is guided by a fitness function that measures the level of functional or non-functional improvement.

At each GI iteration, usually a new version of the software is generated, its fitness computed by executing the test suite against it, and then the best versions found are stored for further use during the optimisation process. This process may go on for hundreds or thousands of iterations, each of which imposing the cost of executing the test suite against the candidate solution. The efficiency of GI becomes a problem when the software is accompanied by costly test suites, which has been pointed out in APR work [13], [14].

GI tools already implement some efficiency improvement mechanisms. For example, Gin [26], an all-purpose GI tool for Java programs, performs in-memory compilation which removes the overhead of writing transformed classes to disk before compiling. Gin also applies a profiling phase before starting the optimisation in order to identify target classes and methods, thus avoiding the transformation of uncovered code. However, the execution can still take several hours, even for relatively small programs [16]. We hypothesise that RTS can provide a significant efficiency improvement for GI.

As far as we are aware, there is only one related work in the literature. Mehne et al. [14] used an ad-hoc RTS (called "Test-Case Pruning") and a localisation technique with GenProg [2], an APR tool for C programs. Their technique only applies patches in specific places, selects test cases that cover the patched functions, and then executes only these test cases when the patch needs to be validated. The results show that Test-Case Pruning provides a speed-up of 1.8× with no additional overhead, whilst mostly maintaining the correctness of the patches.

Differently from the work of Mehne et al. [14], we propose a more in-depth evaluation of the impact of RTS on multiple GI properties, such as safety, efficiency, improvement capability, and the trade-off between these properties. Furthermore, we investigate RTS in the context of non-functional improvement as opposed to APR, though the techniques can also be applied in the APR context. We chose to target runtime improvement as fitness evaluation is even more costly than in the case of APR, as it requires multiple runs of each of the software variants to get reliable fitness measures. Therefore, efficiency of improvements of the whole GI process would be most beneficial.

The next section presents the Research Questions and the experimental set-up used to answer them.

III. Research Questions

In this work we aim to answer the following Research Questions (RQs):

RQ1. Effectiveness: How effective is regression test selection in the context of Genetic Improvement of software?

RQ2. Efficiency: What is the efficiency gain when using Regression Test Selection with Genetic Improvement?

RQ3. Trade-Off: What is the trade-off between efficiency and efficacy of the Genetic Improvement process with various Regression Test Selection strategies in different application scenarios?

In the following subsections we discuss in detail the motivation for each of the RQs and the measures we defined to gather the answers. In Section IV we describe the techniques, tools and program subjects we used in our empirical study.

A. RQ1 – Effectiveness

How effective is regression test selection in the context of genetic improvement of software? – This question is asked to evaluate whether RTS harms the general functionality of GI. We define herein "effectiveness" as a combination of functional validity and improvement levels of the software produced by the GI optimisation process. Specifically, we are concerned that RTS techniques may discard important test cases, and consequently the improved software resulting from the GI optimisation may fail when tested against the whole test suite. Moreover, depending on the selected test cases, the non-functional improvement capability of GI can be affected. If this is the case, then the use of RTS with GI may impose serious disadvantages that can affect the overall results of GI, rendering its application infeasible in practice.

We define the "Relative Safety" (RS) measure as follows:

$$RS(s_i, p) = \frac{|PT_{s_i,p}|}{|T|} \tag{1}$$

where $|T|$ is the number of test cases in the test suite T of program p; and $|PT_{s_i,p}|$ is the number of passing test cases in T when executed against the best improved version of p obtained by the strategy s in the i-th independent run. Effectively, this measure computes the percentage of test cases in the whole test suite T that pass when executed against the improved software version. The greater the RS, the safer the RTS technique.

We further analyse the results of the experiments in order to investigate whether the RTS techniques have any impact on the final outcome of GI in terms of improvement. For this end, we define the "Relative Improvement Change" (RIC) measure as follows:

$$RIC(s_i, p) = \frac{improvement(s_i, p)}{originalImprovement(p)} \tag{2}$$

$improvement(s_i, p)$ is the level of improvement (fitness value) of a valid version of the improved program p obtained by the strategy s_i in the i-th independent run; and $originalImprovement(p)$ is the average level of improvement for the program p when using GI without RTS. In summary, RIC computes the magnitude of the improvement obtained by a given strategy when compared to the results of GI with no RTS strategy. Thus, if $RIC >= 1.0$, then it means that the respective RTS strategy improves the capability of GI to improve software, otherwise there is a negative impact.

The results of RS and RIC are compared using the Kruskal-Wallis statistical test [27] and Vargha-Delaney $\hat{A}_{12}$

effect size [28]. The former is used to assess if the difference between the techniques is statistically significant across the 20 independent runs, whereas the latter measures the magnitude of the difference. Both tests are non-parametric, thus they do not assume normal distribution of the data.

B. RQ2 – Efficiency

What is the efficiency gain when using regression test selection with genetic improvement? – This question focuses on the main benefit of using RTS with GI: efficiency gain. We want to unveil what is the magnitude of the savings in terms of GI execution time during the optimisation process.[2] Whilst our implementation of GI works on improving the execution time of a program, the RTS techniques aim at improving the execution time of GI itself.

The overhead of RTS techniques comes mainly from the data collection on test cases and class dependencies during such a phase, and from the filtering of test cases for subsequent use (Section II-A). As one can infer, these tasks can become quite costly when a program is accompanied by thousands of methods and test cases. Certainly, it is undesirable to use an RTS technique that introduces more overhead than the efficiency gain it provides. Therefore, the cost of the strategies is also included in the efficiency evaluation.

We define the "Relative Cost" (RC) of a strategy as follows:

$$RC(s_i, p) = \frac{cost(s_i, p)}{originalCost(p)} \tag{3}$$

where $cost(s_i, p)$ is the sum of overhead time and optimisation time of the strategy s at the i-th independent run for program p; and $originalCost(p)$ is the average cost of the default implementation of GI without any RTS for program p. The lower the RC, the better. If the result of RC is greater or equal than 1.0, then we can state that the RTS technique does not reduce the overall cost of GI in that specific context. On the other hand, if the result is lower than 1.0, then we can quantify how much execution time can be saved by the strategy and compare to others, e.g., a RC of 0.4 means that the strategy saved 60% in execution time.

Similarly to RQ1, we apply the Kruskal-Wallis and Vargha-Delaney $\hat{A}_{12}$ tests over RC to identify statistical differences between the different test case selection strategies.

C. RQ3 – Trade-Off

What is the trade-off between efficiency and efficacy of the genetic improvement process with various regression test selection strategies in different application scenarios? – Finally, RQ3 takes into account the two measured properties and weighs them on a multi-objective space to identify if the trade-off is positive in different scenarios. For example, we want to identify whether the strategies provide greater cost savings than safety loss.

In order to answer this question, we first weigh the relative cost of each strategy versus its safety. If the strategies yield $RS = 1$ and $RIC \leq 1$, i.e., the improved programs do not fail when tested with the whole test suite and do not harm the improvement capabilities of GI, then the comparison is straightforward: the cheapest strategy provides the best trade-off between the measures. However, in case of obtaining results that are conflicting (i.e., failing test cases or negative changes in improvement), we have to perform a more careful investigation on the real trade-off.

With that in mind, we define a few GI application use cases and evaluate the trade-offs of each RTS strategy in such scenarios. The first use case is when the engineer wants to find the perfect improvement, i.e., they let the GI optimisation process execute until the end and then select the best improved software. The second use case takes place when the engineer needs a fast improvement, regardless of how good this improvement is. In other words, the engineer stops the optimisation process after the first positive improvement found by GI in which the software does not fail. The last scenario concerns diversity. In this case, the engineer wants to find a plethora of improved software versions, and then choose one (or many) from the set of non-failing ones. In all of the posed use cases, we evaluate the efficiency of each strategy against a different property of interest.

For the "perfect improvement" use case (herein abbreviated as P_{improv}), the property is the level of improvement of the best and valid found improved version. If a strategy is both faster to execute and provides better improvement, then it is naturally preferred for this scenario.

In the "fast improvement" use case (F_{improv}), the property of interest is the validity of the first improved version found. The improved version must pass all test cases in the test suite, not only the ones selected by the RTS technique, otherwise the engineer stopped the optimisation and was left with no valid software. The level of improvement for F_{improv} is not important, as long as it is a positive one.

Finally, in the "diverse improvement" use case (D_{improv}), the engineer will choose an improved software of their liking, thus the property of interest is the number of valid and positive improvement versions. The more options the engineer has, the better suited the RTS strategy is for this use case. The underlying question posed here is: how fast the RTS techniques can successfully fulfil each of these use cases? This question is answered in a qualitative manner using the concept of Pareto optimality [29].

IV. Experimental Design

In this Section we describe in detail the techniques, the program subjects and the tools we use in our empirical study.

A. Techniques

In order to answer our research questions, we use two state-of-the-art RTS techniques (based on the Ekstazi and STARTS tools – Section II-A) in combination with Gin [26]. Namely, we compare the following strategies in our experiment:

- GI – GI with no RTS;

[2]Not to be confused with the savings in execution cost of the improved version of the program.

- GI+Random – randomly selects a subset of test cases without guidance;
- GI+Ekstazi – GI using Ekstazi as a dynamic analysis RTS technique;
- GI+STARTS – GI using STARTS as a static analysis RTS technique.

We did not use the default test case selection mechanism implemented in Gin, since it showed to be infeasible. For instance, whilst Ekstazi and STARTS both took less than 10 minutes to execute on *commons-codec*, Gin's selection took 11 hours. The GI+Random strategy is used as a matter of sanity check.

We compare these implementations using seven subject programs over 20 independent runs (more details in Section IV-C). Multiple independent runs are needed to cater for the stochastic nature of search based algorithms [30]. The result of each independent run (best improved version p' of the program p) is collected and then evaluated in terms of number of passing test cases.

Conveniently, Gin applies a profiling mechanism before the actual optimisation process to gather information about "hot spots" in the code, which are later focused on by the GI optimisation. The output of this preprocessing is a `csv` file containing which methods and how many times they were executed, along with the set of test cases to test each method during the optimisation process. If no RTS technique is selected, then the whole test suite is assigned to test all methods equally. On the other hand, if an RTS technique is selected, then it collects dependency data and assigns the test cases to each "hot spot" based on its own selection mechanism.

In order to answer questions about efficiency of the GI process with and without RTS, we sum the profiling overhead and the execution time of the GI process. Because all runs share the same stopping condition as a constant number of fitness evaluations (configuration parameters are presented in Section IV-C), we can determine which strategy can finish the optimisation process the fastest and how much time can be saved in relation to the default GI implementation without RTS, whilst also considering the overhead incurred by each technique.

B. Experimental Procedure

For each program under improvement, each of the 20 independent runs is preceded by a profiling task. Hence, first we perform the profiling task without any RTS, each of which generating a profiling `csv` file containing the hot spots, and all test cases are assigned to all hot spots. The execution time of this phase (original overhead) is then stored and serves as a baseline for further comparisons. Then, we compute the overhead of the other strategies over the 20 independent runs.

After performing the experiments without RTS, we collect and compute the average execution time. The execution time of a single run is simply the execution time taken to complete the optimisation process. Then, we compute the execution time of the other strategies over the 20 independent runs.

TABLE I
SUBJECT PROGRAMS. LLOC: NUMBER OF LOGICAL LINES OF CODE (EXECUTABLE LINES); #T: NUMBER OF TEST CASES IN THE PROGRAM'S TEST SUITE; T. LLOC: NUMBER OF LOGICAL LINES OF TEST CODE; COV: STATEMENT AND BRANCH COVERAGE PERCENTAGES OBTAINED BY THE TEST SUITE; TEST TIME: EXECUTION TIME OF THE TEST SUITE (MM:SS).

Program	LLOC	#T	T. LLOC	Cov	Test Time
codec-1.14	9 044	1 081	13 276	96/91	00:15
compress-1.20	25 978	1 170	22 059	84/75	01:39
csv-1.7	1 845	325	4 864	89/85	00:06
fileupload-1.4	2 425	82	2 284	80/76	00:04
imaging-1.0	31 320	583	7 427	73/59	00:52
text-1.3	8 703	898	12 872	97/96	00:05
validator-1.6	7 409	536	8 352	86/76	00:11

Finally, we sum the execution time of both the profiling task and the optimisation process for each strategy, program, and independent run. This is the total computational cost of GI with the respective strategy, which can tell us whether the total cost of such a strategy is lower than the total cost of the default GI implementation.

At the end, for each program, strategy, and independent run, we obtain a a testing cost and a set of improved software variants. Then the software variants are tested against the whole test suites in order to check for validity. In order to allow reproducibility, we provide a replication package at https://figshare.com/s/440c2105bad3259bda6f.

C. Subject Programs

We focus on the non-functional GI optimisation [1] of execution time, i.e., the improvement goal of the GI implementation is to reduce the overall execution time of the program under improvement.[3] The seven subject programs are presented in Table I. These programs are part of the Apache Commons project[4], a set of well-known and widely used libraries. We selected such programs because they are different in size, build with no errors, comply with all the requirements needed to run with Gin, have relatively large and passing test suites, and have different testing times. Furthermore, differently from related work [16], the use of non-trivial programs provides another investigation angle on the cost and results of non-functional GI. In other words, with such extensive experimentation, we can evaluate the RTS tools in a wider range of scenarios.

D. Genetic Improvement Framework

We chose Gin [26] for our experiments. It has been designed specifically for the improvement of Java programs. Gin provides several optimisations for running GI on Java software, including in-memory compilation which removes the overhead of writing transformed classes to disk before compiling. It also applies a profiling phase that helps identify which parts of code are covered by a given test suite and thus preventing uncovered parts from being modified.

[3]Not to be confused with the execution time of GI itself.
[4]https://commons.apache.org/

TABLE II
ALGORITHM PARAMETERS.

Parameter	Value
Population Size	40
Generations	20
Test Repetitions	10
Independent Runs	20
Tournament Percentage	20%
Mutation Probability	50%

TABLE III
PERCENTAGE OF SELECTED TEST CASES. MEDIAN ACROSS ALL 20 INDEPENDENT RUNS.

Program	+Ekstazi	+STARTS	+Random
codec	4.65%	4.65%	35.96%
compress	4.60%	16.99%	67.96%
csv	72.26%	96.45%	30.97%
fileupload	40.24%	42.68%	47.56%
imaging	2.11%	81.34%	39.52%
text	4.45%	4.45%	37.25%
validator	16.04%	29.66%	45.52%
Median	4.65%	29.66%	39.52%

The search algorithm used in the experiments is Genetic Programming (GP) [31], which has recently been updated in Gin. It uses tournament selection, uniform crossover and a mutation strategy that picks between a delete, replace, copy and swap mutation operators uniformly at random before applying the selected operator to program statements at the abstract syntax tree level. The parameters were set based on previous work we found that uses this algorithm [2], [32]. We use the default runtime fitness evaluation as implemented in Gin and set the number of test evaluations for each program to 10 for more reliable runtime measurements. Table II presents the parameters used by the algorithm.

V. RESULTS

This section presents the results of the experiments and provides answers to the RQs formulated in the previous section. Table III presents the number of test cases selected by each strategy for each program. This selection was performed in the profiling phase and the time taken to do such a task is computed as overhead.

A. Answer to RQ1 – Effectiveness

The first part of the effectiveness analysis concerns the Relative Safety (RS – Equation 1) of the best patches generated by the GI algorithm in each independent run when re-executed against all test cases, as opposed to tested only by the test cases selected by the RTS techniques.

Our first finding is that almost all resulting patches are valid. The only exceptions are Gin+Random results for *commons-csv* and *commons-text*, and Gin+STARTS for *commons-text*.

Gin+Random generated five invalid patches in five independent runs for which 60 test cases failed in total, i.e., the resulting patches were deemed invalid by the whole test suite. Gin+STARTS only generated one invalid patch (out of 140) for which two test cases failed. Hence, the RS of Gin+Random and Gin+STARTS is always higher than 0.995, and for Gin+Ekstazi RS is always 1.0.

The statistical tests showed no difference between the results, thus we can state that the RTS techniques can safely be used with GI.

The second part of this RQ concerns the Relative Improvement Change (RIC – Equation 2) of GI when using the RTS strategies. Unlike RS, we observed significant changes in the results. Table IV presents the median RIC of each strategy, alongside the Kruskal-Wallis p-value results, whereas Table V presents the effect size results for the RIC comparisons.

For all programs, GI+Ekstazi presented significant positive or statistically equivalent RIC to the best one, i.e., by using Ekstazi with GI, the quality of the resulting patches was at least equal or better in terms of obtained improvement. Similarly, STARTS also presented favourable results for six out of seven programs. For six out of seven ($\approx$86%), the differences between GI and GI+Ekstazi or GI+STARTS are statistically large.

One possible explanation for this positive impact on improvement is that, during the optimisation process with RTS, considerably fewer tests are executed. In such a case, the execution time differences between an improved version of the software and the original one are more significant to the search process. In other words, even with a small improvement of a few hundred milliseconds, the improvement is deemed

TABLE IV
RQ1. MEDIAN RELATIVE IMPROVEMENT CHANGE (RIC) OF STRATEGIES. GREATER VALUES ARE BETTER. BEST RIC VALUES OR VALUES STATISTICALLY EQUIVALENT TO THE BEST ONES ARE HIGHLIGHTED IN BOLD (P-VALUES < 0.05).

Program	GI	+Ekstazi	+STARTS	+Random	p-value
codec	1	**3.36**	**2.29**	0.95	**1.10e-06**
compress	1	**2.53**	**5.81**	**3.96**	**2.07e-07**
csv	1	**2.89**	**3.8**	**1.68**	**7.10e-04**
fileupload	**1**	**2.55**	**1.97**	**2.08**	6.50e-02
imaging	**1**	**0.64**	**0.95**	**0.46**	0.55240
text	1	**2.40**	0.95	0.57	**2.70e-03**
validator	1	**4.92**	**2.81**	**4.04**	**1.90e-04**
Median	1	**2.55**	2.29	1.68	–

TABLE V
RQ1. EFFECT SIZES FOR THE RELATIVE IMPROVEMENT CHANGE (RIC). EFFECT SIZES GREATER THAN 0.5 MEAN POSITIVE IMPROVEMENT FOR THE LEFT STRATEGY. DIFFERENCES: N = NEGLIGIBLE, S = SMALL, M = MEDIUM, AND L = LARGE. LARGE EFFECT SIZES ARE HIGHLIGHTED IN BOLD.

Program	GI/+Ekstazi	GI/+STARTS	+Ekstazi/+STARTS
codec	**0.04 (L)**	**0.14 (L)**	0.56 (S)
compress	**0.13 (L)**	**0.005 (L)**	0.3 (M)
csv	**0.16 (L)**	**0.2 (L)**	0.49 (N)
fileupload	**0.26 (L)**	0.4 (S)	0.63 (S)
imaging	0.58 (S)	0.48 (N)	0.48 (N)
text	**0.29 (L)**	0.56 (S)	**0.73 (L)**
validator	**0.11 (L)**	**0.22 (L)**	0.66 (M)

TABLE VI
RQ2. MEDIAN RELATIVE COST (RC) OF STRATEGIES. LOWER VALUES ARE BETTER. BEST RC VALUES (OR VALUES STATISTICALLY EQUIVALENT TO THE BEST ONES) ARE HIGHLIGHTED IN BOLD (P-VALUES < 0.05).

Program	GI	+Ekstazi	+STARTS	+Random	p-value
codec	1.00	**0.39**	**0.40**	0.77	**3.11e-09**
compress	1.00	**0.32**	**0.38**	0.89	**2.12e-12**
csv	1.00	1.03	**0.93**	**0.82**	**7.97e-03**
fileupload	**1.00**	1.11	**0.93**	**1.05**	**2.81e-02**
imaging	1.00	0.79	0.98	**0.40**	**6.48e-09**
text	1.00	**0.67**	**0.78**	1.15	**2.60e-05**
validator	1.00	**0.82**	**0.47**	1.01	**1.50e-02**
Median	1.00	0.79	**0.78**	0.89	–

TABLE VII
RQ2. EFFECT SIZES FOR THE RELATIVE COST (RC). EFFECT SIZES GREATER THAN 0.5 MEAN GREATER COST FOR THE LEFT STRATEGY. DIFFERENCES: N = NEGLIGIBLE, S = SMALL, M = MEDIUM, AND L = LARGE. LARGE EFFECT SIZES ARE HIGHLIGHTED IN BOLD.

Program	GI/+Ekstazi	GI/+STARTS	+Ekstazi/+STARTS
codec	**0.95 (L)**	**0.89 (L)**	0.55 (N)
compress	**0.99 (L)**	**0.98 (L)**	0.30 (M)
csv	0.49 (N)	0.65 (M)	0.70 (M)
fileupload	0.35 (S)	0.63 (S)	**0.74 (L)**
imaging	**0.80 (L)**	0.52 (N)	**0.24 (L)**
text	**0.82 (L)**	**0.76 (L)**	0.35 (S)
validator	0.63 (S)	**0.73 (L)**	0.64 (M)

more important by the GI algorithm because the cost of testing the program is also relatively small. Without RTS, the same execution time improvement would be "diluted" by thousands of test cases that do not test the changed code. Hence, the GI algorithm puts more emphasis on such small improvements and better explores the search space around those modifications.

Answer to RQ1: State-of-th-art RTS strategies are feasible when used with GI. In our experiments, they showed almost 100% safety, with only one improved software version failing when tested against all test cases. Moreover, such techniques presented positive changes in the improvements obtained by the GI algorithm.

B. Answer to RQ2 – Efficiency

This Section presents the results for the efficiency RQ. Figure 1 depicts the Relative Cost (RC) (Equation 3) of the strategies in relation to GI without RTS, e.g., a RC of 0.4 means that using GI with a given RTS strategy costs 60% less in terms of execution time when compared to the cost of using GI with no RTS. Similarly, Table VI presents the median RC for each strategy and the respective results of the Kruskal-Wallis p-value test in the last column. Finally, Table VII presents the Vargha-Delaney $\hat{A}_{12}$ effect size results for the pairwise comparisons. It is worth restating that the cost includes both overhead and speed-up of the optimisation process.

For six out of seven programs ($\approx$86%), using either Ekstazi or STARTS as RTS strategy showed a significant improvement in execution time (p-value < 0.05). Overall, GI+Ekstazi and GI+STARTS can save up to 68% in relative execution time (22% on average). Moreover, for five out of seven programs ($\approx$71%), this difference is considered statistically large according to the effect size analysis.

The results do not always show a relationship between the number of test cases (Table III) and the relative cost (i.e., the fewer the test cases the lower the cost). This happens because some test cases might be more expensive than others to run. For example, Ekstazi selects considerably fewer test cases for *fileupload*, *imaging*, and *validator*, yet the relative cost to run them is greater than the cost paid by other strategies which select more (but cheaper) test cases for these projects.

We investigated the projects more closely and we found that, projects characteristics play a role. The cost reduction does not seem high when there is a common dependency in the source code (e.g., *csv*) or few methods are tested by multiple test cases (e.g., *text*). In such cases, the RTS strategies end up selecting almost all test cases because they test the improved and highly dependent method, or the improved method is thoroughly tested with expensive test cases. On the other hand, the cost reduction is high when multiple computationally expensive test cases test different parts of the code (e.g., *compress*). In other words, the test cases are more isolated (i.e., test only the unit) and the targeted method only requires a few inexpensive tests.

Figure 2 presents the cumulative cost of the profiling and optimisation phases of each strategy over the 20 independent runs. For programs with expensive test suites (e.g., *commons-compress* and *commons-imaging*), the RTS techniques saved in total from a couple of hours to more than a day of computational time.

It is worth noting that random test case selection during profiling does not add any overhead. However, the selected test cases still incur an execution cost, and the random selection sometimes selects fewer test cases, sometimes selects almost all test cases, hence the differences in results with respect to no test selection.

In our specific case, the total amount of time required to run the experiments without RTS was over 180 CPU hours, whereas with Ekstazi and STARTS this time was reduced to 114 and 118 hours respectively. In other words, using RTS saved us approximately 64 hours of execution time on average (more than a third of the cost). Shorter execution times will allow GI adoption for larger programs, for which the running cost might still be prohibitive and environmentally unfriendly without RTS.

Considering both patch validity and changes in improvement (Section V-A), and the overall efficiency of the strategies, GI+STARTS and GI+Ekstazi are preferred. GI+Random fails more often and does not provide much benefit in improvement and cost savings. Henceforth, we will not discuss the results of GI+Random and will focus our attention on the other two RTS strategies.

Answer to RQ2: Using RTS in combination with GI can significantly reduce the overall relative cost of the optimisation process (up to 68% depending on the program). The reduction

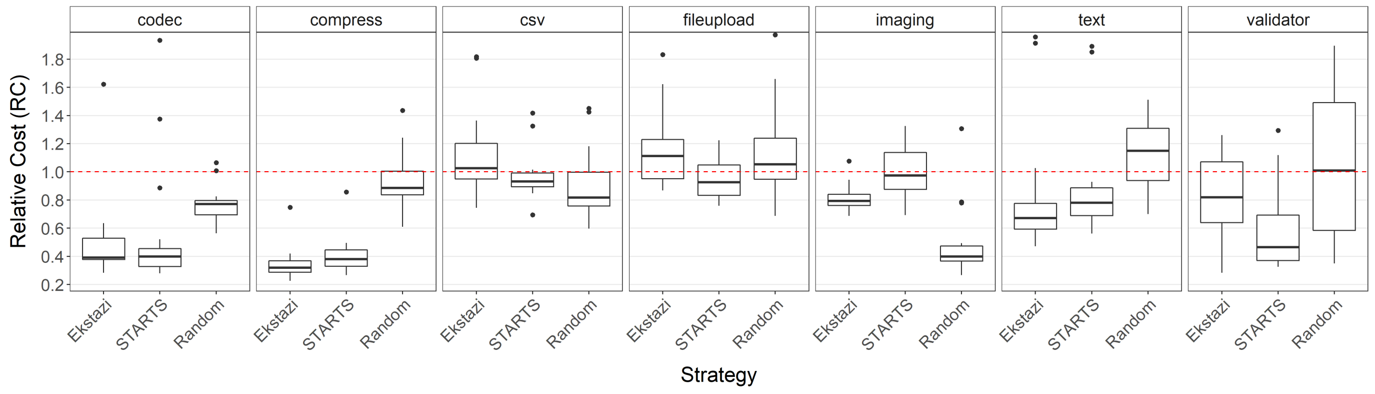

Fig. 1. **RQ2. Relative Cost (RC) of strategies.** Lower values are better. The dashed line represents the median cost of GI without RTS.

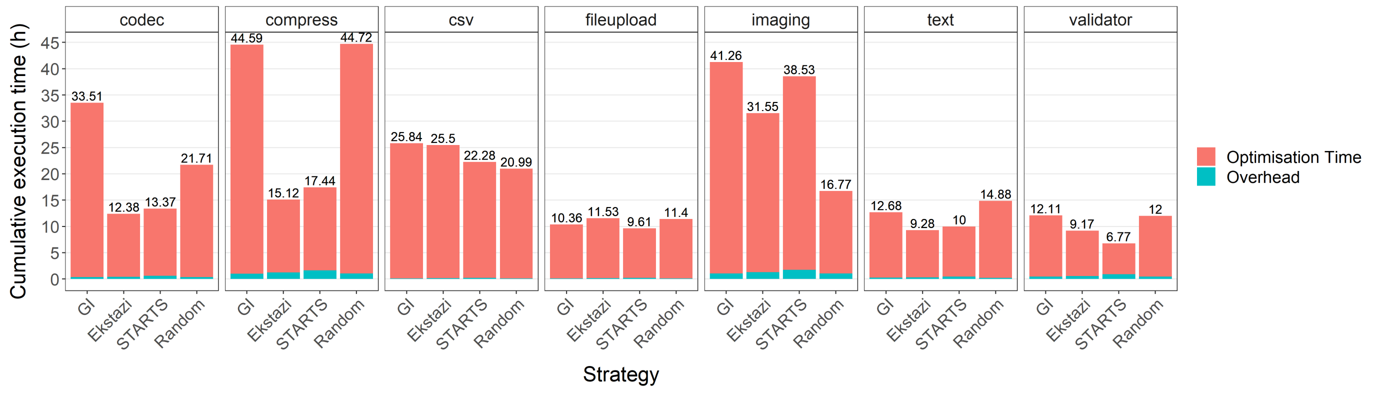

Fig. 2. **RQ2. Cumulative execution times** Results are for four GI strategies i) without RTS, ii) and iii) with RTS, and iv) GI with random test selection.

is statistically significant in 86% of the cases and large in 71% of the cases. Furthermore, when considering the cumulative total cost of experimentation, using RTS can save more than a third of the execution time.

C. Answer to RQ3 – Trade-Off

This section presents the results for the three considered use case scenarios of GI: “perfect improvement” (P_{improv}), “fast improvement” (F_{improv}), and “diverse improvement” D_{improv}.

For P_{improv}, the best strategy in this scenario is the one that is able to find the best valid improvement. Table VIII shows the median best improvement in seconds found for each program. GI+Ekstazi obtained the best improved software in most of the cases (four out of seven), whilst also obtaining the highest median improvement across projects.

Since the RTS strategies are safe when used in combination with GI (with the exception of GI+STARTS for *commons-csv* as shown in Section V-A), then the comparison for the F_{improv} use case is straightforward: the strategy that can find a positive improvement the quickest is preferred. Considering this scenario, Table IX shows how long each strategy took in seconds to find the first positive and valid improvement. On average, GI+STARTS finds an improvement in the first 244 seconds of the search process, and for three out of seven programs it is also the quickest strategy.

TABLE VIII
RQ3. P_{improv} **USE CASE.** MEDIAN IMPROVEMENT FOUND IN SECONDS. GREATER VALUES ARE BETTER.

Program	GI	+Ekstazi	+STARTS
codec	1.29	**4.35**	2.96
compress	2.46	6.23	**14.28**
csv	0.82	2.25	**2.95**
fileupload	0.05	**0.13**	0.11
imaging	11.98	7.63	**12.62**
text	1.46	**3.52**	1.40
validator	0.29	**1.44**	0.82
Median	1.29	**3.52**	2.95

Finally, for the last use case D_{improv}, we are concerned with the diversity of improved software found by the strategies. Table X presents the median number of different valid and positive improvements found by each strategy. For five out of seven programs, GI+Ekstazi finds a wider variety of improvements.

Answer to RQ3: If the engineer is concerned with either finding the perfect improvement or a wider variety of positive and valid improved software, then using GI+Ekstazi is the best option. However, if the main objective is to find a positive and valid improvement as fast as possible, then GI+STARTS is recommended. In most of the cases considered (i.e., 19 out

TABLE IX
RQ3. F_{improv} **USE CASE.** MEDIAN TIME IN SECONDS EACH STRATEGY TOOK TO FIND A POSITIVE AND VALID IMPROVEMENT. LOWER VALUES ARE BETTER.

Program	GI	+Ekstazi	+STARTS
codec	711.32	215.76	**188.19**
compress	730.20	251.14	**244.01**
csv	**388.12**	474.82	425.26
fileupload	287.02	**261.31**	354.38
imaging	768.03	**562.79**	659.11
text	237.69	137.69	**137.38**
validator	131.78	**79.96**	142.68
Median	388.12	251.14	**244.01**

TABLE X
RQ3. D_{improv} **USE CASE.** MEDIAN NUMBER OF POSITIVE AND VALID IMPROVEMENTS FOUND. GREATER VALUES ARE BETTER.

Program	GI	+Ekstazi	+STARTS
codec	6	**16**	9
compress	13.50	28.50	**41.50**
csv	26	**48**	24
fileupload	2	**4**	2
imaging	**27.50**	21	23
text	12	**19.50**	18.50
validator	36	**44**	7
Median	13.5	**21**	18.5

of 21 comparisons) using GI+RTS provides better results than traditional GI without RTS. Therefore, we recommend the use of RTS in future GI work.

VI. Related Work

The work most related to ours is the one that applies regression testing selection techniques to a genetic improvement process. To the best of our knowledge, only the work of Mehne et al. [14] has investigated any kind of RTS in this context, and only for APR. The authors defined their own RTS technique and evaluated the results in terms of speed-up. Their results show a speed-up of up to $1.8\times$ the original cost of automated program repair of C programs using GenProg [2].

Other works use other kinds of regression techniques, such as test case prioritisation and sampling. Venugopal et al. [15] proposed and evaluated a history-based test case prioritisation for APR. The approach consisted in prioritising test cases that are most likely to fail whilst also sampling test cases in order to make the software variant to fail faster. The authors achieved up to 57.5% in execution time savings during patch validation.

Similarly, Qi et al. [13] proposed *TrpAutoRepair*, an on-line prioritisation technique for APR. The idea behind the approach is to avoid the overhead of pre-gathering information about test cases executions, and use information already obtained during the execution of candidate patches. *TrpAutoRepair* was able to perform at least as well as GenProg for 15 out of 16 programs, while significantly improving the repair efficiency, as measured in the number of test case executions.

Fast et al. [25] incorporated random sampling of test cases in the fitness function computation for APR. Their approach first samples a subset of passing test cases and all failing test cases. The software variant is only tested against the remaining test cases if all test cases in the subset pass. The authors also compared their approach to a regression technique for test suite minimisation based on a Genetic Algorithm (GA) [23]. The results showed that their approach is able to reduce the computational effort needed to test patches in 81%.

APR is concerned exclusively with one functional property (i.e., bug fixing), while GI concerns with any functional (e.g., addition of a new feature, bug-fixing) and non-functional improvement (e.g., runtime, energy consumption). Only one previous work [14] uses RTS for GI functional improvement by applying a single ad-hoc RTS dynamic technique. Other works investigate other types of regression techniques, such as test suite minimisation and prioritisation, which are inherently different to RTS.

Our work differentiates from the ones presented in this section since we focus solely on RTS and we consider non-functional improvement (as opposed to APR). State-of-the-art RTS techniques have not been extensively evaluated in the context of GI, specially in regards to non-functional improvement. We are the first to investigate the impact of any test selection strategy in the context of non-functional automated software improvement using GI and with both dynamic and static RTS techniques.

Furthermore, all related work focuses on program repair and uses the same tool, namely GenProg. We note that the GP implementation in Gin also follows the GenProg search strategy. However, unlike previous work, our focus is on Java software.

In this work, through extensive experimentation with real-world Java programs, we evaluate the benefits of RTS along multiple angles: efficiency, effectiveness, and the underlying trade-off between those properties, i.e., we evaluate the safety, efficiency, and improvement impact of RTS. Especially the last angle, to the best of our knowledge, has not been considered before.

VII. Threats to Validity

a) Threats to External Validity: As it happens to most software engineering papers with empirical evaluations, the set of programs used in the experiments might not be representative of the whole population. In order to mitigate this threat, we have selected well-know, non-trivial software projects of different sizes, test times, and coverages. Another threat regards to the fact that we only compared the results of two RTS techniques, one based on dynamic and one based on static analysis. Although there are other techniques in the literature [17], it would be infeasible to compare all of them, thus we decided to compare only state-of-the-art [20].

b) Threats to Internal Validity: To reduce internal threats, we used the same configuration for all experiments. This configuration was used in previous work [2], [32], thus allowing for further experimental comparisons. Moreover, we executed all experiments on the same machine with the same resources.

With these precautions, we intended to prevent other factors to impact the results of our experiments.

c) Threats to Construct Validity: The execution times of optimisation algorithms such as GI may fluctuate due to their stochastic nature. In order to cater for these fluctuations, we performed multiple independent runs and analysed the results with statistical significance tests and effect size analysis, as suggested by Arcuri and Briand [30]. To further mitigate noise in the execution time measurements, we repeated each test case execution 10 times and considered the median as a more accurate data source. Finally, we took extra care when selecting statistical tests to avoid making assumptions that would jeopardise the validity of our results (e.g., we used non-parametrised tests because we could not assume normal distribution of the data).

VIII. Conclusions

Genetic improvement (GI), an Artificial Intelligence technique, has been successfully used to improve various software properties, ranging from reduction of software's runtime [5]–[7], optimisation of energy [10]–[12] and memory [8], [9] consumption, through to bug fixing [2]–[4] and addition of new software features [33]. However, it has yet to see wider uptake. A major obstacle is its high demand on runtime of the GI process to find the desired improvements.

The biggest bottleneck for runtime of a GI process comes from the fitness evaluation, which requires running the whole test suite for each and every evolved program version, in order to ensure that regression bugs are not introduced during the GI process.

Regression Test Selection (RTS), a traditional Software Engineering approach, aims to reduce testing effort by selecting only the most relevant tests for a given task. However, it has not yet been tried in the context of genetic improvement of software.

We investigated the use of two state-of-the-art RTS techniques in an open source GI framework and conducted an extensive study on seven large real-world software to show that these can indeed speed-up the GI process without impacting the validity and improvement gain of the evolved software. In particular, we show that integration of RTS does not hinder the validity of the improved patches. In fact, all evolved software (but one out of 280) when tested against the test cases selected using either RTS approach, always passed the whole test suite, thus showing safety.

We observed efficiency gains of the whole genetic improvement process of up to 68%. Given that the GI process might run for hours, days, or even weeks, such an efficiency gain is non-trivial, impacting not only the execution times of scientific experiments, but also the carbon footprint of the servers used to run GI in the industry [34].

Therefore, we recommend the integration of RTS techniques in test-based automated improvement software. We also recommend the static STARTS RTS technique for finding quick improvements, while we conjecture the dynamic Ekstazi might be best when the improvement gain or software validity is of most concern.

Our proposed software engineering approach will thus reduce the feedback time from the automated software improvement system to the developer. Therefore, we hope it will contribute to wider and faster uptake of GI techniques.

There are still several questions to be answered that could help speed up the process even further. Other test case selection strategies could be investigated. Moreover, it would be useful to know what characteristics a given test suite should have in order to be effective with the GI processes. Initial work has been done in this direction, investigating standard test suite measures [16], [35], yet no strong correlation between test suite metrics and their effectiveness in the improvement process has been found. If known, test case selection for GI could be further improved by applying such metrics by providing further guidance to RTS. This is a direction we would like to investigate in future work.

Acknowledgements

This research is funded by the ERC advanced fellowship grant 741278 (EPIC: Evolutionary Program Improvement Collaborators) and by the EPSRC Fellowship grant EP/P023991/1 (Automated Software Specialisation Using Genetic Improvement).

References

[1] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic Improvement of Software: A Comprehensive Survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2018.

[2] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[3] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 5:1–5:53, 2020.

[4] G. An, A. Blot, J. Petke, and S. Yoo, "Pyggi 2.0: language independent genetic improvement framework," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 1100–1104.

[5] F. de Almeida Farzat, M. de Oliveira Barros, and G. H. Travassos, "Challenges on applying genetic improvement in javascript using a high-performance computer," *J. Softw. Eng. Res. Dev.*, vol. 6, p. 12, 2018.

[6] W. B. Langdon and M. Harman, "Optimising Existing Software with GP," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 1–18, 2015.

[7] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation," *IEEE Transactions on Software Engineering*, vol. 44, no. 6, pp. 574–594, 2017.

[8] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, S. Silva and A. I. Esparcia-Alcázar, Eds. ACM, 2015, pp. 1375–1382.

[9] E. M. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, V. Sarkar and R. Bodík, Eds. ACM, 2013, pp. 317–328.

[10] B. R. Bruce, J. Petke, M. Harman, and E. T. Barr, "Approximate oracles and synergy in software energy search spaces," *IEEE Trans. Software Eng.*, vol. 45, no. 11, pp. 1150–1169, 2019.

[11] N. Burles, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, J. Swan, and N. Veerapen, "Object-oriented genetic improvement for improved energy consumption in google guava," in *Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*, ser. Lecture Notes in Computer Science, M. de Oliveira Barros and Y. Labiche, Eds., vol. 9275. Springer, 2015, pp. 255–261.

[12] E. M. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 639–652.

[13] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 180–189.

[14] B. Mehne, H. Yoshida, M. R. Prasad, K. Sen, D. Gopinath, and S. Khurshid, "Accelerating Search-Based Program Repair," in *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 227–238.

[15] Y. Venugopal, P. Quang-Ngoc, and L. Eunseok, "Modification point aware test prioritization and sampling to improve patch validation in automatic program repair," *Applied Sciences (Switzerland)*, vol. 10, no. 5, pp. 1–14, 2020.

[16] M. Lim, G. Guizzo, and J. Petke, "Impact of Test Suite Coverage on Overfitting in Genetic Improvement of Software," in *Proceedings of the Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2020.

[17] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[18] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 211–222.

[19] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A Framework for Checking Regression Test Selection Tools," in *Proceedings of the 2019 41st International Conference on Software Engineering (ICSE)*, vol. 2019-May, 2019, pp. 430–441.

[20] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM Press, 2016, pp. 583–594.

[21] O. Legunsen, A. Shi, and D. Marinov, "STARTS: STAtic regression test selection," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 949–954.

[22] L. Chen and L. Zhang, "Speeding up Mutation Testing via Regression Test Selection: An Extensive Study," in *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 58–69.

[23] M. Gendreau and J.-Y. Potvin, *Handbook of Metaheuristics*, 3rd ed. Springer, 2019.

[24] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7007 LNCS, pp. 1–59, 2011.

[25] E. Fast, C. L. Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference (GECCO)*, 2010, pp. 965–972.

[26] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: Genetic Improvement Research Made Easy," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. New York, NY, USA: ACM, 2019, pp. 985–993.

[27] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[28] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[29] C. A. C. Coello, "Multi-objective Optimization," in *Handbook of Heuristics*. Springer International Publishing, 2018, pp. 1–28.

[30] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[31] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[32] Z. Y. Ding, Y. Lyu, C. S. Timperley, and C. L. Goues, "Leveraging program invariants to promote population diversity in search-based automatic program repair," in *Proceedings of the 6th International Workshop on Genetic Improvement, GI@ICSE 2019, Montreal, Quebec, Canada, May 28, 2019*, J. Petke, S. H. Tan, W. B. Langdon, and W. Weimer, Eds. ACM, 2019, pp. 2–9.

[33] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 257–269.

[34] C. Calero and M. Piattini, *Green in Software Engineering*. Springer Publishing Company, Incorporated, 2015.

[35] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2948–2979, 2018.