Multi-Objective Genetic Improvement: A Case Study with EvoSuite

James Callan and Justyna Petke

University College London, United Kingdom
{james.callan.19,j.petke}@ucl.ac.uk

Abstract. Automated multi-objective software optimisation offers an attractive solution to software developers wanting to balance often conflicting objectives, such as memory consumption and execution time. Work on using multi-objective search-based approaches to optimise for such non-functional software behaviour has so far been scarce, with tooling unavailable for use. To fill this gap we extended an existing generalist, open source, genetic improvement tool, Gin, with a multi-objective search strategy, NSGA-II. We ran our implementation on a mature, large software to show its use. In particular, we chose EvoSuite — a tool for automatic test case generation for Java. We use our multi-objective extension of Gin to improve both the execution time and memory usage of EvoSuite. We find improvements to execution time of up to 77.8% and improvements to memory of up to 9.2% on our test set. We also release our code, providing the first open source multi-objective genetic improvement tooling for improvement of memory and runtime for Java.

Keywords: Genetic Improvement · Multi-Objective Optimisation. · Search-Based Software Engineering

1 Introduction

Performance is one of the key properties of software. Programs that are laggy, consume a lot of resources, are not only a source of user complaints, but can render such software unsustainable and unusable. Even though there have been extensive studies on software performance issues, e.g., [9], and tools have been proposed to automatically improve software's performance, whether through compiler optimisation, parameter tuning, genetic improvement, or other, few consider the interplay between the different non-functional properties [8]. Such automated tooling is needed, given that changes that improve one non-functional property might negatively influence another.

Improving the speed of a program may have unintended consequences. A popular strategy would be caching of intermediate computation results, e.g., in array structures. This, however, leads to increased memory use. Furthermore, if arrays are large enough, the time cost of array operations might outweigh the computational time savings. It is thus important that we consider memory usage when optimising the execution time of an application. In fact finding more memory efficient versions of software can be beneficial to it's speed by saving

expensive garbage collection and page swapping operations. Although multi-objective search algorithms seem best fit for this problem domain, to the best of our knowledge, there is no tool available that provides this facility, despite such work being proposed in the past [13].

With this in mind, we extended an existing Genetic Improvement (GI) [11] framework, Gin [4], with a multi-objective search strategy[1], namely NSGA-II [6]. We chose GI as it is an approach which can be applied to any source code without the need for tuning or domain expertise. GI utilises search algorithms to find patches which can improve the program with respect to a given objective. GI has already been successfully used to fix bugs, optimise program's runtime, memory, energy consumption, and other [11]. GI has the advantage of being ambivalent to the particular search algorithm used to explore the landscape of patches, thus we can very easily plug in multi-objective algorithms to improve both memory and execution simultaneously or find good trade-offs between them.

To show usefulness of our implementation we target a large, popular, mature piece of software — EvoSuite [7], a tool which utilises Genetic Programming in an attempt to automatically generate test suites for Java programs. EvoSuite then generates and minimises a set of assertions for each test. This allows the tests to detect regressions in future versions of software. EvoSuite is often run with a time limit for test generation for each target class. Once time limit is reached, EvoSuite stops, regardless of whether a particular coverage objective was achieved. It is thus important that EvoSuite can efficiently explore the search landscape, and evaluate generated tests. By improving the speed of EvoSuite we can increase the amount of test cases it can generate and evaluate in the given time limit. At the same time we don't want such improvements to happen at the cost of unnecessary memory use.

Our results are encouraging. We report improvements of up to 78% in runtime and 9% in memory use for 10 methods in EvoSuite software. The best patches removed redundant yet expensive checks, and change the scope of try catch statements. We hope that researchers and practitioners alike find these results encouraging, to apply multi-objective GI to other software, and continue research in this direction. There is more to be explored: which multi-objective algorithms are best fit for search-based software improvement? which other properties could we target? and other. We release our code [1] to facilitate future work.

2 Background

Genetic Improvement uses automated search to improve existing software [11]. GI takes a section of source code and the tests which cover it and searches through a landscape of potential patches in order to find those which improve a given software property. Standard GI operators delete, replace, or copy code fragments, such as statements or lines. Testing is also used as standard as a proxy for capturing software's functional correctness, and to measure the software improvement property of interest. For instance, for runtime improvement, fitness measure of a given program variant will be the time taken by the given test suite.

[1] A pull request can be found at https://github.com/gintool/gin/pull/89.

The most popular search algorithm for GI has been genetic programming, however, local search has been recently shown to be as effective [3]. Although White et al. [12] were the first to propose multi-objective (MO) search to improve software's non-functional behaviour, they evolved full programs rather then patches making their approach only applicable to toy software. Basios et al. [2]'s work is closest to what we want to achieve. They used MO to improve memory consumption and runtime of Java applications. However, they used specialised mutations, targeting data structures only. Furthermore, they have not made their code available. This leaves the question of how effective standard GI operators are at MO optimisation unanswered.

3 Approach

We pose that ***multi-objective (MO) Genetic Improvement (GI) provides a useful generalist approach for automated software optimisation***.

In order to prove this statement we incorporate multi-objective search into an existing GI framework. We target improvement of non-functional software behaviour, as it's been shown that changes that improve such properties are often non-obvious and their impact on other software properties is hard to predict [5]. We also aim to improve a large, mature piece of code, that comes with an extensive (99% line coverage) test suite. Given the effort put into development of such a piece of software, we expect it will be challenging to find improvements. Thus, if any are found, it will provide strong evidence for usefulness of multi-objective GI.

4 Methodology

In our empirical study we use an existing GI tool, and extend it with a multi-objective algorithm, namely, NSGA-II [6], as it's one of the most popular MO algorithms and proved successful in previous related work [2]. Otherwise, we use the most common GI settings. In particular, we mutate statements, and use 4 standard GI mutation operators, as the building blocks for our generated patches. Each can either *delete* a statement, *copy* one statement from one location to another, *replace* a statement with another, or *swap* locations of two statements. Moreover, we set each run of GI to consists of 40 individuals and 10 generations, for a total of 400 evaluations as shown to be effective in previous GI work [10]. We repeat each GI run 10 times, to account for the non-deterministic nature of NSGA-II. We also separately evaluate each patch found 20 times, to account for noise in fitness measure, as it's often encountered when measuring non-functional properties of software.

GI tool Recent survey of GI tooling, revealed that [13] few GI tools can be easily applied to unseen software. After closer investigation we chose Gin [4], as it is the only one to implement fitness functions for at least two non-functional software properties, namely runtime and memory consumption. Moreover, it provides profilers for both properties, thus helps automatically identify the most time and memory consuming parts of code. Runtime fitness measure takes the elapsed time on a set of tests. Memory fitness measure simply calculates memory use before and after a test is run.

Target Software As our target software we chose EvoSuite — a tool for automatic test generation. It has 1.1 million lines of code, it's been developed for 11 years, and comes with an extensive test suite. We ran Gin's memory and runtime profilers on the evosuite-client module, which contains the code for actual test generation. The profilers' output provides us with a list of methods with the largest impact on memory and execution time, along with the tests that cover those methods. Unfortunately, at this point we discovered a bug in Gin's test runners. EvoSuite uses an example project in a different package for many of it's tests and these tests are not compatible with Gin, we chose to discard the methods covered by these tests and focus on those which had all passing tests. This resulted in 27 methods from the execution time profiler and one method from the memory profiler. We further filtered out methods with less than 5 lines of code as they would be too small for improvements to be found. From here we selected the method found by the memory profiler and the top 9 slowest methods with more than 5 lines of code, giving us 10 methods to attempt to improve. The line coverage of the tests on each method can be found in Table 1.

5 Results and Discussion

In this section we present the improvements which we found to EvoSuite using our multi-objective genetic improvement approach.

Table 1. Table showing execution time improvements found by GI. Numbers in brackets indicate the effect the patch had on the other property, i.e., memory use.

	Execution Time Imp.		
Method	Median	Max	Line Coverage
MersenneTwister.nextGaussian	55.84%	67.12% (-1.1%)	100%
TestFactory.addConstructor	34.9%	37.98% (-2.13%)	73%
RegexDistanceUtils.cacheRegex	12.83%	30.29% (-0.53%)	100%
DistanceCalculator.visit	27.88%	60.62% (-0.81%)	84%
FileIOUtils.recursiveCopy	0.39%	0.42% (-2.33%)	100%
TestCodeVisitor.visitPrimitiveStatement	42.42%	44.44% (-1.02%)	80%
DistanceEstimator.getDistance	48.62%	49.86% (-6.28%)	87%
TestCodeVisitor.getClassName	67.54%	78.77% (-7.05%)	85%
DistanceCalculator.getStringDistance	22.46%	25.63% (-0.67%)	70%
StringHelper.StringRegionMatches	51.91%	58.28% (-2.9%)	80%

Over our 10 runs we find improvements for every single method which we tried to improve, with improvements to runtime of up to 78.8% and improvements to memory of up to 9.2% (see Table 1, and our repository for all Pareto Fronts [1]). Interestingly, the method highlighted by Gin's memory profiler was the only method in which we could not find any improvements to memory. The method in question copies files from one place to another, in doing so it loads the contents of the files being copied into memory 2048 bytes at a time. Perhaps reducing the size of this buffer would reduce the memory usage, at the cost of execution

Table 2. Table showing memory improvements found by GI. Numbers in brackets indicate the effect the patch had on the other property, i.e., runtime.

	Memory Imp.	
Method	Median	Max
DistanceEstimator.getDistance	2.72%	5.81% (20.49%)
DistanceCalculator.visit	2.73%	4.43% (9.96%)
TestFactory.addConstructor	2.51%	4.12% (18.83%)
StringHelper.StringRegionMatches	2.04%	3.64% (17.49%)
MersenneTwister.nextGaussian	1.64%	4.58% (-4.89%)
TestCodeVisitor.getClassName	3.37%	9.2% (1.89%)
RegexDistanceUtils.cacheRegex	2.49%	8.46% (-7.44%)
FileIOUtils.recursiveCopy	0.00%	0.00% (-0.23%)
DistanceCalculator.getStringDistance	6.52%	6.76% (-12.85%)
TestCodeVisitor.visitPrimitiveStatement	4.3%	5.76% (11.37%)

time, but our mutation operators are not able to make this kind of change. Using mutation operators which modify constants could lead to further improvements.

We find that, in all cases, the best improvements to execution time lead to memory usage increasing, mostly by small amounts. However, in one case, it increased by almost 6%. Improvements to memory lead to improvements to execution time in 6 cases. This could be due to fewer GC calls. In 4 cases, the best memory improvements also lead to an increase in execution time. These patches offer developers a choice over which property is more important to them, and could also allow multiple versions of EvoSuite to be made available to systems with different hardware resources.

A patch to the TestCodeVisitor.getClassName method was one which improved execution time the most, finding an improvement of almost 80%, on the example EvoSuite class called Tutorial_Stack. This patch leads to a 4.8% improvement on the number of generations evaluated. It is made up of 3 edits, 1 delete, 1 copy, and 1 replace statement edits. The main execution time improvement from the patch comes from removing a conditional which checks whether or not the current ClassLoader for the system under test has the class and then gets the class's Canonical. The area of code which is removed is wrapped in a try catch which ignores exceptions and is able to fail without consequence on the rest of the method. The code is also accompanied by a comment which states that the code is irrelevant in normal use of EvoSuite and only triggered during testing.

We also find a patch to the TestCodeVisitor.getClassName method that provides the greatest memory consumption reduction. This patch changes the scope of try statement which changes the way in which Java releases resources, thus decreasing memory usage by a small amount. Many memory improvements, however, offer marginally improved speed for slightly improved memory. This may be preferable to the much faster but more memory intensive patches also produced.

All patches (see Table 1 and 2) were subsequently run on the whole EvoSuite test suite, showing no regression errors.

Cost of Genetic Improvement Each run improving all 10 methods took a median time of 48 minutes, with the slowest run taking 75 minutes and the quickest taking only 23 minutes. The difference between runs is due to the number of compiling patches generated. Runs in which a large number of patches fail to compile will need to run significantly fewer tests and thus complete quicker. We believe this is a relatively small cost for the improvements we found.

6 Conclusion

We extended an existing GI tool to provide the first open source multi-objective genetic improvement tool for Java [1], that can improve software's runtime and memory consumption out-of-the-box. We applied it to the EvoSuite test generation tool. We found improvements to both execution time for all methods improved and memory to all but one of the methods which we improved. We found that the NSGA-II algorithm was able to effectively explore the search landscape of patches, finding good trade-offs between memory and execution. Our approach was relatively fast and fully automatic, requiring no domain expertise.

Acknowlegements This work was funded by the EPSRC grant EP/P023991/1.

References

1. https://github.com/SOLAR-group/EvoSuiteGI
2. Basios, M., Li, L., Wu, F., Kanthan, L., Barr, E.T.: Darwinian data structure selection. In: ESEC/SIGSOFT FSE. pp. 118–128. ACM (2018)
3. Blot, A., Petke, J.: Empirical comparison of search heuristics for genetic improvement of software. IEEE TEVC **25**(5), 1001–1011 (2021)
4. Brownlee, A.E.I., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: genetic improvement research made easy. In: Auger, A., Stützle, T. (eds.) GECCO. pp. 985–993. ACM (2019)
5. Bruce, B.R., Petke, J., Harman, M., Barr, E.T.: Approximate oracles and synergy in software energy search spaces. IEEE TSE **45**(11), 1150–1169 (2019)
6. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In: PPSN. LNCS, vol. 1917, pp. 849–858. Springer (2000)
7. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: SIGSOFT FSE. pp. 416–419. ACM (2011)
8. Hort, M., Kechagia, M., Sarro, F., Harman, M.: A survey of performance optimization for mobile applications. IEEE TSE pp. 1–1 (2021)
9. Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S.: Understanding and detecting real-world performance bugs. In: PLDI. p. 77–88. PLDI '12 (2012)
10. Motwani, M., Soto, M., Brun, Y., Just, R., Goues, C.L.: Quality of automated program repair on real-world defects. IEEE TSE **48**(2), 637–661 (2022)
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 TEVC **22**(3), 415–432 (2018)
12. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE TEVC **15**(4), 515–538 (2011)
13. Zuo, S., Blot, A., Petke, J.: Evaluation of genetic improvement tools for improvement of non-functional properties of software. In: Fieldsend, J.E., Wagner, M. (eds.) GECCO '22. pp. 1956–1965. ACM (2022)