Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement

Case Study: Improvement of MiniSAT's Running Time

Aymeric Blot[0000−0003−0485−5279] and Justyna Petke[0000−0002−7833−6044]

CREST, University College London, London WC1E 6BT, UK
a.blot@cs.ucl.ac.uk, j.petke@ucl.ac.uk

Abstract. Genetic improvement (GI) uses automated search to find improved versions of existing software. While most GI work use genetic programming (GP) as the underlying search process, focus is usually given to the target software only. As a result, specifics of GP algorithms for GI are not well understood and rarely compared to one another. In this work, we propose a robust experimental protocol to compare different GI search processes and investigate several variants of GP- and random-based approaches. Through repeated experiments, we report a comparative analysis of these approaches, using one of the previously used GI scenarios: improvement of runtime of the MiniSAT satisfiability solver. We conclude that the test suites used have the most significant impact on the GI results. Both random and GP-based approaches are able to find improved software, even though the percentage of viable software variants is significantly smaller in the random case (14.5% vs. 80.1%). We also report that GI produces MiniSAT variants up to twice as fast as the original on sets of previously unseen instances from the same application domain.

Keywords: Genetic improvement (GI) · Genetic programming (GP) · Search-based software engineering (SBSE) · Boolean satisfiability (SAT)

1 Introduction

Genetic improvement (GI) [11, 19] uses automated search to find improved versions of existing software. GI literature focuses on both improvement of functional properties, such as automated bug repair or introduction of new functionality, as well as improvement of non-functional properties such as running time, or memory or energy consumption.

Genetic programming (GP) has been used most often so far as the GI search process [11]. Even though previous work use GP as a theoretic common framework, most of it implements or uses very specific variants and parameter values for the GP algorithms that led to evolution of improved software. In order to shift the focus from the target software to the GI process itself, so it can be

better understood, it becomes increasingly necessary to be able to compare and analyse all these proposed search processes.

In this paper, we aim to provide insights on how to compare GI approaches and improve the protocol for applying GI techniques. We consider an existing non-functional improvement GI scenario used in previous work [3, 12, 13], the improvement of the running time of MiniSAT [4] on combinatorial interaction testing instances, and a diverse range of various GP-based and random-based search processes. We consider the following research questions:

RQ1 (Effectiveness): *How often are noticeable improvements found?*
RQ2 (Efficiency): *How significant are the improvements found?*
RQ3 (Robustness): *How critical are the GP parameter values for GI?*
RQ4 (Consistency): *What is the impact of test cases on the results of GI?*

This paper is structured as follows. First, Section 2 provides the necessary GI background. Next, Section 3 presents the GP structure that will be used in the experiments. Section 4 then describes the experimental protocol and which specific GI search processes are compared. Experimental results are presented and discussed in Section 5. Finally, Section 6 concludes this paper.

2 Genetic Improvement (GI)

Genetic improvement (GI) uses automated search to find improved versions of existing software. In this section, we detail on how the software to be improved will be represented, how it will be modified, and how mutant fitness is assessed. In addition to the related work mentioned in this section see [11] for a more comprehensive survey of GI work.

2.1 Software Representations

This work focuses, as a lot of previous work [11], on processing software source code based on its underlying abstract syntax tree (AST). The main advantage of producing mutated source code, in contrast to, for example, producing mutated binary code [15], is that source code mutations and, in particular, patches can be expected to be much more easily understood and thus accepted by software developers [17].

Source Code Representation. Implementation-wise, this paper uses the latest version of the PyGGI[1] framework, and in particular its XML tree representation introduced in [1]. SrcML[2] is used to obtain an XML tree for the AST of the original source code file, which is then stripped down to only consider statements inside functions. No specific instrumentation is performed; the only source code modification applied is the addition of explicit brackets around pseudo

[1] `https://github.com/coinse/pyggi`
[2] `https://www.srcml.org/`

blocks (e.g., *if* statements containing a single line statement with no surrounding bracket) so that modifications of the AST are correctly translated back when generating the modified source code. A successful alternative to AST-based representations is BNF grammar-based representation (e.g., GISMOE [3,6,7,12–14]).

Mutants Representation. In contrast to the earliest GI work [2], in which populations of entire programs were evolved, most GI work nowadays consider intermediary representations for mutants, focusing on the changes that are applied to the original software. In this paper, the possible changes (or *edits*) considered are deletions, replacements, and insertions, relatively to any sub-tree of the original software AST. Mutants are then simply represented as a sequence of edits, only translated into source code and compiled for fitness assessment.

2.2 Fitness Assessment

The non-functional software property to be optimised is computational speed of software, in particular, its average running time. For all purposes, the original software will be considered already functionally correct and the correctness of mutants will be assessed in comparison with the original software execution (e.g., by providing the same output). Non-functional mutant will be discarded immediately. Previous work have shown that considering a multi-objective approach and using degree of functionality as another objective is a viable alternative that can sometimes even lead to a semantic gain [7].

Running time can be an extremely unreliable property to measure precisely, strongly impacted by the environment, with good measurements only achievable after a sufficient number of repetitions. As a proxy measure, previous work used, for example, the number of lines of code executed by the software (e.g., [11]). Major drawbacks include heavy source code instrumentation, same weight given to every statement, omission of impact of standard or external libraries, and an arguably strong impact of the compiler optimisation procedure. Instead, we propose to use the total number of executed CPU instructions as reported through the `perf` UNIX kernel monitoring tool[3]. While the number of CPU instructions still does not provide a deterministic measure [16], preliminary results have shown it to be well correlated with running time and several orders of magnitude more stable even when executing experiments in parallel on a single machine.

3 Genetic Programming (GP)

Since the inception of genetic improvement (GI), many variants of genetic programming (GP) have been successfully applied to the task of improving existing software. In this work, we focus on and extend a GP structure that has been used in recent non-functional GI work [7, 12, 13], and particularly has already been used on the specific software improvement scenario that we will consider.

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

This structure is detailed in Figure 1. It considers a fixed size population of n individuals that it will evolve until the training budget is exhausted. Individuals of the first generation are generated by considering a single random mutation of the original source code. Then, for all subsequent generations, offspring are generated in five successive steps.

Selection. The fittest individuals of the previous population are selected as parents. In this paper, selection simply amounts to discarding invalid mutants and sorting the remaining individuals according to their fitness.

Elitism. The best p_e parents are simply added back untouched to better ensure gene transmission.

Crossover. The best p_c parents are considered successively and crossed with another parent picked uniformly at random to produce a single offspring. GP crossovers in the GI literature include concatenation (e.g., in [7, 12, 13]), 1-point crossover (e.g., in [5, 10, 18]), or uniform crossover (e.g., in [9]).

Mutation. The first p_m best parents are considered successively again and mutated once. The most common mutations include either removing an edit from the edit sequence, selected uniformly at random, or appending a new edit at the end of the edit sequence.

Regrow. Finally, if not enough offspring have been generated (e.g., if the previous generation could not yield enough valid parents) then new individuals are generated at random with a single mutation.

This structure differs from the previous one by two major points: it includes an elitism step and enables explicit parameterisation. In the previous work up to $n/2$ parents were selected, and each of them had two offspring, one through crossover and one through mutation, with the risk of completely discarding the genetic material of a parent when both offspring are unsuccessful. In the worst case, the entire population can be decimated in a single population only to restart evolution from scratch [7]. The elitism step tries to alleviate this issue by providing a way to safely carry the best mutations over to the next generation.

4 Experimental Setup

In this section, we present the GI scenario—i.e., the target software to be improved, MiniSAT [4], together with the application scenario, combinatorial interaction testing instances—, clarify implementation specifics such as how the source code of MiniSAT is represented, how it will be modified, and how performance is assessed, before finally detailing the experimental protocol.

4.1 MiniSAT

This paper targets the automatic improvement of MiniSAT [4], a well-known Boolean satisfiability (SAT) solver. MiniSAT is open-source and can be downloaded online[4]. It has been used several times in previous GI work, such as in [1, 3, 12–14].

[4] `http://minisat.se/MiniSat.html`

```
▷ n: population size
▷ p_e, p_c, p_m: number of parents selected for elitism, crossover, mutation
procedure GP(n, p_e, p_c, p_m)
    ▷ Initial generation, generated at random
    pop ← [ ]
    while |pop| < n do
        mutant ← new mutant
        append mutant to pop
    end while
    ▷ Subsequent generations
    repeat
        offspring ← [ ]
        ▷ (1) Selection (here: filter and sort)
        parents ← selection(pop)
        ▷ (2) Offspring by elitism
        for all parent ∈ parents[0 ... k_e] do
            append parent to offspring
        end for
        ▷ (3) Offspring by crossover
        for all parent1 ∈ parents[0 ... k_c] do
            parent2 ← individual from parents (uniformly at random)
            mutant ← crossover(parent1, parent2) or crossover(parent2, parent1)
            append mutant to offspring
        end for
        ▷ (4) Offspring by mutation
        for all parent ∈ parents[0 ... k_m] do
            mutant ← mutation(parent1)
            append mutant to offspring
        end for
        ▷ (5) If not enough parents: fill with random mutants
        while |offspring| < n do
            mutant ← new mutant
            append mutant to offspring
        end while
        pop ← offspring
    until training budget exhausted
    return overall best mutant ever evaluated
end procedure
```

Fig. 1: Genetic programming search

Two versions of MiniSAT were previously considered, `minisat2-070721` and `minisat-2.2.0`, based on the winning entries of SAT-Race 2006 and 2009, respectively. We used the latest version and focus on a single file, `core/Solver.cc`, which contains the code pertaining to the search process.

As input, we use combinatorial interaction testing (CIT) instances, a GI scenario proposed in [12] and reused in later work. In particular, we use the

130 instances described in [13], split following the 5 "bins" according to their satisfiability (SAT or UNSAT) and the time taken by MiniSAT to solve them[5].

4.2 Experimental Protocol

The purpose of GI is to obtain, from a given software, a *better* software, in terms of either functional or non-functional property. In previous work, especially in the case of automated bug fixing, only one test suite was often used, leading frequently to overfitting, as pointed out in [8]. For that reason, it is necessary to split data into at least two disjoint sets of inputs in order to properly control for generalisation, which has been done in the non-functional work to-date. Additionally, previous GI work [7, 11] has shown that, especially using GP and edit list as representation for the mutated software, an intermediary filtering step was extremely useful to reduce bloat, focus on fewer edits, and improve generalisation. Our experimental protocol thus consists of the following steps:

Pre-processing. The set of inputs is disjointly split between training, validation, and test inputs. When considering multiple bins of input, all of them are split independently and then interwoven so that each of the training, validation, and test input sets also contains the same number of sub-bins, thus ensuring that they all follow the same distribution of bins as the entire set of inputs.

Training. Training is the main, most important, and most computationally expensive part of the experimental protocol. Starting from an initially empty mutant, the search process (e.g., GP) produces incrementally *better* individuals, before returning a single mutated software when the training budget is exhausted. The final training mutant can, for example, be the best individual of the last GP generation, or the best overall mutant for random search. No new software modification is to be investigated beyond this point.

Validation. The final training mutant very often overfits to the training data. The validation steps tries, by considering previously unseen input, to filter out mutations that do not seem to generalise. This step also allows for a simpler and more understandable software to be returned.

Test. Again, new unseen data is used to reassess the performance of the final validated mutant. It is extremely important that the final software is not modified: any decision or analysis process, including singling out individual mutations, should have been performed during the validation process.
Additionally, we also reassess the performance of the final training mutant to control the impact of the filtering performed during the validation step.

Additionally, to select the training, validation, and test sets, we propose to use a repeated procedure based on nested k-fold cross-validation. This procedure, illustrated in Figure 2, ensures a fair usage of every instance across the

[5] Fastest SAT instances, fastest UNSAT instances, SAT instances, UNSAT instances, and slowest (both SAT or UNSAT) instances. Respective bin sizes: 50, 37, 17, 18, 8.

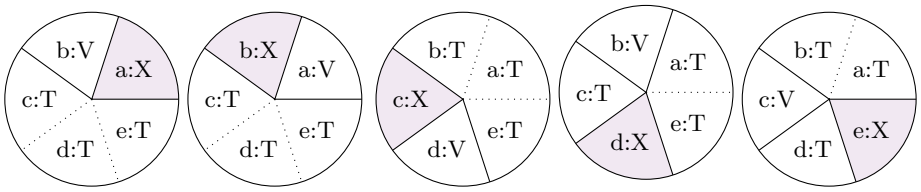


Fig. 2: Example of nested 5-fold cross validation using a single fold for test (X) and $k-1$ folds for both validation (V: single random fold from the remaining $k-1$ folds) and training (T: all remaining $k-2$ folds). Each of the five folds is successively used once for the test step (X).

multiple steps and control the consistency of the GI process. Firstly, inputs (independently for each bin) are shuffled and split into k disjoint subsets (or *folds*). For k successive sets of experiments, each of the k folds is successively used during the test step, leaving $k-1$ folds for both training and validation. Within these $k-1$ folds, one is picked uniformly at random for the validation subset, leaving the last $k-2$ for the training subset. In the experiments, we use $k=5$, thus five sets of experiments will be conducted, each using 60% of each bin of inputs in the training step, 20% in the validation step, and 20% in the test step.

Note that while the validation and test steps will use every single of their respective instances, search processes will not necessarily use all the training instances during the training step. For example, previous GI work using GP advocated sampling a single instance from each bin before every generation [7], while for random search we will simply use a fixed subset of instances. The reason is the computational cost of evaluating software on all instances: impractical and inefficient for each and every mutant generated during training, but necessary to reliably assess performance in the two other much shorter steps. Finally, so that multiple approaches can be fairly compared, it is critical that every search process, disregarding their specific instance usage strategy, is given equal opportunity to all training instances: in no circumstance should the training set be tailored to a specific search process.

4.3 Search Processes

A total of eight GP search processes are compared, together with a baseline constituted of four random searches. The GP search processes follow the structure introduced in Figure 1, with four different population sizes $n \in \{10, 20, 50, 100\}$ for a total budget of 2000 mutant evaluations. Half of them will use the elitism mechanism with $p_e = 0.1 \cdot n$ and $p_c = p_m = 0.45 \cdot n$, carrying the best 10% individuals to the next population[6], while the other half will follow previous work with $p_e = 0$ and $p_c = p_m = 0.5 \cdot n$. These very small populations sizes are justified by the large amount of computational time used for fitness computation; the successful use of a population of $n = 10$ is corroborated in [7] while

[6] After rounding, we use $\{p_e, p_c, p_m\} = \{1, 5, 4\}$, $\{2, 9, 9\}$, $\{5, 23, 22\}$, and $\{10, 45, 45\}$

$n = 100$ is used, for example, in [13]. Furthermore, we deviate from previous work using a similar GP structure with a concatenation crossover in favour of a 1-point crossover, preliminary experiments having shown that the former generates unreasonable and unsustainable amount of bloat, especially with very small populations (thus at equal training budget, more generations).

In addition to these eight GP search processes, four random searches are included as a baseline, in which new individuals are simply generated independently and uniformly at random and the final mutant is the one with overall best fitness. These random searches are parameterised with the maximum number m of edits that are generated for each new mutant. We consider $m = 1$ (i.e., each mutant contains a single random edit), and $m = 2, 5, 10$ to enable generating more complex mutants.

The only difference between the two types of search processes is the number of training instances used and therefore the subsequent training budget. On the one hand, to compute fitness GP will use as in previous work five instances that are resampled at the start of every generation (a single instance from each bin). It arguably implies an initial very unreliable fitness in terms of both functional and non-functional properties, but the evolutionary process ensures that the longer a mutation lives in the population, the more instances it has been trained on and thus increasing reliability. On the other hand, fitness in random search cannot rely on subsequent evaluations so instead more instances are used: twenty in total, four instances from each bin. As a direct consequence, to keep the overall same number of software execution and ensure fair comparison, the training budget is reduced to 500 mutant evaluations, one forth of the GP training budget.

In the experiments the four approaches using GP without elitism will be referred to as $GP(n)$, with $GP_e(n)$ used for the four approaches using GP with elitism and $Rand(m)$ for the four random-based approaches.

4.4 Filtering

Two successive filtering procedures are applied during the validation step. The first filtering is based on the assumption that GP-based search may produce a large amount of bloat. Every edit is successively removed from the edit sequence and discarded if its omission has no impact on the mutated source code. More precisely, this filtering targets patterns in which, for example, a single statement is deleted multiple times, modified, then deleted. It is usually very cheap as it does not require any fitness computation, but has, however, no impact on the mutant performance.

The second filtering (from [13]) aims to improve generalisation by discarding edits that fail to generalise on previously unseen instances. The fitness of every edit is first computed independently, then edits are sorted by fitness, and the final mutant is constructed by adding edits one at a time if their addition has a positive impact. This process consumes at most twice as many fitness evaluations as the size of the edit sequence. This specific filtering works best when edits are independent.

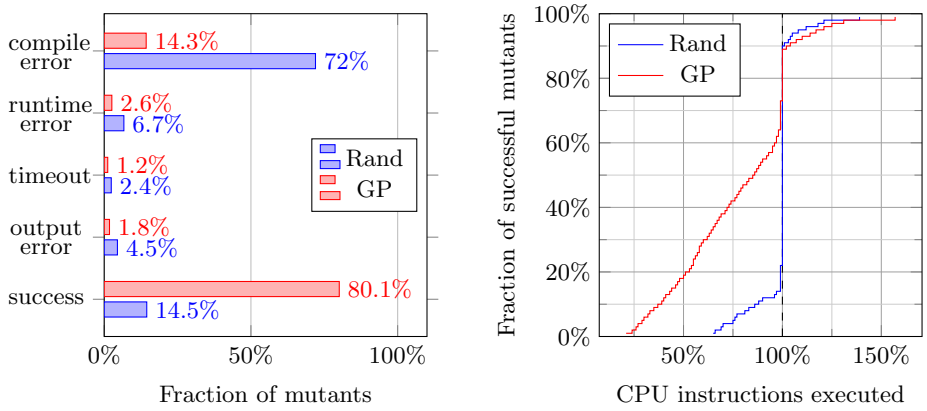


Fig. 3: Distribution of evaluation outcomes for random- and GP-based approaches

5 Results and Discussion

Training and validation step 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. The testing step was conducted sequentially on a single core.

5.1 Overall Training Results

The 40 GP training runs required between 7 to 13 hours to complete, with an average of 10 hours. The four random-based approaches required between 30 minutes to 5 hours, with an average of about 2 hours. Variance in training time can be explained by instances with different processing time being sampled.

Figure 3 shows the distribution of outcome of every mutant during the training step, separated between random search and GP, together with the empirical cumulative distribution function (ECDF), i.e., the fraction of the successful mutants better than a given fitness. Because fitness values are computed using different instances they are normalised and indicated as a ratio with the fitness of the original software on the same instances. Random-based approaches generated 10000 mutants, with only 14.5% of them viable and very few with a noticeable impact. 72% failed to compile, and the remaining 13.5% either crashed, stalled, or produced an incorrect satisfiability output. GP-based approaches generated 80000 mutants, within which only 14.3% failed to compile while 80.1% were successful, with a very large fraction of them reporting large improvements over the original software. This indicates that a very high efficiency for the 1-point crossover for combining existing mutations and generating valuable mutants.

Table 1: Experimental results for all variants (first split).

Training			Validation				Test			
Search	Size	CPU	Size'	CPU	Size*	CPU*	CPU	Time	CPU*	Time*
$GP(10)$	16	99.9%	11	99.9%	7	99.9%	99.9%	99.2%	99.9%	100.4%
$GP(20)$	32	92.7%	12	123.4%	5	93.5%	40.5%	52.7%	67.4%	76.1%
$GP(50)$	23	69.6%	11	102.6%	3	99.4%	77.7%	91.3%	99.6%	98.8%
$GP(100)$	16	63.8%	13	111.3%	4	99.9%	87.7%	100.3%	99.9%	100.6%
$GP_e(10)$	1304	33.5%	26	114.4%	13	90.8%	44.1%	50.5%	62.8%	70.6%
$GP_e(20)$	268	57.7%	21	105.5%	4	91.0%	43.7%	57.1%	63.0%	71.1%
$GP_e(50)$	15	78.2%	7	123.6%	5	96.7%	80.0%	87.9%	98.5%	99.6%
$GP_e(100)$	6	64.8%	6	107.1%	2	100.0%	36.2%	45.7%	100.0%	99.3%
$Rand(1)$	1	66.5%	1	114.0%	0	–	89.2%	101.4%	–	–
$Rand(2)$	2	67.0%	2	114.5%	0	–	89.7%	102.5%	–	–
$Rand(5)$	1	75.0%	1	109.0%	0	–	60.5%	66.9%	–	–
$Rand(10)$	2	74.9%	2	107.2%	1	100.0%	63.3%	66.3%	100.0%	99.3%

Size, Size', Size*: patch size (number of edits) of the final training mutant, of the cleaned-up mutant, and of the final validation mutant.
CPU, CPU* (Time, Time*): percentage of CPU instructions (running time) of the final training and validation mutant, compared to the unmodified software.

5.2 Comparison of Approaches

Table 1 to Table 5 respectively report on the performance of the twelve approaches over the five repetitions and splits of instances. For each approach we first report, for the training step, the size and fitness estimate of the final training mutant. Then, for the validation step we report the cleaned-up size of the final training mutant and its fitness, and the size and fitness after filtering. Finally, for the test step, we report, for both the final training mutant and the final validation mutant, the fitness in terms of both the number of CPU instructions and the actual running time. Again, fitness values are indicated as a ratio of the fitness with the original software on the same instances. An illustration of the relationships between these results is presented in Figure 4.

Firstly, while the results are mostly consistent within a single instance split, they greatly differ from one split to another. Furthermore, the instance sets of the validation and the test steps induce extremely different results, albeit being from identical size and sampled from the same distribution. This difference in results points towards a high heterogeneity in the data set, easily explained by the small number of SAT instances used (only 130 CIT instances).

The results do not show any noticeable impact on neither the size of the population nor the use of elitism on the performance of the GI process. While a statistical analysis using a considerably larger amount of GI runs for each approach might yield better insight on the impact of GP parameters, significant difference in performance within the selected parameter values is unlikely. The four random-based approaches also show no significant difference in performance.

Table 2: Experimental results for all variants (second split).

Training			Validation				Test			
Search	Size	CPU	Size	CPU	Size⋆	CPU⋆	CPU	Time	CPU⋆	Time⋆
$GP(10)$	71	38.6%	27	155.3%	3	93.6%	127.1%	135.8%	88.0%	88.2%
$GP(20)$	40	20.0%	13	117.7%	6	99.7%	98.5%	113.4%	99.6%	99.7%
$GP(50)$	16	54.9%	13	165.7%	5	99.9%	151.9%	205.6%	100.0%	100.1%
$GP(100)$	5	55.6%	5	136.6%	1	100.0%	126.4%	154.0%	100.0%	99.8%
$GP_e(10)$	45	38.6%	29	100.7%	2	100.0%	90.9%	91.5%	100.0%	99.8%
$GP_e(20)$	62	15.7%	19	117.7%	7	100.0%	102.9%	119.8%	100.0%	99.7%
$GP_e(50)$	31	44.2%	12	118.8%	5	99.7%	97.8%	112.7%	99.7%	99.6%
$GP_e(100)$	19	49.9%	12	123.8%	3	93.4%	90.7%	105.5%	81.2%	84.6%
$Rand(1)$	1	48.3%	1	151.1%	0	–	114.9%	123.0%	–	–
$Rand(2)$	2	44.4%	2	107.9%	1	100.0%	105.1%	106.8%	100.0%	99.7%
$Rand(5)$	2	70.2%	2	108.0%	1	100.0%	89.9%	86.8%	100.0%	100.5%
$Rand(10)$	1	51.8%	1	107.4%	0	–	91.9%	96.1%	–	–

Table 3: Experimental results for all variants (third split).

Training			Validation				Test			
Search	Size	CPU	Size	CPU	Size⋆	CPU⋆	CPU	Time	CPU⋆	Time⋆
$GP(10)$	28	69.4%	9	102.5%	6	99.8%	76.5%	76.0%	99.7%	100.8%
$GP(20)$	63	86.7%	10	107.7%	6	100.0%	130.2%	138.0%	100.0%	99.8%
$GP(50)$	7	26.2%	7	151.1%	1	100.0%	74.9%	71.6%	100.0%	100.3%
$GP(100)$	8	60.8%	6	109.3%	1	100.0%	62.9%	64.1%	100.0%	98.4%
$GP_e(10)$	19	69.3%	5	93.2%	3	100.0%	98.5%	98.2%	100.0%	99.5%
$GP_e(20)$	1	100.0%	1	100.0%	1	100.0%	100.0%	100.4%	100.0%	99.4%
$GP_e(50)$	21	25.7%	9	111.2%	3	100.0%	102.2%	113.2%	100.0%	100.8%
$GP_e(100)$	5	48.2%	5	109.2%	2	93.6%	76.3%	74.6%	76.3%	74.7%
$Rand(1)$	1	64.9%	1	107.7%	0	–	50.1%	49.0%	–	–
$Rand(2)$	1	65.7%	1	118.1%	0	–	100.3%	108.4%	–	–
$Rand(5)$	2	52.2%	2	107.9%	1	100.0%	86.9%	89.0%	100.0%	100.3%
$Rand(10)$	3	69.0%	3	98.8%	2	98.8%	104.3%	105.3%	104.3%	106.9%

Focusing on the very first set of experiments (Table 1), most of GP-based and random-based approaches report final mutants using around 70% of the number of CPU instructions executed compared to the original software, with the best mutant reporting using only as much as 33.5%. Mutants generated by GP approaches used between 6 and 26 edits, with the best mutants of random-based approaches containing only a single or two edits. The training performance of all mutants failed to generalise to the set of validation instances, with only three mutants surviving the filtering process with more that 5% fitness improvement. All three mutants did then further generalise on the final test set of instances,

Table 4: Experimental results for all variants (fourth split).

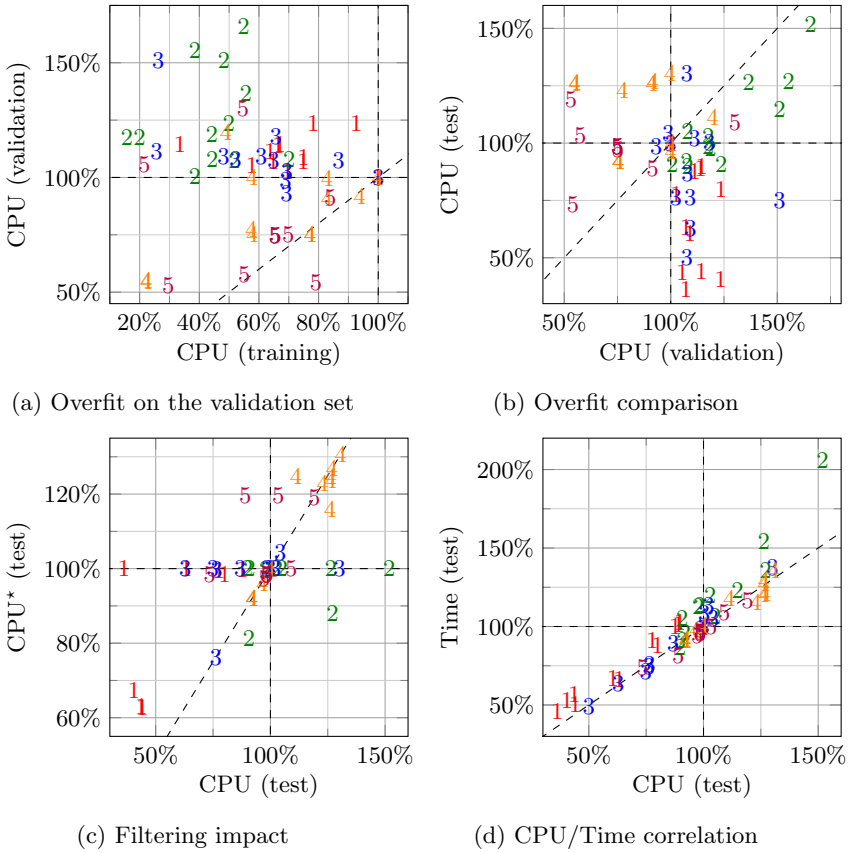
Training			Validation				Test			
Search	Size	CPU	Size	CPU	Size⋆	CPU⋆	CPU	Time	CPU⋆	Time⋆
$GP(10)$	26	93.8%	9	91.6%	6	91.6%	126.9%	121.4%	126.9%	121.1%
$GP(20)$	54	22.2%	13	55.0%	6	50.2%	126.3%	129.7%	124.7%	128.2%
$GP(50)$	9	82.8%	7	91.0%	6	54.0%	126.0%	120.7%	115.8%	111.8%
$GP(100)$	7	57.8%	5	75.4%	3	75.4%	92.0%	91.8%	92.0%	91.7%
$GP_e(10)$	2	99.8%	2	99.9%	2	99.9%	99.8%	101.0%	99.8%	101.1%
$GP_e(20)$	49	22.2%	9	54.9%	8	49.8%	126.2%	127.7%	123.8%	124.9%
$GP_e(50)$	6	82.8%	6	99.7%	4	99.7%	130.6%	135.6%	130.6%	134.6%
$GP_e(100)$	10	48.9%	9	119.6%	5	50.1%	111.2%	118.1%	124.7%	127.8%
$Rand(1)$	1	57.4%	1	77.2%	1	77.2%	122.8%	115.6%	122.8%	114.8%
$Rand(2)$	1	77.1%	1	75.4%	1	75.4%	92.0%	90.9%	92.0%	92.1%
$Rand(5)$	3	57.7%	3	99.9%	1	99.8%	96.4%	98.3%	96.1%	99.2%
$Rand(10)$	1	77.1%	1	75.4%	1	75.4%	92.0%	91.2%	92.0%	91.1%

Table 5: Experimental results for all variants (fifth split).

Training			Validation				Test			
Search	Size	CPU	Size	CPU	Size⋆	CPU⋆	CPU	Time	CPU⋆	Time⋆
$GP(10)$	0	100.0%	–	–	–	–	–	–	–	–
$GP(20)$	36	21.6%	16	105.6%	4	75.1%	timeout	timeout	97.1%	96.9%
$GP(50)$	6	83.9%	5	91.5%	3	53.5%	89.0%	81.7%	119.8%	115.0%
$GP(100)$	4	54.7%	4	130.1%	1	100.0%	109.1%	108.8%	100.0%	99.4%
$GP_e(10)$	0	100.0%	–	–	–	–	–	–	–	–
$GP_e(20)$	88	29.6%	15	53.2%	11	53.2%	119.2%	116.6%	119.2%	116.5%
$GP_e(50)$	14	79.1%	9	54.1%	4	49.2%	73.5%	74.3%	98.6%	103.3%
$GP_e(100)$	20	55.1%	12	57.6%	5	53.4%	103.4%	99.9%	119.8%	116.8%
$Rand(1)$	1	65.4%	1	74.9%	1	74.9%	98.4%	96.4%	98.4%	98.9%
$Rand(2)$	1	65.4%	1	74.9%	1	74.9%	98.4%	96.7%	98.4%	96.8%
$Rand(5)$	1	65.6%	1	75.0%	1	75.0%	98.6%	95.7%	98.6%	95.5%
$Rand(10)$	1	69.8%	1	75.3%	1	75.3%	97.3%	94.6%	97.3%	95.6%

with a final improvement of 25% to 30% in running time. However, many unfiltered mutants, albeit slower on the validation set, showed on the test set much larger improvements in both number of CPU instructions and running time.

For this first split of instances, it appears that mutants generalised to the test instances but not the validation instances. The same situation occurs in Table 3, while Table 4 and Table 5 show the inverse to be true. Finally, on the three cases (first three folds) in which the validation step is able to catch overfitting on the training step, it efficiently is able to fix it on the test step.



(a) Overfit on the validation set

(b) Overfit comparison

(c) Filtering impact

(d) CPU/Time correlation

Fig. 4: Results correlations after the training, validation, and test steps

5.3 Comparative Analysis

Figure 4 illustrates various relationships between mutant performance at different points of the experiments. Dashed lines highlight 100% thresholds and the identity function ($x = y$). Data points are simply denoted by the index of the fold they used in the test step.

Figure 4 (a) shows the overfitting of the final training mutants on the validation set of instances. As expected, almost every single mutant overfits on the training set, with roughly half of the mutants using more CPU instructions than the original software on the previously unseen validation instances.

Figure 4 (b) should ideally show very correlated results, as performance should be similar on both validation and test sets of instances, both unseen and following the same distribution. Instead, it highlights a previous conclusion: the set of CIT instances is too small to be randomly divided in five fair subsets.

Figure 4 (c) shows the impact of the filtering step, comparing the performance on the test step of the final training mutant and the filtered one. Unfortunately,

filtering had in most cases either no impact or a negative impact. This is due to validation and test sets of instances being inconsistent.

Figure 4 (d), finally, shows the clear correlation between performance in terms of the number of CPU instructions executed and the running time. This, together with the very high stability of CPU instructions readings even in parallel contexts, confirms it as an excellent measure of computational speed.

5.4 Research Questions

RQ1 (Effectiveness): *How often are noticeable improvements found?*
In all but five GI runs improvements from 5% to 79% in the number of CPU instructions were found on training instances. In slightly more than half of the runs some of the mutations had a noticeable ($> 5\%$) impact during either the validation or the test step. However, considering only the performance on the filtered mutants, only nine GI runs had a noticeable positive impact during the test step. Seven of these GI runs used a GP search process, while the two other used random search.

RQ2 (Efficiency): *How significant are the improvements found?*
The improvements of the nine most successful GI runs vary between 8% and 37% in terms of the number of CPU instructions, and between 8% and 30% in terms of running time. Furthermore, among the many results with significant ($> 5\%$) improvements on unseen instances (validation and test) about two-thirds show improvements of at least 25%.

RQ3 (Robustness): *How critical are the GP parameter values?*
No particular impact of parameter values is noticed for neither GP-based approaches nor random-based approaches. While more GI runs based on GP ultimately produced significant improvements, performance of GP-based approaches was similar to the performance of random-based approaches. This could be partly attributed to the data set heterogeneity.

RQ4 (Consistency): *What is the impact of test cases on the results of GI?*
As clearly demonstrated in the experiments, results are strongly impacted by the way instances are split. The same final mutant trained on 60% of the instances can be reported as 50% faster on 20% of previously unseen instances while being at the same time 25% slower of the remaining 20% of equally unseen instances. As a consequence, it is highly recommended for future GI work to report repeated performance using multiple data splits, following, for example, the experimental protocol described in this paper. Failure to do so might result in overlooking major weaknesses in the dataset and highly overestimated final software performance.

6 Conclusions

This paper presented and compared several GP approaches for a GI scenario in which a Boolean satisfiability solver, MiniSAT, was evolved to optimise running time on combinatorial interaction testing instances. Number of CPU instructions

was proposed as an alternative to source code instrumentation and was shown to be a reliable indicator of computational speed. Following a protocol based on repeated experiments, it showed that performance of GI processes was highly impacted by heterogeneity in the data set. While the training steps resulted in a very high number of mutants with excellent performance on *either* of the validation or test steps, very few of them had a significant impact after the complete GI process. Overall, GP approaches are mostly indistinguishable from one another and yet more efficient and effective than random search, suggesting that more consistent and reliable approaches are yet to be proposed.

The proposed protocol, with repeated experiments and disjoint validation and test sets, shows the potential for obtaining even better results than in previous work. Moreover, it shows that the largest impact on the performance lies in the set of test suites used, which requires further investigation in future work. Regardless, even in the simplest random search case, improvements can be found. However, the question of which is the most efficient and effective search process in GI remains open with this work being the first step towards answering that question.

Acknowledgement

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

References

1. An, G., Blot, A., Petke, J., Yoo, S.: PyGGI 2.0: Language independent genetic improvement framework. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019). pp. 1100–1104. ACM (2019). https://doi.org/10.1145/3338906.3341184
2. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: Proceedings of the Congress on Evolutionary Computation (CEC 2008). pp. 162–168 (2008). https://doi.org/10.1109/CEC.2008.4630793
3. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015). pp. 1327–1334. ACM (2015). https://doi.org/10.1145/2739480.2754752
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). LNCS, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
5. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 4th Genetic and Evolutionary Computation Conference (GECCO 2009). pp. 947–954. ACM (2009). https://doi.org/10.1145/1569901.1570031
6. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nvidia template. In: Proceedings of the Congress on Evolutionary Computation (CEC 2010). pp. 1–8 (2010). https://doi.org/10.1109/CEC.2010.5585922

7. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1), 118–135 (2015). https://doi.org/10.1109/TEVC.2013.2281544
8. Le, X.D., Chu, D., Lo, D., Le Goues, C., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2017). pp. 593–604. ACM (2017). https://doi.org/10.1145/3106237.3106309
9. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: 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 2012). pp. 3–13. IEEE Computer Society (2012). https://doi.org/10.1109/ICSE.2012.6227211
10. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012). https://doi.org/10.1109/TSE.2011.104
11. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: A comprehensive survey. IEEE Transactions on Evolutionary Computation **22**(3), 415–432 (2018). https://doi.org/10.1109/TEVC.2017.2693219
12. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Proceedings of the 17th European Conference on Genetic Programming, Revised Selected Papers (EuroGP 2014). LNCS, vol. 8599, pp. 137–149. Springer (2014). https://doi.org/10.1007/978-3-662-44303-3
13. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Transactions on Software Engineering **44**(6), 574–594 (2018). https://doi.org/10.1109/TSE.2017.2702606
14. Petke, J., Langdon, W.B., Harman, M.: Applying genetic improvement to MiniSAT. In: Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE 2013). LNCS, vol. 8084, pp. 257–262. Springer (2013). https://doi.org/10.1007/978-3-642-39742-4_21
15. Schulte, E.M., DiLorenzo, J., Weimer, W., Forrest, S.: Automated repair of binary and assembly programs for cooperating embedded devices. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013). pp. 317–328 (2013). https://doi.org/10.1145/2451116.2451151
16. Weaver, V.M., Terpstra, D., Moore, S.: Non-determinism and overcount on modern hardware performance counter implementations. In: Proceedings of the 2013 International Symposium on Performance Analysis of Systems & Software (ISSTA 2013). pp. 215–224. IEEE (2013). https://doi.org/10.1109/ISPASS.2013.6557172
17. Weimer, W.: Patches as better bug reports. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006). pp. 181–190 (2006). https://doi.org/10.1145/1173706.1173734
18. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). pp. 364–374. IEEE Computer Society (2009). https://doi.org/10.1109/ICSE.2009.5070536
19. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation **15**(4), 515–538 (2011). https://doi.org/10.1109/TEVC.2010.2083669