A General-Purpose Framework for Genetic Improvement

Francesco Marino[1], Giovanni Squillero[1], and Alberto Tonda[2(✉)]

[1] Politecnico di Torino, Corso Duca Degli Abruzzi 24, 10129 Torino, Italy
francesco.marino@studenti.polito.it, giovanni.squillero@polito.it
[2] UMR GMPA, AgroParisTech, INRA, Université Paris-Saclay, 1 Av. Brétignières, 78850 Thiverval-Grignon, France
alberto.tonda@grignon.inra.fr

Abstract. Genetic Improvement is an evolutionary-based technique. Despite its relatively recent introduction, several successful applications have been already reported in the scientific literature: it has been demonstrated able to modify the code complex programs without modifying their intended behavior; to increase performance with regards to speed, energy consumption or memory use. Some results suggest that it could be also used to correct bugs, restoring the software's intended functionalities. Given the novelty of the technique, however, instances of Genetic Improvement so far rely upon ad-hoc, language-specific implementations. In this paper, we propose a general framework based on the software engineering's idea of mutation testing coupled with Genetic Programming, that can be easily adapted to different programming languages and objective. In a preliminary evaluation, the framework efficiently optimizes the code of the md5 hash function in C, Java, and Python.

Keywords: Genetic improvement · Genetic programming · Linear genetic programming · Software engineering

1 Introduction

The term "genetic improvement" has been commonly used to denote the science of applying genetic, breeding principles and biotechnology to *improve* plants and animals, that is, to maximize the expression of their genetic potential making them more productive for human use. While such techniques are dated back to 1700 s, the very same term has been quite recently renovated in a completely different context: computer science. Nowadays among evolutionary computation scholars, Genetic Improvement (GI) denotes the application of evolutionary, search-based optimization methods to the improvement of existing software.

The hope to automatically improve, let alone create, software has been a driving force of evolutionary computation. In 1992, John Koza asked "how can computers be made to do what is needed to be done, without being told exactly how to do it?", then tried to answer the question by introducing the paradigm of Genetic Programming (GP) [1]. Despite an unquestionable series of successes,

J. Handl et al. (Eds.): PPSN XIV 2016, LNCS 9921, pp. 345–352, 2016.
DOI: 10.1007/978-3-319-45823-6_32

GP cannot be used to evolve from scratch a full *computer program* able to solve a generic problem, yet.

With the more recent GI, practitioners are tackling an apparently easier problem: given the code of an existing program, GI strives to tweak it in order to reach a specific goal, such as: improve speed, reduce memory usage, reduce code length, remove bugs, etc. Such a technique raised the interest of the scientific community, and several works on the subject have appeared in literature, in the last few years [2–4]. Despite the interest, so far each application of GI has been developed in-house by researchers, with solutions designed for specific problems and computing languages, with little to none code re-usability.

In this paper, we propose a general-purpose framework for GI, able to tackle problems in any computer language and with user-definable goal. The approach exploits an existing general-purpose evolutionary algorithm (EA), and is tested on a simple but challenging test case, the md5 hash function. For three different languages, C, Java, and Python, the proposed methodology is proven able to reduce the code size of the function without introducing errors, given a target number of items for which to generate a hash value.

2 Background

2.1 Genetic Improvement

GI was introduced as a technique able to automatically modify the source code of existing software, optimizing its performance with regards to user-defined metrics [2]. GI was originally based on Genetic Programming (GP) [1]: individuals are encoded as linear graphs, each one representing a series of permutations on the code, ranging from commenting blocks, to swapping two lines, to change the initialization of a variable. Even if the fitness evaluation is specifically tailored for each application, the general idea is always to improve code behavior with respect to one or more objectives, all the while maintaining the software's functionalities.

GI has been successfully applied to different case studies, in order to decrease energy consumption [3], improve speed [2], specialize a program to optimize some specific functions [4], and minimize memory usage [5], respectively. Recent results, aiming at repairing the firmware of a router, prove that GI is also able to act on compiled code, correcting bugs without direct access to the source or to test suites [6]. The rising interest of the evolutionary research community for the topic culminated in a first workshop on the subject, organized during the GECCO conference in 2015[1].

As it is common for techniques in the early stages of research, case studies of GI use ad hoc tools, usually developed from scratch for the specific application. Now that GI is getting more and more adopted, a general-purpose framework could be extremely helpful to practitioners and researchers alike, speeding up prototyping and development of new ideas.

[1] http://geneticimprovementsoftware.com/.

2.2 μGP

μGP (MicroGP) [7,8] is an evolutionary toolkit. Originally devised to evolve assembly-language programs for test program generation [9], it was later expanded to a general-purpose open-source project[2] and exploited for several applications, ranging from Bayesian network structure learning [10] to analyzing the behavior of wireless network routing protocols [11], from adapting the number of cards in reactive pull systems [12] to the detection of power-related software errors in industrial verification processes [13].

What makes μGP suitable to tackle such a wide range of diverse problems is its design, based on a distinct separation between the description of individuals in the target application, the evolutionary core, and the fitness evaluator. In essence, the framework evolves a set of linear directed graphs, where each node represents a macro, that can in turn present several parameters. The description of the macros is specified by the user through a configuration file. When individuals are evaluated, the macros in each node are converted to text, and the resulting file is passed to a user-designed evaluator program. For a high-level depiction of the framework, see Fig. 1.

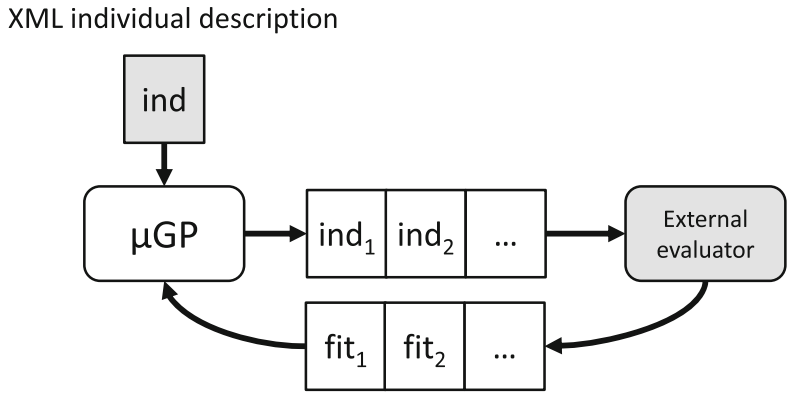

Fig. 1. High-level structure of the μGP toolkit. In order to prepare the framework to tackle a new application, only the parts in gray (XML description file and external evaluator) need to be modified by the user.

3 Proposed Approach

Given the rising interest around applications of GI, we propose a general-purpose framework, to ease prototyping and development. The framework is based on the μGP evolutionary toolkit, and can be quickly adapted to new GI applications, across different languages, without any need to recompile the source code, simply acting on configuration files and fitness evaluation.

[2] μGP is hosted on SourceForge http://ugp3.sourceforge.net/.

Table 1. Set of MT operations selected for the proposed framework.

Short-code	Description
CAR	Arithmetic operator replacement
CAS	Assignment operator replacement
CBI	Bitwise operator replacement
CCO	Logic connector replacement
CLO	Logic operator replacement
CST	Constant value replacement
CUN	Unary operator replacement
DEL	Statement deletion

An individual encodes a sequence of operations to be performed on the target code. Such operations are inspired by Mutation Testing (MT), a technique devised in the early 1970s to evaluate the quality of a test suite [14]. The basic idea is to slightly *mutate* a program, emulating developers' errors. All such mutants are eventually used to assess the effectiveness of a test suite in discriminating bug-free software.

As for GI, most approaches exploiting MT are either problem or language specific. However, being a well-established technique, one can find in literature lists of mutation operators that can be applied to programs [15–20].

Our approach exploits the possibility to mutate a source program. We select a compact list of standard MT operations that are both general and relevant for all languages. It is important to notice that some MT functions have the clear purpose of causing a fault, and they have not been considered here. Table 1 shows the set of selected operations.

The proposed approach is summarized in Fig. 2.

3.1 Evolutionary Core

An individual is a variable-length sequence of modifications. Each modification is encoded as an operation (see Table 1) and one or more operands. Possible operands, e.g., the list of used operators or the list of used constants, are precomputed with static analysis.

The evolutionary core is the out-of-the-box μGP. μGP mutates and recombines individuals using classical genetic operators. In more details: an operation may be substituted with another operation, or its operands changed. Two individuals may be mixed using one-cut, two-cut, and uniform crossover operators.

Since the list of operations to be performed on the code is not language-specific, a parser is required to translate the generic high-level operations to language-specific ones. After the modified program has been generated, it is then tested on a set of test cases to ensure that the features are maintained and to evaluate the quality of the improvement reached, as for standard GI procedure.

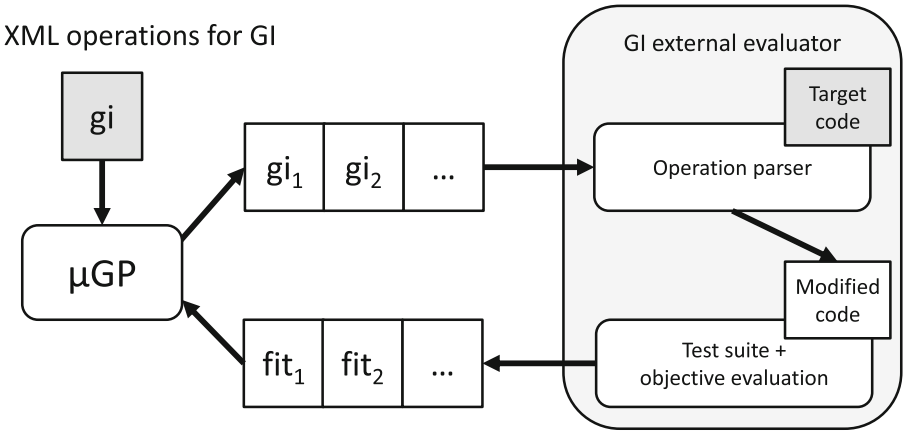

Fig. 2. Structure of the proposed approach. The μGP toolkit is used to generate a sequence of operations to be performed on the target code. The resulting modified program is then run through a test suite, in order to assess its functionality, and then evaluated with regards to a user-defined metric, for example speed, memory usage, etc.

4 Experimental Evaluation

In order to assess the suitability of our approach, we test the proposed framework on the MD5 function [21], a small, yet paradigmatic, case study. The MD5 message-digest algorithm is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32-digit hexadecimal number. MD5 has been utilized in a wide variety of cryptographic applications and is also commonly used to verify data integrity despite the fact that it is now considered unsuitable for further cryptographic use.

The classical MD5 implementation computes the value starting from the key and performing a series of arithmetic and binary operations on it. The experiments aim at *improving* the classical MD5 by reducing its size while still guaranteeing zero collision of the generated hashes on different fixed input set.

The necessity to optimize a general algorithm applied in a reduced scenario is not uncommon. And calculating hashes in an embedded system, thus for a known and fixed number of keys is a plausible scenario.

Experiments tackle the same function implemented in three different languages: C, Java and Python. Each function was improved in four scenarios, for a different number of keys: 8, 256, 1,024, and 4,096. The parameters used by the μGP in all the experiments are shown in Table 2. The evolution was stopped after 50 generations with no improvement in the fitness of the best individual.

In total, twelve different experiments were executed, producing four improved versions of the original function, in each language. Every improved program showed a reduced size, while guaranteeing to produce no collisions over the set of keys. Each run has been repeated 10 times. Results are shown in Table 3.

Table 2. μGP parameters

Parameter	Value	Description
μ	30	Population size
ϵ	1	Elite size
λ	20	Genetic operators applied in each generation
α	0.8	Self-adaptation inertia
τ	2	Tournament size
M	300	Maximum number of generations
St	50	Steady-state threshold

Table 3. Hash function size reduction compared among languages and test suites (10 repetitions)

# keys	C	Java	Python
8	46 %	40 %	36 %
256	41 %	37 %	39 %
1,024	41 %	36 %	36 %
4,096	37 %	38 %	44 %

Although preliminary, results are interesting. In both C and Java the achieved improvement is larger when the key set is smaller, as expected. The less different inputs are used, the more lines can be removed. Indeed, it must be noted that the tool is able to *modify* the original programs, tweaking constants or changing operators, and not only removing lines. For the Python implementation, on the other hand, the *type* of the keys and not their simple number, is the most important element. Thus, the tool is able to improve the original MD5, but improvement are not directly connected with the size of the key set.

Table 4 reports the average computational resources required to run the experiments. The tool was executed on a i7 computer with 16 GB of RAM, using a Linux-based operating system. Column **CPU** shows the total time required

Table 4. Time elapsed in each experiment (10 repetitions)

# keys	CPU (h:mm)			Generations		
	C	Java	Python	C	Java	Python
8	2:06	4:20	3:10	162	54	50
256	2:22	4:26	5:09	300	69	147
1,024	2:03	6:25	6:26	300	72	206
4,096	3:50	3:58	5:56	248	53	185

to run μGP, the tool for applying changes, and the evaluator. Column **Generations** reports the average number of generations before a steady-state is reached.

5 Conclusions

Genetic Improvement (GI) is a recently presented evolutionary technique for software engineering, able to automatically modify the source code of a program, increasing its performance with regards to energy/memory consumption or speed of execution. While the methodology has been proven to be rather promising, all solutions found in literature are ad-hoc implementations, often devised from scratch for a specific application. In this paper, we presented a generic framework for GI, able to target different programming languages and different objectives, requiring only minor tweaking on the part of the user. The proposed approach is experimentally tested on simple case studies in Python, C++ and Java, and the results show that it is able to satisfactorily perform in all instances. Future works will focus on providing a Graphical User Interface for the framework, and releasing a full test set of benchmarks for GI, in different languages and for different objectives.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, vol. 1. MIT Press, Cambridge (1992)
2. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation (99) (2013)
3. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1327–1334. ACM (2015)
4. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., García-Sánchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K. (eds.) EuroGP 2014. LNCS, vol. 8599, pp. 137–149. Springer, Heidelberg (2014)
5. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1375–1382. ACM (2015)
6. Schulte, E.M., Weimer, W., Forrest, S.: Repairing COTS router firmware without access to source code or test suites: a case study in evolutionary software repair. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation. GECCO Companion 2015, pp. 847–854. ACM, New York (2015)
7. Squillero, G.: MicroGP - an evolutionary assembly program generator. Genet. Prog. Evol. Mach. **6**(3), 247–263 (2005)
8. Squillero, G.: Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs. Computing **93**(2–4), 103–120 (2011)

9. Corno, F., Sánchez, E., Squillero, G.: Evolving assembly programs: how games help microprocessor validation. IEEE Trans. Evol. Comput. **9**(6), 695–706 (2005)
10. Tonda, A., Lutton, E., Squillero, G., Wuillemin, P.-H.: A memetic approach to Bayesian network structure learning. In: Esparcia-Alcázar, A.I. (ed.) EvoApplications 2013. LNCS, vol. 7835, pp. 102–111. Springer, Heidelberg (2013)
11. Bucur, D., Iacca, G., Squillero, G., Tonda, A.: The impact of topology on energy consumption for collection tree protocols: an experimental assessment through evolutionary computation. Appl. Soft Comput. **16**, 210–222 (2014)
12. Belisário, L.S., Pierreval, H.: Using genetic programming and simulation to learn how to dynamically adapt the number of cards in reactive pull systems. Expert Syst. Appl. **42**(6), 3129–3141 (2015)
13. Gandini, S., Ruzzarin, W., Sanchez, E., Squillero, G., Tonda, A.: A framework for automated detection of power-related software errors in industrial verification processes. J. Electron. Test. **26**(6), 689–697 (2010)
14. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **4**, 34–41 (1978)
15. King, K.N., Offutt, A.J.: A FORTRAN language system for mutation-based software testing. Softw. Pract. Exp. **21**(7), 685–718 (1991)
16. Delamaro, M.E., Maldonado, J.C., Mathur, A.: Proteum-a tool for the assessment of test adequacy for C programs users guide. In: PCS, vol. 96, pp. 79–95 (1996)
17. Ma, Y.S., Offutt, J., Kwon, Y.R.: Mujava: an automated class mutation system. Softw. Test. Verif. Reliab. **15**(2), 97–133 (2005)
18. Derezińska, A., Rudnik, M.: Quality evaluation of object-oriented and standard mutation operators applied to C# programs. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 42–57. Springer, Heidelberg (2012)
19. Derezińska, A., Hałas, K.: Operators for mutation testing of python programs. Research report (2014)
20. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
21. Rivest, R.: The Md5 Message-digest Algorithm. Princeton, RFC (1992)