Empirical Comparison of Search Heuristics for Genetic Improvement of Software

Aymeric Blot and Justyna Petke

***Abstract*—Genetic improvement uses automated search to improve existing software. It has been successfully used to optimise various program properties, such as runtime or energy consumption, as well as for the purpose of bug fixing. Genetic improvement typically navigates a space of thousands of patches in search for the program mutation that best improves the desired software property. While genetic programming has been dominantly used as the search strategy, more recently other search strategies, such as local search, have been tried. It is, however, still unclear which strategy is the most effective and efficient. In this paper, we conduct an in-depth empirical comparison of a total of 18 search processes using a set of 8 improvement scenarios. Additionally, we also provide new genetic improvement benchmarks and we report on new software patches found. Our results show that, overall, local search approaches achieve better effectiveness and efficiency than genetic programming approaches. Moreover, improvements were found in all scenarios (between 15% and 68%). A replication package can be found online: https://github.com/bloa/tevc_2020_artefact.**

***Index Terms*—Genetic Improvement, Search-Based Software Engineering, Stochastic Local Search, Genetic Programming**

I. Introduction

GENETIC improvement (GI) [1] uses automated search in order to improve existing software. GI arose as a separate field of research only in the last few years, yet it has already been successfully applied to a wide range of programs. GI has been used to improve both functional properties of software, such as fixing of bugs [2] and addition of new functionality [3], as well as non-functional software properties, such as reduction of software's running time [4], memory [5], and energy [6] consumption. GI-evolved code changes have been adopted into development [7], with a couple of companies using GI during bug fixing process [8], [9].

Roughly speaking, GI takes existing software and mutates it until a variant is found that is better than the original program with respect to some desired property. GI has been applied at the source code [4], binary, and assembly level [10]; however, with explainability and exploitability in mind, most GI works target source code [1]. Mutation granularity also varies, with changes applied at the line [4], abstract syntax tree (AST) [11] and expression level [5]. The most common operators found in the literature are deletion, replacement, and insertion of code found in the original software. There is also work using *transplantation*, injecting code from other programs [3].

The search space for code changes is vast. Considering the simplest case with deletions, replacements, and insertions applied at the line level, there are $l+l*(l-1)+l*(l-1)$ single potential edits, where l is the number of relevant lines in the given software. Furthermore, this number grows exponentially when combinations of edits are considered. Therefore, metaheuristics, such as genetic programming, have been proposed to navigate the GI search space, with successful results.

Many different GI frameworks and toolkits have been developed and used, often using variations in their core search process. Nowadays GI search processes are based on either genetic programming (GP) [12]–[14], with, for example, the GISMOE framework [4], or stochastic local search [15], with, for example, the recent PyGGI [16]–[18] or Gin [19], [20] frameworks. However, though literature search strategies have proven themselves to be effective in practice, they have not yet been empirically compared and analysed.

In order to come closer to answering the question which search strategy is the most efficient and effective in GI, a few studies tried to answer the question of what the search space of code changes looks like in practice [21]. In each work a particular GI framework was used, and none compared the different possible search strategies. Nevertheless, it was shown that the search space is largely neutral, i.e., many mutations lead to software variants that pass the same number of test cases as the original code. If such a search space is navigated efficiently, it will open up great possibilities for improvement of non-functional properties of software, such as running time, energy or memory consumption. It is worth mentioning that most of the studies looked into the search space of code mutations in the context of bug fixing. We believe that part of the reason for this focus is that there is no standard benchmark for GI for improvement of non-functional properties.

In this study, we set up an empirical comparative study to learn more about the GI search process and ultimately find out which search strategy is the most successful in navigating the GI search space. We focus on the following research questions:

RQ1: How generalisable are the generated patches?
RQ2: How consistent is GI performance?
RQ3: How often do GI approaches find improvements and how good are the improvements found?
RQ4: Which search heuristic is overall the most effective?

Specifically, our contributions include:

- a comprehensive empirical study of 18 search strategies;
- new benchmarks and target software for non-functional genetic improvement;
- implementation of new search strategies in PyGGI.

A. Blot and J. Petke are with the Department of Computer Science, University College, London WC1E 6BT, U.K. (e-mail: a.blot@cs.ucl.ac.uk; j.petke@ucl.ac.uk).

This work was supported by UK EPSRC Fellowship EP/P023991/1.

Manuscript received XXXX XX, 20XX; revised XXXX XX, 20XX.

II. Related Work

GI approaches have been traditionally built upon GP systems. In automated program repair (APR), the very influential GenProg tool [11] cemented the idea that genetic approaches could be used to repair bugs and improve software's functional properties. Efficiency of alternative GenProg components (representation, mutation, crossover) have been studied in several studies [22], [23]. There has been an exponential increase in the number of publications on GI in recent years, improving various GI components, and expanding to new programming languages and application domains ([6], [17], [18], and many others). Very few though compare just the search component.

Arcuri [24] compared GP, hill climbing and random search for the purpose of repairing small Java programs, showing superiority of GP in terms of fault fixing ability. Qi et al. [25] implemented a random search strategy within GenProg and concluded that random search can be as effective as GP, whilst improving efficiency of the search process. The authors, however, changed the regression testing strategy for their random search. ASTOR, a comparison framework for APR approaches has also been developed, though no study yet exists that compares just the search component [26]. In the context of improvement of non-functional properties, we are the first to compare different GP search strategies and random search in [27]. That case study suggested that all surveyed approaches were able to produce improved software versions, although resulted in no clear winner. Finally, there have been studies directly investigating the program search space of GI problems [21], for example reporting on large number of neutral variants [28]. Rather than primarily relying on empirical observations, these studies can bring better insight the search process and indirectly hint towards which type of search should be preferred.

III. Genetic Improvement

In this section we deal with the general algorithmic details of GI. First, we present edit sequences, the usual GI solution representation. Then, we detail several types of search strategies[1], which will be compared in the experiments: (1) random search, used as a baseline; (2) genetic programming, the original and search strategy of many GI systems; (3) local search, as implemented in recent GI frameworks—PyGGI [18], Gin [20]. Finally, we also describe the validation procedure shared between all search processes.

A. Edit Sequences

GI work usually processes software at either line or abstract syntax tree (AST) level. If earlier work used to evolve software directly [29], nowadays most work use more lightweight representations. Instead of representing the mutated software with a full copy of the original code, focus is only given to the changes necessary to obtain the desired mutant. With the assumptions that fixed or improved software can be obtained with very little modification, using code already existing in the original software [30], current GI approaches use a sequence of elementary edits that are to be applied in order. Usual types of edits are deletions, replacement, or insertion of code. Because edits are completely disassociated from the actual software—they use the position of the line or the node of the AST—this makes edit sequences very easy to generate and manipulate.

B. Random Search

Random search is arguably the simplest search strategy: every iteration a new mutant is generated, modifying the original software by up to m edits. At the end of the procedure the mutant yielding the best fitness is returned. Random search is a purely exploratory strategy, with no intensification. In particular, combinations of good edits can only happen if those edits are generated during the same iteration. Likewise, if the final incumbent mutant contains multiple edits, no information is carried on whether they all are necessary or if some of them are either unnecessary or even have a negative impact on the final fitness. Random search provides a very simple and natural baseline, whilst also providing many sample-related statistics for the target software.

C. Genetic Programming

Genetic programming (GP) [12] has been used in GI since the inception of the field [29], and since in most of GI work both for the improvement of functional and non-functional properties. In contrast to random search, which was a strict exploratory procedure, GI simultaneously evolves a population of solutions, using both mutation and crossover.

We describe a very simple GP search process used in many GI works. The initial population is obtained by generating n individuals with a single random edit. Then, in each generation invalid individuals are filtered and up to $n/2$ parents are selected, starting with those having the best fitness. For each single parent, another valid individual from the population is selected and used through crossover to obtain a single offspring. Each single parent is then again used to obtain a second offspring, this time through mutation. At the end of the generation, the offspring obtained by crossover and those obtained by mutation are evaluated and combined to form the next population. The valid individuals of the previous generation are considered as parents and sorted by fitness (as defined in Section IV-B). If there are fewer than $n/2$ valid parents, new individuals with a single random edit are generated until the new population reaches n individuals. Multiple types of GP crossover have been proposed in the context of GI, including concatenation crossover (e.g., in [4], [31], [32]), 1-point crossover (e.g., in [2], [33], [34]), or uniform crossover (e.g., in [11]). We detail the four crossover methods compared in the experiments.

Concatenation The edit sequences of the two parents are simply concatenated.

Single-point The edit sequences of the two parents are uniformly split at random into two sub-sequences, and one offspring is created by concatenating the head of the first one with the tail of the second one.

[1] Pseudo code for all algorithms can be found in supplementary material.

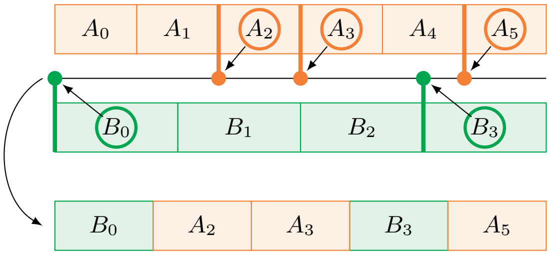

Fig. 1. Uniform interleaved crossover: half of the edits of both parents are selected (here: A_2, A_3, A_5, B_0, and B_3), and then recombined while keeping their original relative positions.

Uniform concatenation The edit sequences of both parents are concatenated, then each edit is dropped with 50% probability.

Uniform interleaved Half of the edits of both parents are selected uniformly at random and then intertwined at their relative positions to create the offspring (see Figure 1).

Finally, mutation of an edit sequence involves with equal probabilities either the deletion of an edit selected uniformly at random (only if the sequence is non-empty) or the creation of a new edit, appended at the end of the sequence.

GP search processes for non-functional GI usually use a subset of test inputs to estimate the quality of software variants, re-sampling a new subset at the beginning of each generation. This re-sampling enables GP to consider a smaller subset, thus speeding the search considerably; if quality estimation is necessarily worse and more noisy, genetic material able to survive multiple generations will ultimately be assessed using more inputs and therefore be more reliable.

In the experiments, we will distinguish two variants of GP: first a GP search process that returns the best individual in the last generation, as described in this section. Then, a GP search process that returns the best individual evaluated over the entire search: for this purpose, and to avoid overfitting to a very small input subset, at the end of each generation the best individual is reassessed over a larger subset of inputs, constant over all generations. This variant will be denoted as GP^r.

D. Local Search

In contrast to GP, local search only relies on mutation to find better software variants. We present three local search strategies: first improvement, best improvement, and tabu search.

1) First Improvement: The random search strategy generates new mutants by applying edits to the original software. Consider instead modifying the previous best mutated software so-far found and the result is a *first improvement* exploration strategy. The original software is now iteratively and gradually improved, building the final mutant step by step rather than hoping to generate it already fully constructed. Other common names for the first improvement strategy include hill climbing, or iterative improvement. Furthermore, neutral neighbours are also accepted, i.e., mutants with different edits but a similar fitness, on the basis that we are accepting (or removing) edits that are not detrimental to the final mutant. Finally, in common with both other local search strategies, first improvement follows the mutation procedure described for GP.

First improvement provides the simplest means of building improved software step by step. Because in this work first improvement is not iterated with restarts, this strategy can be trapped by local optima, i.e., solutions with only neighbours of decreased fitness, or severely hindered by neutral plateaus, i.e., sets of solutions with similar fitness that are connected by the neighbourhood relationship. Finally, because edits with no or insignificant impact will be accepted on the same basis as those resulting in large fitness improvement it can be expected that the edit sequence of the final mutant might contain a proportion of unnecessary edits.

2) Best Improvement: First improvement performs a partial exploration of the neighbourhood, stopping as soon as a non-degrading neighbour is evaluated and accepted. This can lead to a slow convergence of the search process when better neighbours are available. In this situation, an exhaustive exploration of the neighbourhood enables the process to select a more efficient search step by accepting the neighbour achieving the maximal improvement. Unfortunately, GI neighbourhoods are far too large, and neighbours far too expensive to evaluate, to exhaustively explore the neighbourhood of any solution. For that reason, our best improvement strategy compromises by considering reasonable fixed-size subsets of s neighbours of the neighbourhood of the current solution. If no improving neighbour is found, the search does not stop but simply continues generating a new subset of s neighbours.

Best improvement provides an alternative to first improvement that delays the acceptance of edits of insignificant impact, at the cost of an additional parameter, the partial neighbourhood size. The addition of this parameter, however, would be anyway necessary in order to iterate both exploration strategies with the goal of escaping local optima and plateaus.

One possible drawback of the best improvement strategy is that when multiple very good edits are found during the same partial neighbourhood exploration only the best one will be accepted while the other would be simply discarded. However, this drawback could be easily overcome with, for example, maintaining a queue of the most promising edits. For the sake of simple comparison we nevertheless still only consider the simple strategy described in this section.

3) Tabu Search: Both first improvement and best improvement exploration strategies are simple descent local search strategies, which can be trapped in local optima and plateaus. For this reason we consider a third local search-based strategy: tabu search. Based on the best improvement strategy, it enables escaping local optima by accepting the best neighbours from the partial neighbourhood, even if this one worsens the fitness of the current solution. Additionally, to avoid alternating between the same solutions, accepted solutions are stored in a fixed-length *tabu list* and ignored in the next few iterations.

E. Validation Step

It is expected that mutated software produced by GI approaches overfits on training data. Given the separable nature of the GI representation, most GI work includes a validation step that filters in the produced mutant individual edits harmful to generalisation. Following Blot and Petke [27], experiments will use a two-stage validation step.

The first stage does not require running the mutated software. On the assumption that the edit sequence may contain large amounts of bloat, especially with GP search strategies, each edit is sequentially extracted from the edit sequence and the resulting sequence is applied and the resulting software compared to the one using the complete sequence. If the two mutants are identical, then the extracted edit is filtered out. While this procedure does not modify in any way the resulting software, it is very cheap and can greatly reduce the size of the sequence, and thus the cost of the subsequent stage.

The second stage, originally by Petke et al. [32], considers all remaining edits independently, computing their respective individual fitness, before ranking them and constructing the final sequence by applying each one of them in order, keeping only those that improve fitness.

IV. Experimental Setup

In this section, we present the experimental protocol used in our experiments in order to compare GI approaches.

A. Framework

There are currently two general GI frameworks available: PyGGI [16]–[18] and Gin [19], [20]. While both are designed to ease prototyping and quick GI experiments and thus adapted to our research, the later solely focuses on Java software with language specific features, while the former is language-agnostic. As we mostly target C/C++ software but also consider a Java scenario, we chose PyGGI for our experiments.

B. Fitness

We target improvement of a non-functional software property: efficiency in terms of computational speed. This is most easily understood as the time required by a given software to finish its workload. However, measurements are extremely imprecise and can be strongly impacted by the state of the hosting machine, even when considering deterministic software.

To obtain more reliable measurements and better assessment of software quality, the number of lines of code was used in previous work as a proxy measure (e.g., [32]), at the cost of heavy source code instrumentalisation. Furthermore, the same weight was given to every line of code, and impact of calls to standard or external libraries were omitted. We use instead the number of CPU instructions as measured by the Linux `perf`[2] command, as it provides an alternative several orders of magnitude more consistent.

Other GI work on non-functional fitness have targeted for example memory [5] or energy [6] usage. We chose to focus on software speed only, at the risk of less general conclusions, because of its universal relevance and to better control our observations on a single type of scenario: the most common [1].

C. Experimental Protocol

In order to fairly compare our different GI approaches we base our experimental protocol on the standard k-fold cross-validation scheme, introduced in previous GI work [27], with some modifications to better suit the specifics of our GI scenarios. The complete protocol is detailed as follows:

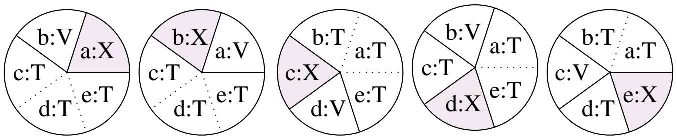

Fig. 2. Example of nested 5-fold cross validation using a single fold for test (X) and $k-1$ folds for both validation (V: single random fold from the remaining $k-1$ folds) and training (T: all remaining $k-2$ folds). Each of the five folds (a to e) is successively used once for the test step (X).

Pre-processing. The unmodified software is first repeatedly run on every instance of the given dataset, in the conditions of the training step, and the software average performance is used to determine the experiment's constants. If the dataset is not already naturally separated into bins of similar instances, those can be fixed at this point.

Cross-validation. Each bin of the dataset is shuffled and divided into k subsets of similar size. The number of folds of the k-fold cross-validation scheme is determined using the size of the bins: it should be possible to divide each bin into k disjoint subsets so that the entire dataset can be divided into k disjoint sets, following the same distribution of instances and large enough so that quality estimation is reliable. The kth fold of the dataset is obtained by considering the kth subset of each bin. One fold is used for the test step, a disjoint one for the validation step, while the $k-1$ others are merged together to be used during the training step.

Training budget. Computed based on the average time T required to run the original software on every single instance of the dataset to be both long enough so that improvements can be found but short enough so that the experiments can be completed in reasonable time. We chose to use one hundred time the average running time on one fold (i.e., $100{*}T/k$). Because not every instance is used and some invalid variants are very quickly discarded (e.g., failure to compile), this budget resulted in the experiments in several hundred to several thousand evaluations, enough for GI to find significant runtime improvements for all benchmarks.

Training. All GI approaches are given equal chance to use any or all of the instances for the $k-2$ folds selected for training. Starting from the original unmodified software, new variants are iteratively considered until the training budget is exhausted and a final individual is returned.

Validation. The software variant resulting from the training step usually overfits to the training instances and can contain bloat. The validation step provides new unseen instances that can be used to investigate every individual modification and construct a final software variant.

Test. Once more, new unseen instances are used to assess the generalisation of the *final* mutated software.

Functional confirmation. The mutated software is run on the entire dataset to ensure its functional properties.

Repetitions. The training, validation, and test steps are repeated as many times as there are folds in the dataset. Each time a different fold is used for the test step, so that every instance is used for generalisation purposes

[2] https://perf.wiki.kernel.org/index.php/Main_Page

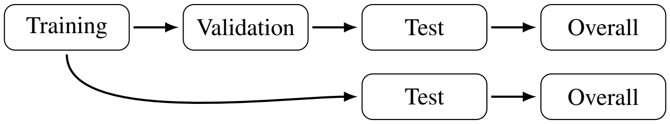

Fig. 3. Experimental protocol for a single search process on a single scenario. For each training run fitness of both validated and non-validated final software variants are assessed over the test fold and all test cases altogether.

exactly once, as pictured in Figure 2 [27] (for five folds).

Statistical validation. The Friedman test is used to detect statistical difference between the different GI approaches and provide an overall ranking. Pairwise comparisons are then examined using the Siegel post hoc analysis when the Friedman test reveals significant statistical differences between approaches.

Figure 3 provides a top-level visualisation of our experimental protocol, for a single search strategy on a single scenario. In addition to the *typical* GI protocol, the fitness of final training software variants are directly reassessed on test data and on all data, bypassing the validation step, in order to provide more information about the search processes and quantify the benefit of the validation step. Note that for each scenario the represented scheme is repeated as many times as there are folds in the dataset, varying the distribution of test cases between each step of the protocol.

D. GI Search Processes

First, as a baseline, we consider four approaches based on random search: $Rand(k)$ denotes that mutants with up to k random edits are generated. We choose k in $1, 2, 5, 10$ to gauge to which extent random search can find combinations of useful edits. Then, we consider several approaches using GP, studying the impact of four different crossover operators: GPc (concatenation), $GP1p$ (single-point), $GPuc$ (uniform concatenation), and $GPui$ (uniform interleaved). For each, we consider a *standard* version and one in which the best individual of every generation is reassessed on a predetermined set of training instances; the later are denoted with an r (e.g., GPc and GPc^r). Following previous work, we use a population of 100 individuals [32]. Finally, we consider approaches using local search: $First$, $Best$, and $Tabu$. For each approach, we also indicate in subscript how many training instances they use: either a single instance drawn from each bin (e.g., GPc_1), or two from each bin in an attempt to avoid overfitting for approaches using a fixed set (e.g., $First_2$). Finally, we investigate variants of every GI approaches when no validation step is performed for a total of $18 * 2 = 36$ GI approaches; for those we reuse the existing results of the computationally expensive training step.

V. Scenarios

A. Selection Criteria

Previous works have repeatedly proven the worth of using GI on many different software optimisation scenarios, considering many different target: software applications, granularity levels, performance measures, and data sets. In order to focus our study on the comparison of search strategies as detailed in Section IV we selected eight various software optimisation scenarios with the selection criteria detailed hereafter.

First, notwithstanding our goal of selecting diverse scenario it was necessary to consider ones that were similar enough to enable general conclusions on search process performance. To that effect, we decided to only focus on improvement of software source code—in contrast to, e.g., improving binary code directly. Then, to implement and fairly compare the different search processes we considered the PyGGI framework [18], because it has been designed for that purpose and is one of the very few agnostic GI frameworks that easily accommodate for target software of multiple programming languages. Additionally, our experimental protocol uses repetitions and cross validation in order to avoid biases due to the selection of training, validation, and test sets. It requires that a sufficient number of input cases are available, so that a reliable fitness can be computed on each of the three steps. Furthermore, because input cases are equiprobably distributed among the three steps, that improvements can be found on the training distribution—studies such as Bowtie in [4], that use different distributions for training and test, cannot therefore be considered. Finally, GI experiments are notoriously computationally expensive, because many target software variants have to considered in order to find improvements. To keep the cost of our experiments reasonable it is necessary to keep training budget relatively short; it follows that only software for which compilation and fitness assessment is short enough is selected.

B. SAT Solvers

MiniSAT[3] is a well-known Boolean satisfiability (SAT) solver. It has been used several times in previous GI work [18], [31], [32], [35], [36]. We consider two implementations of MiniSAT: the latest C++ one by its original authors (minisat-2.0.0) and a Java reimplementation, Sat4j[4] version 2.3.4. We evolve the main solving algorithm `core/Solver.cc` and `org/sat4j/minisat/core/Solver.java`, respectively. The GI scenario we consider is very simple. The only constraint we impose on mutated software is that it should be able to produce, compared to the original software, the same satisfiability result for all tested SAT instances.

1) Combinatorial Interaction Testing: To assess the performance of the considered SAT solvers, we first consider instances from the combinatorial interaction testing (CIT) field as used in previous work [27], [32]. Due to previous concerns about the size and heterogeneity of the dataset [27], we included some additional hard instances and reorganised the bins to improve relevance of fitness estimation: instead of 130 instances separated into five bins (two for SAT instances, two for UNSAT, one for hard instances) we consider 146 instances separated into eight bins (four for SAT instances, four for UNSAT), giving more weight to harder instances and making sure that all estimates are based on multiple SAT and UNSAT hard instances.

[3] http://minisat.se/MiniSat.html
[4] http://www.sat4j.org/

TABLE I
SCENARIO OVERVIEW

Software	Language	# files	# AST nodes	Dataset	# runs	Running time	# bins	# folds	Training budget
MiniSAT	C++	1	429	CIT	146	5.5 minutes	8	8	1.1 hours
MiniSAT	C++	1	429	Uniform-3-SAT	1000	10.8 minutes	10	10	1.8 hours
Sat4j	Java	1	962	Uniform-3-SAT	1000	23.0 minutes	10	10	3.8 hours
OptiPNG	C	2	$790 + 375$	Colour Images	100	5.6 minutes	5	5	2.0 hours
OptiPNG	C	2	$790 + 375$	Greyscale Images	100	2.7 minutes	5	5	0.9 hours
OptiPNG	C	2	$790 + 375$	Mixed Images	200	8.3 minutes	10	10	1.4 hours
MOEA/D	C++	2	$171 + 151$	Functions	45	2.9 minutes	1	9	0.4 hours
NSGA-II	C++	2	$192 + 151$	Functions	45	14.6 minutes	1	9	2.2 hours

2) Uniform Random-3-SAT: In addition to CIT instances, we also consider instances from the SATLIB [37] benchmark collection. In particular, we use 1000 instances from the Uniform Random-3-SAT benchmark[5], generated at the phase transition [38], i.e., as it happens the region of the 3-SAT problem space associated to the hardest 3-SAT formulas. We consider ten bins of 100 instances, with 50, 100, 150, 200, and 250 variables: five of satisfiable instances and five of unsatisfiable instances, for a total of 1000 instances.

C. PNG Optimisation

For this scenario, we target OptiPNG, a PNG file size optimiser. PNG files are images that are stored using lossless compression. The PNG specification allows many different types of images types and combinations of options; there are many ways to encode a specific matrix of pixels. The space of possible compression configurations being computationally intractable, OptiPNG simply applies in order increasingly large subset of manually selected schemes. While to our present knowledge no GI work previously targeted OptiPNG, it already had some vulnerabilities fixed by fuzzing, another field of automated software engineering. We target two files, `src/optipng/optipng.c` and `src/optipng/optim.c`, containing the code pertaining to select and run the selected compression schemes. We constrain mutated software to those able to produce output PNG of size as least as small as the original OptiPNG.

1) Colour and Greyscale Benchmarks: We consider three benchmarks of PNG images, following the dataset used in Petke et al. [32]. There are in total ten sets of twenty images from various types and sources, half of them colour and half of them greyscale, of geometric shapes, face images, houses, galaxies, and scenes from everyday life, respectively.

The first benchmark consists of the five sets of colour images, the second benchmark consists of the five sets of greyscale images, while the third benchmark consists of all the ten sets. Overall, OptiPNG reduces the size of 13 images by less than 0.1%, of 69 images by 0.1%–1%, of 89 images by 1%–5%, of 20 images by 5%–10%, and 9 images by more than 10% (average: 2.78%, best: 26.03%).

D. Multi-Objective Evolutionary Algorithms

NSGA-II [39] and MOEA/D [40] are both extremely popular and wide-spread evolutionary algorithms for multi-objective optimisation. They have both stood the test of time and are still nowadays used as a baseline for many new multi-objective approaches. We reuse the experimental scenario of Li and Zhang [41], in which the authors of MOEA/D compare two C++ implementations of MOEA/D and NSGA-II over "complicated Pareto sets". In practice both implementations partially share the same C++ code, in which nine functions to optimise are hardcoded.

We consider two scenarios, in which the source code of MOEA/D and NSGA-II are evolved separately. In the first one, we target the files `DMOEA/dmoeafunc.h` and `common/recombination.h`, while in the second one we target the files `NSGA2/nsga2func.h` and `common/recombination.h`. In the original paper the two algorithms are compared using a single run on each function. To get more generalisable results, we compute the final fitness by averaging several individual runs using different starting random seeds. For both algorithms we will use five random seeds, as a trade-off between more reliable fitness quality assessment and running time costs. In both scenarios, we constrain the mutated evolutionary algorithm to those able to produce reasonable Pareto sets on each of the nine functions. By *reasonable* we mean here Pareto sets no worse than a given percentage compared to the original algorithm on the same function. We report results obtained using a 110% threshold (i.e., accepting software variants producing solution quality no worse than 10% worse than the original software).

E. Scenario Overview

Table I summarises the eight GI scenarios we consider in the experiments. For software, the table details the language, the number of files evolved and the size of the AST; for datasets it shows the total number of instances available before cross validation, the time to run all instances, the number of bins instances have been classified into, and how many times the GI process will be repeated. Finally, Table I also shows the respective complete training budgets—i.e., the time to perform as many training steps as there are folds in the protocol.

Note that we will not consider the combination of Sat4j and CIT instances. First, Sat4j is extremely slow in solving every instance of the CIT dataset (7.4 hours), making it by far the most time-expensive scenario. Then, following previous use of the dataset [32] binning was done as to follow the MiniSAT performance distribution, which we found to not correspond to Sat4j performance.

[5] https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

TABLE II
TOTAL EXPERIMENTAL RUNNING TIME FOR EACH STAGE [IN DAYS]

Scenario	Training	Validation	Test	Overall
MiniSAT (CIT)	7.0	0.8	0.1	0.9
MiniSAT (Uniform)	13.4	3.0	0.3	3.2
Sat4j (Uniform)	29.4	3.2	0.8	6.7
OptiPNG (Colour)	8.9	1.1	0.2	1.1
OptiPNG (Grey)	4.3	0.6	0.1	0.5
OptiPNG (Both)	12.4	1.2	0.3	3.2
MOEA/D (110%)	3.9	0.4	0.1	0.9
NSGA-II (110%)	18.6	2.3	0.5	4.4
Total	97.9	12.6	2.5	20.8

TABLE III
PERCENTAGE OF FINAL MUTANTS YIELDING $> 5\%$ IMPROVEMENT.

Scenario	Training	Validation	Test	Overall
MiniSAT (CIT)	95.8%	57.6%	36.8%	4.9%
MiniSAT (uniform)	92.8%	87.2%	85.0%	81.7%
Sat4j (uniform)	60.6%	19.4%	18.9%	16.1%
OptiPNG (colour)	24.4%	17.8%	17.8%	17.8%
OptiPNG (grey)	24.4%	17.8%	17.8%	17.8%
OptiPNG (both)	15.6%	11.7%	8.3%	8.3%
MOEA/D (110%)	49.4%	45.1%	42.6%	40.1%
NSGA-II (110%)	39.5%	34.6%	34.0%	29.6%

VI. RESULTS[6]

Training and validation steps were conducted in parallel on four cores of a dedicated (8×3.4GHz, 16GB RAM) Intel i7-2600 machine, running CentOS-7 with Linux kernel 3.10.0 and GCC 4.8.5. Subsequent steps—test, functional confirmation—were conducted sequentially on a single core. While all C++ software are single-threaded, Sat4j compilation and execution use multi-threading. To ensure fair experiments, all Sat4j runs used two cores each, enforced using CPU affinity. Finally, compilation is performed as instructed by the original software; for example, MiniSAT uses the `-O3` optimisation option.

Table II reports, for each step of the experimental protocol and for each of the eight scenarios, the combined running time of all runs of all approaches. Due to the number of approaches and the repetitions of the cross-validation scheme, despite considering very short training budgets training runs alone required more than three months of CPU time.

In this section, we first investigate the relevance of the chosen scenarios: how generalisable are the generated patches to unseen tests (RQ1) and how consistent is GI performance (RQ2). We then discuss the best patches found for every scenario. Finally we comment on the success rate of GI (RQ3) and compare the performance of every GI approach (RQ4).

A. Scenario Analysis

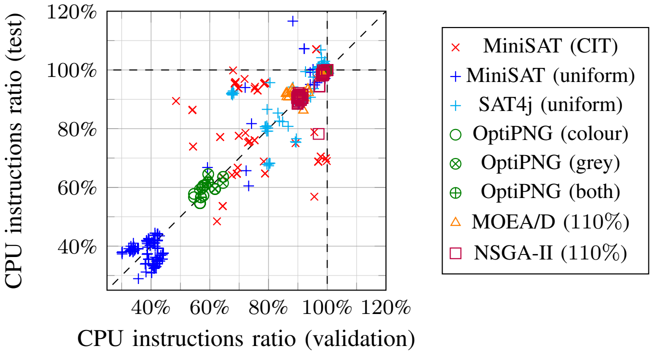

Fig. 4. Comparison between fitness on a single fold (validation & test steps).

Table III shows for each scenario the percentage of GI runs yielding a final software yielding a final software at each step: after the search, on the validation fold, on the test fold, and overall on all instances, with the additional constraint of being at least 5% faster than the original software. First, results on the MiniSAT (CIT) benchmark show that even while manually accounting with instance heterogeneity and fine tuning the separation into bins most GI approaches fail to generalise; although better than in previous work [27], many of the validated software fail on test instances, and almost none generalise to every instance of the dataset. On all other benchmarks, while a consequent drop may be observed after training, due to training overfit, variants that undergo validation usually generalise on all instances. Solutions to reduce this overfit include considering more instances or reducing the number of folds, to refine fitness approximation on each fold, or considering more instances during training, slowing the GI process by considering more reliable approximations.

Answer to RQ1: (Generalisability) On average, 84% of the significant improvements confirmed during validation are also significant in the entire dataset (Table III, ratio between columns *Validation* and *Overall*). Excluding the more heterogeneous MiniSAT (CIT) scenario, the rate raises to 95%.

Figure 4 shows the correlation between the fitness of the final mutant on the validation and test folds. Mutants failing to generalise to the test fold are not represented. Similarity between fitness values indicate that the two selected folds follow similar distributions of instances. This is the case for all scenarios except clearly the MiniSAT (CIT) one and in a lesser measure the two other SAT scenarios. Additionally, fitness values for all other scenarios are generally well clustered, meaning that successful GI runs found very similar improvements.

Table IV presents, for each approach, the performance ratio of the software variant the most efficient across all repetitions of the approach on each scenario, computed using all available instances. A ratio of 100% indicates that the best of the final software variant performs exactly the same as the original software. No ratio is indicated when none of runs using the approach resulted in a better software variant. The best ratio for each scenario is emboldened.

Answer to RQ2: (Consistency) Speedups from 15% to 68% can be found for all scenarios, with difficulties greatly differing from scenario to scenario. Manual investigation of final patches revealed further increased improvements. While performance largely depends of the scenario considered, local search and genetic programming found identical or semantically equivalent software variants in most cases.

[6] Additional training statistics, complete statistical analysis, and details and performance of patches found are provided in the supplementary material.

TABLE IV
BEST SINGLE FITNESS RATIO FOUND AFTER VALIDATION, USING ALL INSTANCES FROM ALL FOLDS.

	MiniSAT		Sat4j	OptiPNG			MOEA/D	NSGA-II
Algo	CIT	Uniform	Uniform	Colour	Grey	Both	110%	
$Rand_2(1)$	—	38%	**84%**	—	99%	**59%**	**90%**	86%
$Rand_2(2)$	—	39%	**84%**	**57%**	62%	100%	**90%**	91%
$Rand_2(5)$	—	**32%**	**84%**	**57%**	62%	**59%**	**90%**	91%
$Rand_2(10)$	—	38%	**84%**	**57%**	62%	**59%**	**90%**	91%
GP_1c	98%	36%	**84%**	—	99%	**59%**	**90%**	91%
$GP_1 1p$	97%	38%	**84%**	**57%**	99%	100%	**90%**	91%
$GP_1 uc$	87%	38%	**84%**	—	—	100%	**90%**	91%
$GP_1 ui$	97%	38%	**84%**	—	99%	—	**90%**	**85%**
$GP_1^r c$	87%	38%	**84%**	**57%**	99%	**59%**	**90%**	91%
$GP_1^r 1p$	—	36%	**84%**	—	62%	100%	**90%**	91%
$GP_1^r uc$	87%	38%	**84%**	**57%**	62%	100%	**90%**	**85%**
$GP_1^r ui$	87%	38%	**84%**	—	99%	100%	**90%**	**85%**
$First_1$	97%	35%	100%	**57%**	**61%**	**59%**	**90%**	89%
$Best_1$	**81%**	36%	**84%**	**57%**	62%	**59%**	**90%**	90%
$Tabu_1$	—	36%	99%	**57%**	62%	**59%**	**90%**	91%
$First_2$	97%	36%	**84%**	**57%**	62%	**59%**	**90%**	90%
$Best_2$	**81%**	36%	99%	**57%**	62%	**59%**	**90%**	90%
$Tabu_2$	**81%**	36%	99%	**57%**	62%	**59%**	**90%**	90%

B. Mutated Software Comparison

The changes leading to the biggest performance improvements are discussed below. They have all been manually verified to be semantically valid.

1) MiniSAT: The largest runtime improvement (22%) in the optimisation of MiniSAT for the CIT domain scenario comes from the addition of a `return false` statement in the `litRedundant` method, which disables a nonessential search optimisation. In the case where the solver was run on the uniform random SAT instances, one mutation that led to 61% speed-up simply cancelled restarts that would have happened after a specified number of conflicts reached. Interestingly, similar improvement was obtained by halving the number of conflicts required before a restart.

2) Sat4J: Interestingly, similar optimisations were found in Sat4J. In particular, a replacement operation caused the solver to restart more frequently, leading to 84% speedup. Disabling recording of learnt clauses also led to similar speed-ups.

3) OptiPNG: In all three scenarios the best performances are found using a single edit: in `optipng/optim.c`, when applying the selected PNG format configuration, calls to the `png_set_compression_window_bits()` function are deleted. This function is used by libpng to setup in zlib the size of the sliding window of the LZ77 compression algorithm. While not a parameter explicitly optimised by OptiPNG, the compression size is set to its smallest value in some cases; the reasons for this constraint are unclear. By removing it, zlib always uses the default/largest value, and OptiPNG runs use on average 40% less CPU instructions and are 20% faster.

4) MOEA/D: Both MOEA/D and NSGA-II implementations periodically compute the current population fitness for display purposes. For MOEA/D, this intermediary fitness computation amounts to 10% of the total work load, and can be disabled without any consequence. Many similar software variants have been found, including removing the computation, modifying the reference front filename so it is not found, preventing it being loaded, emptying it, or trivialising the fitness computation function. These mutations are only valid because our GI scenario recomputes the final fitness externally..

5) NSGA-II: For NSGA-II, the removal of the intermediary fitness computation is also found, but only yields a 1.4% improvement. Instead, several algorithmic changes are found. Density evaluation can be modified for a 13% speed improvement and at the minor cost of 3% solution quality (removal of a `rank++` statement in `CNSGA2::eval_dens`). The `CNSGA2::fill_union` function can be tweaked to always accept new individuals disregarding repetitions for a 11% speed improvement with no impact on the final solution quality. Finally, the best variant—found during training only—also simplifies the the dominance ranking procedure for an overall speedup of 50% and a 0.2% solution quality loss.

C. Evolutionary Computation Comparison

1) Overall Results: Table V presents the testing results for each approach. The cross-validation results in k runs, from which we report the best and median fitness values, together with the percentage of runs yielding improved performance.

Answer to RQ3: (Performance) With the notable exception of the MiniSAT (uniform) scenario, for which almost every GI run yielded excellent improvements, most GI runs were unfruitful, which can be clearly seen with very low supports and median performance close to 100%. This can in part be attributed to the very short training budget allowed to GI approaches, necessitated by our large-scale comparison study. However, longer training budgets will not necessarily always lead to better final results as risks of overfitting will also increase. Across all repetitions the best performances have been obtained in around 12% of all runs.

Focusing on the 18 approaches using validation, there are statistical differences in the performance on the test step. Table VI presents the results of the Siegel post hoc analysis. Statistical analysis and ranking use all independent

TABLE V
BEST AND MEDIAN FITNESS RATIO WITH PROPORTION OF SUCCESSFUL RUNS, USING ALL INSTANCES FROM THE SINGLE TEST FOLD.

	MiniSAT (CIT)			MiniSAT (Uniform)			Sat4j (Uniform)			OptiPNG (Colour)		
Algo	Best	Median	Success	Best	Median	Success	Best	Median	Success	Best	Median	Success
$Rand_2(1)$	79%	—	(38%)	34%	40%	(90%)	68%	96%	(50%)	—	—	(0%)
$Rand_2(2)$	65%	—	(38%)	34%	40%	(90%)	68%	**90%**	(70%)	57%	—	(20%)
$Rand_2(5)$	57%	—	(38%)	**29%**	40%	(90%)	68%	99%	(70%)	57%	—	(40%)
$Rand_2(10)$	—	—	(0%)	34%	40%	(90%)	**67%**	—	(40%)	**55%**	—	(20%)
GP_1c	69%	96%	(62%)	31%	42%	(100%)	92%	99%	(70%)	—	—	(0%)
GP_11p	69%	96%	(75%)	34%	43%	(90%)	79%	—	(40%)	58%	—	(20%)
GP_1uc	65%	96%	(62%)	34%	40%	(80%)	80%	—	(30%)	—	—	(0%)
GP_1ui	69%	—	(38%)	33%	40%	(100%)	81%	—	(40%)	—	—	(0%)
GP^r_1c	64%	—	(38%)	33%	39%	(100%)	81%	100%	(50%)	**55%**	—	(20%)
GP^r_11p	54%	96%	(62%)	31%	41%	(90%)	90%	—	(30%)	—	—	(0%)
GP^r_1uc	54%	98%	(50%)	33%	40%	(100%)	91%	—	(40%)	58%	—	(20%)
GP^r_1ui	54%	98%	(50%)	34%	39%	(100%)	92%	—	(40%)	—	—	(0%)
$First_1$	64%	84%	(100%)	31%	39%	(100%)	98%	100%	(50%)	**55%**	—	(40%)
$Best_1$	69%	90%	(88%)	32%	41%	(90%)	81%	100%	(60%)	**55%**	—	(40%)
$Tabu_1$	**48%**	**82%**	(88%)	32%	39%	(90%)	98%	—	(20%)	**55%**	—	(40%)
$First_2$	65%	99%	(62%)	33%	40%	(90%)	68%	99%	(70%)	**55%**	—	(20%)
$Best_2$	65%	85%	(88%)	32%	39%	(90%)	98%	100%	(50%)	**55%**	—	(20%)
$Tabu_2$	65%	85%	(88%)	33%	**38%**	(100%)	98%	100%	(50%)	**55%**	—	(20%)

	OptiPNG (Grey)			OptiPNG (Both)			MOEA/D (110%)			NSGA-II (110%)		
Algo	Best	Median	Success	Best	Median	Success	Best	Median	Success	Best	Median	Success
$Rand_2(1)$	99%	—	(20%)	60%	—	(30%)	**86%**	93%	(100%)	**78%**	**98%**	(78%)
$Rand_2(2)$	64%	99%	(60%)	100%	—	(40%)	90%	**92%**	(89%)	91%	99%	(67%)
$Rand_2(5)$	61%	**64%**	(100%)	59%	**100%**	(70%)	92%	93%	(89%)	90%	99%	(56%)
$Rand_2(10)$	**59%**	—	(40%)	**57%**	—	(30%)	92%	**92%**	(100%)	91%	99%	(78%)
GP_1c	99%	—	(20%)	62%	—	(20%)	91%	98%	(89%)	90%	99%	(78%)
GP_11p	99%	—	(20%)	100%	—	(10%)	92%	98%	(67%)	91%	—	(44%)
GP_1uc	—	—	(0%)	100%	—	(20%)	91%	98%	(100%)	90%	99%	(67%)
GP_1ui	99%	—	(20%)	—	—	(0%)	91%	98%	(100%)	90%	99%	(67%)
GP^r_1c	99%	—	(40%)	62%	—	(10%)	91%	98%	(100%)	90%	99%	(78%)
GP^r_11p	**59%**	99%	(60%)	100%	—	(10%)	91%	98%	(100%)	89%	**98%**	(78%)
GP^r_1uc	65%	—	(20%)	100%	—	(10%)	91%	98%	(100%)	90%	99%	(67%)
GP^r_1ui	99%	—	(20%)	100%	—	(10%)	92%	98%	(89%)	90%	**98%**	(78%)
$First_1$	**59%**	99%	(60%)	**57%**	**100%**	(50%)	**86%**	98%	(78%)	89%	99%	(56%)
$Best_1$	**59%**	99%	(60%)	60%	—	(30%)	91%	98%	(89%)	88%	—	(44%)
$Tabu_1$	**59%**	99%	(60%)	**57%**	—	(30%)	**86%**	98%	(78%)	88%	**98%**	(67%)
$First_2$	**59%**	—	(40%)	60%	—	(30%)	92%	98%	(78%)	89%	99%	(56%)
$Best_2$	**59%**	—	(40%)	**57%**	—	(30%)	92%	98%	(89%)	90%	**98%**	(78%)
$Tabu_2$	**59%**	—	(40%)	60%	—	(10%)	92%	98%	(78%)	90%	**98%**	(67%)

runs without averaging by scenario. Results are presented pairwise, a dot ($\cdot$) indicating a small statistical difference ($p < 0.1$) and asterisks significant statistical differences ($*$: $p < 0.05$; $**$: $p < 0.01$; ⁂: $p < 0.001$).

Answer to RQ4: (Overall) Local search approaches clearly outperform all other approaches on the test data. Approaches using genetic programming and random search perform similarly. However, there are overall not much statistical differences between approaches: when computing Vargua-Delaney A_{12} [42] significant statistical differences are tied to small effect size, with very few medium effect sizes. These results are a clear sign that current GI performance could be significantly improved if better search approaches are considered, e.g., based on local search heuristics.

2) Random-based Approaches: There is no significant statistical difference between the 8 random-based approaches on the test set (Friedman test: $p = 0.088$). Still, we find that the validation step has overall a slightly adverse impact on performance; this can be explained by the lack of intensification-related overfit and marginal false-positive during filtering.

3) GP-based Approaches: There are statistical differences between the 16 GP-based approaches on the test set (Friedman test: $p = 0.021$). Table VII shows the results of the Siegel post hoc analysis. While no single approach statistically outperforms every other, some observations still can be made. Every approach using reassessment of the best solution every generation outranks its non-reassessing counterpart, despite being slower. Similarly, approaches using the uniform interleaved crossover always outranks their counterpart using the uniform concatenation crossover. Overall, the best three GP-based approaches use the concatenation crossover.

GP-based approaches produced significantly fewer software variants than other approaches. The reason lies in the fact that the crossover of valid parents has a high probability of also producing valid offspring. In practice, GP use most of the training budged recombining known mutations, while local search have more opportunities to explore software variants.

TABLE VI
SIEGEL POST HOC STATISTICAL ANALYSIS, ALL VALIDATED APPROACHES, ALL TEST DATA. (FRIEDMAN TEST: $p = 4.1 \times 10^{-8}$)

Approach	$First_1$ (v)	$Best_2$ (v)	$Tabu_1$ (v)	$Tabu_2$ (v)	$Best_1$ (v)	$First_2$ (v)	$Rand_2(5)$ (v)	GP^r_1c (v)	GP_1c (v)	$Rand_2(2)$ (v)	GP^r_11p (v)	$Rand_2(10)$ (v)	GP^r_1ui (v)	GP_1ui (v)	GP^r_1uc (v)	GP_1uc (v)	$Rand_2(1)$ (v)	GP_11p (v)	Rank
$First_1$ (v)						*	*	*	*	**	***	***	***	***	***	***	***	***	6.95
$Best_2$ (v)										·	*	**	**	**	**	**	***	***	7.82
$Tabu_1$ (v)											*	*	*	**	**	**	**	***	8.11
$Tabu_2$ (v)											*	*	*	**	**	**	**	***	8.20
$Best_1$ (v)											*	*	*	**	**	**	**	***	8.21
$First_2$ (v)	*													·	·	*	*	**	8.96
$Rand_2(5)$ (v)	*													·	·	·	*	**	9.00
GP^r_1c (v)	*															·	·	**	9.25
GP_1c (v)	*																·	**	9.33
$Rand_2(2)$ (v)	**	·															·	**	9.36
GP^r_11p (v)	***	*	*	*	*													·	10.14
$Rand_2(10)$ (v)	***	**	*	*	*														10.35
GP^r_1ui (v)	***	**	*	*	*														10.47
GP_1ui (v)	***	**	**	**	**	·	·												10.64
GP^r_1uc (v)	***	**	**	**	**	·	·												10.67
GP_1uc (v)	***	**	**	**	**	*	·	·											10.81
$Rand_2(1)$ (v)	***	***	**	**	**	*	*	·	·	·									10.92
GP_11p (v)	***	***	***	***	***	**	**	**	**	**	·								11.84

TABLE VII
SIEGEL POST HOC STATISTICAL ANALYSIS, ALL GP-BASED APPROACHES, ALL TEST DATA. (FRIEDMAN TEST: $p = 0.021$)

Approach	GP^r_1c (v)	GP_1c (v)	GP^r_1c	GP^r_11p (v)	GP^r_11p	GP^r_1ui (v)	GP^r_1ui	GP^r_1uc (v)	GP^r_1uc	GP_1ui (v)	GP_1ui	GP_1uc (v)	GP_1uc	GP_1c	GP_11p	GP_11p (v)	Rank
GP^r_1c (v)										·	·	*	*	*	**	***	7.20
GP_1c (v)											·	·	*	*	*	***	7.30
GP^r_1c												·	*	*	*	**	7.51
GP^r_11p (v)														·	·	**	7.89
GP^r_11p															·	*	8.00
GP^r_1ui (v)																*	8.32
GP^r_1ui																·	8.51
GP^r_1uc (v)																·	8.52
GP^r_1uc																·	8.55
GP_1ui (v)	·															·	8.58
GP_1ui	·	·														·	8.67
GP_1uc (v)	*	·	·														8.90
GP_1uc	*	*	*														9.23
GP_1c	*	*	*	·													9.33
GP_11p	**	*	*	·	·												9.42
GP_11p (v)	***	***	**	**	*	*	·	·	·	·	·						10.07

4) Local Search-based Approaches: There are statistical differences between the 12 local search approaches on the test set (Friedman test: $p = 6.7 \times 10^{-4}$). Table VIII shows the results of the Siegel post hoc analysis. Very clearly, approaches using the validation step outrank approaches that do not, confirming the downside of local search approaches to overfit to the training data. Interestingly, approaches using more training instances perform worse, trend that should likely reverse itself with the use of longer training budgets.

TABLE VIII
SIEGEL POST HOC STATISTICAL ANALYSIS, ALL LOCAL SEARCH APPROACHES, ALL TEST DATA. (FRIEDMAN TEST: $p = 6.7 \times 10^{-4}$)

Approach	$First_1$ (v)	$Best_1$ (v)	$Tabu_1$ (v)	$Best_2$ (v)	$Tabu_2$ (v)	$First_2$ (v)	$First_1$	$Best_2$	$Tabu_1$	$Best_1$	$Tabu_2$	$First_2$	Rank
$First_1$ (v)			·	**	**	**	**	***	***	***	***	***	4.70
$Best_1$ (v)								·	*	*	*	*	5.70
$Tabu_1$ (v)	·									·	*	*	5.91
$Best_2$ (v)	**												6.34
$Tabu_2$ (v)	**												6.63
$First_2$ (v)	**												6.64
$First_1$	**												6.72
$Best_2$	***	·											6.92
$Tabu_1$	***	*											6.94
$Best_1$	***	*	·										7.03
$Tabu_2$	***	*	*										7.16
$First_2$	***	*	*										7.30

VII. CONCLUSIONS

In this article, we investigated how search heuristics influence effectiveness and efficiency of genetic improvement processes, in the context of non-functional improvement. Using a thorough experimental protocol based on cross-validation, we compared the performance of 18 search processes across 8 various improvement scenarios, concerned with runtime improvement. Due to a lack of standard GI scenarios in the literature that would be suitable for easy cross-comparisons, we proposed new benchmarks and target software, for which possible improvements and found patches are reported. In particular, for one of the selected software, OptiPNG, a particularly interesting patch has been shared through a pull request. Similarly, in addition to the implementation of the various search processes described in this paper, multiple patches and improvements have been shared with the PyGGI developers.

Regarding the performance of search processes, our results indubitably show that GI still is a very young research field, as most approaches considered fail to produce consistent results, while random search, usually well crushed in most previous work, was able to produce here remarkably good results. Genetic programming approaches, usually preferred as the original and traditional search process, are in our experiments bested by local search based approaches. Overall, all results point to the importance of using well suited search process, while simultaneously indicating the very large gap that is yet to be filled regarding optimal GI performance.

In future work, we intend to better study the fitness landscape of GI scenarios. Indeed, little is known about them, with a short survey published only last year [21]. If our results showed evidence of the possible improvements a more suitable search process can yield, there is no doubt fitness landscape analysis will lead to a deeper understanding of what evolutionary process should be preferred for GI. Moreover, one could also explore how the best performing approaches evolve against time. Finally, another possible direction for future research would be to investigate in more detail the validation step of GI approaches and how to better recognise promising combinations of edits.

References

[1] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, 2018.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[3] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 257–269.

[4] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 1, pp. 118–135, 2015.

[5] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 1375–1382.

[6] J. Dorn, J. Lacomis, W. Weimer, and S. Forrest, "Automatically exploring tradeoffs between software output fidelity and energy costs," *IEEE Trans. Softw. Eng.*, vol. 45, no. 3, pp. 219–236, 2019.

[7] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman, "Improving CUDA DNA analysis software with genetic programming," in *Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 1063–1070.

[8] S. O. Haraldsson, J. R. Woodward, A. E. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: How genetic improvement became an overnight success," in *Genetic and Evolutionary Computation Conference*. ACM, 2017.

[9] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: automated end-to-end repair at scale," in *International Conference on Software Engineering: Software Engineering in Practice*. IEEE / ACM, 2019, pp. 269–278.

[10] E. Schulte, W. Weimer, and S. Forrest, "Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Genetic and Evolutionary Computation Conference*. ACM, 2015.

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

[12] J. R. Koza, *Genetic programming – On the programming of computers by means of natural selection*, ser. Complex adaptive systems. MIT Press, 1992.

[13] W. B. Langdon and R. Poli, *Foundations of genetic programming*. Springer, 2002.

[14] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. lulu.com, 2008.

[15] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[16] G. An, J. Kim, S. Lee, and S. Yoo, "PyGGI: Python General framework for Genetic Improvement," in *Korea Software Congress*, 2017, pp. 536–538.

[17] G. An, J. Kim, and S. Yoo, "Comparing Line and AST granularity level for program repair using PyGGI," in *International Workshop on Genetic Improvement*. ACM, 2018, pp. 19–26.

[18] G. An, A. Blot, J. Petke, and S. Yoo, "PyGGI 2.0: Language independent genetic improvement framework," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 1100–1104.

[19] D. R. White, "Gi in no time," in *Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1549–1550.

[20] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *Genetic and Evolutionary Computation Conference*. ACM, 2019, pp. 985–993.

[21] J. Petke, B. Alexander, E. T. Barr, A. E. Brownlee, M. Wagner, and D. R. White, "A survey of genetic improvement search spaces," in *Genetic and Evolutionary Computation Conference*. ACM, 2019, pp. 1715–1721.

[22] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Genetic and Evolutionary Computation Conference*. ACM, 2012, pp. 959–966.

[23] V. P. L. Oliveira, E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior, "Improved representation and genetic operators for linear genetic programming for automated program repair," *Empir. Softw. Eng.*, vol. 23, no. 5, pp. 2980–3006, 2018.

[24] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, 2011.

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

[26] M. Martinez and M. Monperrus, "ASTOR: a program repair library for Java (demo)," in *International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[27] A. Blot and J. Petke, "Comparing genetic programming approaches for non-functional genetic improvement – Case study: Improvement of MiniSAT's running time," in *European Conference on Genetic Programming*, ser. LNCS, vol. 12101. Springer, 2020, pp. 68–83.

[28] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A journey among Java neutral program variants," *Genet. Program. Evolvable. Mach.*, vol. 20, no. 4, pp. 531–580, 2019.

[29] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Congress on Evolutionary Computation*. IEEE, 2008, pp. 162–168.

[30] E. T. Barr, , Y. Brun, , P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 306–317.

[31] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *European Conference on Genetic Programming*, ser. LNCS, vol. 8599. Springer, 2014, pp. 137–149.

[32] ——, "Specialising software for different downstream applications using genetic improvement and code transplantation," *IEEE Trans. Softw. Eng.*, vol. 44, no. 6, pp. 574–594, 2018.

[33] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference*. ACM, 2009, pp. 947–954.

[34] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[35] J. Petke, W. B. Langdon, and M. Harman, "Applying genetic improvement to MiniSAT," in *International Symposium on Search Based Software Engineering*, ser. LNCS, vol. 8084. Springer, 2013, pp. 257–262.

[36] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 1327–1334.

[37] H. H. Hoos and T. Stützle, "SATLIB: An online resource for research on SAT," in *Satisfiability*. IOS Press, 2000, pp. 283–292.

[38] I. P. Gent and T. Walsh, "The SAT phase transition," in *European Conference on Artificial Intelligence*, 1994, pp. 105–109.

[39] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.

[40] Q. Zhang and H. Li, "MOEA/D: A multiobjective evolutionary algorithm based on decomposition," *IEEE Trans. Evol. Comput.*, vol. 11, no. 6, pp. 712–731, 2007.

[41] H. Li and Q. Zhang, "Multiobjective optimization problems with complicated Pareto sets, MOEA/D and NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 13, no. 2, pp. 284–302, 2009.

[42] A. Vargha and H. D. Delaney, "A critique and improvement of the "cl" common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000.

Aymeric Blot received the Doctoral degree in computer science from the University of Lille, Lille, France, with a focus on automated algorithm design. He is a Research Associate at the Centre for Research on Evolution, Search and Testing, University College London, London, U.K.

Justyna Petke is a Principal Research Fellow and Proleptic Associate Professor at the Centre for Research on Evolution, Search and Testing, University College London, London, U.K. She has published articles on the applications of genetic improvement.