

Empirical Comparison of Runtime Improvement Approaches: Genetic Improvement, Parameter Tuning, and Their Combination

Thanatad Songpetchmongkol

Department of Computer Science

University College London

London, United Kingdom

thanatad.songpetchmongkol.22@ucl.ac.uk

Aymeric Blot

IRISA

University of Rennes

Rennes, France

aymeric.blot@univ-rennes.fr

Justyna Petke

Department of Computer Science

University College London

London, United Kingdom

j.petke@ucl.ac.uk

Abstract—Software can be optimised in various ways, for instance, by modifying its source code or adjusting its compiler and runtime parameters. To automate these tasks, algorithm configuration and genetic improvement have been proposed – the former modifies parameters and the latter source code. Many tools have been introduced to automate such changes. However, these tools typically only work at a single code level, optimising either parameter values or source code, but not both. In 2022, Blot and Petke [1] introduced MAGPIE, a framework capable of simultaneously searching for improvements at multiple granularity levels. Our literature review revealed that the best search strategies in genetic improvement and algorithm configuration that may generalise to both domains are based on local search and genetic algorithms, respectively. We compared these two approaches for improving the execution time of the MiniSAT solver, and also explored their performance on the joint search space of parameter and source code edits. Our results show that genetic improvement with first improvement local search led to the best results, improving MiniSAT’s execution time by 18.05%.

Index Terms—genetic improvement, algorithm configuration, parameter tuning, hyperparameter optimisation

I. INTRODUCTION

Software is everywhere and nowadays plays a huge role in our daily lives. Everyone relies on software to complete tasks such as replying to emails, finding the shortest route to a restaurant, etc. Consequently, software is consistently updated and improved to meet users’ needs.

There are diverse methods by which engineers can optimise software. Selecting the right programming language for a task is the first step to software optimisation. If software requires high performance and precise control over data structures, then the engineer should use a low-level language such as C, C++ or Assembly. Conversely, programming languages such as Go or Python would for example be more appropriate choices for higher-level tasks such as web development.

Various methods can be used to optimise existing software. On the one hand, much of the existing software has a set of parameters that one could configure. Similarly, a compiler also has many configuration options that can affect the generated binary files. Algorithm configuration facilitates the automated

setting of such parameters. On the other hand, software optimisation by directly manipulating source code can be achieved via genetic improvement [2], [3].

In 2022, Blot and Petke [1] introduced MAGPIE (*Machine Automated General Performance Improvement through Evolution of software*), a framework for improving functional and non-functional properties of software. In MAGPIE’s original paper, the tool was shown to be capable of significantly improving the execution time of a diverse set of programs. Moreover, the MAGPIE’s authors claimed to be the first to combine three optimisation approaches into a single tool: compiler, parameter tuning, and genetic improvement. They used a first improvement local search as the search algorithm, which was applied to all three optimisation approaches. However, such a search algorithm might not be the best search strategy for all optimisation approaches.

Here, we aim to improve MAGPIE by implementing search algorithms that have been empirically shown to perform well for each optimisation strategy. We investigate how effective the search is when exploring the joint search space of parameter and source code changes. In our experiments, we used algorithms considered as the state-of-the-art search strategies in each domain: algorithm configuration¹ (AC) and genetic improvement (GI).

Currently, there are two main search strategies in the genetic improvement domain: local search (LS) and genetic programming (GP). Blot and Petke [4] showed that the first improvement local search performs best². As for algorithm configuration, there exists a plethora of search strategies. Unfortunately, as far as we know, no study empirically evaluates all search strategies in the domain. Most publications usually compare between three to four search strategies, but many more exist. Nonetheless, we found a paper from the hyper-algorithm configuration domain by Yang and Shami [5], who empirically evaluated the largest number of search strategies, including those considered state-of-the-art in the AC domain.

¹We note that compiler flag and hyper-parameter optimisation can be regarded as another form of algorithm configuration.

²Best out of overall 16 GP and LS strategies investigated.

Thus, we decided to use Yang and Shami [5]’s empirical results as one of the criteria for selecting a search strategy for algorithm configuration, choosing a variant of genetic algorithm used by the authors.

Next, we extended MAGPIE’s implementation to include state-of-the-art search strategies from the AC and GI domains. For both search strategies, we first focused separately on parameter tuning (AC) and source code modification (GI), before exploring the joint search space of parameter and code edits, for a total of six experiments. As target of our case study, we selected the MiniSAT solver improvement of its execution time, following previous work in both AC and GI.

Our results show that genetic improvement with first improvement local search (18.05% speed-up) performs slightly better than algorithm configuration with genetic algorithm (14.32% speed-up). For the joint search space, first improvement local search found a 9.88% speed-up, while genetic algorithm did not find any improving variant. From the result, we can see that there is a potential for exploring the joint search space of edits, yet further modifications to the search strategy procedure are required in order to obtain better results. We observed that the tournament selection mechanism in the genetic algorithm is capable of selecting the best individuals from previous to proceeding generation. Yet, there is no guarantee that those genes will be retained as they might undergo mutation. In such a case, elitism would be an appropriate choice for preserving the best genes from the preceding generations in future work.

Overall, our results on MiniSAT show that on this example, the best search strategy in MAGPIE is genetic improvement with first improvement local search (18.05% speed-up) followed by algorithm configuration with first improvement local search (17.75% speed-up) and genetic improvement with genetic algorithm (16.12% speed-up). More experiments are needed to generalise our findings.

II. RESEARCH QUESTIONS

Our main goal is to explore how well each state-of-the-art search technique performs in AC and GI domain. We also want to measure how much can be gained from combining the best of the two domains. Given our goals, we present the three following research questions.

RQ1: How effective are AC and GI at software performance improvement?

We want to see how much runtime reduction can be obtained from tuning parameters using algorithm configuration compared to modifying software using genetic improvement. This is achieved by running GI and AC with the state-of-the-art search strategies identified in each domain.

RQ2: How effective is the simultaneous exploration of the joint search space of parameter values and software edits for runtime improvement?

Given that we established how well each technique performed in RQ1, we want to see how effective it is to explore both domains concurrently. This is possible thanks to MAGPIE, which provides only one software representation shared

across all domains. To answer this, we run AC + GI with the state-of-the-art search strategies from each domain.

RQ3: Which search strategy is best for improving software performance?

To answer this we compare results from all experiments. We also need to perform an analysis of validity of statement edits, to ensure they do not break functionality which is only verified via testing in MAGPIE.

III. SEARCH STRATEGIES SELECTION

We conducted a literature review in order to establish the state-of-the-art search strategies in AC and GI. We found that although Blot and Petke [4] provide a comprehensive review for search strategies in GI, no paper empirically compares all search strategies in the AC domains. Nonetheless, we came across a paper from Schede et al. [6] that performs a literature review on AC techniques that go back as far as 2002. Yet, the authors did not provide any empirical comparison. Yang and Shami [5] empirically evaluate various search strategies for hyper-parameter optimisation which also includes state-of-the-art techniques from the AC domain. Hence, we used these two papers to guide our selection process. Additionally, we have the following criteria for search selection:

- SC1: The optimisation technique is generic and generalisable for both AC and GI.
- SC2: The optimisation technique has been shown to perform well on various benchmarks.
- SC3: The experimental time is realistically feasible.

Genetic Improvement. Both LS and GP are well-known in the GI domain and have been proven effective in many works. First, they satisfy the SC1 as MAGPIE had proved it in the original publication. Second, there is only one paper that empirically compares these search heuristics within the GI context. Blot and Petke [4] showed that an LS variant outperformed the GP approach. In particular, the first improvement local search ranked highest. Hence, LS and GP satisfy all selection criteria, and we chose first improvement local search for our study with GI. For brevity, we used the term LS to refer to this LS variant for the remainder of the paper.

Algorithm Configuration. AC field is more mature than GI and many search strategies have been proposed. Unfortunately, we have not found any recent comprehensive survey that evaluates all search strategies. Hence, we considered each in turn. After considering multiple approaches (e.g., Grid/Random Search [7], Particle Swarm Optimisation [8], Golden Parameter Search [9], and others), we chose the Genetic Algorithm (GA) presented by Yang and Shami [5], as it was shown to be competitive with other approaches and can easily generalise to the GI domain, fulfilling all our criteria.

In this section, we present the pseudo-code and explain our GA algorithm, as well as the modifications we added to extend it to the GI domain.

The main components of a genetic algorithm are a population, chromosomes, and genes [10, Chapter 1, Chapter 8]. A population is a group of solutions, which are also known as chromosomes. Each solution has zero or more edits. These

TABLE I
THE DEFAULT PARAMETERS' VALUE FOR OUR GA IMPLEMENTATION

Parameter	Default Value
Population size	10
Default size	2
Tournament selection size	3
Chromosome crossover probability	0.5
Gene crossover probability	0.5
Chromosome mutation probability	0.2
Gene mutation probability	0.1
Statement insertion probability	0.5

edits are known as genes. For AC, a gene represents a single parameter. Thus, there are as many genes as the number of parameters that need to be configured. Conversely, in the case of GI, a gene represents a statement mutation.

See Table I for GA's default values.

We implemented two modifications to the original algorithm (for the initial population generation and mutation, as specified below), to adapt the algorithm to the GI and AC domains. Our GA implementation works as follows:

- 1) Initialise the population with a default value.
 - AC = default parameter value
 - GI = empty patch
- 2) Mutate N individual from the population with a probability of 0.1, where N is *population_size* – *default_size*.
- 3) Evaluate the population.
- 4) Select offspring by tournament selection from the entire population.
- 5) Perform crossover on each selected pair of offspring.
- 6) Perform mutation on each individual in the population.
- 7) Evaluate the offspring.
- 8) Assign offspring to the population.
- 9) Steps 4–8 are repeated until the budget is exhausted.

The top-level pseudocode for our GA implementation is provided in our artefact [11].

Population Initialisation. The standard procedure in a GA is to generate an initial population by randomising all parameters' values [10, Chapter 8]. Doing so ensures that the population uniformly represents the entire search space. Yet, we found that such a practice does not work well when there is a large number of parameters. We note that Yang and Shami [5] used programs with up to seven parameters only. From the preliminary result, we found that, by randomising all 25 parameter values, GA took at least 30 generations to find an individual that successfully compiled and ran without facing any run timeout or code error. Thus, our initialisation process works by injecting a default population of *default_size* and mutating the rest of the population using the mutation function. Given the low mutation rate, the algorithm can mimic gradual adjustment of parameter values.

Selection. Selection is the process of selecting the fittest individual from the population and ensuring that these individuals survive to the next generation [10, Chapter 8].

Selection is only performed once in our GA implementation, during the beginning of every generation. This is because the population size remains fixed throughout the whole process. We used 'tournament selection' with a tournament size of three. Tournament selection works by selecting a group of N individuals at random from the whole population. Then, a single individual with the best fitness value from this group is selected.

Crossover. Crossover is a method of exchanging genes between two parents. There are two layers of crossover probability: chromosome and gene. GA first checks whether to perform crossover between a pair of parents with a probability of 0.5. If it passes the first check, the uniform crossover is performed between each gene of the two parents with a probability of 0.5. The crossover is achieved by either swapping the value for categorical parameters or randomising two new values between the existing values for numerical parameters. In the case of statement edit, we simply swap the edit between parents in a similar manner to a categorical parameter. Otherwise, both parents get to keep the same gene, and no swapping is performed. Pseudo-code for crossover is provided in our artefact [11].

Mutation. Mutation is a method of introducing diversity to the current population. It is used with crossover to ensure that each gene can access a full range of possible values [10, Chapter 9]. The existing mutation procedure [5] does not support edits at code level. Thus, we adopt the approach from genetic programming (GP).

Chromosomes in GA have a 0.2 probability to undergo mutation. In that case, every gene in the mutated individual have a 0.1 probability to be mutated, and if the GI domain is considered, a new statement edit (e.g., Statement Insertion, Deletion, or Replacement) is generated with probability 0.5. As for individual gene mutations, a new parameter value is randomly generated for edits related to categorical and numerical parameters, GA randomly selects a new value, and the edit is simply deleted in the case of statement edits. Pseudo-code for mutation is provided in our artefact [11].

Difference between GA and GP. There are multiple key differences between our GA and the standard GP approach used in GI (later, just "GP"). First, GP individuals only encode the differences to the original software. Conversely, in the AC domain individuals in our GA are initialised with a full set of genes corresponding to the parameters' default values. Then, selection in GP is random within valid software variants, whilst our GA uses tournament selection. Further, due to the sparse nature of GP patch representation, the crossover used is usually a simple concatenation. Our GA uses more traditional uniform and average crossovers. Finally, the mutation in the GP approach used in GI consists in either adding a new edit or removing an existing one. In contrast, in our GA every individual gene is simultaneously considered for mutation. For brevity, we used the term GA to refer to our genetic algorithm variant explained here for the remainder of the paper.

IV. EMPIRICAL STUDY.

In this section, we present the methodology used to answer our research questions.

Experimental Protocol. We follow the experimental protocol from the original MAGPIE paper [1], modified to use a time budget rather than a step budget. This is because GI usually generates fewer valid software variants since source code modification can result in compilation errors and, in some cases, runtime errors. However, this is not an issue for AC as it targets the exposed parameters rather than the source code. Hence, with the same number of maximum steps, AC will evaluate more valid variants than GI. We thus propose to use timeout as a more realistic measure. Each approach will be given the same budget to find improvements. To set the maximum time budget, we followed Blot and Petke [4] as calculated from Eq. 1:

$$time = 100 \cdot T / K \quad (1)$$

where T is the time (in sec) required for the original software to run every training instance, and K is the number of folds in the procedure’s cross-validation.

Given the adoption of the new protocol, we conducted a preliminary experiment to confirm that such a time limit was enough for AC and GI on all k-fold. With 247 training instances from MiniSAT, it took around 20 minutes to run. Based on Equation 1, the final time budget is roughly three hours. The results confirm that every k-fold for both AC and GI managed to find an improvement within three hours. See Figure 5 in the artefact [11] for k-7’s preliminary result. Although not all folds managed to find significant improvements, the results still confirm that (1) Equation 1 provides a time that is sufficient for all search strategies and (2) this protocol can be used as a common ground for performance comparison between the AC and GI domains.

The experiment is divided into three phases: training, validating, and testing. We used k-fold cross-validation in the training and validation phase. Initially, we divide available instances into two groups: the training set and the test set. We use the training set for the training and validation phase, while the test set is for the testing phase. The training set is further divided into K groups where K is the number of folds. The training phase is repeated K times for each benchmark. Next, in the validation phase, we use other $K - 1$ subsets to validate the final software variants. Finally, in the testing phase, we use the test set to ensure that there is no overfitting from the previous phases. For instance, if there are 5-folds and we use the first fold for training, we then would use the other four folds for validation.

In this study, we ran experiments with two search strategies: first improvement local search (LS) and the genetic algorithm (GA), both outlined in the previous section. We ran three experiments for each search strategy, which included exploring the GI and AC search spaces (i.e., searching over source code and parameter edits, respectively) and simultaneously exploring the joint search space (i.e., AC + GI). In total, we ran six experiments.

MAGPIE Optimisation Framework. From the literature review, we found that only MAGPIE allows for simultaneous search across parameter settings and code edits. Given our goals, we decided to use MAGPIE in this study.

Ever since the publication of MAGPIE, in 2022, various modifications have occurred with the framework. Various algorithms have been included, such as genetic programming. We treat the MAGPIE snapshot 5b85d5³ as a baseline for our implementation. We only added our GA, as outlined in the previous section, to the framework along with various helper functions scattered throughout the code base. However, we did not modify any existing underlying mechanism of the framework.

Software Benchmark. In this research, we used the same software benchmark from the original MAGPIE paper. Among the four benchmarks, we selected the well-known SAT solver MiniSAT_HACK_999ED_CSSC⁴, which has 25 exposed parameters as provided in ACLib [12]. We chose MiniSAT because it has been widely used as a benchmark in optimisation-related publications and has undergone iterative improvements by human experts. We evolved the core/Solver.cc file and used data instances from CircuitFuzz⁵, with 247 training instances and 277 test instances, as in previous work.

To set up MiniSAT for MAGPIE, we generated the XML file for the core/Solver.cc file and added script files for compilation and execution of the binary.

Experimental Environment. All experiments were performed on a machine with CentOS 7 and kernel version 3.10.0, Intel(R) Xeon(R) CPU E5520 @2.27GHz. We used Python 3.10.1 and GCC 9.3.1 with the `-O3` flag. Following the original MAGPIE paper, we measured the fitness value with the Linux `perf` command, focusing on the CPU instruction count. We note here that, throughout our experiment phase, we were the only ones using the machine. Our experiments took 258.55 CPU hours to complete.

V. RESULTS AND DISCUSSION

In this section, we present our empirical results and analysis.

A. RQ1: Algorithm Configuration and Genetic Improvement

Figure 1 summarises the results from the test phase in terms of CPU instruction counts for each k-fold. The results between AC with GA and GI with LS are similar, with only a 4% difference.

AC performs reasonably well given the current GA’s approach, where there is only a 0.01 probability of mutating each gene. Seven variants are not overfitting to the training instances, but only four manage to generalise to unseen test instances. These provide speed-ups of 13.31%, 4.15%, 14.32%, and 3.75% for folds k-5, 6, 8 and 9, respectively [11]. Regarding runtime measurement, the best variant reduces the runtime from 1077 to 808 seconds, on 247 training instances. We measured this using the Linux `time` command.

³<https://github.com/bloa/magpie/tree/5b85d5dc11380abbfd5acaab33f8cc8f53d3f8e>

⁴http://aclib.net/cssc2014/solvers/minisat_HACK_999ED_CSSC.tar.gz

⁵http://aclib.net/cssc2014/instances/circuit_fuzz.tar.gz

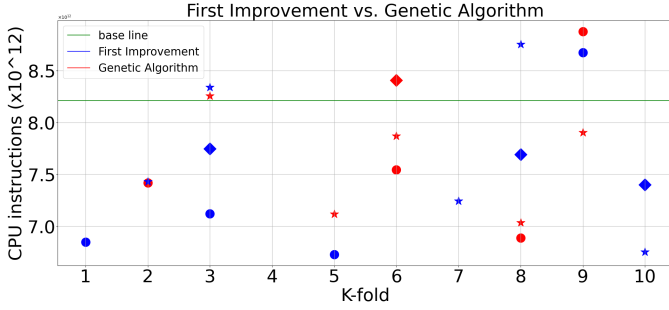


Fig. 1. Comparison between local search and genetic algorithm, from the test phase. [Star: algorithm configuration, Circle: genetic improvement on Statement, and Diamond: Joint search space]

For genetic improvement, we ran the experiment with the first improvement local search algorithm. Three of the ten variants generated did not overfit, and passed the test set. Despite the different budget strategy, which makes the number of total steps on average (689 steps) much less than the original MAGPIE’s experiment setting (fixed 1000 steps), within a roughly 3-hour budget, the best variant found provides a 18.05% speed-up, which reduces runtime from 1077 seconds down to 992 seconds, on 247 training instances.

Furthermore, these results are worth noting as each experimental results are better than the original MAGPIE paper’s experiment result. E.g., with first improvement local search, Blot and Petke [1] reported 11% and 15% speed-ups for AC and GI, respectively. In our case, program variant found in the AC with GA experiment led to a 14.32% speed-up and GI with LS experiment led to a 18.05% speed-up over original software.

Nonetheless, these are heuristic algorithms (i.e., local search and genetic algorithm), which means the workflow and results are non-deterministic. Although the latest result shows an improvement over previous work, it does not mean the latter runs will always perform better. Yet, what we found is that there exists room for runtime improvement of the MiniSAT benchmark. Most importantly, we demonstrate that the optimisation can be achieved automatically.

Moreover, we noticed a common pattern in many runs – certain program variants, when preserved from generation to generation, led to faster convergence to the best known solution. We conjecture that it would be good to keep these individuals from generation to generation as these contain a set of good genes and only discard them when a better individual is found. This mechanism is known as ‘elitism’. Yet, we note here that our GA did not use elitism, but these phenomena occur by chance and turn out to be mimicking elitism’s behaviour.

Figure 2 shows the k-10 (i.e., fold 10) of local search. There is only one major speed-up of 19.51% within the first 40 minutes for GI. For the remaining time, there is rarely any improvement and the final variant’s runtime is only improved to 19.54%. On the other hand, GA frequently manages to find improvements, with a final 32.00% speed-up. From Figure



Fig. 2. Relative fitness % with respect to original software for each patch generated in K-10 found in the first improvement local search experiment during the training phase.

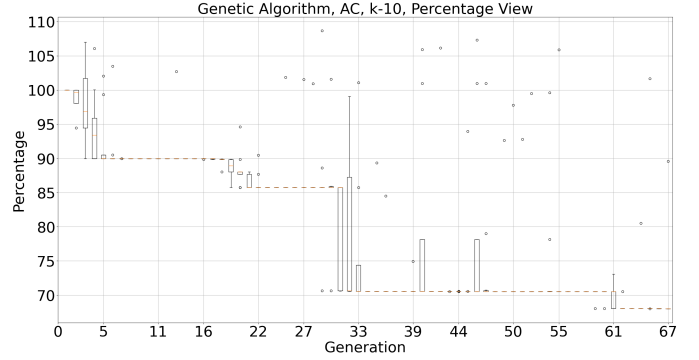


Fig. 3. Relative fitness % with respect to original software for each patch generated in K-10 found in the algorithm configuration with genetic algorithm experiment, during the training phase.

3, we see that, for each improvement, elite individual(s) are responsible for leading the whole population to convergence. After cross checking with the output log, we found that these individuals were lucky enough and did not participate in any genetic operator and remained unmodified. Thus, with the help of unmodified elite individuals, GA can find a better variant than LS, given the same instances and budget.

Conversely, a set of good genes was ignored if elite individuals were not preserved. For instance, k-4 could be considered as the hardest fold to optimise as GI with LS took around 90 minutes before managing to find an improvement, of 13.02%. Likewise, GA also faces the same obstacle as the majority of the population manages to find a tiny improvement of 2.36% speed up. Yet, the interesting aspect of GA is where, in the 13th generation of k-4, an outlying individual manages to find a 23.53% speed-up while the majority has not found any speed-up at all. In 14th gen, this same individual was selected from tournament selection. However, it participated in mutation and crossover operations that deteriorated the overall performance. In particular, run timeout error occurred, causing set of genes to not be moved forward to the next generation. However, this would not happen if elitism is applied as elite individuals would be preserved.

With the above investigation, we believe that elitism should

TABLE II
SINGLE BEST FITNESS IMPROVEMENT FROM ALL TEST INSTANCES AND REPORTED IN % OF CPU INSTRUCTIONS. FOR JOINT SEARCH SPACE OF GENETIC ALGORITHM, NONE OF THE FOLDS PRODUCE VARIANTS THAT MANAGE TO GENERALISE TO THE UNSEEN TEST INSTANCES.

Scenario	AC	GI	Joint
local search	-17.75%	-18.05%	-9.88%
genetic algorithm	-14.32%	-16.12%	N/A

be applied as it can increase GA’s performance. Doing so will allow that set of genes to survive to the proceeding generations with a promise of leading the whole population to the best known setting as seen in some folds.

However, elitism could deter GA from introducing diversity to the population. This is because elite individuals are likely to be selected from the tournament selection and participate in the genetic operators. By mutating the same genes over and over again, GA will not explore the whole search space and evolved solutions might get stuck in suboptimal solution while better ones would never be explored.

To answer RQ1, both search techniques perform similarly on the MiniSAT benchmark. Three and five variants of GI with LS and AC with GA, respectively, manage to generalise to the unseen test instances. The best variant evolved by GI with LS managed to find a variant that offered a 18.05% speed-up over the original software, while AC with GA managed to find a 14.32% speed-up. Nonetheless, we believe that GA’s performance can be further increased by introducing elitism.

B. RQ2: Exploring Joint Search Space

Table II reports the best fitness of program variants evolved using search on both parameter and code changes simultaneously. LS only produces three non-overfitting variants. These variants manage to generalise to the unseen test instances, specifically k-3, k-8 and k-10, which provide speed-ups of 5.64%, 6.30%, and 9.88%, respectively.

On the other hand, the GA’s result looks more promising during the preceding phases as four variants manage to survive the validation phase. Unexpectedly, none of the four variants generalises to the unseen test instances. Although one variant completed the run within the time limit, it slowed down by 2.38%.

Initially, we expected the GA to perform similarly to LS in the joint search space, due to the results of RQ1. Given the unexpected result, we investigated the number of generations and edits generated by MAGPIE for both algorithms. Regarding generations, Blot and Petke [1] used a fixed 1000 steps and reported an improvement of 40% on test instances for MiniSAT. In our case, the number of generations on average was 631 and 157 for LS and GA, respectively. Given smaller generation sizes, it is likely that there was not enough time for the algorithm to find an improvement. We conjecture that additional time would be necessary as the search space size

increases. Alternatively, more intelligent search strategies are necessary to explore this larger search space.

We also looked at the distribution of the types of edits for LS and GA. Upon close inspection of edits, we found that LS’s edits are more diverse, meaning that statement edits are split between various operations: Statement Insertion, Deletion, and Replacement. In contrast, GA’s individuals usually contain one or two statement edit types. It’s unclear to what extent this observation influences the result, thus further experiments might be needed.

We also investigated the number of parameter and statement edit(s) for each k-fold of each algorithm, for the joint search space [11]. We concluded that the majority of the edits from variants generated in the experiments that use GA mostly modify parameter values, which is, on average, around five times more than edits that modify source code. Conversely, edits from LS modify more statements than parameter values which is around twice as much. From the above observation, we conjecture that an edit’s representation can influence the numbers of each type of edit. This is because GA edit’s representation contains many more genes related to parameters than genes related to statements. Thus, there are more parameter edits for GA to mutate. The same phenomenon could have happened if GA represented all statements instead of storing only modified statements. Yet, this would not occur in practice as this approach would blow up the chromosome as there would be as many genes as the number of statements. This would also create a massive load on memory which we should avoid.

For LS, the probability of selecting a particular mutation operator is likely to influence the type of edits appearing in an evolved software variant. To create a new edit, LS selects a mutation operator from a list of possible edits, which includes one parameter edit and three statement edit operators (e.g., insertion, deletion, and replacement). So, there is a higher probability that LS will select a statement edit operator. In future work, we aim to modify the probability of selecting parameter edit and statement edit operators in LS.

Nonetheless, we conducted additional experiments (with the same time budget) on three search spaces with GA by increasing the population size from 10 to 100 as it was shown effective in previous GI work. Similarly, we found that higher population size led to better performance for AC and AC+GI search space. The best variants provide speed-ups of 18.61%, 19.06%, and 5.59% for AC, GI and AC+GI, respectively. Upon closer inspection, many edits simply remove assert statements, which is something one might want to avoid in practice. Nevertheless, some led to true improvements, by, e.g., modifying internal parameters of MiniSAT.

To answer RQ2, when searching the joint search space of parameter values and code edits, we found that the first improvement local search can find a software variant with a 9.88% speed-up. However, with the genetic algorithm, MAGPIE cannot find any improving variant that generalised to the test set.

TABLE III
FINAL RANKING FROM THE TEST PHASE. WE SELECT THE SINGLE BEST
VARIANT FROM EACH EXPERIMENT.

Rank	Technique	Speed-up
1	GI with LS	18.05%
2	AC with LS	17.75%
3	GI with GA	16.12%
4	AC with GA	14.32%
5	AC + GI with LS	9.88%
6	AC + GI with GA	N/A

C. RQ3: Best Search Strategy in MAGPIE

Table III reports the final ranking of the single best variant from a total of six experiments. The improvements, in terms of speed-ups, range from 9.88% to 18.05%. From the table, we can see that LS always manages to find improving software variants, whilst GA failed to find improving variants in the joint search space.

In the case of GI with LS, more than half of the generated mutants failed to compile or did not pass training. Nonetheless, the largest improvement was found in this experiment. At the same time, AC + GI with LS resulted in fewer erroneous mutants than GI with LS.

As expected – because no modification was performed on the source code – none of the experiments on the AC search space resulted in errors, although some occasionally ran into timeouts.

Ultimately, we aim to demonstrate that these algorithms and techniques can be used in an industrial setting and solve real problems. Hence, the generated edit should be correct, meaning a human developer would accept these modifications as if they were modified by a fellow colleague and allow them to be merged into the trunk. Nonetheless, this is not any new practice, but rather one of the vital processes in software engineering, which is ‘code review’. That being said, we conducted a code review on all statement edits of the best variants from Table III. We did not conduct a code review on parameter edits because these parameters are intentionally exposed by developers for the users to tune. Thus, we assume that such modifications to their values will not lead to erroneous behaviour.

We classified patches into two categories: corrects and incorrects. Correct patches modify the original software execution without affecting the final result. They can modify how the software works internally (e.g., whether to select a variable at random or not) or affect its output, which can lead to reporting inaccurate statistical data in the console. For instance, removing a `num_prop++` expression will make MiniSAT report a wrong number of propagations that occur before reaching a solution. Conversely, incorrect patches can be proved to potentially lead to incorrect results. For instance, an incorrect MiniSAT patch may affect its final report (“satisfiable” or “unsatisfiable”) on some given instances. Manual examination of all patches revealed no incorrect patch.

However, we found that some of the patches contained edits

that removed assert statements. Assertions help ensure that the program is in a valid state before the execution proceeds. Removing them may potentially introduce various issues such as unexpected behaviours, security vulnerabilities (e.g., buffer overflows, out-of-bound reads), etc. Although no such issues occurred in this study, it is critical to exercise caution when merging any automatically generated code.

To answer RQ3, on MiniSAT, MAGPIE with first improvement local search produced the most execution time-improving program variants. First, LS finds variants that generalise to unseen instances in all experiments. Second, in every search space, the best variant generated by LS always outperformed the best variant generated by GA. We note here that all improving program variants included statement edits that deleted assert statements, which can be risky and may lead to potentially unstable or unsecure code. However, such risks can be mitigated through thorough code review and extensive testing.

VI. THREATS TO VALIDITY

This section discusses threats to validity and how each can affect our experimental results.

Benchmark. In this study, we only experimented on a single benchmark. Although results are promising and provide insights into the capability of each search strategy to find improving software variants, it is worth noting that these observations might not apply to other benchmarks. Yet, empirical results demonstrated that automatically optimising software at different granularity levels concurrently is possible without any human interference. Further investigation is necessary, considering more benchmarks, to confirm that the results generalise.

Patch Quality. In a typical software engineering workflow, engineers gather requirements, design the system, implement, test, and perform code reviews before pushing updated code to the main branch. However, by automatically optimising software, many of these steps are ignored, which can pose a significant threat to the code base. In our case, MAGPIE only handles the development and testing steps.

Code review ensures that code is well written and meets established standards. It can assist in detecting and preventing the introduction of new bugs to the existing codebase. In this research, we manually inspected the patches generated to assess their correctness. However, we found edits removing assert statements, which probably should have been left unmodified. We do not recommend automatically merging the generated patches without thorough manual code review.

Search Heuristic. LS and GA are both meta-heuristic algorithms. In particular, this means that their executions are non-deterministic, as highlighted in Section V-A. Most importantly, results from a single run do not necessarily reflect the underlying performance distribution. Our experimental protocol is based on k-fold cross-validation to alleviate these limitations.

VII. CONCLUSIONS

Software optimisation has a long-standing history in computer science. For decades, researchers and practitioners from all over the world have been trying to find ways to make software perform better, included but not limited to reducing execution time, improving response time, or consuming less energy. These optimisations can be achieved at various levels, such as compiler optimisation, parameter tuning, or source code modification.

Two research domains focus on parameter tuning: AC and HPO. It is worth noting that compiler optimisation and HPO can both be modeled using AC, as they fundamentally involve setting and tuning parameters and strategies. Additionally, source code modifications can be achieved through GI.

There are many frameworks and algorithms that are capable of optimising software. Yet, these are only capable of optimising at a single level – either parameter or source code – but not both. In 2022, Blot and Petke [1] introduced MAGPIE, a framework that is capable of simultaneously exploring and optimising both parameters and source code. It provides a first improvement local search, which has been empirically shown to perform well; however such a search heuristic is not state-of-the-art in all mentioned optimisation domains.

Blot and Petke [4] empirically showed that local search algorithms perform best in the GI domain. On the other hand, Yang and Shami [5] had empirically shown that genetic algorithms perform on par with other methods in optimising various ML model parameters. Most importantly, both approaches have the potential to generalise to both application domains as well as their joint optimisation.

We conducted a total of six experiments that targeted the execution time improvement of MiniSAT, a well known SAT solver used in both AC and GI previous work. We used GI with LS and GA, AC with LS and GA, and explored the joint search space of edits of AC and GI using LS and GA.

From our experiments, we found that GI with LS produced slightly better program variants (18.05% speed-up) than AC with GA (14.32% speed-up). In the case of the joint search space, the result is unexpected for GA as no variant leads to any improvement on the test instances. Conversely, LS manages to find a 9.88% speed-up improvement. Lastly, the search strategy that led to finding program variants with the largest speed-up was GI with LS followed by AC with LS and GI with GA with speed-ups of 18.05%, 17.75%, and 16.12%, respectively. Lastly, we found edits deleting assert statements in multiple improving software variants. Whilst such edits generally do not impact the software final results, they may occasionally lead to unexpected behaviours and security vulnerabilities. Hence, manual code review is necessary to assess the impact of automatically generated code before any integration into production code.

To enable others to replicate and extend our work, we provide links to our MAGPIE version and scripts to run the benchmark in our artefact [11].

VIII. ACKNOWLEDGEMENTS

We would like to thank Dr Yang for explanations on the implementation of the GA algorithm [5].

REFERENCES

- [1] A. Blot and J. Petke, “MAGPIE: Machine automated general performance improvement via evolution of software,” arxiv, 2022. [Online]. Available: <https://arxiv.org/abs/2208.02811>
- [2] W. B. Langdon and B. J. Alexander, “Genetic improvement of OLC and H3 with Magpie,” in *IEEE/ACM International Workshop on Genetic Improvement, GI@ICSE 2023, Melbourne, Australia, May 20, 2023*. Melbourne, Australia: IEEE, 2023, pp. 9–16. [Online]. Available: <https://doi.org/10.1109/GI59320.2023.00011>
- [3] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, “Genetic improvement of software: A comprehensive survey,” *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2018.
- [4] A. Blot and J. Petke, “Empirical comparison of search heuristics for genetic improvement of software,” *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 5, pp. 1001–1011, 2021.
- [5] L. Yang and A. Shami, “On hyperparameter optimization of machine learning algorithms: Theory and practice,” *Neurocomputing*, vol. 415, pp. 295–316, 2020.
- [6] E. Schede, J. Brandt, A. Tornede, M. Wever, V. Bengs, E. Hüllermeier, and K. Tierney, “A survey of methods for automated algorithm configuration,” *Journal of Artificial Intelligence Research*, vol. 75, pp. 425–487, 2022.
- [7] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *J. Mach. Learn. Res.*, vol. 13, no. 1, p. 281–305, feb 2012.
- [8] Y. Li, Y. Zhang, G. Zhou, and Y. Gong, “Bayesian optimization with particle swarm,” in *2021 International Joint Conference on Neural Networks (IJCNN 2021)*, 2021, pp. 1–6.
- [9] Y. Pushak and H. H. Hoos, “Golden parameter search: Exploiting structure to quickly configure parameters in parallel,” in *15th Genetic and Evolutionary Computation Conference (GECCO 2020)*. ACM, 2020, pp. 245–253.
- [10] A. P. Engelbrecht, *Computational Intelligence*, 2nd ed. Hoboken, NJ: Wiley-Blackwell, Oct. 2007.
- [11] S. Group, “Empirical comparison of runtime improvement approaches,” 2025, accessed: 2025-01-30. [Online]. Available: <https://github.com/SOLAR-group/empirical-comparison-of-runtime-improvement-approaches>
- [12] F. Hutter, M. López-Ibáñez, C. Fawcett, M. T. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Stützle, “Aclib: A benchmark library for algorithm configuration,” in *Learning and Intelligent Optimization (LION 8)*, ser. LNCS, vol. 8426. Springer, 2014, pp. 36–40.