Evaluation of genetic improvement tools for improvement of non-functional properties of software

Shengjie Zuo, Aymeric Blot, Justyna Petke

Evaluation of Genetic Improvement Tools for Improvement of Non-functional Properties of Software

Shengjie Zuo
University College London
London, United Kingdom
shengjie.zuo.19@alumni.ucl.ac.uk

Aymeric Blot
University College London
London, United Kingdom
a.blot@cs.ucl.ac.uk

Justyna Petke
University College London
London, United Kingdom
j.petke@ucl.ac.uk

ABSTRACT

Genetic improvement (GI) improves both functional properties of software, such as bug repair, and non-functional properties, such as execution time, energy consumption, or source code size. There are studies summarising and comparing GI tools for improving functional properties of software; however there is no such study for improvement of its non-functional properties using GI. Therefore, this research aims to survey and report on the existing GI tools for improvement of non-functional properties of software. We conducted a literature review of available GI tools, and ran multiple experiments on the found open-source tools to examine their usability. We applied a cross-testing strategy to check whether the available tools can work on different programs.

Overall, we found 63 GI papers that use a GI tool to improve non-functional properties of software, within which 31 are accompanied with open-source code. We were able to successfully run eight GI tools, and found that ultimately only two —Gin and PyGGI— can be readily applied to new general software.

CCS CONCEPTS

• **Software and its engineering** → *Search-based software engineering*; *Software evolution*; *Extra-functional properties*; • **General and reference** → **Surveys and overviews**.

KEYWORDS

genetic improvement, survey, tooling, non-functional properties

ACM Reference Format:
Shengjie Zuo, Aymeric Blot, and Justyna Petke. 2022. Evaluation of Genetic Improvement Tools for Improvement of Non-functional Properties of Software. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3520304.3534004

1 INTRODUCTION

Genetic improvement (GI) is an automated software engineering approach that applies search strategies to optimise existing software [69]. It has many applications, including both improving functional properties of software, e.g., automated program repair [57] or feature transplantation [71], and non-functional properties, such as execution time [85], energy consumption [19], memory usage [88], or code size [20].

A significant amount of research was conducted on automated software repair specifically, thus several studies summarising and comparing tools for software repair exist [66, 79]. Furthermore, there are many repair tools available, that are frequently compared with each other [9, 89]. On the other hand, non-functional properties (NFP) refer to the constraints on how software implements and delivers their functionalities [78]. NFP are considered to be as important as functional properties [80]; for example, many software failures were shown to be caused by unsatisfied non-functional properties [26], and slow response times may lead to customers rejecting software [33]. While there are many GI tools tackling NFP improvement —such as for example GISMO [44, 71, 72], locoGP [22], Gin [14], and PyGGI [1, 2]— they are seldom reused and compared, and there is no existing work overviewing all the available GI tools for NFP improvement.

In this paper, we therefore conduct a literature review on NFP-improving GI tooling and we discuss and compare their availability, usability, and generalisability. We hope that this work may contribute to improvement, development, dissemination, and adoption of GI tools in the community, as well as ultimately their increased use in real-world industrial context.

Our contributions thus include:

(1) a literature review resulting in 63 papers using GI tooling for NFP improvement, 31 of which having associated available open-source code;
(2) a usability study of the found open-source GI tools; and
(3) a generalisability study of 8 distinct GI tools.

The overarching conclusion of our study is that much of GI research work does not come with reusable implementations. Furthermore, of the available GI tools only two, Gin [14] and PyGGI [1], can be easily be applied to new software.

2 BACKGROUND

Genetic improvement (GI) is a relatively new research field [69]. Although its foundation can be traced back to the early days of computer science, this field arose as an active one only in the last ten years. GI takes an existing software system and using search-based methods, generates variants that improve it with respect to a given fitness function. Depending on the fitness considered, GI can improve a given software system's functional properties, correcting bugs [56] or transplanting new functionalities [5], as well as improve non-functional properties, such as for example execution time, energy consumption, memory usage, or code size.

The software under consideration is most often modified by GI at the level of source code, although some work successfully operated

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/3520304.3534004

at binary or assembly level [74]. One reason is the production of readable patches that can then be more easily understood and accepted by the software's developers [82]. GI tools operating at the source code level can use lines of code directly, but often use other types of representation, such as an abstract syntax tree (AST) [56] or notation similar to BNF grammar [44].

In addition to the different representations of source code, there are various search strategies to generate patches based on the processed code. The most widely used algorithms include random search, local search, and genetic programming. Random simply applies, uniformly at random, a mutation operator, usually deletion, insertion, and replacement [67]. Because of its simplicity, it is often used as a baseline. Local search [34] also only relies on mutations, but is used to iteratively search for better and better software variants by considering sequences of mutations. Starting from the original software (therefore an empty sequence), local search in GI either appends a new mutation to the current sequence, accepting it if the corresponding software variant is considered better. Existing mutations can also be removed, as a way to decrease bloat, keep the overall sequence length reasonable, and avoid overfitting. The different types of edits are usually considered with fixed probabilities, proportions of which can significantly impact search performance [76]. Genetic programming (GP) [36, 54], on the other hand, combines both mutations and crossover to evolve populations of software variants. GP has been used for GI since the field inception, and strategies of some successful tools in program repair are based on GP [28, 57, 83]. The most common types of crossover used in GI are the concatenation crossover [44], that simply combines two sequences of mutations, and 1-point crossover [57], that combines the start of the sequence from one individual with the end of the sequence of a different one. Overall, whilst GP has been for a very long time the privileged GI search process, local search has recently been used more often, as it's simpler and can be equally effective [12].

3 RESEARCH QUESTIONS

In order to figure out which GI tools are available and how they work, we set out the following research questions:

RQ1: Which state-to-the-art GI tools target non-functional properties (NFP) of software? The first question is meant to fill in the blank in current research in terms of summarising existing GI tooling for NFP improvement. We are interested in how many papers used GI tools for NFP improvement thus far and how many of those tools are available for use.

RQ2: How many GI tools found in previous work can actually run? Then, we would like to check availability, hardware and software requirements, and determine which GI tool can actually be used.

RQ3: How do existing GI tools work, and can they be applied to different programs? Finally, we want to check how easy it is to apply existing tools to software to which the tools have not yet been run on. This is a critical consideration separating research artefact from actual stand-alone tools that could be widely used by researchers and developers.

Source	Filters
IEEE Xplore	Metadata with the exact key words of 'genetic improvement' Publication time between 2016 and 2022
ACM Digital Library	Title OR Abstract with the exact words of 'genetic improvement' Publication year between 2016 and 2022
Living Survey on GI	Publication year between 2016 and 2022 Conference and journal papers only

Table 1: Filters for the collections

	Papers		
Source	Total	On NFP	With code
Petke et al. [69]	66	27	19
ACM Digital Library	35	15	4
IEEE Xplore	57	10	9
Living survey on GI	264	63	45

Table 2: Results of the literature review

4 LITERATURE REVIEW

A literature review was conducted to answer RQ1. In order to review publications on GI tools efficiently, relevant papers published before 2016 were retrieved from an existing comprehensive survey [69]. For the papers published after 2016, we continue the literature review based on the collections of IEEE Xplore, the ACM Digital Library and the Living Survey on Genetic Improvement[1]. Details on the filters used during the literature review are presented in Table 1. Moreover, to further filter the relevant papers consistently, we apply the following rules:

(1) The paper should focus on non-functional properties of software.
(2) The paper should propose, implement, or reuse a tool that is shown to improve the performance of software in terms of non-functional properties.
(3) The paper should include evaluation on example programs or real-world projects.

The result of the literature review is summarised in Table 2. The comprehensive survey on GI [69] details 66 core GI publications published up to 2016, including 27 on NFP improvement and 19 with GI tools. For the papers published after 2016, the ACM Digital Library yielded 35 publications on GI, with 15 on NFPs and 4 with GI tools. IEEE Xplore yielded 57 entries, with only 10 on NFPs and 9 using GI tools. Finally, the living survey on GI yielded 246 GI publications published since 2016, with 63 relating to NFPs and 45 using GI tools. Overall, we found a total of 63 publications and 7 PhD theses [3, 6, 16, 29, 84, 86] that use GI tools to improve software's NFPs. Because for each of them we found corresponding publications, we won't include these PhD theses in the remainder of our study.

[1] http://geneticimprovementofsoftware.com/learn/survey

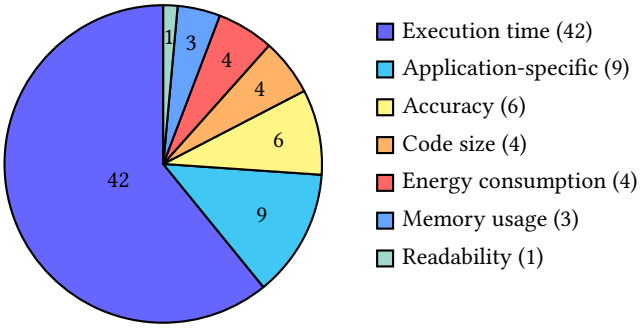

Figure 1: Distribution of non-functional properties in GI tool literature.

We then surveyed in more detail the NFP considered in each of the 63 papers. Time is the concern addressed in the vast majority of papers, with 34 papers considering execution time [1, 2, 7, 8, 10, 14, 15, 17, 24, 32, 35, 39, 41–44, 47–50, 55, 58–63, 68, 70–72, 75, 77, 87, 88], number of CPU or bytecode instructions [4, 11, 12, 21, 22, 85], or also loading time [23]. Other NFPs include code size [25, 38, 90, 91], energy consumption [13, 18, 19, 27], memory usage [7, 8, 88], accuracy of the underlying algorithm [30, 31, 59, 60, 62, 81], readability [73], or other application-specific NFPs [37, 40, 45, 46, 51–53, 64, 65]. A summary is presented in Figure 1. Furthermore, a few pieces of work considered multiple NFPs [7, 8, 27, 59, 60, 62, 88].

As for tool availability, we looked for URLs in papers as well as searched GitHub for the papers' titles or DOIs. Overall, we found open-source code associated with 31 papers, either on GitHub, SourceForge, or the authors' research website pages. In particular, we found many variations of the GISMO tool[2] that targets C/C++ code. Furthermore, despite having sometimes no direct mention in the publication itself and no available code, by communicating with authors we were able to confirm that GISMO and its underlying representation format were used in many more research publications [17, 18, 41–44, 48–50, 55, 62, 70–72]. We note that in some GI papers (e.g., [4, 65, 85]), a GI tool was built on top of a general evolutionary framework; whilst those frameworks are still available, we were unable to obtain the source code specific to the GI tool itself. Moreover, a few tools were listed on GitHub without a license, which would have prohibited their use. We raised the issue with the authors of such tools in those instances, who have promptly added a permissive license.

4.1 Summary

Detailed information on the 31 papers from which we were able to find available code is presented in Table 3. Table 4 details the 32 additional papers describing work that used GI tools, yet their source code was unavailable.

Answer to RQ1: We found 63 papers whose authors used GI tools for improvement of non-functional properties of software. We found source code associated with only 31 of those papers.

[2]http://www0.cs.ucl.ac.uk/staff/ucacbbl/gismo/

5 USABILITY STUDY

In this section, we conduct a set of experiments to answer RQ2: whether the GI tools we found can actually be used. We downloaded and tested all found tools following their requirements and the instructions given by their developers.

5.1 Experiments

Within the 31 papers detailed in Table 3 we can find 13 distinct tools[3]: a tool for shader simplification [75], GISMO [70], locoGP [22], HOMI [87], PyGGI 2.0 [1], GGGP [39], Optmizer [24], a tool for data optimisation [51], Gin [14], PowerGAUGE [27], GEVO [60], DFAHC [25], and finally a tool dedicated to routing protocols [64]. However, two of them — the tool for data optimisation and the tool for routing protocols — target application-specific NFPs and cannot be expected to be easily applied to different software, as non-trivial manipulation of the software to be improved is required. Thus, we only focus on the remaining 11 GI tools. Finally, we only considered a single version of each tool, either the latest version, the most general one, or if possible, the version with the fewest software and hardware dependencies.

5.2 Methodology

All tools were tested on the local virtual machines under their required environments. Because of the different environmental requirements, four local virtual machines running Ubuntu 20.04 and Ubuntu 16.04 were set up, with different versions of Java and C/C++ tools.

The steps to check whether the tools can be run are listed below. Tools were deemed unable to work if they failed any of those steps.

(A) Whether the testing machines meet hardware requirements.
(B) Whether the dependencies can be installed.
(C) Whether the tool can be compiled.
(D) Whether the tool can successfully run on data provided with its associated publication.

5.3 Results

Of the 11 tools investigated, we were able to run 8 without any issues.

We were unable to meet hardware requirements for GEVO [60]. The tool is available, but requires CUDA-compatible GPU hardware as an essential dependency.

We were unable to install dependencies for Optmizer [24]. The source code of the tool used in this paper is available on GitHub, but it relies on outdated NodeJS dependencies that do not resolve on a fresh install. In particular, the dependency tree seems to trigger 23 different deprecated packages, and the installation ultimately fails to complete.

Finally, we were unable to run HOMI [87]. The source code of the tool used in this paper is available, but no instruction is provided. After close observation, the bash file of `run.sh` was decided to be the file to run this tool based on the names of files and the source code. This file can be run, and the information shows that the tool is running the genetic programming algorithm to improve its subjects. However, information about current iteration simply

[3]We cite in brackets the version of software we examined in detail.

Table 3: Papers with open source GI tools targeting non-functional properties of software

Year	Paper	Tool	Properties	Language
2011	[75] Genetic Programming for Shader Simplification	*unnamed*	GPU time	GLSL (C)
2014	[42] Genetically Improved CUDA C++ Software	GISMO	Execution time	CUDA (C++)
	[50] Improving 3D Medical Image Registration CUDA Software with Genetic Programming	GISMO	Execution time	CUDA (C++)
	[70] Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class	GISMO	Execution time	C
2015	[22] locoGP: Improving Performance by Genetic Programming Java Source Code	locoGP	Instructions executed	Java
	[48] Improving CUDA DNA Analysis Software with Genetic Programming	GISMO	Execution time	CUDA (C)
	[44] Optimizing Existing Software With Genetic Programming	GISMO	Execution time	C++
2016	[17] Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV	*unnamed*	Execution time	C++
	[87] HOMI: Searching Higher Order Mutants for Software Improvement	HOMI	Execution time	C
2017	[2] PYGGI: Python General Framework for Genetic Improvement	PYGGI	Execution time	Java
	[39] Improving SSE Parallel Code with Grow and Graft Genetic Programming	GGGP	Execution time	C
2018	[24] Challenges on applying genetic improvement in JavaScript using a high-performance computer	Optmizer	Execution time	JavaScript
	[51] Evolving Better Software Parameters	*unnamed*	Function accuracy	C
	[71] Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation	GISMO	Execution time	C++
2019	[1] PyGGI 2.0: Language Independent Genetic Improvement Framework	PyGGI	Execution time	C++
	[14] Gin: Genetic Improvement Research Made Easy	Gin	Execution time	Java
	[27] Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs	PowerGAUGE	Energy consumption and accuracy	C/C++
	[49] Evolving AVX512 Parallel C Code using GP	GISMO	Execution time	C
	[60] Genetic Improvement of GPU Code	GEVO	Execution time and accuracy	C++
	[62] Applying genetic improvement to a genetic programming library in C++	GISMO	Execution time and accuracy	C++
	[68] Software Improvement with Gin: A Case Study	Gin	Execution time	Java
2020	[11] Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement Case Study: Improvement of MiniSAT's Running Time	PyGGI	CPU instructions	C++
	[15] Injecting Shortcuts for Faster Running Java Code	Gin	Execution time	Java
	[37] Automatically Evolving Lookup Tables for Function Approximation	*unnamed*	Function accuracy	C/C++/Java
	[41] Genetic Improvement of Genetic Programming	GISMO	Execution time	C++
	[45] Evolving sqrt into 1/x via software data maintenance	*unnamed*	Function accuracy	C
	[58] GEVO: GPU Code Optimization Using Evolutionary Computation	GEVO	Execution time	C++
2021	[12] Empirical Comparison of Search Heuristics for Genetic Improvement of Software	PyGGI	CPU instructions	C/C++/Java
	[25] Evolving JavaScript Code to Reduce Load Time	DFAHC	Code size	JavaScript
	[46] Genetic Improvement of Data for Maths Functions	*unnamed*	Function accuracy	C
	[64] Genetic Improvement of Routing Protocols for Delay Tolerant Networks	*unnamed*	Delivery probability	Java

Table 4: Papers with unavailable GI tools targeting non-functional properties of software

Year	Paper	Tool	Properties	Language
2008	[4] Multi-objective Improvement of Software Using Co-evolution and Smart Seeding	ECJ	CPU cycles	Java
2011	[85] Evolutionary Improvement of Programs	ECJ	Instructions executed	C
2013	[21] The Emergence of Useful Bias in Self-focusing Genetic Programming for Software Optimisation	*unnamed*	Instructions executed	Java
	[72] Applying Genetic Improvement to MiniSAT	GISMO	Execution time	C++
2015	[18] Reducing Energy Consumption Using Genetic Improvement	GISMO	Energy consumption	C
	[38] Removing the Kitchen Sink from Software	*unnamed*	Code size	C
	[43] Grow and Graft a Better CUDA pknotsRG for RNA Pseudoknot Free Energy Calculation	GISMO	Execution time	CUDA (C)
	[88] Deep Parameter Optimisation	*unnamed*	Execution time and memory usage	C
	[90] Embedding Adaptivity in Software Systems using the ECSELR framework	ECSELR	Code size	Java
2016	[35] Automatic Improvement of Apache Spark Queries using Semantics-preserving Program Reduction	Hylas	Execution time	Spark (SQL)
	[55] API-Constrained Genetic Improvement	GISMO	Execution time	C++
	[63] Genetic Programming: From Design to Improved Implementation	GISMO	Processing time	C++
	[65] A General-Purpose Framework for Genetic Improvement	μGP	Hash function size	C/Java/Python
2017	[7] Optimising Darwinian Data Structures on Google Guava	ARTEMIS	Execution time and memory usage	Java
	[13] Search-based energy optimization of some ubiquitous algorithms	Opacitor	Energy consumption	Java
	[32] Genetic Improvement of Runtime and its Fitness Landscape in a Bioinformatics Application	JM	Execution time	C/C++
	[31] The use of predictive models in dynamic treatment planning	JM	Accuracy	Python
	[47] Genetically improved BarraCUDA	GGGP	Execution time	CUDA (C)
	[77] Polytypic Genetic Programming	Polytope	Execution time	Scala
	[81] Trading between quality and non-functional properties of median filter in embedded systems	*unnamed*	Accuracy	C
	[91] Online Genetic Improvement on the Java virtual machine with ECSELR	ECSELR	Code size	Java
2018	[8] Darwinian Data Structure Selection	ARTEMIS	Execution time and memory usage	C++/Java
	[23] Investigating the Evolvability of Web Page Load Time	*unnamed*	Load time	JavaScript
	[30] Predicting changes in quality of life for patients in vocational rehabilitation	JM	Accuracy	Python
	[53] Evolving Better RNAfold Structure Prediction	GISMO	Accuracy	C
	[61] Novelty Search for Software Improvement of a SLAM System	GISMO	Execution time	C++
	[73] Evolving Exact Decompilation	BED	Readability	C
2019	[10] A comparison of tree- and line-oriented observational slicing	T-ORBS	Execution time	C/C++/Java
	[19] Approximate Oracles and Synergy in Software Energy Search Spaces	GISMO	Energy consumption	C/C++
	[52] Genetic improvement of data gives binary logarithm from sqrt	*unnamed*	Function accuracy	C
	[40] Genetic Improvement of Data gives double precision invsqrt	*unnamed*	Function accuracy	C
2020	[59] GEVO-ML: a proposal for optimizing ML code with evolutionary computation	GEVO-ML	Execution time and accuracy	CUDA (C)

returns the following error: `could not find or load main class executable.Evolve`. It is difficult to determine the reason for this error because without proper documentation we do not even know whether the environmental settings are correct for this tool.

5.4 Summary

Section 4 exposed 63 papers that used GI tools to improve software's NFPs. Upon investigation, we found a total of 13 distinct GI tools with open-source code, 11 of which deemed possible to be run on other software. Further 3 had to be excluded due to lack of required hardware, software dependencies, and required documentation details. Ultimately, we were able to successfully run 8 GI tools.

Answer to RQ2: From our literature review we found 13 distinct GI tools that target software's NFPs, 8 of which we were able to run without any issues.

6 GENERALISABILITY STUDY

Section 5 revealed eight working GI tools. To answer RQ3, these tools were investigated in more detail to examine how they work and how easily they can be applied to other software.

6.1 Methodology

We apply a cross-testing strategy to check whether the tools can work on different software. More precisely, we applied each tool to a new software, chosen to have been previously targeted by another similar tool. This strategy ensures that targeted software are not chosen blindly, and that there are known improvements to be found for all tools.

Because of the significant amount of time required to undertake complete GI experimentation, we only checked whether the tools can be run on different software. This means that GI runs were terminated early as soon as it could be decided that the tool can successfully run. Therefore, the records and results of the tools were not analysed.

In particular, we conducted the following experiments[4]:

(1) Testing Gin with SAT4J, which is the software improved by PyGGI 2.0 in previous work [12].
(2) Testing PyGGI 2.0 with Gson, which is the software improved by Gin in previous work [68].
(3) Testing LocoGP with Gson.
(4) Testing the tool for shader simplification with MiniSAT, which is the software used in previous work on a GISMO-based tool [70].
(5) Testing the GISMO-based tool with RNAfold, which is the software improved by GGGP in previous work [39].
(6) Testing the tool for OpenCV with MiniSAT.
(7) Testing GGGP with MiniSAT.
(8) Testing PowerGAUGE with MiniSAT.

6.2 Evaluation of Gin

Gin[5] is a well-developed tool. It can operate on the source code of the target software at both line-level and AST-level. Unlike other GI work that focuses on NFP improvement, it targets methods, rather than entire class files for improvement. Furthermore, Gin supports both local search and genetic programming algorithms for genetic improvement. By default, it improves runtime, though program repair, and memory consumption can be improved using the latest version of Gin.

The experiment of running Gin on SAT4J was successful. It was possible to run the profiling function of Gin on SAT4J (which finds the most time-consuming methods in a given project), although the processing was time-consuming and was stopped in 2 minutes when we found this function works. Also, we provided a simulated profiling result, and Gin worked with that file to try to improve SAT4J. This processing was also stopped after 2 minutes because we believe this tool is likely to work fine for SAT4J.

As to the available instructions of the tool, we find that the documentation of Gin is one of the best ones among all the available tools in this section. Several frequently used commands are presented in the README instruction, and an example case of applying Gin on spatial4j is provided for reference. Moreover, all the commands used in this research are well explained, meaning that it is easy to get the explanation of the arguments for this command and the detailed information about what the arguments are for.

Moreover, Gin works well with Maven and Gradle. It can automatically find the classpath for these projects, saving time for the settings of the subjects. Also, It has the profiling function, which is very useful for finding the methods used frequently in the projects and helps users decide which methods to improve. Gin also provides a function to automatically validate generated patches on a given test suite.

However, Gin has some limitations. Firstly, as the developers indicated in the README instruction, the documentation of this tool is not complete, which may cause trouble when users intend to use it for projects that are not in Maven or Gradle. Also, because of the limitation of the GI research circle, there are not many developers who can help with the maintenance of Gin. We found some issues from three years ago that were still not solved.

In summary, Gin is the GI tool with the most detailed documentation among all the tools we investigated. Also, it provides relatively complete functions for Maven and Gradle projects, making it easy to use. We thus conclude it can be easily used for different projects. We flag the need for better maintainance of the tool.

6.3 Evaluation of PyGGI

PyGGI[6] can operate on the target software's source code at both line-level or AST-level. Unlike Gin, which only works for Java, PyGGI can target software written in Python, C, Java, and others. It provides only a local search algorithm (although variants of it in later research also implement GP). PyGGI requires users to provide a script to execute the target software and its variants. The default fitness is the execution time of the software but can also be manually provided. The documentation of PyGGI also provides information on how to run it for the purpose of program repair.

The experiment on Gson was successful. We modified the example code of `improve_java.py`, which is the file to define the software to be improved, and `TestRunner.java`, which is the file to define how the patches are to be validated in PyGGI. Since PyGGI does not have a profiling tool to find the frequently used Java files, we use the same file of `GsonBuilder.java` in Gson and define the test file as `GsonBuilderTest.java`. We do not validate the generated patches with the complete test files because this experiment only tests whether PyGGI can work on other software. Therefore, we only test the patches with one test file, which saves the time to modify the source code in `TestRunner.java`. The improvement is terminated after several iterations, and the result shows that PyGGI can be applied to other software.

The validation procedure of PyGGI is easy to modify for different software. Although PyGGI cannot automatically find the tests and execute them to validate the generated patches, users can understand the code of validation definition in the provided example and choose their preferred validation method. Taking Gson as an example; users can choose different ways to validate the patches, such as using `mvn test` or using JUnit to run the tests chosen by

[4]We frequently chose MiniSAT, if not previously improved by a given GI tool, due to quick setup required.
[5]https://github.com/gintool/gin
[6]https://github.com/coinse/pyggi

users. PyGGI only requires the result of the validation, which is represented in the format of "true/false" and the execution time of the tests in milliseconds.

However, PyGGI does not have the tool for analysing the patches recorded in the log file. Only the changes on the source code of the best patch in each epoch are shown in the terminal output, and this only happens when the best patch has a better fitness score than the original code.

Also, the documentation of PyGGI is not as detailed as that of Gin, but it is unlikely to cause serious difficulty for using the tool because the source code of PyGGI is straightforward and easy to understand.

In summary, PyGGI is an easy-to-use tool and can be used for different software, written in different programming languages. Better documentation and further development of the patch analyser are suggested.

6.4 Evaluation of LocoGP

LocoGP[7] modifies the source code to the AST representation and applies the genetic programming algorithm to improve a given Java program. The evaluation of the performance of the modified code relies on the number of instructions used in execution to calculate the fitness score.

However, this tool is unlikely to be used for large projects such as Gson. There is no instruction on how to apply this tool to general software, meaning that we have to refer to the example file of `Ascon128V11DecryptProblem.java` to modify the tool for Gson. After close inspection, we find that this tool requires users to complete considerable programming work to define the subject to be improved. In this experiment, we selected `GsonBuilder.java` for improvement and had to modify eleven functions in the example file to make this example file suitable for Gson.

Moreover, tests defined in Gson cannot be used directly in this tool. This tool requires that all tests should be defined in classes. A list of all test classes for validation is also required, which helps the tool retrieve information, including the results of tests and the number of cases. Moreover, this tool does not rely on JUnit to execute the test cases but applies the simple method to directly check whether the output is consistent with the expected one. It means that the original tests written with JUnit are no longer helpful for this tool, and all the testing files have to be re-written for this tool.

Therefore, the workload for using the tool for Gson is highly significant. We have to define the improvement by imitating the example code and re-define thousands of tests to make them available for this tool. Because of the unacceptable preparation work to use this tool, this tool is not tested with Gson.

In summary, the usability of this tool is not satisfactory, especially for large-scale Java projects. Users have to modify a significant amount of the source code to define the software to be improved and create new test classes for the software.

[7] https://github.com/codykenb/locoGP

6.5 Evaluation of the tool for shader simplification

This tool (reimplementation[8]) modifies software source code at the AST level. It uses genetic programming to improve the target software. As for patch fitness it relies on both rendering time and error. The tool is easy to use as is does not require dependency installation.

This tool is very unlikely to improve MiniSAT. After close observation of the source code, this tool is likely only useful for the shader software. For the core Python files determining how this tool behaves, they are designed for shader simplification only. For the validation, the relevant files are “evaluator.py" and “fresnel.py". In these files, the improved AST is transformed into the OpenGL Shading Language, which is the language used in shader programs, and evaluated the improved code by rendering the new code of the shader. Therefore, necessary modification is required if this tool is planned to be used for different software.

Meanwhile, the files used for genetic programming are also closely related to the shader software. The `generate_individual` function in the `gp.py` file requires a shader variant as an input, meaning that this genetic programming algorithm is unlikely to work on software like MiniSAT. Furthermore, as its name shows, the `shader.py` file is mainly about representing the operators in the shader programs and converting shader programs to the genetic programming tree, which is unlikely to be suitable for software not related to a shader. Therefore, it can be concluded that the core algorithms used in this tool are targeted to improve shader software only.

Since the tool is specifically designed for shader software and the source code is targeted at shader programs, it is not easy to re-write the code for other kinds of software. The workload can be significant because all the files in this tool have to be reviewed and modified. Therefore, MiniSAT is not tested with this tool.

In summary, although the tool for shader simplification is easy to use, it is likely to work with shader software only. Significant modification is required for the core algorithms and validation code if users intend to use this tool for other kinds of software.

6.6 Evaluation of the GISMO-based tool

This tool[9] uses a BNF grammar to represent the target software's source code. It implements a genetic programming algorithm to improve the fitness of software variants. For the evaluation of the generated patches, this tool inserts and uses a counter incremented at each executed statement, however, execution time is also presented in the final result.

The experiment of running the GISMO-based tool on RNAfold is not successful because the tool fails to find the program to be improved in BNF format. There is no general instruction provided on how to apply this tool to different software, and we did not find any comment in the source code that may contribute to the modification. Therefore, we fail to find a way to transfer the source code of RNAfold into the BNF format. Moreover, it can still be challenging to modify the code even if the source code is in BNF format due to the lack of direction.

[8] https://github.com/fabianishere/shadevolution
[9] http://www0.cs.ucl.ac.uk/staff/W.Langdon/gismo

However, we find that the GISMO framework has been used for different software in other research, such as the work conducted by Langdon [41]. Therefore, it is very likely that this framework can work on different software, but we fail to make it because of the poor documentation and missing instruction about how to use this tool for general software.

In summary, the GISMO-based tool failed to improve RNAfold because of the missing method to generate the BNF representation and the lack of necessary instructions for applying this tool on different software. It is suggested that the developers of the tool should improve the documentation and conduct essential maintenance.

6.7 Evaluation of the tool for OpenCV

This tool[10] uses deep parameter optimisation [88] to form a more extensive search space for optimisation and tuning the found parameters with the NSGA-II algorithm. Modifications of the source code are determined at the level of lines, and the generated patches are evaluated in terms of execution time and accuracy.

However, the testing with MiniSAT failed. In order to understand the failure better, the previous publication [17] is reviewed. As the paper indicates, three steps are required while using the tool:

(1) finding the location of the deep parameters;
(2) exposing the deep parameters;
(3) tuning the parameters.

The error occurred in the second step. We learn from the example provided with the tool that the `replace.hpp` file, which defines integer constants, is required as an input for this step to expose the deep parameters in the files found in Step 1. However, there was no such file defining the integer constants for MiniSAT. Although we intended to solve the problem, we did not find any information about how to generate this file from the paper and the README instruction.

Therefore, applying the tool for MiniSAT is unlikely to be completed because of the error in exposing the deep parameters. However, it is possible that this tool can be used in other similar projects, especially the ones containing a large number of parameters if the step of defining parameter constants can be well explained or automated. However, the performance of this tool can be determined mainly by the number of deep parameters in the software to be improved.

6.8 Evaluation of GGGP

Similarly to GISMO, from the same author, this tool[11] represents the source code of the software using a BNF grammar and implements a genetic programming algorithm. For the evaluation of the generated patches, this tool relies on the test cases and uses the execution time of the tests as the fitness score.

However, we were unable to use this tool to improve MiniSAT. A README file is present but provides no instruction on how to apply this tool to another program. We tried but ultimately were unsuccessful in modifying the RNAfold example to accommodate MiniSAT instead. More precisely, we were unable to fix errors pertaining to the `RE_gp.bat` file. Inspection of the source code was also unhelpful as the code includes no comments.

Therefore, unless additional documentation can be provided, this tool is not likely to be used on other software.

6.9 Evaluation of PowerGAUGE

This tool[12] harnesses the GenProg [57] software[13] to apply a genetic programming algorithm to the targeted software. Fitness computation is to be manually provided, although code samples provide examples for execution time and output-related fitness functions.

We were unable to use this tool on MiniSAT. PowerGAUGE evolves and thus requires access to assembly files for the targeted software. Whilst those may generally not be hard to obtain, in practice it could require rewriting the entire compilation pipeline, which even in the case of the fairly simple MiniSAT was unreasonable.

Overall, it is unlikely that this tool can easily be applied to other, especially complex, software.

6.10 Summary

We tried to assess the generalisability of all eight GI tools we could run by applying them to new software. For only two tools, Gin and PyGGI, we were able to find adequate documentation to do so. Ultimately we were unsuccessful in making any of the six other tools work on new software. We note that some tools target specialist software (e.g., shaders), and thus cannot be easily applied on general software.

Answer to RQ3: Gin and PyGGI are the only two GI tools are application-agnostic and can be easily applied to improve new software.

7 CONCLUSION

In this paper we investigated genetic improvement (GI) tooling for improvement of non-functional properties (NFP) of software. More precisely, we focused on the available GI tools described in the literature, whether they were available, whether they were actually usable, and whether they could easily be applied to software, to which they have not been applied to in previous work.

In the survey, we found 63 relevant papers, within which 31 come with associated open-source code. The usability study exposed 11 different general GI tools, but only 8 that we were able to run without any issues. Furthermore, the generalisability study ultimately showed that within these eight GI tools only two — Gin and PyGGI — can be readily applied to new software for improvement of non-functional properties of software. We recommend addition of more detailed documentation and better maintenance of current GI tooling.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 1100–1104.

[10] https://github.com/BobbyRBruce/DPT-OpenCV
[11] http://www0.cs.ucl.ac.uk/staff/ucacbbl/gggp
[12] https://github.com/dornja/powergauge
[13] https://github.com/squaresLab/genprog-code

[2] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 2017. PyGGI: Python General framework for Genetic Improvement. In *Proceedings of the Korea Software Congress (KSC 2017)*. 536–538.

[3] Andrea Arcuri. 2009. *Automatic software generation and improvement through search based techniques*. Ph. D. Dissertation. University of Birmingham, UK.

[4] Andrea Arcuri, David Robert White, John A. Clark, and Xin Yao. 2008. Multi-objective Improvement of Software Using Co-evolution and Smart Seeding. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL) (LNCS, Vol. 5361)*. Springer, 61–70.

[5] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 257–269.

[6] Michail Basios. 2019. *Darwinian Code Optimisation*. Ph. D. Dissertation. University College London, UK.

[7] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2017. Optimising Darwinian Data Structures on Google Guava. In *Proceedings of the 9th International Symposium on Search Based Software Engineering (SSBSE 2017) (LNCS, Vol. 10452)*. Springer, 161–167.

[8] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2018. Darwinian data structure selection. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 118–128.

[9] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE 2020)*. IEEE, 907–918.

[10] David W. Binkley, Nicolas Gold, Syed S. Islam, Jens Krinke, and Shin Yoo. 2019. A comparison of tree- and line-oriented observational slicing. *Empirical Software Engineering* 24, 5 (2019), 3077–3113.

[11] Aymeric Blot and Justyna Petke. 2020. Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement – Case Study: Improvement of MiniSAT's Running Time. In *Proceedings of the 23th European Conference on Genetic Programming (EuroGP 2020) (LNCS, Vol. 12101)*. Springer, 68–83.

[12] Aymeric Blot and Justyna Petke. 2021. Empirical Comparison of Search Heuristics for Genetic Improvement of Software. *IEEE Transactions on Evolutionary Computation* 25, 5 (2021), 1001–1011.

[13] Alexander E. I. Brownlee, Nathan Burles, and Jerry Swan. 2017. Search-Based Energy Optimization of Some Ubiquitous Algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence* 1, 3 (2017), 188–201.

[14] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 2019. Gin: Genetic improvement research made easy. In *Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2019)*. ACM, 985–993.

[15] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. 2020. Injecting Shortcuts for Faster Running Java Code. In *Proceedings of the Congress on Evolutionary Computation (CEC 2020)*. IEEE, 1–8.

[16] Bobby R. Bruce. 2018. *The Blind Software Engineer – Improving the Non-Functional Properties of Software by Means of Genetic Improvement*. Ph. D. Dissertation. University College London, UK.

[17] Bobby R. Bruce, Jonathan M. Aitken, and Justyna Petke. 2016. Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV. In *Proceedings of the 8th International Symposium on Search Based Software Engineering (SSBSE 2016) (LNCS, Vol. 9962)*. Springer, 238–243.

[18] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement. In *Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015)*. ACM, 1327–1334.

[19] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. 2019. Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Transactions on Software Engineering* 45, 11 (2019), 1150–1169.

[20] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, 135–146.

[21] Brendan Cody-Kenny and Stephen Barrett. 2013. The Emergence of Useful Bias in Self-focusing Genetic Programming for Software Optimisation. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE 2013) (LNCS, Vol. 8084)*. Springer, 306–311.

[22] Brendan Cody-Kenny, Edgar Galvan Lopez, and Stephen Barrett. 2015. locoGP: Improving Performance by Genetic Programming Java Source Code. In *Companion Material Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015 companion)*. ACM, 811–818.

[23] Brendan Cody-Kenny, Umberto Manganiello, John Farrelly, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2018. Investigating the Evolvability of Web Page Load Time. In *Proceedings of the 21st International Conference on Applications of Evolutionary Computation (EvoApp 2018) (LNCS, Vol. 10784)*. Springer, 769–777.

[24] Fábio de Almeida Farzat, Márcio de Oliveira Barros, and Guilherme Horta Travassos. 2018. Challenges on applying genetic improvement in JavaScript using a high-performance computer. *Journal of Software Engineering Research and Development* 6 (2018), 12.

[25] Fábio de Almeida Farzat, Márcio de Oliveira Barros, and Guilherme H. Travassos. 2021. Evolving JavaScript Code to Reduce Load Time. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1544–1558.

[26] Darshan Domah and Frank J. Mitropoulos. 2015. The NERV methodology: A lightweight process for addressing non-functional requirements in agile software development. In *SoutheastCon 2015*. 1–7. https://doi.org/10.1109/SECON.2015.7133028

[27] Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. 2019. Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs. *IEEE Transactions on Software Engineering* 45, 3 (2019), 219–236.

[28] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 4th Genetic and Evolutionary Computation Conference (GECCO 2009)*. ACM, 947–954.

[29] Saemundur O. Haraldsson. 2017. *Genetic Improvement of Software – From Program Landscapes to the Automatic Improvement of a Live System*. Ph. D. Dissertation. University of Stirling, UK.

[30] Saemundur O. Haraldsson, Ragnheidur D. Brynjolfsdottir, Vilmundur Gudnason, Kristinn Tomasson, and Kristin Siggeirsdottir. 2018. Predicting changes in quality of life for patients in vocational rehabilitation. In *Proceedings of the Evolving and Adaptive Intelligent Systems (EAIS 2018)*. 1–8.

[31] Saemundur O. Haraldsson, Ragnheidur D. Brynjolfsdottir, John R. Woodward, Kristin Siggeirsdottir, and Vilmundur Gudnason. 2017. The use of predictive models in dynamic treatment planning. In *Proceedings of the Symposium on Computers and Communications (ISCC 2017)*. IEEE Computer Society, 242–247.

[32] Saemundur Oskar Haraldsson, John R. Woodward, Alexander E.I. Brownlee, Albert V. Smith, and Vilmundur Gudnason. 2017. Genetic Improvement of Runtime and its fitness landscape in a Bioinformatics Application. In *Companion Material Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO 2017 companion)*. ACM.

[33] Chih-Wei Ho, M.J. Johnson, L. Williams, and E.M. Maximilien. 2006. On agile performance requirements specification and testing. In *AGILE 2006*. 6 pp.–52. https://doi.org/10.1109/AGILE.2006.41

[34] Holger H. Hoos and Thomas Stützle. 2004. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann.

[35] Zoltan A. Kocsis, John H. Drak, Douglas Carson, and Jerry Swan. 2016. Automatic Improvement of Apache Spark Queries using Semantics-preserving Program Reduction. In *Companion Material Proceedings of the 11th Genetic and Evolutionary Computation Conference (GECCO 2016 companion)*. ACM, 1141–1146.

[36] John R. Koza. 1992. *Genetic programming*. MIT Press.

[37] Oliver Krauss and William B. Langdon. 2020. Automatically Evolving Lookup Tables for Function Approximation. In *Proceedings of the 23th European Conference on Genetic Programming (EuroGP 2020) (LNCS, Vol. 12101)*. Springer, 84–100.

[38] Jason Landsborough, Stephen Harding, and Sunny Fugate. 2015. Removing the Kitchen Sink from Software. In *Companion Material Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015 companion)*. ACM.

[39] W. Langdon and R. Lorenz. 2017. Improving SSE Parallel Code with Grow and Graft Genetic Programming. In *Companion Material Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO 2017 companion)*. ACM, 1537–1538.

[40] William B. Langdon. 2019. Genetic Improvement of Data gives double precision invsqrt. In *Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2019 companion)*. ACM, 1709–1714.

[41] William B. Langdon. 2020. Genetic Improvement of Genetic Programming. In *Proceedings of the Congress on Evolutionary Computation (CEC 2020)*. IEEE, 1–8.

[42] William B. Langdon and Mark Harman. 2014. Genetically Improved CUDA C++ Software. In *Proceedings of the 17th European Conference on Genetic Programming (EuroGP 2014) (LNCS, Vol. 8599)*. Springer, 87–99.

[43] William B. Langdon and Mark Harman. 2015. Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In *Companion Material Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015 companion)*. ACM, 805–810.

[44] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (2015), 118–135.

[45] William B. Langdon and Oliver Krauss. 2020. Evolving sqrt into 1/x via software data maintenance. In *Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2020 companion)*. ACM, 1928–1936.

[46] William B. Langdon and Oliver Krauss. 2021. Genetic Improvement of Data for Maths Functions. *ACM Transactions on Evolutionary Learning and Optimization* 1, 2 (2021), 7:1–7:30.

[47] William B. Langdon and Brian Yee Hong Lam. 2017. Genetically improved BarraCUDA. *BioData Mining* 10, 1 (2017), 28:1–28:11.

[48] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015)*. ACM, 1063–1070.

[49] William B. Langdon and Ronny Lorenz. 2019. Evolving AVX512 Parallel C Code Using GP. In *Proceedings of the 22nd European Conference on Genetic Programming (EuroGP 2019) (LNCS, Vol. 11451)*. Springer, 245–261.

[50] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. 2014. Improving 3D medical image registration CUDA software with genetic programming. In *Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO 2014)*. ACM, 951–958.

[51] William B. Langdon and Justyna Petke. 2018. Evolving Better Software Parameters. In *Proceedings of the 10th International Symposium on Search Based Software Engineering (SSBSE 2018) (LNCS, Vol. 11036)*. Springer, 363–369.

[52] William B. Langdon and Justyna Petke. 2019. Genetic improvement of data gives binary logarithm from sqrt. In *Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2019 companion)*. ACM, 413–414.

[53] William B. Langdon, Justyna Petke, and Ronny Lorenz. 2018. Evolving Better RNAfold Structure Prediction. In *Proceedings of the 21st European Conference on Genetic Programming (EuroGP 2018) (LNCS, Vol. 10781)*. Springer, 220–236.

[54] William B. Langdon and Riccardo Poli. 2002. *Foundations of genetic programming*. Springer.

[55] William B. Langdon, David Robert White, Mark Harman, Yue Jia, and Justyna Petke. 2016. API-Constrained Genetic Improvement. In *Proceedings of the 8th International Symposium on Search Based Software Engineering (SSBSE 2016) (LNCS, Vol. 9962)*. Springer, 224–230.

[56] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE, 3–13.

[57] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[58] Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020. GEVO: GPU Code Optimization Using Evolutionary Computation. *ACM Transactions on Architecture and Code Optimization* 17, 4 (2020), 33:1–33:28.

[59] Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020. GEVO-ML: a proposal for optimizing ML code with evolutionary computation. In *Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2020 companion)*. ACM, 1849–1856.

[60] Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019. Genetic Improvement of GPU Code. In *Proceedings of the 6th International Workshop on Genetic Improvement (GI@ICSE 2019)*. ACM, 20–27.

[61] Víctor R. López-López, Leonardo Trujillo, and Pierrick Legrand. 2018. Novelty Search for Software Improvement of a SLAM system. In *Companion Material Proceedings of the 13th Genetic and Evolutionary Computation Conference (GECCO 2018 companion)*. ACM.

[62] Víctor R. López-López, Leonardo Trujillo, and Pierrick Legrand. 2019. Applying genetic improvement to a genetic programming library in C++. *Soft Computing* 23, 22 (2019), 11593–11609.

[63] Víctor R. López-López, Leonardo Trujillo, Pierrick Legrand, and Gustavo Olague. 2016. Genetic Programming: From design to improved implementation. In *Companion Material Proceedings of the 11th Genetic and Evolutionary Computation Conference (GECCO 2016 companion)*. ACM, 1147–1154.

[64] Michela Lorandi, Leonardo Lucio Custode, and Giovanni Iacca. 2021. Genetic Improvement of Routing Protocols for Delay Tolerant Networks. *ACM Transactions on Evolutionary Learning and Optimization* 1, 1 (2021), 4:1–4:37.

[65] Francesco Marino, Giovanni Squillero, and Alberto Paolo Tonda. 2016. A General-Purpose Framework for Genetic Improvement. In *Proceedings of the 14th International Conference on Parallel Problem Solving from Nature (PPSN XIV) (LNCS)*. Springer, 345–352.

[66] Hiroki Nakajima, Yoshiki Higo, Haruki Yokoyama, and Shinji Kusumoto. 2016. Toward Developer-like Automated Program Repair – Modification Comparisons between GenProg and Developers. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016)*. IEEE Computer Society, 241–248.

[67] Justyna Petke, Brad Alexander, Earl T. Barr, Alexander E.I. Brownlee, Markus Wagner, and David R. White. 2019. A Survey of Genetic Improvement Search Spaces. In *Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2019 companion)*. ACM, 1715–1721.

[68] Justyna Petke and Alexander E. I. Brownlee. 2019. Software Improvement with Gin: A Case Study. In *Proceedings of the 11th International Symposium on Search Based Software Engineering (SSBSE 2019) (LNCS, Vol. 11664)*. Springer, 183–189.

[69] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432.

[70] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. 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 (EuroGP 2014) (LNCS, Vol. 8599)*. Springer, 137–149.

[71] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2018. Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation. *IEEE Transactions on Software Engineering* 44, 6 (2018), 574–594.

[72] Justyna Petke, William B. Langdon, and Mark Harman. 2013. Applying Genetic Improvement to MiniSAT. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE 2013) (LNCS, Vol. 8084)*. Springer, 257–262.

[73] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving Exact Decompilation. In *Workshop on Binary Analysis Research (BAR@NDSS 2018)*.

[74] Eric Schulte, Westley Weimer, and Stephanie Forrest. 2015. Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair. In *Companion Material Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015 companion)*. ACM.

[75] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic Programming for Shader Simplification. *ACM Transactions on Graphics* 30, 6 (2011), 152.

[76] Marta Smigielska, Aymeric Blot, and Justyna Petke. 2021. Uniform Edit Selection for Genetic Improvement: Empirical Analysis of Mutation Operator Efficacy. In *Proceedings of the 43rd International Conference on Software Engineering: Workshops (ICSE 2021 Workshops)*. ACM, 1–8.

[77] Jerry Swan, Krzysztof Krawiec, and Neil Ghani. 2017. Polytypic Genetic Programming. In *Proceedings of the 20th European Conference on Applications of Evolutionary Computation (EvoApp 2017) (LNCS, Vol. 10200)*. Springer, 66–81.

[78] Richard Taylor, Nenad Medvidovic, and Eric. Dashofy. 2009. Designing for Non-Functional Properties. In *Software Architecture: Foundations, Theory, and Practice*. Wiley, Chapter 12.

[79] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *Proceedings of the 9th International Symposium on Search Based Software Engineering (SSBSE 2017) (LNCS, Vol. 10452)*. Springer, 99–114.

[80] Mahrukh Umar and Naeem Ahmed Khan. 2011. Analyzing Non-Functional Requirements (NFRs) for software development. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. 675–678. https://doi.org/10.1109/ICSESS.2011.5982328

[81] Zdenek Vasícek and Vojtech Mrazek. 2017. Trading between quality and non-functional properties of median filter in embedded systems. *Genetic Programming and Evolvable Machines* 18, 1 (2017), 45–82.

[82] Westley Weimer. 2006. Patches as better bug reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006)*. ACM, 181–190.

[83] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE, 364–374.

[84] David Robert White. 2009. *Genetic Programming for Low-Resource Systems*. Ph. D. Dissertation. University of York, UK.

[85] David R. White, Andrea Arcuri, and John A. Clark. 2011. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation* 15, 4 (2011), 515–538.

[86] Fan Wu. 2017. *Mutation-Based Genetic Improvement of Software*. Ph. D. Dissertation. University College London, UK.

[87] Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. 2016. HOMI: Searching Higher Order Mutants for Software Improvement. In *Proceedings of the 8th International Symposium on Search Based Software Engineering (SSBSE 2016) (LNCS, Vol. 9962)*. Springer, 18–33.

[88] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep Parameter Optimisation. In *Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015)*. ACM, 1375–1382.

[89] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 20.

[90] Kwaku Yeboah-Antwi and Benoit Baudry. 2015. Embedding Adaptivity in Software Systems using the ECSELR framework. In *Companion Material Proceedings of the 10th Genetic and Evolutionary Computation Conference (GECCO 2015 companion)*. ACM, 839–844.

[91] Kwaku Yeboah-Antwi and Benoit Baudry. 2017. Online Genetic Improvement on the java virtual machine with ECSELR. *Genetic Programming and Evolvable Machines* 18, 1 (2017), 83–109.