Synthetic Benchmarks for Genetic Improvement

Aymeric Blot, Justyna Petke

Synthetic Benchmarks for Genetic Improvement

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) uses automated search to find improved versions of existing software. If over the years the potential of many GI approaches have been demonstrated, the intrinsic cost of evaluating real-world software makes comparing these approaches in large-scale meta-analyses very expensive. We propose and describe a method to construct synthetic GI benchmarks, to circumvent this bottleneck and enable much faster quality assessment of GI approaches.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

Genetic improvement; Search-based software engineering

ACM Reference Format:
Aymeric Blot and Justyna Petke. 2020. Synthetic Benchmarks for Genetic Improvement. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392175

1 INTRODUCTION

Genetic improvement (GI) [7] uses automated search to find improved versions of existing software. In the last decade many works have repeatedly demonstrated the potential of GI for the improvement of both functional and non-functional properties of software. The GI field however suffers from major segmentation, due to a lack of standard GI benchmarks, as work usually focuses on different software, at different granularity levels, for different application scenarios, using slightly different search processes. If many of these steps are determined by the problem at hand (e.g., the programming language, the fitness function), many others require to be manually fixed, at great costs, using expert knowledge.

There is still much to learn on how to, given a specific application scenario, select the most suitable approach. Ideally one would simply select a set of various different software, compute a set of features (e.g., programming language, size, application), and perform a comprehensive study to understand relationships between features and approaches. This is currently infeasible due to the inherently very high cost of running GI experiments. Indeed, in essence all GI approaches suffer from the inevitable bottleneck that is fitness assessment, requiring in most cases to repeatedly run the (slow) targeted software. Although GI runs are easily parallelisable, the product of the number of scenarios, number of search algorithms, number of parameter configurations, and repetitions required for statistical validation, drastically reduces the scope for quick experimentation. Additionally, meta-approaches such as automated algorithm configuration, automated algorithm selection, or even genetic improvement of the GI approaches are also currently infeasible, being orders of magnitude slower than the GI approaches.

To enable such large-scale experiments, it is necessary to be able to avoid the expense of repetitive fitness evaluation. We propose to construct new, artificial scenarios, in which the fitness evaluation would not require any costly computation. These new scenarios would use synthetic data, automatically generated following statistics obtained from real scenarios obtained beforehand using preliminary analyses. Generating artificial data is often done when data is too scarce or too expensive to obtain [4], and mostly prevalent in machine learning and data science [5]. Other possible solutions may include simulating the execution of software variants, or computing surrogate models to predict its performance [2].

2 FORMALISM

GI focuses on improving some measure of a given software, typically its ability to clear a test suite without error or a performance measure such as running time, or memory or energy consumption [7]. GI can be formalised [1] as an abstract optimisation problem, as shown in Equation 1, given the space S of variants of the target software s_0, a distribution $\mathcal{D}$ (e.g., the test suite of an automated program repair scenario or the inputs or instances of a non-functional property optimisation scenario), a cost metric $o : S \times \mathcal{D} \to \mathbb{R}$, and a statistical population parameter E (e.g., the arithmetic mean).

$$\text{(GI)} \quad \begin{cases} \text{optimise} & E[o(s,i),\ i \in \mathcal{D}] \\ \text{subject to} & s \in S \end{cases} \tag{1}$$

We propose to completely substitute in Equation 1 the costly fitness estimation $E[o(s,i),\ i \in \mathcal{D}]$ of the software variant s, requiring compilation and subsequent multiple executions of s over elements of $\mathcal{D}$, by a single query to synthetic model, following the list of mutations between s and s_0. In practice, the model needs to associate with each possible software variant s both a final state—e.g., for cases when the mutated software failed to compile, or provides an invalid result—and the actual measure of the quality of s.

3 PROPOSAL

For the sake of simplicity, we assume an extremely simple scenario, in which: (1) the targeted software is represented by an abstract

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392175

syntax tree (AST) of n statements, (2) software variants are represented by lists of mutations, (3) mutations are either deletions or replacements of statements, (4) the quality measure of a software is the average running time over a set of inputs, and (5) validity of a variant can be checked simply by comparing outputs.

3.1 Analysing the Original Software

GI search spaces are huge. Even by only considering deletions and replacements, there are around $n + n^2$ different mutations, and therefore around $\sum_{k=1}^{m} (n+n^2)^k$ software variants up to m mutations to consider. In order to keep the complexity of our model low, we will suppose that the contribution of mutations are all independent.

The first step of our proposition is to collect statistics from the targeted real-world software. Unless n is extremely large, it should be reasonable to consider all deletions, but mutations such as replacements will necessarily require uniform random sampling. Each of the selected mutations is to be individually investigated, each time recording if the mutated variant compiled, successfully returned an acceptable output, and the associated running time normalised to the running time of the original software. Aggregating these data provides for each type of mutation the distributions that will be used to create the synthetic data.

3.2 Forging the Synthetic Model

We need to construct a model that can, given a list of mutations, provide a consistent and coherent alternative to compiling and running the associated mutated software. The model is ultimately equivalent to a simple hash table, with software variants as keys and results as values. As mentioned before, we consider here a very simplistic model in which the contributions of individual mutations are independent, so the hash table *only* needs to contain around $n + n^2$ keys, while the values are randomly generated according to the previously acquired statistics.

To generate values *on-the-fly* and avoid to actually storing that much data, we propose to rely instead on the pseudo-random number generator. From a canonical string representation of the mutation, supposedly unique, salted with the *root* random seed of the model, it is easy to generate a hash that can be used as a random seed to regenerate the characteristics of that particular mutation.

To then aggregate the individual contributions, we propose to abide by the following rules. First, if one or more mutations fail to compile, the complete patch is deemed to have failed to compile. Similarly, a mutation leading to a wrong output will cause the complete patch to also lead to a wrong output. If all the individual mutations are valid, the normalised running time of the complete patch is obtained by computing the product of all normalised running times. Finally, multiple appearances of a single mutation are ignored. For example, if a patch contains three mutations, one being 20% faster, one having no impact on running time, and one slowing the software by 5%, then the combined running time will be $0.8 \times 1.0 \times 1.05 = 84\%$ of the original software.

At last, changing the root random seed of the model will modify all hashes and thus provide other instances of models based on the same initial analysis. Performing analyses on many different software with different features will hopefully provide GI researchers with diverse and cheap benchmarks.

3.3 Discussion

The model we propose is purposely simple, and many drawbacks could be alleviated by considering more complex models, e.g., the independence assumption it requires is quite strong. To lessen it one solution would be to sample sets of mutations during the preliminary analysis to then reuse the collected statistics when aggregating individual contributions. Data pertaining to result variability, e.g., noise or affinity to given inputs, could also be integrated in the model. Similarly, the current model uniformly uses unified statistics for all nodes of the AST; clustering samples at the end of the analysis (e.g., by type, content, results, or simply position in the AST) would also lead to a more complex but hopefully better model.

In general, synthetic models should ultimately reflect known GI search spaces [6] and include features that can have a strong impact on GI approaches (e.g., mutational robustness [8], or plastic regions [3]). However, models should also compromise between high complexity and fidelity, and overall speed usage.

Finally, because correctness or consistency of such artificial models cannot really be proven, they cannot replace real-world scenarios but they can provide a very convenient way to prototype new approaches and conduct large-scale comparisons.

4 CONCLUSION

The main bottleneck of GI is software quality assessment, which prevents large-scale comparisons and meta-analyses of GI processes. We presented a method to completely circumvent executing the target software by replacing it by a very simple deterministic synthetic model. If this method can be shown to be efficient in creating reasonable and quick to evaluate models, we believe it to be of great potential for GI researchers.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Aymeric Blot and Justyna Petke. 2019. On Adaptive Specialisation in Genetic Improvement. In *Companion Material Proceedings of the 14th Genetic and Evolutionary Computation Conference (GECCO 2019 companion)*. ACM, 1703–1704.

[2] Nguyen Dang, Leslie Pérez Cáceres, Patrick De Causmaecker, and Thomas Stützle. 2017. Configuring irace using surrogate configuration benchmarks. In *Proceedings of the 12th Genetic and Evolutionary Computation Conference (GECCO 2017)*. ACM, 243–250.

[3] Nicolas Harrand, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monperrus, and Benoit Baudry. 2019. A journey among Java neutral program variants. *Genetic Programming and Evolvable Machines* 20, 4 (2019), 531–580.

[4] Yuri Malitsky, Marius Merschformann, Barry O'Sullivan, and Kevin Tierney. 2016. Structure-Preserving Instance Generation. In *Proceedings of the 10th International Conference on Learning and Intelligent Optimization (LION 10) (LNCS)*, Vol. 10079. Springer, 123–140.

[5] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. 2016. The Synthetic Data Vault. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics (DSAA 2016)*. IEEE, 399–410.

[6] 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.

[7] 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.

[8] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2014), 281–312.