Speeding Up Genetic Improvement via Regression Test Selection

GIOVANI GUIZZO, DAVID WILLIAMS, MARK HARMAN, JUSTYNA PETKE, and FEDERICA SARRO, University College London, London, United Kingdom

Genetic Improvement (GI) uses search-based optimisation algorithms to automatically improve software with respect to both functional and non-functional properties. Our previous work showed that Regression Test Selection (RTS) can help speed up the use of GI and enhance the overall results while not affecting the software system's validity. This article expands upon our investigation by answering further questions about safety and applying a GI algorithm based on Local Search (LS) in addition to the previously explored Genetic Programming (GP) approach. Further, we extend the number of subjects to 12 by analysing five larger real-world open-source programs. We empirically compare two state-of-the-art RTS techniques combined with GP and LS for these 12 programs. The results show that both RTS techniques are safe to use and can reduce the cost of GI by up to 80% and by 31% on average across programs. We also observe that both search-based algorithms impact the effectiveness gains of GI differently, and that various RTS strategies achieve differing gains in terms of efficiency. These results serve as further evidence that RTS must be used as a core component of the GI search process to maximise its effectiveness and efficiency.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Software performance*; **Genetic programming**;

Additional Key Words and Phrases: Genetic Improvement, Regression Test Selection, Search-Based Software Engineering

ACM Reference format:
Giovani Guizzo, David Williams, Mark Harman, Justyna Petke, and Federica Sarro. 2024. Speeding Up Genetic Improvement via Regression Test Selection. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 196 (November 2024), 31 pages.
https://doi.org/10.1145/3680466

1 Introduction

Genetic Improvement (GI) involves using search-based techniques to automatically improve existing software properties [32]. The properties under improvement can be functional (e.g., bug

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). For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to any accepted manuscript version arising.

Authors' Contact Information: Giovani Guizzo, University College London, London, United Kingdom; e-mail: g.guizzo@ucl.ac.uk; David Williams, University College London, London, United Kingdom; e-mail: david.williams.22@ucl.ac.uk; Mark Harman, University College London, London, United Kingdom; e-mail: mark.harman@ucl.ac.uk; Justyna Petke, University College London, London, United Kingdom; e-mail: j.petke@ucl.ac.uk; Federica Sarro (Corresponding author), University College London, London, United Kingdom; e-mail: f.sarro@ucl.ac.uk.

ACM 1557-7392/2024/11-ART196
https://doi.org/10.1145/3680466

fixing [1, 24, 41, 46]) or non-functional (e.g., runtime [8, 23, 33], memory usage [35, 43] and energy consumption [5, 6, 36]). The level of improvement of a property is measured by a fitness function, which guides the search towards better software over multiple iterations. At each iteration, the GI technique generates multiple software variants potentially better than the original software w.r.t. the fitness function, but must preserve the desired software behaviour (i.e., pass the tests in the software's test suite). If a software variant fails any functional tests, it is deemed invalid.

Despite GI's appealing benefit of automatically improving software properties, as one can infer, the process of generating and testing many variants of a given software is computationally expensive [8, 29, 34, 40], especially when the software has a costly test suite. Even for programs with relatively small test suites, GI executions can take many hours or even days of computation [28]. One solution for this high cost is to select and execute only a subset of test cases relevant to the modifications made in the variant instead of executing the entire test suite. We have shown in a preliminary work [16] that this can be achieved with the use of well-established **Regression Test Selection (RTS)** techniques [45].

RTS has been extensively studied in the **Software Engineering (SE)** literature [45], with the primary purpose of selecting subsets of tests from a test suite to allow for a more efficient regression test process. Such techniques typically determine dependencies for tests (e.g., based on which parts of the source code they reach) and focus on selecting "affected tests," or tests dependent on the changes made in the program's latest revision. RTS differs from Regression Test Minimisation (which permanently removes redundant test cases from the test suite) and Regression Test Prioritisation (which defines a test case order for testing) because it selects a subset of test cases for the imminent testing task based on the context of the changes.

In our previous work [16], we hypothesised that existing RTS techniques can be powerful assets for improving the effectiveness and efficiency of the whole GI process, and we carried out an empirical investigation on the effects of RTS on the non-functional GI process by considering various contexts and tradeoffs in different scenarios. Our experiments consisted of three RTS techniques (one random, one dynamic and one static technique), one GI algorithm based on **Genetic Programming (GP)** [20], and seven real-world open-source projects from the Apache Commons suite.[1] Such investigation was crucial because, until then, we did not know the effects RTS had on the effectiveness (i.e., to which extent it affected the capability of GI in finding better variants), efficiency (i.e., to which extent it reduced the cost of GI more than it introduced overhead) and safety (i.e., to which extent the generated variants deemed valid would still be valid when tested against the whole test suite) of GI. Our previous work showed that RTS is not only safe, but can even significantly speed up the overall GI process by up to 68%. Surprisingly enough, speeding up the GI execution also led to better-improved software variants because the algorithm could spend the spared resources on finding new variants rather than executing unrelated tests.

However, our previous work did not address the usage of various GI algorithms, opting only to consider GP in those experiments. Depending on the type of algorithm used to search for improved software variants, different RTS techniques may be more suitable or display different outcomes. Moreover, although the Apache Commons suite comprises many well-known and large software, there is still room for improvement in the generalisability of our findings. In order to close these research gaps, in this article, we extend our previous work by including **Local Search (LS)** [4, 13, 19] as an additional GI algorithm as well as five larger open-source programs, making a total of 12 real-world software investigated herein. This work does not only extend the type and number

[1] https://commons.apache.org/

of algorithms and programs, but also enriches our previous set of **Research Questions (RQs)** with new sub-questions, which have been designed to comprehensively tackle all aspects involved with the multiple RTS techniques and GI algorithms regarding safety, effectiveness and efficiency, as well as tradeoffs encountered in the GI process. By answering these questions, we aim to present a more detailed overview of the benefits and drawbacks of each RTS technique in the context of non-functional GI using different types of search algorithms.

Our results show that both state-of-the-art RTS techniques we employed (Ekstazi [14] for dynamic RTS and STARTS [27] for static RTS) are safe to use in conjunction with GI. Ekstazi successfully selected all fault-revealing test cases in all of its 480 independent GI executions, while STARTS neglected a fault-revealing test case only once out of its 440 runs. Considering effectiveness, roughly half of the time (54.4%), GI with RTS maintained the same level of program variant runtime improvement as GI without RTS. Further, RTS even provided a benefit (a better level of improvement relative to that of the average variant generated without RTS) in 30.4% of cases. However, its impact on GI efficiency is where RTS truly shone in our results, where we found that RTS could reduce the cost of the entire process by up to 80% for some programs, providing a significant speed up in 83.3% of cases. Ekstazi provided a median execution time reduction of 31%, whereas that of STARTS was only 6%. Across our entire experimentation, the cost of GI+Ekstazi was less than half of GI, saving over six weeks (or 1,000 hours) of computational resources. This observation, combined with our safety and effectiveness findings, makes us believe that RTS is essential for a more sustainable GI. Additionally, we noted which factors influence GI performance more significantly when analysing impacts on effectiveness and efficiency. For instance, we found that effectiveness results depended more on the search algorithm in use, while switching RTS techniques had the greatest impact on efficiency. Finally, our work presents the tradeoffs of adopting each algorithm-RTS combination. While we determined that GI+Ekstazi using GP was our recommendation as the most balanced in tradeoffs for most scenarios, we provide insights allowing engineers to choose which combination would best suit their needs based on priorities such as best improvement, fastest improvement and diversity of program variants generated by the GI process.

In summary, the main contributions of this paper are:

- Large-scale experimentation with 12 open-source software, three RTS techniques and two state-of-the-art GI search algorithms.
- Comprehensive quantitative and qualitative result analysis, comparing algorithms and RTS techniques with three different metrics and two statistical tests.
- Answers to multiple RQs designed to evaluate the impact of RTS on GI from many application angles.
- Provision of our GI and RTS source code as an open-source software available at: https://github.com/gintool/gin.
- Provision of a replication package, available at: https://figshare.com/s/52a5092425c64648467e.

2 Background

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

2.1 RTS

Regression Testing concerns assessing whether the software's pre-existing behaviour is impaired by a given change [45]. Conventionally, the software is tested against its entire test suite whenever a new change is performed. However, as the test suite grows in complexity and size, the cost of such re-testing becomes infeasible. To avoid the re-execution of the whole test suite and consequently speed up the regression testing process, researchers have proposed many regression testing strategies [45]. Among these strategies, the most common are test suite minimisation, test case prioritisation and test case selection.

Test suite minimisation aims to permanently remove irrelevant, redundant, or obsolete test cases from the test suite. Test case prioritisation focuses on re-ordering the test cases during their execution such that faults are detected earlier in the testing process. Finally, RTS techniques select only a subset of test cases to execute when a new change to the software is performed. The main goal of RTS is to avoid the execution of test cases that are unable to reveal faults in the modified code. Unlike test suite minimisation and test case prioritisation, RTS relies on information about the changes made between software versions (or variants, in the context of this work). As this work focuses on RTS, the following presents it in more detail.

According to Yoo and Harman [45], an RTS technique must select a subset of test cases T' from the whole test suite T that contains all available test cases able to reveal faults in a given program variant p' of the original program p. A test case t is fault revealing in relation to p and p' if t yields different outputs for both versions ($t(p) \neq t(p')$), meaning t needs to be executed against both variants. Assuming that $t(p)$ halted and produced the correct result, t will only be able to reveal a fault in p' if t traverses the modified code of p' or used to traverse a now deleted piece of code in p'. Therefore, if an RTS technique selects all available test cases in T that traverse modifications in p', then such a technique is called *safe*. In other words, an RTS technique must avoid the execution of irrelevant test cases in relation to the modifications performed in the software. In the context of this work, a "relevant" test case should (i) traverse the modified code, (ii) create a state of error in the **System Under Test (SUT)** and (iii) reveal this erroneous state as a failed assertion. A test case may be considered irrelevant if it fails to meet any of these three conditions. However, due to the difficulty of checking the second and third conditions, RTS tools typically only select tests based on the first requirement. Thus, the RTS techniques used in this work select affected tests (those transitively dependent on the modified code) rather than relevant ones.

In this article, we employ three RTS techniques: a random selection technique, a modification-based technique using dynamic analysis, and a firewall approach based on static analysis [45]. The random technique selects test cases at random without any additional information and is only used as a baseline. The dynamic approach is implemented by the Ekstazi tool [14], while the firewall one is implemented by the STARTS tool [27]. Ekstazi implements a three-phase process to select affected test cases. The first phase (analysis) involves discarding unaffected tests by considering each test class and seeing whether all of its dependent classes' checksums have remained the same since the previous execution. If a test class meets this condition, it is not selected for testing. The second phase (execution) runs the remaining tests. The third phase (collection) can be done in parallel to the second or sequentially at a later point, and involves instrumenting the bytecode and recording which classes are accessed when each test case is executed. Ekstazi computes the checksum for the list of files associated with each test class and stores it alongside the dependencies for the next analysis phase.

Conversely, STARTS [27] employs a static approach for RTS based on the concept of class firewalls. A class firewall for a given class C is the set of classes that could be affected if C is modified [22]. STARTS computes class firewalls using a **Type-Dependency Graph (TDG)**, delimiting which types need to be retested after a code change. Type dependencies are determined using the constant pool for each classfile, and the TDGs are constructed from this data in a type-to-test dependency file. To select impacted tests, STARTS uses the same checksum function as Ekstazi to check whether any types have been modified and returns the set difference between the latest test suite and the set of test cases *not* associated with the changed types. Similarly to Ekstazi, the TDG computation phase for the following execution can be run either in parallel to the other steps or later.

We selected these tools for our experiments as both have been extensively evaluated in literature [7, 16, 26, 37, 38, 47] and have shown to produce safe results with low execution costs. For example, Chen and Zhang [7] used RTS tools to speed-up Mutation Testing and have shown that both tools are able to select the appropriate test cases for a given mutant. In our preliminary work [16], we followed this intuition that RTS can be used for more than just Regression Testing. We applied RTS within the GI process and obtained conclusive evidence of the benefits it can provide in such a context.

2.2 GI and Efficiency

GI consists of using search-based techniques to improve the properties of an existing software [32]. Search-based algorithms use intelligent heuristics to search for approximate solutions for a given problem when the exact optimum solution cannot be found in a feasible time [13]. GI is part of the more general field of Search-Based SE [18], which aims to solve hard SE problems through the application of search algorithms.

During the GI optimisation process, a search algorithm searches for software transformations (i.e., patches) that can improve a set of functional or non-functional software properties. Functional improvement is usually associated with **Automated Program Repair (APR)** [12, 29, 34], which consists of, as the name implies, searching for transformations that can repair faults in a program. Non-functional improvement, on the other hand, focuses on finding transformations that maintain the functional behaviour of the software while also improving properties such as memory usage [35, 43], execution time [8, 23, 33], energy consumption [5, 6, 36] and others. In both cases, the functional behaviour of software is measured by the test suite, i.e., if all test cases pass after the transformation, then the GI algorithm assumes that the existing behaviour is maintained. Test case execution is also used as a source of information for the non-functional properties, e.g., test case execution time being used as a measure of runtime improvement. All of this information is incorporated into a “fitness function” that guides the search process towards solutions that are more fit for solving the given problem.

In non-functional GI, each iteration results in a set of patches, which are then used to attain new software variants that are potentially better than the original w.r.t. fitness function measurement. As previously mentioned, such variants are executed against the test suite to gather execution information and compute the fitness. Since the search process is stochastic and based mainly on trial and error (a variant can be worse than the original program), the search for software variants is performed throughout numerous iterations, each imposing the cost of executing the test suite against the candidate solutions. As one can infer, the cost of such a process quickly becomes prohibitive, especially when the test suite is computationally expensive [16, 29, 34].

In order to speed up the GI process, some tools [4] already implement a few strategies, such as in-memory compilation that removes the writing and reading of source files before compiling. Another common strategy is to specifically target only relevant methods and classes, e.g., APR tools usually trace failing test cases and only focus on repairing classes that the test cases reached. On the other hand, non-functional GI tools perform profiling in a preliminary step to find costly pieces of code to improve. Nevertheless, such tools still can take several hours or days of execution, even for relatively small programs [25, 28].

As shown in our previous work [16], selecting fewer test cases during the evolutionary process is a viable way of further reducing the cost of evaluating programs. In this study, we performed an initial evaluation of the impact of RTS on multiple GI aspects, such as effectiveness, efficiency,

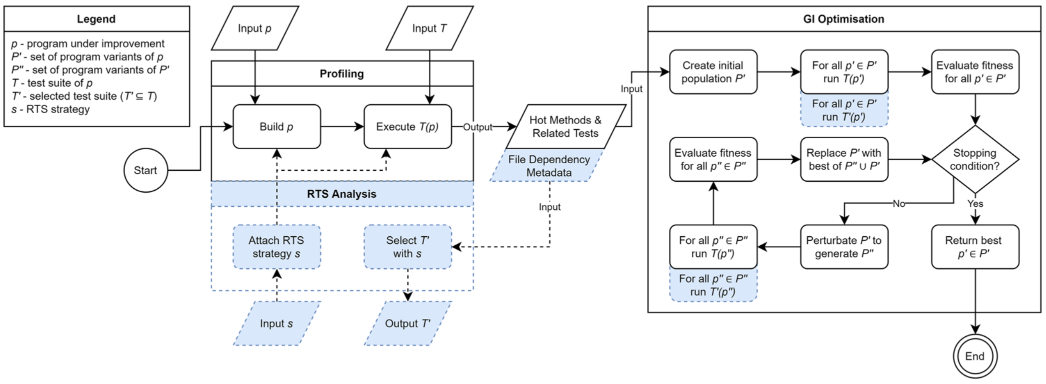
Legend
p - program under improvement
P' - set of program variants of p
P'' - set of program variants of P'
T - test suite of p
T' - selected test suite (T' ⊆ T)
s - RTS strategy
Input p
Input T
Profiling
Build p
Execute T(p)
Output
Start
RTS Analysis
Attach RTS strategy s
Select T' with s
Input s
Output T'
Hot Methods & Related Tests
File Dependency Metadata
Input
Input
GI Optimisation
Create initial population P'
For all p' ∈ P' run T(p')
For all p' ∈ P' run T'(p')
Evaluate fitness for all p' ∈ P'
Stopping condition?
Evaluate fitness for all p'' ∈ P''
Replace P' with best of P'' ∪ P'
No
Yes
For all p'' ∈ P'' run T(p'')
For all p'' ∈ P'' run T'(p'')
Perturbate P' to generate P''
Return best p' ∈ P'
End

Fig. 1. GI process of the Gin tool and proposed steps to allow the usage of RTS.

safety and the tradeoff an engineer has to face when choosing an RTS technique. However, there are still a few unanswered questions in this context, such as:

(1) How do RTS techniques behave with different types of GI algorithms?
(2) Which GI algorithms can benefit the most from RTS cost reduction?
(3) Which GI algorithms provide the most cost-effective search when using RTS?

The following section describes how we enhanced GI to use RTS and presents the main challenges and benefits.

3 Proposed GI with RTS Approach

The findings of our previous study [16] indicated that incorporating RTS in the GI process has the potential to dramatically reduce computational resource requirements, saving more than a third of the execution time during experiments compared to GI without RTS (from 180 hours to 116 hours). We adopt a similar approach in this work, intending to further investigate GI performance with RTS and which factors are most impactful. For our experiments, we use Gin [4], a GI tool for the improvement of Java programs. Gin has two main phases: profiling and optimisation. Figure 1 presents the general process of applying Gin with all its phases and which additional features we implemented to enable the usage of RTS. Essentially, the RTS techniques are incorporated as part of Gin's profiling phase, allowing the RTS tools to determine which tests are relevant to execute when targeting different methods for non-functional improvement. Thus, when Gin generates and develops program variants, these are evaluated using the subset of test cases selected by the RTS tools (as opposed to the entire test suite). For an in-depth description of Gin's approach to GI without RTS, please refer to its introductory paper by Brownlee et al. [4]. The following subsections will describe each of the major phases in Gin and how these phases are affected by the integration of RTS strategies.

3.1 Profiling

In the initial phase, profiling, the main goal is to capture execution information about test cases and tested methods from the unmodified program p. This is a one-time operation where p is compiled, and the entire test suite T is executed. During its execution, Gin uses either **Java Flight Recording (JFR)** or hprof (used in this work) to sample which methods are executed during testing. Specifically, Gin records each tested class and method, which tests were used for profiling, the number of times a method was executed by the tests, and the execution time of the entire profiling procedure. From this sampling, it generates a list of methods and assigns a list of associated test cases to each method. This execution information is stored in a CSV file for use in the second phase (optimisation). Further, Gin identifies "hot methods," i.e., costlier methods in terms of execution time that are more likely to benefit from the GI optimisation. Without RTS, the resulting CSV file contains a list of all test cases related to all methods identified by JFR or hprof.

Since most RTS techniques [14, 27] require a complete execution of the test suite T before performing test case selection, we incorporate the RTS techniques during Gin's profiling phase. For Ekstazi (dynamic RTS), the tool's Java agent is incorporated during the testing phase of Gin's profiling. This allows Ekstazi to collect detailed information from the test case execution traces, but also makes it compulsory to execute the entire test suite at least once during profiling. In the case of STARTS (static RTS), Gin includes the Abstract Syntax Tree analysis during build time. Unlike Ekstazi, STARTS using static analysis means test suite execution during the profiling phase is not required. Since both RTS tools generate their output in proprietary formats, Gin captures their results and parses them to a standardised format (identical to that of the CSV files generated without RTS) before concluding the profiling phase. The result is a reduced test set $T' \subseteq T$ for each hot method, specifically tailored to avoid the execution of unaffected test cases that would not be able to reveal faults in a method, given that it is modified during the GI process. This information about the combination of hot methods-test sets replaces the original trace information of Gin. It should be noted that Ekstazi and STARTS have different levels of granularity than Gin's profiling phase when selecting test cases. Gin functions at a method level, while STARTS and Ekstazi can only associate test cases at a class level. Gin handles this mismatch in granularity conservatively by assigning all tests selected for a given class to each of its methods.

3.2 Optimisation

The second phase, optimisation, consists of the actual GI process, where a given algorithm searches for program variants (P'). Gin focuses on optimising one method at a time, usually the costliest one identified in the profiling phase.

The first step is to generate a random initial population P' of program variants (solutions). Gin works with a patch representation (chromosome) for the solutions, i.e., it searches for patches that modify the original program p to transform it into a potentially improving variant $p' \in P'$. The patch is represented by a sequence of edits (genes) containing the edit operation and targeted code statement. The available edit types are: (i) Delete—deletes a statement; (ii) Copy—copies a statement; (iii) Replace—replaces a statement with another existing one and (iv) Swap—swaps two existing statements. Hence, each initial solution p' consists of a patch with a single random edit to the original program.

The second step is to run the test suite T on all solutions $p' \in P'$. Since a patch cannot be executed against the test suite, Gin first applies the patch to the code, performs an in-memory compilation of the modified source code, and then executes the test suite against the modified program. If an RTS technique is used during the optimisation phase, instead of running all test cases in T, Gin uses the reduced test set T' selected specifically for the method under improvement. The results of the test case execution are stored and used in the next step, the fitness evaluation.

Since we focus on the non-functional improvement of runtime execution, the fitness function is the difference in execution time between the original program p and the program variant p' using the test suite T

$$\uparrow fitness(p', T) = runtime(T(p)) - runtime(T(p')). \tag{1}$$

When an RTS technique is being used, Gin uses the runtime of the reduced test set T' to perform the fitness evaluation

$$\uparrow fitness\_rts(p', T') = runtime(T'(p)) - runtime(T'(p')). \tag{2}$$

The original program is also tested against T' as it would be misleading to compare the runtime of the entire test suite T for p and that of T' for p'. The objective of the optimisation is to find the variant p' that maximises this function. If the program variant p' does not pass the test cases (i.e., it contains a fault due to the patch applied to the code), Gin assigns it the maximum possible value (Double.MAX_VALUE) as its execution time, resulting in very low fitness value for p' according to Equation (2). This mechanism guides the GI algorithm towards programs that maintain the functional behaviour of the software since a fast but faulty program is undesirable in the context of non-functional GI. Additionally, if Gin is unable to find any test-passing variants, it simply outputs the original program.

Next, Gin starts the iterations (generations) and continues until the stopping criterion is met, which is a fixed number of generations set as a parameter by the engineer. In each generation, the first step is to perturbate the solutions in the population P' and generate a new offspring population P''. This step is where the search algorithms implemented in Gin differ.

The GP [20] algorithm is based on **Genetic Algorithms (GA)** [9] and performs two perturbations, crossover and mutation, to generate an offspring population P'' of size $|P'|$. Gin uses its own version of the Uniform Crossover operator [15] to generate new solutions. First, it selects two parent solutions p'_1 and p'_2 from P'. Then each edit (gene) has a given probability $0 < \alpha \leq 1$ of being copied from p'_1 to the first child p''_1 and from p'_2 to the second child p''_2, in the order they appear in the parents. Then, the edits of the other parent are copied to the other child with the same probability α. After performing the crossover, Gin applies the mutation operator with a given probability $0 < \beta \leq 1$ to the children P'' to introduce new random edits. The goal of the crossover operator is to carry genetic information from the parents to the children, whereas the mutation operator is used to introduce diversity into the population.

Unlike GA, the LS algorithm [4, 13, 19] does not use crossover, but rather a more simplistic approach to generate an offspring population P'' of size one. Starting from the best solution $p' \in P'$ found so far, it mutates p' to generate a single solution p''. It works by searching for “neighbour” solutions p'' of the best one p', instead of generating offspring from the global population P', hence the name “local”. The advantage is a more exploitative strategy, which can be effective in complex problems such as non-functional GI.

Regardless of how the solutions are generated, whether with GP or LS, the next step is to execute the test cases on the newly generated offspring population P''. Similarly to the initial population evaluation, each solution $p'' \in P''$ is executed against the entire test suite T or against the test case subset T' when an RTS technique is used. The execution information is then used to compute the fitness of all $p'' \in P''$. After the fitness evaluation, the replacement operation occurs, which joins both P' and P'' and selects the best solutions in the union to survive and become parents (P') in the next generation. When the stopping condition is met, Gin outputs the best variant p' w.r.t. the fitness function.

3.3 Validation Phase

When using RTS, one must perform an additional procedure in order to validate whether the RTS technique discarded important tests that could reveal faults in the program variants. In other words, since we only use a subset of test cases T' of T during the optimisation phase, it may be the case that a program variant deemed as valid during optimisation (passes T') is in fact faulty (does not pass T). To achieve this, we re-execute all the valid program variants found during optimisation against the entire test suite T and record the results. During this step, we also record the real improvement obtained in relation to T and the execution time needed to perform this task.

In this article, we conduct extensive experimentation to assess how feasible our proposed approach is, taking into account the safety and other effects RTS techniques may have in the context of GI. The following section details the RQs used to guide the experimentation of our work.

4 RQs

The experiments of this work are designed to answer the following RQs:

- *RQ1. Safety:* How safe is RTS in the context of non-functional GI?
  RQ1.1. RTS Comparison: How safe are different RTS techniques?
  RQ1.2. Algorithm Comparison: How does RTS technique safety impact the output of different GI algorithms?
- *RQ2. Effectiveness:* How does the use of RTS impact the effectiveness of non-functional GI?
  RQ2.1. RTS Comparison: How is GI effectiveness affected by different RTS techniques?
  RQ2.2. Algorithm Comparison: How do different GI algorithms benefit from RTS in terms of effectiveness?
- *RQ3. Efficiency:* What is the efficiency gain when using RTS with GI?
  RQ3.1. RTS Comparison: How is GI efficiency affected by different RTS techniques?
  RQ3.2. Algorithm Comparison: How do different GI algorithms benefit from RTS in terms of efficiency?
- *RQ4. Tradeoff:* What is the tradeoff between efficiency and effectiveness of the GI process with various RTS strategies in different application scenarios?

Since we use multiple algorithms and RTS techniques in our experimentation, we also aim to analyse the different application contexts in more depth. Therefore, for all RQs, we create sub-RQs to compare the algorithms' performance and whether these results vary with different combinations of RTS and algorithm types. Furthermore, we analyse the results in terms of software variant runtime improvement. Thus, non-functional GI is hereby defined as GI for runtime improvement during software execution. In the following subsections, we describe and motivate each RQ in the context of our work.

4.1 RQ1—Safety

How Safe is RTS in the Context of Non-Functional GI?—This question is designed to evaluate the safety of the RTS techniques when used in conjunction with GI. As defined in Section 2, an RTS technique is considered "safe" if it selects all the test cases from the test suite that can reveal the potential faults in the modified program. In this context, we are concerned that the RTS techniques may discard affected test cases that could reveal a fault in a given program variant. If this is the case, the GI algorithm will wrongfully deem faulty program variants as functionally adequate due to the RTS technique failing to select important test cases, rendering its application infeasible in practice. Thus, after finishing the evolutionary process of GI with the reduced test set, we execute the best obtained variants against the entire test suite to check for faults, as described in Section 3.3.

It is important to note that, although the test suite is a crucial indicator of the presence of faults in a given program, it cannot prove the absence of faults, i.e., the program's correctness [10]. This is discussed further in Section 7. Hence, herein, safety is defined as the adequacy of the program w.r.t. the expected behaviour represented by the entire test suite, rather than the correctness of the program. This concept of adequacy also extends to the use of the term "valid" throughout this text. A given program variant p' is valid if it passes all test cases in the original program p test suite T.

In order to measure the level of safety of a given RTS technique, we define the **"Relative Safety" (RS)** measure as follows:

$$\uparrow \mathrm{RS}(p', T) = \frac{|passing(T(p'))|}{|T|} \tag{3}$$

where $|T|$ is the number of test cases in the test suite T of a program p; and $|passing(T(p'))|$ is the number of passing test cases in T when executed against a given program variant p' of p. In other words, this measure calculates the percentage of test cases in the entire test suite T that pass when executed against a given program variant p'. Therefore, the greater the RS, the safer the technique is.

Such variant p' is obtained from the GI algorithm execution, which is the best (greatest fitness) adequate variant (passes all test cases selected by the RTS technique) found during the evolutionary process. The only (reasonable) assumption for RS is that all tests in T pass when executed against p. This is a common pre-requisite in non-functional GI [4, 16], since the GI algorithm may not yield valid variants if the original program is faulty.

In order to answer sub-RQs 1.1 (how safe are different RTS techniques?) and 1.2 (how does RTS technique safety impact the output of different GI algorithms?), we compare the results by RTS strategy and by algorithm used respectively. The objective is to unveil differences in the impact of RTS by the multiple techniques used in our experiments. We answer each sub-RQ individually alongside the more general RQ answer.

4.2 RQ2—Effectiveness

How Does the use of RTS Impact the Effectiveness of Non-Functional GI?—We further analyse the experiment results to unveil the potential effects of using RTS on the improvement capabilities of GI. In other words, we want to analyse to what extent the RTS techniques affect the final non-functional improvement in runtime obtained by the GI algorithms. Since the RTS techniques aim to speed up the evolutionary process, and the fitness function precisely measures the speed up of program variants, we expect to find results significantly different from the conventional GI process without RTS. To this end, we define the **"Relative Improvement Change" (RIC)** measure as follows:

$$\uparrow \mathrm{RIC}(p', T) = \frac{runtime\_improvement(T(p', p))}{runtime\_improvement\_avg(T(P'', p))}, \tag{4}$$

where $runtime\_improvement(T(p', p))$ is the runtime improvement (fitness value) of a valid variant p' obtained by GI with RTS compared to the runtime of the original program p; P'' is the set of all valid variants obtained from the GI algorithm without RTS; and $runtime\_improvement\_avg$ $(T(P'', p))$ is the average runtime improvement relative to p for all variants p'' in P''

$$runtime\_improvement\_avg(T(P'', p)) = \frac{runtime\_improvement(T(p'', p))}{|P''|}. \tag{5}$$

This average improvement acts as a reference point to determine how RTS affects the (runtime) performance of program variants obtained through GI. Thus, the greater the RIC value, the better, and for a given variant p', an RIC value larger than one indicates p' performs better than a typical variant computed using GI without RTS. All improvements are computed with the whole test suite T.

In summary, RIC measures the proportional improvement gain of a given variant p' compared to the average improvement gain without RTS. Therefore, if RIC > 1.0, it means that the variant p' obtained using GI with RTS has a better level of improvement on average than when not using RTS. If this is the case, then using RTS enhances the improvement capabilities of GI. If RIC < 1.0, using RTS negatively impacts such capabilities.

Similarly to RQ1, we perform additional comparisons to answer sub-RQs 2.1 (how is GI effectiveness affected by different RTS techniques?) and 2.2 (how do different GI algorithms benefit from RTS in terms of effectiveness?). The objective is to check whether any RTS strategy (Ekstazi/STARTS) or search algorithm (LS/GP) is more effective than their counterparts in the same setting.

4.3 RQ3—Efficiency

What is the Efficiency Gain When Using RTS with GI?—By answering this question, we intend to quantify the speed-up gained from using the RTS techniques when executing two different GI algorithms, namely LS and GP. As opposed to RQ2, this RQ focuses on the speed-up of the GI process itself rather than that of the software variants obtained by the GI algorithm.

Although running fewer tests during the search process will intuitively provide some speed up, we want to analyse whether the cost of the additional profiling steps (i.e., collecting the dependencies between source files and test cases and performing test case selection for each identified hot method) incurred by the RTS tools make their usage infeasible and, if not, what is the resulting speed up when taking this added cost into account. In the context of our approach to non-functional GI with RTS (outlined in Section 3), we refer to the cost of these additional profiling steps as the overhead of using the RTS techniques. These steps become increasingly complex as program and test suite size increases, and the overhead may not be trivial when the program under improvement is accompanied by many test cases and source files.

To evaluate the cost of the GI process with RTS, we used the **"Relative Cost" (RC)** metric, which is defined as follows:

$$\downarrow \mathrm{RC}(s,p) = \frac{cost(s(p)) + profiling(s(p))}{average\_original\_cost(p)}, \tag{6}$$

where $cost(s(p))$ is the total execution time of the GI algorithm when using a given RTS strategy s to improve program p; $profiling(s(p))$ is the cost of the profiling phase using strategy s when applied to p; and $average\_original\_cost(p)$ is the averaged cost across all runs when applying the same GI algorithm without RTS on p. As RIC (Section 4.2) is represented in relation to GI without RTS, a similar normalisation is applied for RC. In summary, RC measures the cost (execution time) of the GI algorithm using a given RTS strategy s relative to the same algorithm without RTS. Thus, the lower the RC, the greater the benefit RTS provides to the GI process execution time.

If the RC value is greater than 1.0, then the RTS technique does not speed up the process; rather, it introduces more costs during the profiling phase than time saved during optimisation, making it infeasible to use in practice. However, if the result of RC is lower than 1.0, then it is possible to quantify the speed-up obtained by the strategy. For example, an RC of 0.5 means that the GI process is twice as fast with RTS.

Similarly to RQs 1 and 2, we perform additional comparisons in order to answer sub-RQs 3.1 (how is GI efficiency affected by different RTS techniques?) and 3.2 (how do different GI algorithms benefit from RTS in terms of efficiency?). These are required since we expect different RTS strategies to yield different efficiency. Moreover, the differences between GP and LS may also extend to the efficiency gained from each RTS technique, as each algorithm may take advantage of the efficiency gains provided by the RTS techniques in different ways.

4.4 RQ4—Tradeoff

What is the Tradeoff between Safety, Efficiency and Effectiveness of the GI Process with Various RTS Strategies in Different Application Scenarios?

Finally, RQ4 aims to answer whether different algorithms and RTS techniques weigh differently on specific scenarios. This RQ sets the ground for a more practical view of the problem. In summary, we want to analyse the results and provide guidelines to the engineer on how to choose GI algorithms and RTS techniques better when faced with different priorities. Because no specific algorithm can be the best for all scenarios and must compromise on its tradeoffs [42], unveiling the magnitude of such tradeoffs is crucial.

We created three tradeoff scenarios that comprise three different priorities an engineer may have when performing GI: (i) Perfect Improvement (P_{improv}); (ii) Fast Improvement (F_{improv}) and (iii) Diverse Improvement (D_{improv}).

This first scenario concerns the Perfect Improvement, where the engineer deals with the best improved and valid (passes all test cases) program variant. In other words, this scenario is dedicated to revealing which technique can obtain the best software runtime improvement overall, regardless of how long it takes to execute GI. To find out which technique obtains the perfect improvement, we compare their results in terms of raw runtime improvement.

The second scenario, Fast Improvement, is when the engineer is concerned with finding a valid and improved program variant as fast as possible in the search process. Here, the engineer will focus on finding any improved and valid program variant and then stop the GI execution immediately to avoid wasting resources. In this case, the level of improvement is not important as long as the overall runtime decreases. In order to find out which technique can find improvements the fastest, we analyse at which time the first improved variant was found in the search process.

The third and final improvement scenario is Diverse Improvement, where the engineer wants a wide gamma of program variants to choose from. Having a diverse set of improved programs means the engineer can balance their priorities, inspect the improved code, and make a more critical choice based on a qualitative analysis. For example, a wide set of programs may allow the engineer to choose an improved program with fewer edits or even compare the code of multiple variants to understand how their software's runtime is positively affected. In order to find the most diverse technique, we compute the number of improved variants found during the GI process by each technique.

5 Experimental Design

This section describes how we conducted our experiments to validate and analyse the proposed technique. For all experiments, we used the Gin tool [4] (see Section 3 for more information on Gin's functionality). A prior study comparing 31 GI tools for non-functional improvement found Gin and PyGGI the most accessible tools to apply to new software [48]. Gin was found to be easily configurable and natively supports LS and GP algorithms for GI. Combined with the fact that Gin is scalable to larger projects and that we used it in our previous study [16], this made Gin a straightforward choice for use in these experiments. To allow for reproducibility, we provide a replication package at https://figshare.com/s/52a5092425c64648467e.

5.1 GI Algorithms

To answer our RQs, we use two common GI algorithms [33] already implemented in Gin. As mentioned in Section 3.2, we use GP [20] and LS [4, 13, 19]. The former is the most common algorithm in GI, consisting of a global search, focusing on both exploration and exploitation of the search space, and is generally more expensive computationally. The latter focuses on an LS of neighbouring software variants with fewer modifications, which is less expensive but can more often result in local optima. LS was chosen for these experiments as it is readily available in Gin and has shown promising results in prior research [2], demonstrating comparable, if not better, performance than GP across various improvement scenarios. Since these algorithms perform the search differently, analysing their results can unveil further insights for GI and RTS. They are each set to run for 10 generations with 40 individuals in the population. The mutation probability is

Table 1. Subject Programs

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
gson-2.8.5	8,123	1,050	14,137	83/79	00:05
imaging-1.0*	31,320	583	7,427	73/59	00:52
jcodec-0.2.3	98,126	386	10,556	46/34	00:19
jfreechart-1.5.0	94,203	2,174	39,883	54/46	00:08
joda-time-2.10.14	29,895	4,239	56,404	89/81	00:11
spatial4j-0.9	6,950	466	3,954	79/74	00:09
text-1.3*	8,703	898	12,872	97/96	00:05
validator-1.6*	7,409	536	8,352	86/76	00:11

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). The asterisks mark programs from the Apache suite.

set to 50% and the crossover probability to 100% (GP only). The values for these parameters were selected to remain consistent with our previous investigation [16], and are in line with prior studies using these algorithms [11, 24, 28, 33] Each patch/software variant is run internally 10 times to account for runtime variations.

5.2 RTS Techniques

We consider two state-of-the-art RTS techniques, used both in research and in industry: Ekstazi [14] (dynamic RTS) and STARTS [27] (static RTS). These are described in Section 2.1. We also include a random test case selection as a sanity check. Namely, we compare the following strategies in our empirical evaluation:

- GI—Either GP or LS with no RTS (baseline);
- GI+Random—GI using a random test selection that selects a subset of test cases from the original test suite without guidance;
- GI+Ekstazi—GI using Ekstazi (v5.3.0) as a dynamic analysis RTS technique;
- GI+STARTS—GI using STARTS (v1.3) as a static analysis RTS technique.

To answer questions about the efficiency of the RTS techniques, we compute the time taken for each algorithm-RTS combination by summing the runtime needed for all GI steps, i.e., the time needed to profile the programs, select test cases and perform the search. Since all strategies share a common stopping criterion (number of generations), we can compute how much time is needed to perform the same GI tasks.

5.3 Subject Programs

We compare the algorithms and techniques using 12 programs collected from related work [3, 16, 17, 28, 31]. Table 1 presents details about the subject programs. We selected these programs because they represent a diverse set of domains, with different sizes, numbers of test cases, coverage values and test times, which in turn enhances the generalisability of our evaluation. The test time is the cost of running the program's testing procedure using Maven.

Table 2. Percentage of Selected Test Cases From the Entire Test Suite

Program	GI+Ekstazi	GI+STARTS	GI+Random
codec	**4.55**	**4.55**	35.90
compress	**4.52**	16.92	67.94
csv	72.17	96.44	30.74
fileupload	**39.51**	41.98	53.70
gson	64.47	90.74	**37.25**
imaging	**1.94**	81.31	39.42
jcodec	2.55	**0.46**	47.11
jfreechart	**13.83**	39.62	41.62
joda-time	**0.66**	–	45.97
spatial4j	83.56	100.00	**41.45**
text	**4.35**	**4.35**	37.18
validator	**15.89**	29.53	45.42
Median	**4.55**	29.53	41.53

The lower the percentage, the greater the reduction in number of test cases. Best values are highlighted in bold. A dash represents a technical failure in selecting test cases.

5.4 Experimental Procedure

Each algorithm is run for 20 independent runs on each program to account for the stochastic search process. At the end of each independent run, the algorithm outputs a list of all program variants generated during the search. The valid program variant with the best improvement score is selected for validation against the entire test suite. The result of this validation is then used to compute the efficiency gain (RC—speed-up of the GI process), effectiveness (RIC—improvement achieved by the variant compared to the average improvement from GI without RTS) and safety (RS—how many test cases from the entire test suite fail on the variant).

The results of the independent runs are compared using the Kruskal–Wallis statistical test [21] and Vargha-Delaney $\hat{A}_{12}$ effect size [39]. The former is used to assess if the difference between the techniques is statistically significant across many independent runs, whereas the latter measures the magnitude of the difference. Both tests are non-parametric, meaning they do not assume a normal distribution of the data.

6 Results

This section presents the results of our experiments and answers the RQs described in the previous section. Table 2 presents the number of test cases selected by each strategy for each program. This selection was performed in the profiling phase (as explained in Section 3), and the time taken to complete this task is computed as overhead.

6.1 Answer to RQ1—Safety

As mentioned in Section 5.2, the GI+Random technique is used as a baseline in this experiment. If a random selection of test cases is as safe as others, it means that Ekstazi and STARTS cannot outperform a much simpler strategy such as random, and their results can be attributed to chance.

Our results show that GI+Random failed to provide a safe selection of test cases for 43 out of 480 independent runs (8.9%), i.e., the test cases selected by GI+Random cannot accurately detect bugs in the improved versions of the software as well as the entire test suite in 8.9% of the cases. Using the RS measure, however, we observed that the RS (Equation (3)) of GI+Random is always higher than 0.991, i.e., at most 0.9% of the test cases fail when using a random RTS technique. This is due to the fact that most test cases do not actually reach the statements modified by the GI algorithm, thus passing the validation step.

GI+STARTS executed mostly successfully without test cases failing, except for two cases: GI+STARTS with GP for commons-text and GI+STARTS for joda-time (for both GP and LS). In the first case, one of the 20 GP independent runs for that SUT generated an improved version that failed with the entire test suite. However, only one out of 898 test cases failed in this scenario, yielding an RS result of 0.999, i.e., 99.9% of the test cases passed. On the other hand, for joda-time, STARTS failed to select test cases altogether (as seen in Table 2). Since STARTS works with static analysis and only captures test case relations with classes in compilation time, it could not analyse joda-time's dynamically loaded test suite, thus failing the 40 runs for that SUT (20 GP plus 20 LS runs). All in all, from all 480 independent runs of GI+STARTS, 41 runs (8.5%) yielded an improved version for which STARTS' selection was unsafe or could not work. Disregarding its limitations with joda-time, this number drops to 1 out of 440 runs, or 0.0023%.

Ekstazi, however, was able to perform the test selection for joda-time because it instruments all test cases and class files of the projects, capturing execution information as they run. As a result of Ekstazi's dynamic analysis, all improved versions of all SUTs generated by GI+Ekstazi passed when tested against the entire suite. Hence, GI+Ekstazi obtained a mean RS of precisely one, i.e., 100% of the test cases passed in all scenarios.

Answer to RQ1.1—RTS Comparison. Randomly selecting test cases is not as safe as using RTS techniques, failing to provide a safe selection in 8.9% of cases. State-of-the-art RTS strategies are mostly feasible when used with GI, yielding almost 100% safety. GI+Ekstazi stood out by not neglecting a single important test case in all 480 independent runs, thus obtaining a perfect RS score. Due to the above reasons, GI+STARTS could not be applied to joda-time. Of the remaining program runs, GI+STARTS failed to select one fault-revealing test case, failing only 1 of the 440 runs (0.0023%).

Answer to RQ1.2—Algorithm Comparison. Unsafe results appeared in 44 runs for GP and 40 runs for LS. If we exclude the inability of STARTS to select test cases for joda-time, the numbers are 24 and 20, respectively. Of these 24 and 20 unsafe results, all but one appeared during GI+Random runs (and can thus be attributed to this inherently unsafe RTS technique). The only unsafe result using state-of-the-art RTS techniques appeared in a GI+STARTS run with GP.

Answer to RQ1. We can safely state that Ekstazi always selects all the relevant test cases for GI, whereas STARTS fails in some edge cases due to its static analysis limitations. We have not observed any significant difference in safety between different GI algorithms. State-of-the-art RTS techniques are safe to use with GI for both GP and LS.

6.2 Answer to RQ2—Effectiveness

One concern with the impact of RTS on the general GI effectiveness is that the use of RTS might affect how much improvement can be achieved by the GI algorithms. Therefore, it is important to analyse the results in terms of the RIC—Equation (4) metric we devised in Section 4.2. Tables 3 and 4 present, respectively, the median RIC results and the effect size of the 20 independent runs of our experiments.

From Table 3, we can observe that, for 16 out of 24 (66.6%) cases, using GI without RTS is favourable, i.e., GI without RTS obtains the most effective results or equivalent results to the most

Table 3. *RQ2:* Median Relative Improvement Change (RIC–Equation (4)) Compared to GI without RTS Over 20 Independent Runs

Algorithm	Program	GI	GI+Ekstazi	GI+STARTS	GI+Random	p-value
GP	codec	1.00	**3.36**	2.29	0.95	< **0.001**
	compress	1.00	2.53	**5.81**	3.96	< **0.001**
	csv	1.00	2.89	**3.80**	1.68	**0.001**
	fileupload	1.00	**2.55**	1.97	2.08	0.065
	gson	**1.00**	0.70	0.68	0.86	0.515
	imaging	**1.00**	0.64	0.95	0.46	0.552
	jcodec	1.00	**1.81**	1.18	1.38	0.684
	jfreechart	**1.00**	0.44	0.51	0.22	< **0.001**
	joda-time	1.00	6.74	–	**7.50**	< **0.001**
	spatial4j	1.00	0.93	1.17	**1.23**	0.478
	text	1.00	**2.40**	0.95	0.57	**0.003**
	validator	1.00	**4.92**	2.81	4.04	< **0.001**
	Median	1.00	**2.46**	1.18	1.30	–
LS	codec	1.00	**10.27**	7.22	1.08	**0.001**
	compress	**1.00**	0.16	0.14	0.15	< **0.001**
	csv	**1.00**	0.54	0.59	0.60	**0.016**
	fileupload	**1.00**	0.19	0.20	0.17	< **0.001**
	gson	1.00	1.42	0.82	**2.51**	0.056
	imaging	1.00	1.08	1.11	**1.61**	0.476
	jcodec	**1.00**	0.84	0.89	0.88	0.926
	jfreechart	1.00	**1.84**	1.74	0.25	**0.001**
	joda-time	1.00	2.96	–	**3.19**	< **0.001**
	spatial4j	1.00	0.97	1.25	**2.63**	< **0.001**
	text	**1.00**	0.58	0.41	0.34	**0.002**
	validator	**1.00**	0.96	0.81	0.70	0.799
	Median	1.00	**0.96**	0.82	0.79	–

Greater RIC values are better. The best RIC medians are highlighted in bold. Grey cells are statistically equivalent to the best RIC. The last column shows the p-value result of Kruskal-Wallis. Significant p-values (< 0.05) are highlighted in bold.

effective approach. In fact, all approaches obtained favourable results for 16 to 19 cases, meaning the results are somewhat mixed, showing no strong evidence in favour of a single approach and evidence for similarities between them more often.

Table 4 shows that the difference in effectiveness between GI and GI+Ekstazi/STARTS is not straightforward. We observe that 25 out of 46 (54.4%) pairwise effect size comparisons between GI and GI+Ekstazi/STARTS show medium to negligible effect sizes. Thus, in approximately half of the cases, using state-of-the-art RTS techniques does not affect the capability of the GI algorithms to improve the software runtime. For 14 out of 46 (30.4%) comparisons, using RTS with GI generates largely better software (bold values lower than 0.5), and only for 7 (15.2%), the results are largely worse with RTS (bold values greater than 0.5). However, if we analyse the results by algorithm (i.e., GP and LS), there is a clear indication that RTS has a beneficial impact on the results of GP (as observed in our previous work [16]) more often than when using LS. The first observation supporting

Table 4. *RQ2:* Results of the Pairwise (Group A/Group B) Vargha-Delaney $\hat{A}_{12}$ (VDA) Effect Size Test for the Relative Improvement Change (RIC–Equation (4))

Algorithm	Program	GI/GI+Ekstazi	GI/GI+STARTS	GI+Ekstazi/GI+STARTS
GP	codec	**0.04 (L)**	**0.14 (L)**	0.56 (N)
	compress	**0.13 (L)**	**0.00 (L)**	0.30 (M)
	csv	**0.16 (L)**	**0.20 (L)**	0.49 (N)
	fileupload	**0.26 (L)**	0.40 (S)	0.63 (S)
	gson	0.56 (N)	0.59 (S)	0.55 (N)
	imaging	0.58 (S)	0.48 (N)	0.48 (N)
	jcodec	0.42 (S)	0.44 (N)	0.57 (N)
	jfreechart	**0.78 (L)**	0.72 (M)	0.40 (S)
	joda-time	**0.00 (L)**	–	–
	spatial4j	0.46 (N)	0.41 (S)	0.43 (N)
	text	0.29 (M)	0.56 (N)	0.73 (M)
	validator	**0.11 (L)**	**0.22 (L)**	0.66 (S)
LS	codec	**0.26 (L)**	**0.21 (L)**	0.43 (N)
	compress	**0.94 (L)**	**0.97 (L)**	0.58 (S)
	csv	**0.74 (L)**	0.73 (M)	0.45 (N)
	fileupload	**0.89 (L)**	**0.85 (L)**	0.50 (N)
	gson	0.38 (S)	0.58 (S)	0.68 (M)
	imaging	0.39 (S)	0.40 (S)	0.48 (N)
	jcodec	0.45 (N)	0.46 (N)	0.48 (N)
	jfreechart	**0.20 (L)**	0.35 (S)	0.56 (N)
	joda-time	**0.17 (L)**	–	–
	spatial4j	0.48 (N)	0.36 (S)	0.38 (S)
	text	0.71 (M)	**0.76 (L)**	0.63 (S)
	validator	0.54 (N)	0.57 (N)	0.51 (N)

VDA values greater than 0.5 are better for Group A, and better for Group B when lower than 0.5. Difference magnitudes are abbreviated as: N = negligible, S = small, M = medium, and L = large. Large effect sizes are highlighted in bold.

this notion is that the GP+RTS area of Table 3 is greyer than the LS+RTS area, highlighting more often the achievement of the best results or results equivalent to the best. Secondly, the effect size differences show large magnitudes in favour of GP+RTS more often than GP without RTS. For 10 out of 23 (43.5%) comparisons, GP+RTS yielded largely better results than GP without RTS, whereas GP without RTS yielded largely better results for only one out of 23 (4.3%) comparisons. These results are not as favourable when considering the LS results. Favourable effect sizes for LS+RTS only occur in 4 out of 23 (17.4%) cases, while unfavourable ones occur in 6 out of 23 (26.1%) cases.

As a side note, of the 122,700 valid variants obtained across all combinations of GI algorithms and RTS techniques in our experiments, 88,794 contained swap edits, 76,923 contained delete edits, 68,957 contained copy edits and 58,063 contained replace edits. While this may initially seem unintuitive, copy edits can positively impact variant runtime (e.g., by inserting return statements or breaking out of loops earlier). Additionally, prior work has shown that copy edits combined with other edit types can replicate the effect of different edit operations [30]. For instance, if a given statement is copied to a different location and one of the statements neighbouring the copied statement is subsequently deleted, the outcome is the same as a single replace operation.

In conclusion, we observe that using RTS has no significant impact on GI effectiveness in most cases. These results confirm our expectation since RTS does not change the overall mechanism of the GI algorithms but rather the fitness evaluation process. In a minority of cases where a positive impact is observed, we conjecture this might be because RTS narrows down the test cases used during the improvement search to only the significant ones, thus reducing the number of executions that can introduce noise to the improvement measurement and making the differences in execution time more noticeable. In other words, by using only a subset of the test suite, the execution time of the software under improvement is considerably lower. Thus, even a one-hundred-millisecond improvement is deemed more significant by the GI algorithm. In such a case, the GI algorithm will keep improving such variants. If the entire test suite were used, the same hundred milliseconds of improvement would be "diluted" and deemed less valuable, increasing the odds of discarding the improving variant during evolution.

Answer to RQ2.1—RTS Comparison. For all comparisons between GI+Ekstazi and GI+STARTS, the results showed only negligible to medium effect size differences. Moreover, there is a significant difference for only one of the 23 comparisons, and even this is still a medium effect size. Hence, there is no strong evidence to suggest that STARTS and Ekstazi differ in their impacts on the improvement effectiveness of GI.

Answer to RQ2.2—Algorithm Comparison. When comparing the results obtained for each GI algorithm (i.e., GP and LS), we observed a greater proportion of positive RIC when using RTS with GP (43.5%) than RTS with LS (17.4%). In other words, when using RTS with GP, there is a greater chance of obtaining better software improvements (relative to GP without RTS) than when using LS. While the reasoning behind this finding is unclear, it indicates that the resources saved using RTS are more productively reallocated towards generating better variants with GP.

Answer to RQ2. The use of RTS seems to be detrimental to the effectiveness of GI in only a small proportion of cases (15.2%), offering no change in effectiveness for most of the cases (54.4%) and even improvements in roughly a third of the cases (30.4%). It should also be noted that the impact on effectiveness measured for each program does not correlate with any of the metrics presented in Table 1, suggesting that engineers using RTS when applying GI to larger-scale programs should expect to see similar performance. Our results showed that both RTS techniques have a similar impact on the effectiveness values. On the other hand, the choice of the GI algorithm can yield different effectiveness: GP benefits the most from RTS in terms of effectiveness.

6.3 Answer to RQ3—Efficiency

This section presents the results and answers for RQ3 regarding the efficiency of using GI with the various RTS techniques. The results are presented in the form of RC—Equation (6) compared to using no RTS. Tables 5 and 6 show, respectively, the median RC results and the effect size over the 20 independent runs. Figure 2 depicts the RC values as boxplots for a more fine-grained visualisation.

The first observation is that, in the vast majority of the cases, more specifically for 20 out of 24 (83.3%) group comparisons, using RTS yields statistically significant better results. These results are expected since improving the efficiency of testing is the precise objective of using RTS. On average, Ekstazi obtained the best efficiency when compared to STARTS, with median RC values of 0.72 for GP and 0.58 for LS (i.e., it costs 28% and 42% less than using no RTS with these search algorithms, respectively). In comparison, STARTS achieved RC values of 0.93 (7% reduction) with GP and 0.96 (4% reduction) with LS. Considering the effect size pairwise comparison, for 14 out of 22 (63.6%) cases, there were no large differences between GI+Ekstazi and GI+STARTS. However, for 7 out of 22 (31.8%) cases, GI+Ekstazi showed largely better efficiency than GI+STARTS, while

Table 5. *RQ3:* Median Relative Cost (RC–Equation (6)) Compared to GI without RTS Over 20 Independent Runs

Algorithm	Program	GI	GI+Ekstazi	GI+STARTS	GI+Random	*p*-value
GP	codec	1.00	**0.39**	0.40	0.77	< **0.001**
	compress	1.00	**0.32**	0.38	0.89	< **0.001**
	csv	1.00	1.03	0.93	**0.82**	**0.008**
	fileupload	1.00	1.11	**0.93**	1.05	**0.028**
	gson	1.00	**0.83**	0.99	0.88	< **0.001**
	imaging	1.00	0.79	0.98	**0.40**	< **0.001**
	jcodec	1.00	**0.22**	1.03	1.30	< **0.001**
	jfreechart	1.00	0.34	**0.30**	0.40	< **0.001**
	joda-time	1.00	**0.50**	–	0.71	< **0.001**
	spatial4j	1.00	0.78	1.02	**0.57**	< **0.001**
	text	1.00	**0.67**	0.78	1.15	< **0.001**
	validator	1.00	0.82	**0.47**	1.01	**0.015**
	Median	1.00	**0.72**	0.93	0.85	–
LS	codec	1.00	**0.28**	0.29	0.98	< **0.001**
	compress	1.00	**0.24**	0.27	1.06	< **0.001**
	csv	1.00	0.95	1.02	**0.74**	< **0.001**
	fileupload	1.00	**0.95**	0.96	**0.95**	0.636
	gson	1.00	0.87	1.06	**0.70**	< **0.001**
	imaging	1.00	0.87	0.96	**0.22**	< **0.001**
	jcodec	1.00	**0.20**	1.03	1.09	< **0.001**
	jfreechart	1.00	0.39	0.44	**0.32**	< **0.001**
	joda-time	1.00	**0.45**	–	1.29	< **0.001**
	spatial4j	1.00	0.55	0.82	**0.45**	< **0.001**
	text	1.00	**0.86**	0.99	1.18	**0.002**
	validator	1.00	0.61	1.00	**0.49**	0.088
	Median	1.00	**0.58**	0.96	0.84	–

Lower RC values are better. The best RC medians are highlighted in bold. Grey cells are statistically equivalent to the best RC. The last column shows the *p*-value result of Kruskal–Wallis. Significant *p*-values (< 0.05) are highlighted in bold.

GI+STARTS only showed large favourable efficiency compared to GI+Ekstazi in one out of 22 (4.5%) cases. This efficiency gap is likely due to the differences in each tool's test case selection methods (described in Section 2.1). STARTS is relatively conservative in its selection phase [38], translating to a larger set of tests for a given variant and resulting in longer execution times, thus yielding higher RC values. As seen in Table 2, Ekstazi is typically more precise in selecting only those test cases that could make a given variant fail, meaning failing variants can be identified more efficiently as fewer irrelevant test cases are executed. For one program ("spatial4j"), GI+Random selected the fewest test cases. This behaviour is reasonable to expect in some cases as GI+Random selects tests without considering safety. In this particular case, the targeted method in "spatial4j" is one on which many other components of the software depend. Thus, the majority of the tests from the test suite traversed this method and were deemed necessary by both RTS tools.

Table 6. *RQ3:* Results of the Pairwise (Group A/Group B) Vargha-Delaney $\hat{A}_{12}$ (VDA) Effect Size Test for the Relative Cost (RC–Equation (6))

Algorithm	Program	GI/GI+Ekstazi	GI/GI+STARTS	GI+Ekstazi/GI+STARTS
GP	codec	**0.96 (L)**	**0.89 (L)**	0.55 (N)
	compress	**0.99 (L)**	**0.98 (L)**	0.30 (M)
	csv	0.49 (N)	0.65 (S)	0.70 (M)
	fileupload	0.36 (S)	0.63 (S)	**0.74 (L)**
	gson	**0.83 (L)**	0.49 (N)	**0.16 (L)**
	imaging	**0.80 (L)**	0.52 (N)	**0.24 (L)**
	jcodec	**1.00 (L)**	0.49 (N)	**0.00 (L)**
	jfreechart	**1.00 (L)**	**0.97 (L)**	0.64 (S)
	joda-time	**0.97 (L)**	–	–
	spatial4j	0.70 (M)	0.50 (N)	0.27 (M)
	text	**0.82 (L)**	**0.76 (L)**	0.35 (S)
	validator	0.63 (S)	0.74 (M)	0.64 (S)
LS	codec	**0.97 (L)**	**0.98 (L)**	0.47 (N)
	compress	**1.00 (L)**	**1.00 (L)**	**0.19 (L)**
	csv	0.51 (N)	0.42 (S)	0.40 (S)
	fileupload	0.60 (S)	0.61 (S)	0.50 (N)
	gson	**0.76 (L)**	0.48 (N)	**0.23 (L)**
	imaging	0.66 (S)	0.54 (N)	0.37 (S)
	jcodec	**1.00 (L)**	0.57 (N)	**0.00 (L)**
	jfreechart	**1.00 (L)**	**1.00 (L)**	0.27 (M)
	joda-time	**0.89 (L)**	–	–
	spatial4j	**0.88 (L)**	0.58 (S)	**0.11 (L)**
	text	0.66 (S)	0.48 (N)	0.30 (M)
	validator	0.72 (M)	0.64 (S)	0.43 (N)

VDA values lower than 0.5 are better for Group A, and better for Group B when greater than 0.5. Difference magnitudes are abbreviated as: N = negligible, S = small, M = medium, and L = large. Large effect sizes are highlighted in bold.

Similarly to our findings for effectiveness, the amount of improvement in terms of efficiency for each program when using RTS with GI did not demonstrate a clear relationship with the program metrics in Table 1. We observed efficiency gains in all programs, regardless of size. In general, we believe the efficiency gains from applying RTS will depend on more in-depth factors related to the quality of the test suite provided (e.g., whether many tests cover the same branches, how long they run for and so on) and what proportion of the test suite covers the methods being targeted for improvement. For instance, if there is significant overlap in the source code covered by many test cases, an RTS tool will struggle to reduce the size of the test suite. This was the case for "spatial4j" discussed above, meaning Ekstazi could only reduce the test suite size by 16.44%, while STARTS could not reduce the test suite at all (Table 2).

Although the benefits of using RTS are prominent for the engineer (e.g., they can save up to 80% of execution time by using LS with Ekstazi for jcodec), the benefits are also significant for researchers performing experiments involving GI. Figure 3 shows the cumulative cost in terms of hours of execution time for our entire set of experiments over 20 independent runs.

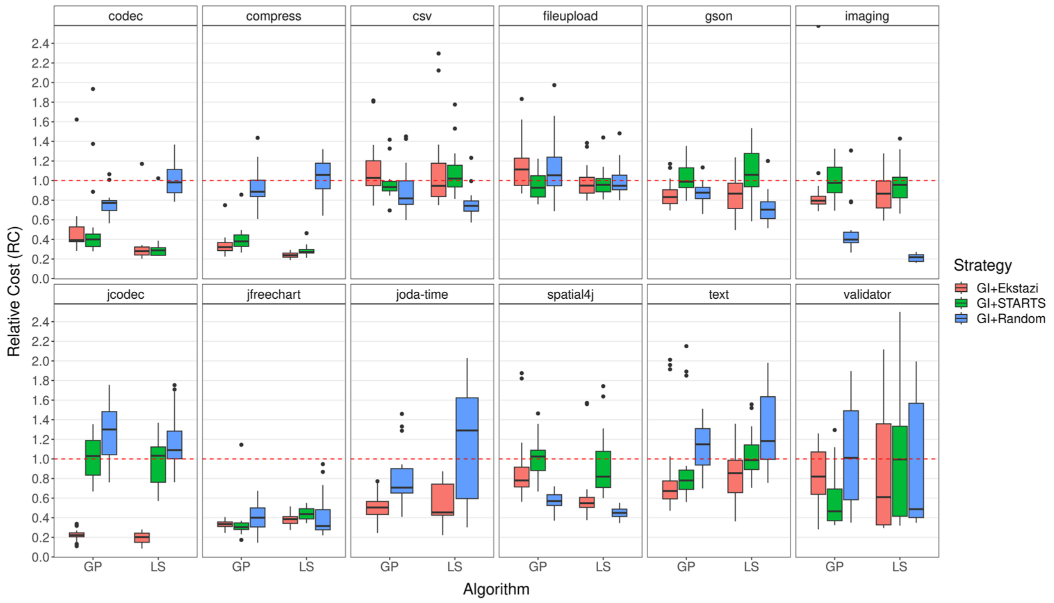

Fig. 2. *RQ3:* RC of strategies. The y-axis shows the RC results, whereas the x-axis shows the two algorithms used: GP and LS. Each boxplot represents the RC result of a given RTS strategy over 20 independent runs. Lower values are better. The dashed line represents the median cost of GI without RTS, i.e., baseline.

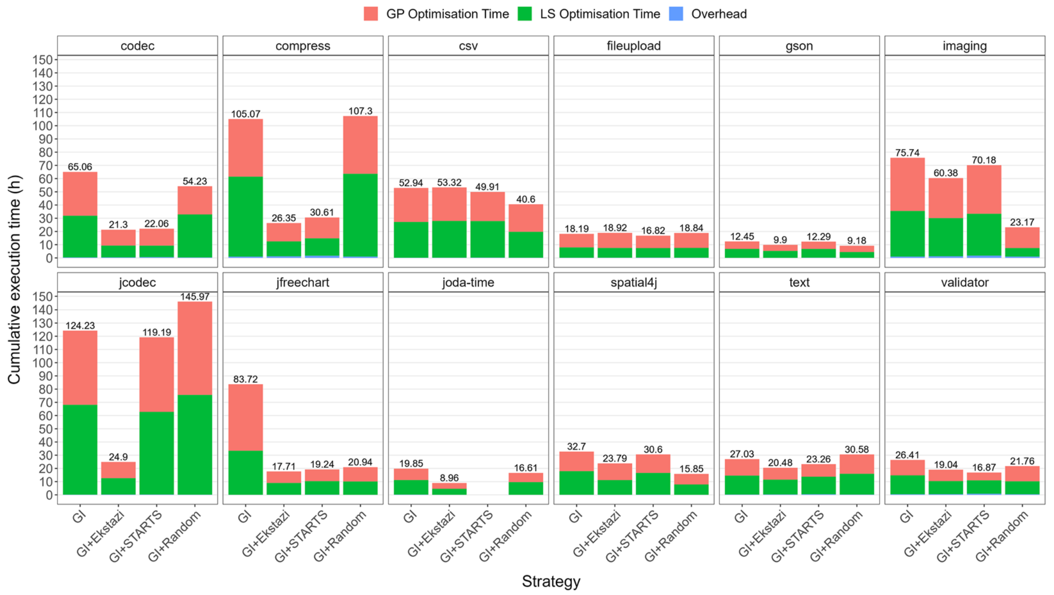

Fig. 3. *RQ3:* Cumulative execution times in hours for all runs. The y-axis shows the cumulative cost, whereas the x-axis shows the different strategies used in our experiments. The results are divided into three phases: (i) Overhead—cost of profiling (collecting dependencies between source files and test cases and performing test case selection); (ii) GP Optimisation Time—cost of the GI optimisation using GP and (iii) LS Optimisation Time—cost of the GI optimisation using LS.

More often than not, GI+Ekstazi and GI+STARTS were much less expensive than using GI alone. This can be seen with the total execution cost of our experiments: (i) GI—1,930 hours (≈11.49 weeks); (ii) GI+Ekstazi—915 hours (≈5.45 weeks); (iii) GI+STARTS—1,233 hours (≈7.34 weeks) and (iv) GI+Random—1,515 hours (≈9.02 weeks). Using GI+Ekstazi, we were able to save more than 1,000 hours of execution time, i.e., roughly six weeks of computational resources.

A case can be made that the high cost of GI could easily be solved by spawning more parallel jobs. However, this approach still would not solve the problem of excessive computational resource consumption. As software becomes more expensive, we believe the concern of sustainability should lie in the hands of the engineers who developed it. To delegate such responsibility to other disciplines (e.g., distributed computing) is to deny accountability for unsustainable engineering. Such practice can be detrimental to the important goal of achieving greener SE.

Gin uses a "fail fast" strategy, meaning it stops the evaluation of a given software variant at the signal of the first failing test case. Consequently, failing variants are cheaper to evaluate than variants for which no test case fails. With this in mind, an RTS technique could incur a higher cost than using no RTS by failing to select relevant test cases. In this situation, even though fewer test cases are considered for execution by GI with RTS than GI without RTS, the algorithm has no signal to stop executing the evaluation on faulty variants since relevant test cases that would fail are not executed. Hence, the execution might go on longer than expected, causing the cost to increase. As seen in Section 6.1, state-of-the-art RTS techniques are safe to use and hardly discard relevant test cases, contrary to Random RTS, which fails more often. Thus, we observed the aforementioned phenomenon in our experiments with GI+Random on programs for which it failed to select relevant test cases. Although its median RC values for each algorithm both fell below one (i.e., cheaper than no RTS), GI+Random was statistically more expensive than using no RTS at all for specific programs such as "jcodec," "joda-time" and "text."

When comparing both algorithms by their RC medians in Table 5, we can see that LS (0.58) showed better reductions than GP (0.72) when using Ekstazi, but the difference is not entirely visible for the other cases. Figure 4 presents the RC results, similarly to Figure 2, but with boxes grouped by algorithm. Since Random RTS is only used as a sanity check and we already compared it in the previous analysis, we omitted it from this figure. For "codec" and "compress," it is clear that LS obtains better efficiency gains with both RTS strategies than GP. On the other hand, GP is more efficient for "jfreechart" and "text." The results are mixed in all other cases.

In contrast to the results of RQ2 that showed that different algorithms impact the effectiveness gains more than the RTS techniques, our efficiency results suggest that efficiency gains stem from RTS techniques rather than the algorithms being used. This is somewhat expected since GI algorithms are commonly designed to achieve better effectiveness and RTS techniques to achieve better efficiency; thus, they differ mainly on those properties.

Answer to RQ3.1—RTS Comparison. GI+Ekstazi showed more efficiency gains when compared to GI+STARTS. Ekstazi yielded a median execution time reduction of 28% with GP and 42% with LS, while STARTS yielded 7% with GP and 4% with LS. When summing the total execution costs of our experiments, GI+Ekstazi took 915 hours, whereas STARTS took 1,233.

Answer to RQ3.2—Algorithm Comparison. In general, both algorithms obtain similar efficiency gains. In a few specific cases (e.g., "codec" and "compress" for LS, and "jfreechart" and "text" for GP), one algorithm showed slightly better RC than the other, but for most cases the difference is not statistically significant.

Answer to RQ3. Using RTS techniques can reduce the cost of the entire GI process by up to 80%. For the vast majority of cases (83.3%), by using Ekstazi we were able to significantly improve the efficiency of GI, reducing on average the cost of executing it by 28% in our experiments with GP and 42% with LS. Ekstazi was also more efficient than STARTS, obtaining largely better efficiency

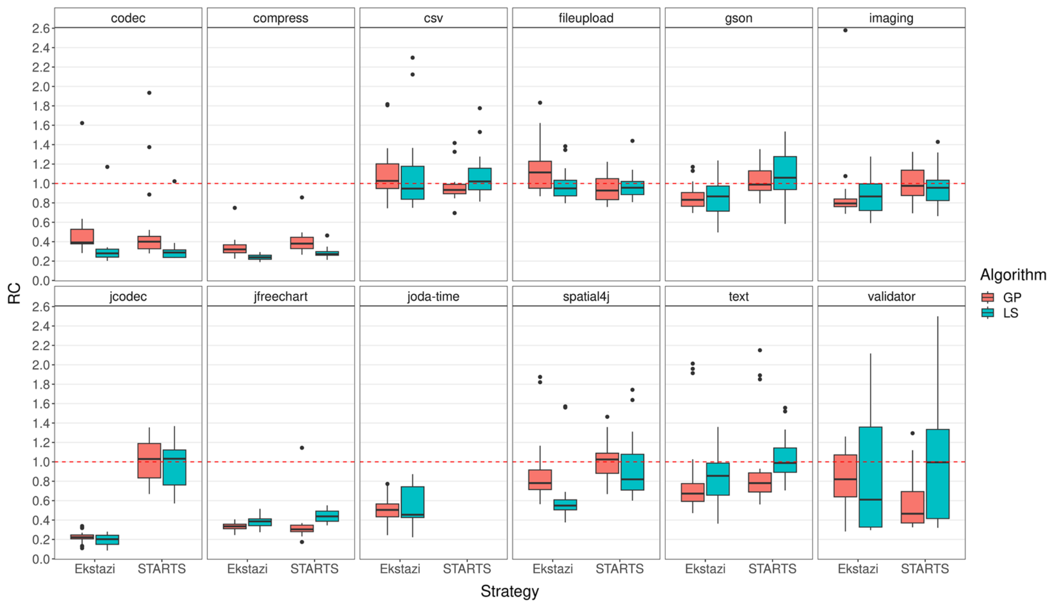

Fig. 4. *RQ3.2:* RC of algorithms. The y-axis shows the RC results, whereas the x-axis shows the two main RTS strategies used: Ekstazi and STARTS. Each boxplot represents the RC result of a given algorithm over 20 independent runs. Lower values are better. The dashed line represents the median cost of either GP or LS without RTS, i.e., baseline.

than STARTS in 31.8% of cases. When looking at the total cost of our experimental procedure, we observed a difference in total execution time of 6 weeks (or 1,000 hours) between the cost of GI and GI+Ekstazi, further showcasing how much RTS is essential for more sustainable GI. Similarly to our findings for effectiveness, we observed efficiency gains when using RTS with GI for all programs regardless of their sizes, meaning RTS is likely to reduce GI execution time when applied to programs at any scale. Overall, the GI algorithms do not seem to prefer a specific RTS technique. Our results show that the efficiency gains stem from the RTS strategies rather than the GI algorithms.

6.4 Answer to RQ4—Tradeoff

This section discusses the performance of the various combinations of GI algorithms and RTS techniques when applied to multiple scenarios, aiming to gain a more practical perspective of the tradeoffs each combination offers. Since our findings from Section 6.1 indicate that it is unsafe to use in practice, we have avoided considering GI+Random when answering this question so as not to mislead the reader: we do not recommend engineers use GI+Random in any of the following scenarios.

The first tradeoff scenario we consider is P_{improv}, which concerns finding the best possible program variant. Table 7 presents the median improvement in seconds of the best program variant found during 20 independent runs, i.e., by how much the execution time of the test suite is reduced by a program variant.

Similarly to our previous results [16], GI+Ekstazi showed the best improvement overall. However, because our previous work only considered GP as a GI algorithm and fewer programs, we observe a few other interesting results in this work.

Table 7. *RQ4:* P_{improv} Scenario

Algorithm	Program	GI	GI+Ekstazi	GI+STARTS
GP	codec	1.29	**4.35**	2.96
	compress	2.46	6.23	**14.28**
	csv	0.78	2.25	**2.95**
	fileupload	0.05	**0.13**	0.10
	gson	**0.29**	0.20	0.20
	imaging	**11.98**	7.63	11.42
	jcodec	6.77	**12.26**	7.95
	jfreechart	**2.06**	0.90	1.05
	joda-time	0.51	**3.46**	–
	spatial4j	2.32	2.16	**2.72**
	text	1.46	**3.52**	1.40
	validator	0.29	**1.44**	0.82
	Median	1.38	**2.86**	2.72
LS	codec	0.34	**3.49**	2.45
	compress	**19.82**	3.16	2.87
	csv	**0.75**	0.41	0.44
	fileupload	**0.40**	0.07	0.08
	gson	0.37	**0.52**	0.30
	imaging	2.48	2.67	**2.74**
	jcodec	**8.38**	7.08	7.46
	jfreechart	0.79	**1.45**	1.38
	joda-time	0.57	**1.69**	–
	spatial4j	2.49	2.41	**3.12**
	text	**0.78**	0.45	0.32
	validator	**0.52**	0.50	0.42
	Median	0.76	**1.57**	1.38

Median improvement in total seconds of improvement of the best variant found. Greater values are better. Best values are highlighted in bold.

First, the improvement obtained is smaller on average when using LS with RTS but better than when using no RTS at all. Looking at the baseline results (GI without RTS), we found that LS performs better for 7 out of 12 programs, and GP performs better for 5 out of 12 programs. On the other hand, GI+Ekstazi performs better using GP rather than LS for 8 out of 12 cases, while GI+STARTS performs better using GP for 8 out 11 cases. In other words, when using GP, the engineer might obtain better results by also using RTS.

Second, GI+Ekstazi obtained the best program variants (on average) for 10 out of 24 cases, whereas GI obtained the best variants for 9 out of 24 cases. Additionally, the overall median improvement achieved by GI+Ekstazi is substantially greater than that of GI (more than double). Therefore, coupled with the fact that GI+Ekstazi is the most efficient option, the tradeoff is clear: GI+Ekstazi can find the best variant while spending less computational resources more often.

Table 8 presents the results for our second scenario, F_{improv}, where the engineer is concerned with finding an improving variant as fast as possible. The table shows the median execution time in seconds until the algorithm found a positive and valid software variant.

Table 8. *RQ4: F_{improv}* Scenario

Algorithm	Program	GI	GI+Ekstazi	GI+STARTS
GP	codec	211.11	50.57	**32.67**
	compress	244.34	**32.27**	36.30
	csv	**63.17**	75.82	106.14
	fileupload	64.07	**53.36**	71.59
	gson	4.28	**2.45**	3.69
	imaging	201.27	**153.25**	188.30
	jcodec	310.69	**59.57**	257.39
	jfreechart	27.48	**8.88**	9.13
	joda-time	76.91	**27.46**	–
	spatial4j	69.88	51.21	**41.79**
	text	31.38	**11.53**	11.57
	validator	27.04	**4.82**	14.41
	Median	66.97	41.42	**36.30**
LS	codec	168.26	**31.00**	31.11
	compress	242.71	36.85	**35.79**
	csv	67.40	**62.53**	84.55
	fileupload	18.57	**13.81**	16.16
	gson	4.56	**3.12**	3.80
	imaging	146.78	**134.66**	162.40
	jcodec	376.17	**38.90**	336.60
	jfreechart	31.26	12.44	**9.34**
	joda-time	33.77	**8.91**	–
	spatial4j	56.06	60.29	**49.05**
	text	33.60	**2.77**	6.71
	validator	35.68	**5.25**	14.08
	Median	45.87	**22.41**	31.11

Median execution time in seconds needed to find the first valid and improving software variant. Lower values are better. Best values are highlighted in bold.

The first observation is somewhat aligned with the results of RQ2: Using RTS significantly speeds up the search for program variants. Thus, the time needed to find the first improving and valid variant is lower for GI+Ekstazi and GI+STARTS. Unlike our previous results [16], where we found that STARTS found the fastest improvements, we observe that GI+Ekstazi finds the first improving variant faster in most cases. This aligns with the findings from Section 6.3, which showed that Ekstazi provides the best efficiency gains overall.

Table 9 presents the median number of valid and improving patches found by each algorithm. These results concern our third scenario, D_{improv}, where the engineer focuses on finding the largest set of improving and valid patches.

Different from our prior study [16], where GI+Ekstazi found the widest variety of program variants, we found that not using RTS results in more diversity. These results are mainly due to the inclusion of LS, where GI without RTS found more variants for 8 out of 12 cases. If we only consider GP, then GI+Ekstazi was able to find more variants for 7 out of 12 cases.

Table 9. *RQ4: D_{improv}* Scenario

Algorithm	Program	GI	GI+Ekstazi	GI+STARTS
GP	codec	6.0	**16.0**	9.0
	compress	11.0	28.5	**41.5**
	csv	25.0	**48.0**	24.0
	fileupload	2.5	**4.0**	2.0
	gson	**48.5**	40.5	25.5
	imaging	**25.5**	21.0	23.0
	jcodec	23.5	**35.0**	23.5
	jfreechart	**121.5**	35.5	73.0
	joda-time	7.0	**8.0**	–
	spatial4j	11.0	5.0	**18.0**
	text	13.0	**19.5**	18.5
	validator	36.0	**44.0**	7.0
	Median	18.2	**24.8**	23.0
LS	codec	**27.0**	9.5	12.5
	compress	**60.5**	51.5	53.5
	csv	**38.0**	34.5	32.0
	fileupload	**28.0**	6.0	6.0
	gson	**73.5**	61.5	53.0
	imaging	16.0	28.0	**30.5**
	jcodec	11.5	38.0	**47.5**
	jfreechart	73.0	**104.5**	67.5
	joda-time	**23.0**	15.0	–
	spatial4j	**27.0**	5.0	20.0
	text	20.0	22.0	**24.0**
	validator	**29.5**	15.0	25.0
	Median	27.5	25.0	**30.5**

Median number of distinct, positive, and valid patches. Greater values are better. Best values are highlighted in bold.

Answer to RQ4. If the engineer is concerned with finding the best possible program variant using GI, then the results are clear: GI+Ekstazi offers the best tradeoff. GI+Ekstazi is able to find the best variants overall with excellent efficiency. This efficiency also makes GI+Ekstazi the best choice when trying to find an improving variant as fast as possible. On the third tradeoff analysis concerning the variety of program improvements, we found that LS provides a wider gamma of patches without RTS, whereas GP yields more diversity with Ekstazi. Overall, it appears that GI+Ekstazi using GP provides the most balanced tradeoff in most scenarios.

7 Threats to Validity

Threats to External Validity. As is common in SE experiments, it is possible that the set of subject programs may not be representative of all software. To mitigate this threat, we have included five new subjects from related work on top of the previous seven, thus forming a diverse set of programs from many domains. As shown in Table 1, the programs used in our empirical evaluation are well-known, non-trivial, of different sizes, have test suites of various sizes and coverages, and are used for different purposes.

Another external threat concerns the fact that we have used only two RTS techniques and two GI algorithms in our experiments, and that Gin can only target one method at a time or multiple methods from the same class (which it handles all in the same way, regardless of their type). Additionally, other GI tools may benefit from the application of RTS approaches in different ways (e.g., due to the various tool execution times and algorithm implementations). In this work, we have included an additional GI algorithm (LS) precisely to improve the generalisability of our results and to unveil the effect of the different state-of-the-art RTS techniques in this new scenario. Although more algorithms, techniques and GI tools could have been used, the computational cost of the experiments would have been prohibitive. Thus, we decided to experiment only with the state-of-the-art [14, 27].

Threats to Internal Validity. We have taken a few measures when designing the experiments to reduce internal threats concerning different environments affecting the results. First, we ran all experiments on the same cluster of machines, giving them a common environment and thus mitigating possible execution variations due to hardware differences. Second, we used the same configuration for all experiments, including the optimisation-stopping condition. Finally, we provided all algorithms and strategies with the same test suite for each program, meaning a single shared benchmark was used to measure execution times.

Since test cases can only reveal the presence of bugs in the software and not the software's correctness [10], our validation can only measure the adequacy of the program variants w.r.t. the available test suites, not their correctness. Therefore, another internal threat concerns the validity of our results if we considered correct patches rather than adequate patches during the validation process. Unfortunately, analysing programs to guarantee correctness is still generally impossible with automated tools and would require an infeasible amount of effort to do so manually. In order to minimise this threat, we have used the original test suites provided with the programs as a baseline for validity. The programs' developers and other open-source collaborators carefully curated these test suites and use them in the continuous integration process to validate pull requests. Despite this, there is still a possibility that these tests are insufficient to test GI-generated patches adequately. One possible solution to mitigate this threat would be to automatically generate new test cases to improve the testing power of such test suites. However, this approach could introduce overfitting in the results [28]. We tried to mitigate this threat in our experiments by including programs with a range of values for test suite coverage. However, handling insufficient test suites remains an open challenge in GI for non-functional properties [32].

Threats to Construct Validity. Given the stochastic nature of the GI algorithms used in this article, we performed 20 independent runs to account for the randomness variation of results. Moreover, we executed each test case 10 times to account for possible fluctuations in their execution time. We also took extra care when selecting and executing the statistical significance and effect size tests to only claim differences in the results when sufficient evidence is found. Moreover, we do not make any statistical assumptions about the data, thus avoiding tests that could jeopardise the validity of our analysis.

8 Related Work

This section describes papers related to the usage of RTS techniques for improving the GI process. As far as we know, only Mehne et al. [29] have investigated RTS in the context of APR, but never for non-functional GI. In their work, the authors define their own ad-hoc RTS technique and evaluate the results in terms of APR speed-up. The results show a speed-up of up to 1.8 times the original cost of APR for C programs using GenProg [24].

Additional studies use various regression techniques other than RTS. Venugopal et al. [40] proposed using test case prioritisation for APR to prioritise test cases that can make the validation fail, thus failing faster when an invalid patch is generated. As shown by their results, the authors were able to save up to 57.5% of execution time. Similarly, Qi et al. [34] proposed *TrpAutoRepair*, a technique that can prioritise test cases for APR in an online fashion. The idea is to avoid the offline training of the techniques and only use information generated during the test validation phase. Fast et al. [12] used incremental random sampling of test cases for patch validation in APR. In summary, their approach selects all failing test cases and a subset of passing test cases. If the patch passes all selected test cases, another sampling is performed. The authors compared their technique with a test suite minimisation based on GA [9]. The authors obtained savings of up to 81% in computational resources.

Although APR can be considered a type of functional GI (i.e., it improves functional properties), it is only a subset of what GI can achieve. Only the work of Mehhne et al. [29] touches the surface regarding the evaluation of the effect of RTS in the context of GI, but then again, it focuses on APR alone. Analysing the effect of RTS in the context of non-functional GI is considerably different than analysing its effect in the context of APR, since RTS impacts precisely what indicates (most of the time) the GI's improvement capabilities: computational resources consumption. In other words, reducing the cost of APR with RTS impacts only the approach's cost, whereas reducing the cost of non-functional GI with RTS impacts the approach's quality as well.

Due to the lack of work investigating the phenomena that can arise from using RTS in a non-functional GI context, we conducted a set of experiments to analyse such phenomena in previous work [16]. Our work differs from the articles mentioned in this section because: (i) we focus on non-functional GI (as opposed to APR); (ii) we use test case selection (as opposed to test suite minimisation and prioritisation); (iii) we focus on the improvement of Java programs (as opposed to C programs) and (iv) and we focus on quantifying the overall effect that RTS has in this context. In our previous work [16], we evaluated the impact of two RTS techniques when using GP in seven real-world Java programs. To the best of our knowledge, it was the first study of its kind.

In this article, we substantially extend our previous work by including a new GI algorithm, five new and larger subject programs, new RQs, and we provide a more thorough description of our approach. Overall, as seen in Section 6, this extension brought to light exciting results that unveiled differences in how RTS behaves in different contexts.

9 Conclusion

Although GI has been successful in improving many software properties [32], the cost of this approach might still be an issue for its widespread adoption. Test case execution for validating improved variants remains the primary source for this high cost. In this article, we tackled this problem by proposing the usage of RTS techniques during the evolutionary process of non-functional GI for improving execution time. We analysed the impact of state-of-the-art techniques from many angles, including safety, effectiveness, efficiency and engineering tradeoffs. With that in mind, we conducted a set of experiments with 12 real-world programs, two state-of-the-art RTS tools (Ekstazi and STARTS) and two GI algorithms (GP and LS) to answer four RQs concerning RTS feasibility in this context.

Our results show that RTS is not only safe, but can also save up to 80% of execution time w.r.t. GI algorithms without RTS. On average, RTS techniques were able to save 31% of runtime. When analysing the total cost of our experiments, we discovered that using Ekstazi (dynamic RTS) resulted in six weeks (approximately 1,000 hours) of execution time savings. Furthermore, RTS also showed little to no negative effect on the GI capabilities of improvement, i.e., most of the time, RTS techniques do not impact the effectiveness of GI algorithms. On the contrary, RTS can potentially improve the final results of non-functional GI. Finally, we showed that Ekstazi can help the GI algorithms find better software variants, find an improving and valid variant faster than other techniques, and provide the engineer with a wider variety of patches.

Given these results, we advise future GI research to use RTS during the evolutionary process. STARTS and Ekstazi have both been implemented and made accessible to anyone using Gin, available at https://github.com/gintool/gin. This type of traditional SE technique can help the entire field advance towards a more sustainable practice without the need for more potent parallelisation hardware. We believe that the results of our work serve as evidence that SE techniques can be used to solve AI problems as well and that there are other SE for AI topics that can be explored in the future.

In future work, we intend to evaluate the usage of other regression techniques, such as test case prioritisation, test suite minimisation, and online test case selection. Other possibilities include using the data collected during the GI execution to not only select fewer test cases but also include newly generated ones in case of insufficient testing. However, other works [28, 44] have suggested that overfitting is an open challenge in non-functional GI, thus requiring more research. We also plan to investigate the effect of flakiness both in functional and non-functional GI.

References

[1] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. 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 '19).* Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.), ACM, 1100–1104. DOI: https://doi.org/10.1145/3338906.3341184

[2] Aymeric Blot and Justyna Petke. 2021. Empirical comparison of search heuristics for genetic improvement of software. *IEEE Transactions on Evolutionary Computation* 25, 5 (2021), 1001–1011. DOI: https://doi.org/10.1109/TEVC.2021.3070271

[3] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. 2020. Injecting shortcuts for faster running java code. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '20),* 1–8. DOI: https://doi.org/10.1109/CEC48606.2020.9185708

[4] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 2019. Gin: Genetic improvement research made easy. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19).* ACM, New York, NY, 985–993. DOI: https://doi.org/10.1145/3321707.3321841

[5] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. 2019. Approximate oracles and synergy in software energy search spaces. *IEEE Transactions on Software Engineering* 45, 11 (2019), 1150–1169. DOI: https://doi.org/10.1109/TSE.2018.2827066

[6] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. Object-oriented genetic improvement for improved energy consumption in Google Guava. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering* (SSBSE '15). Márcio de Oliveira Barros and Yvan Labiche (Eds.), Lecture Notes in Computer Science, Vol. 9275, Springer, 255–261. DOI: https://doi.org/10.1007/978-3-319-22183-0_20

[7] Lingchao Chen and Lingming Zhang. 2018. 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 '18).* IEEE, 58–69. DOI: https://doi.org/10.1109/ICST.2018.00016

[8] Fábio de Almeida Farzat, Márcio de Oliveira Barros, and Guilherme Horta Travassos. 2018. Challenges on applying genetic improvement in JavaScript using a high-performance computer. *Journal of Software Engineering Research and Development* 6 (2018), 12. DOI: https://doi.org/10.1186/s40411-018-0056-2

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. DOI: https://doi.org/10.1109/4235.996017

[10] Edsger W. Dijkstra. 1972. The Humble Programmer. *Commun.* ACM 15, 10 (oct 1972), 859–866. 0001–0782 https://doi.org/10.1145/355604.361591

[11] Zhen Yu Ding, Yiwei Lyu, Christopher Steven Timperley, and Claire Le Goues. 2019. 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 '19). Justyna Petke, Shin Hwei Tan, William B. Langdon, and Westley Weimer (Eds.), ACM, 2–9. DOI: https://doi.org/10.1109/GI.2019.00011

[12] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2010. Designing better fitness functions for automated program repair. In *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference (GECCO '10),* 965–972. DOI: https://doi.org/10.1145/1830483.1830654

[13] Michel Gendreau and Jean-Yves Potvin. 2019. *Handbook of Metaheuristics* (3rd. ed.). Springer, 604 pages.

[14] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*, 211–222. DOI: https://doi.org/10.1145/2771783.2771784

[15] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st. ed.). Addison-Wesley Longman Publishing Co., Inc.

[16] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing genetic improvement of software with regression test selection. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE '21)*, 1323–1333. DOI: https://doi.org/10.1109/ICSE43902.2021.00120

[17] Giovani Guizzo, Federica Sarro, Jens Krinke, and Silvia R. Vergilio. 2022. Sentinel: A hyper-heuristic for the generation of mutant reduction strategies. *IEEE Transactions on Software Engineering* 48, 3 (2022), 803–818. DOI: https://doi.org/10.1109/TSE.2020.3002496

[18] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. 2011. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification.* B. Meyer and M. Nordio (Eds.),Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7007, 1–59. DOI: https://doi.org/10.1007/978-3-642-25231-0_1

[19] Holger H. Hoos and Thomas Stützle. 2004. *Stochastic Local Search: Foundations and Applications*. Elsevier.

[20] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Vol. 1. MIT Press.

[21] William H. Kruskal and W. Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.

[22] David C. Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class firewall, test order, and regression testing of object-oriented programs. *JOOP* 8, 2 (1995), 51–65.

[23] William B. Langdon and Mark Harman. 2015. Optimising existing software with GP. *IEEE Transactions on Evolutionary Computation* 19, 1 (2015), 1–18. DOI: https://doi.org/10.1109/TEVC.2013.2281544

[24] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. 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 '12).* IEEE, 3–13.

[25] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. DOI: https://doi.org/10.1109/TSE.2015.2454513

[26] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. 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 '16).* ACM Press, 583–594. DOI: https://doi.org/10.1145/2950290.2950361

[27] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17).* IEEE, 949–954. DOI: https://doi.org/10.1109/ASE.2017.8115710

[28] Mingyi Lim, Giovani Guizzo, and Justyna Petke. 2020. Impact of test suite coverage on overfitting in genetic improvement of software. In *Proceedings of the Symposium on Search Based Software Engineering (SSBSE '20).* Springer, 188–203.

[29] Ben Mehne, Hiroaki Yoshida, Mukul R. Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating search-based program repair. In *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST '18).* IEEE, 227–238. DOI: https://doi.org/10.1109/ICST.2018.00031

[30] Justyna Petke, Brad Alexander, Earl T. Barr, Alexander E. I. Brownlee, Markus Wagner, and David R. White. 2023. Program transformation landscapes for automated program modification using Gin. *Empirical Software Engineering* 28, 4 (2023), 104. 1573–7616 https://doi.org/10.1007/s10664-023-10344-5

[31] Justyna Petke and Alexander E. I. Brownlee. 2019. Software improvement with gin: A case study. In *Proceedings of the Search-Based Software Engineering: 11th International Symposium (SSBSE '19).* Springer-Verlag, Berlin, 183–189. DOI: https://doi.org/10.1007/978-3-030-27455-9_14

[32] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. 2018. Genetic improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432. DOI: https://doi.org/10.1109/TEVC.2017.2693219

[33] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2017. Specialising software for different downstream applications using genetic improvement and code transplantation. *IEEE Transactions on Software Engineering* 44, 6 (2017), 574–594. DOI: https://doi.org/10.1109/TSE.2017.2702606

[34] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '13).* IEEE, 180–189. DOI: https://doi.org/10.1109/ICSM.2013.29

[35] Eric M. Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. 2013. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS '13).* Vivek Sarkar and Rastislav Bodík (Eds.), ACM, 317–328. DOI: https://doi.org/10.1145/2451116.2451151

[36] Eric M. Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler software optimization for reducing energy. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS '14).* Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.), ACM, 639–652. DOI: https://doi.org/10.1145/2541940.2541980

[37] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), Article 187, 29 pages. DOI: https://doi.org/10.1145/3360613

[38] Min K. Shin, Sudipto Ghosh, and Leo R. Vijayasarathy. 2022. An empirical comparison of four Java-based regression test selection techniques. *Journal of Systems and Software* 186 (2022), 111174. DOI: https://doi.org/10.1016/j.jss.2021.111174

[39] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[40] Yazhini Venugopal, Phung Quang-Ngoc, and Lee Eunseok. 2020. Modification point aware test prioritization and sampling to improve patch validation in automatic program repair. *Applied Sciences (Switzerland)* 10, 5 (2020), 1–14. DOI: https://doi.org/10.3390/app10051593

[41] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. 2024. User-centric deployment of automated program repair at Bloomberg. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24).* Association for Computing Machinery, New York, NY, 81–91. DOI: https://doi.org/10.1145/3639477.3639756

[42] David H. Wolpert and William G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67–82. DOI: https://doi.org/10.1109/4235.585893

[43] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '15*). Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.), ACM, 1375–1382. DOI: https://doi.org/10.1145/2739480.2754648

[44] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979. DOI: https://doi.org/10.1007/s10664-017-9552-y

[45] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. DOI: https://doi.org/10.1002/stv.430

[46] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 5:1–5:53. DOI: https://doi.org/10.1145/3360004

[47] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19*), 430–441. DOI: https://doi.org/10.1109/ICSE.2019.00056

[48] Shengjie Zuo, Aymeric Blot, and Justyna Petke. 2022. Evaluation of genetic improvement tools for improvement of non-functional properties of software. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '22).* ACM, New York, NY, 1956–1965. DOI: https://doi.org/10.1145/3520304.3534004

Received 10 June 2023; revised 20 January 2024; accepted 14 March 2024