On Adaptive Specialisation in 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 uses automated search to find improved versions of existing software. Software can either be evolved with general-purpose intentions or with a focus on a specific application (e.g., to improve it's efficiency for a particular class of problems). Unfortunately, software specialisation to each problem application is generally performed independently, fragmenting and slowing down an already very time-consuming search process. We propose to incorporate specialisation as an online mechanism of the general search process, in an attempt to automatically devise application classes, by benefiting from past execution history.

CCS CONCEPTS

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

KEYWORDS

Genetic Improvement, Search-Based Software Engineering, Software Specialisation, Algorithm Selection, Algorithm Configuration

ACM Reference Format:
Aymeric Blot and Justyna Petke. 2019. On Adaptive Specialisation in Genetic Improvement. In *Proceedings of the Genetic and Evolutionary Computation Conference 2019 (GECCO '19).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Genetic improvement (GI) [4, 7] uses automated search in order to improve existing software. The main applications of GI work include automated program repair and optimisation of non-functional properties such as running time, memory consumption, or software size [4]. In most GI work, the objective is to evolve a single general-purpose software variant. However, following the observation that optimal performance for all possible inputs is generally intractable, some work explicitly targets software specialisation [5, 6].

Unfortunately, software specialisation generally requires manual identification of different sub-classes of applications and to evolve the different specialised software independently, thus slowing down the improvement process proportionally to the number of different sub-classes of applications considered. Furthermore, classifying applications may involve costly feature computation, and feature selection is by itself a complex machine learning problem [2]. Additionally, it can happen that either a single evolved software fits multiple application domains, or that one of the considered application domains could have been divided further.

As an example, we consider Petke et al.'s recent work [6]. They ran three GI processes independently in order to specialise the MiniSAT solver for three different downstream applications. We make the following observations: Firstly, all three types of SAT instances are compatible, and can be tackled using the non-evolved version of MiniSAT. Secondly, it is non-obvious what sort of parameter settings and/or algorithmic changes should be made to boost MiniSAT's performance for the three different applications. Finally, the authors concluded that while it can be expected that some mutations are application-specific, other mutations might be beneficial for all three application domains.

2 PROPOSAL

In the following, we propose an online approach that automatically achieves the goal of software specialisation without the need for assumptions regarding the different application scenarios nor the need for feature identification, computation, or selection.

Following the MiniSAT example, we propose to run a single GI process on the training set comprising the instances of all three original applications. The search process would initially try to find modifications beneficial (i.e., decreasing running time in this case) to the entire training set, until some of the modifications can be used to find a partitioning of the training set for which they are statistically more relevant. Then, the search process would divide the search between the different partitions to specialise between the different subsets of applications. Additionally, such a process would enable the discovery of groups of statistically indistinguishable instances, in terms of running time improvement, on which substantial amount of training might have been otherwise wasted. The new GI process could also propose degradation of functional properties for specific applications (e.g., allowing specialisation to fail unit tests irrelevant to the target application).

Our proposition highlights the following research questions:

RQ1 Can GI search processes be modified to simultaneously accommodate for multiple target applications?
RQ2 How well can heterogeneity be detected during training?
RQ3 What is the cost of repartitioning the training set?
RQ4 How can adaptive specialisation approaches compare against pre-separated independent search?

3 FORMALISM

Algorithm configuration (AC) and algorithm selection (AS) are often seen as two instantiations of automated algorithm design [1]. AC focuses on improving the performance of a given algorithm by

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

modifying the values of its parameters. As in most GI scenarios, AC generally optimises a single general-purpose variant of the target algorithm. AC is often formalised as an abstract optimisation problem, as shown in Equation 1. Given a parameterised target algorithm A, the space Θ of configurations of A, a distribution of instances $\mathcal{D}$, a cost metric $o : \Theta \times \mathcal{D} \to \mathbb{R}$, and a statistical population parameter E, optimising the aggregated performance of the target algorithm A across all instances $i \in \mathcal{D}$.

$$\text{(AC)} \quad \begin{cases} \text{optimise} & E[o(A_\theta, i),\ i \in \mathcal{D}] \\ \text{subject to} & \theta \in \Theta \end{cases} \tag{1}$$

AC is very similar to GI [3]; it can indeed be seen as a subset of GI where the search is restricted to modifying a set of parameters [4]. GI can be formalised following Equation 1 by substituting Θ with the space S of evolved software, and searching for $s \in S$ rather than A_θ. While analogous to the case of non-functional property optimisation, the distribution $\mathcal{D}$ of instances corresponds to test cases in the case of automated program repair.

On the other hand, AS focuses on understanding the relation between algorithm performance and problem instance features. AS can be formalised as shown in Equation 2, where we optimise a mapping $m : \mathcal{I} \to \mathcal{P}$ between every instance $i \in \mathcal{I}$ and a corresponding algorithm in the portfolio $\mathcal{P}$.

$$\text{(AS)} \quad \begin{cases} \text{optimise} & \sum\limits_{i \in \mathcal{I}} o(m(i), i) \\ \text{subject to} & m : \mathcal{I} \to \mathcal{P} \end{cases} \tag{2}$$

In AC as in GI, software specialisation is performed by independently targeting specific applications. This can be seen as a manual AS approach in which the portfolio is obtained one instance at a time.

An example of a related hybridisation between AS and AC is given by the Hydra algorithm [8], in which the mapping between instances and algorithm is obtained by performing successive AC search processes, but this time each final algorithm being compared on every instance. Although specifically proposed for AC, Hydra's approach should also be compatible with GI and could reasonably be adapted and compared with our approach.

The approach we propose is similar to Hydra's, but instead of using AC to populate an AS portfolio, we aim to identify and differentiate applications during the GI search process. More specifically, we want our approach to behave identically to standard GI approaches when confronted with single or homogeneous application scenarios, but to be able to distinguish between different classes when applied to mixed application scenarios. This approach can be formalised as shown in Equation 3.

$$\text{(GI+AS)} \begin{cases} \text{optimise} & \sum\limits_{\mathcal{P}_k \subset \mathcal{D}} E[o(m(\mathcal{P}_k), i),\ i \in \mathcal{P}_k] \\ \text{subject to} & m : \mathcal{P}_k \subset \mathcal{D} \mapsto s \in S \\ & \bigcup\limits_k \mathcal{P}_k = \mathcal{D}, i \neq j \implies \mathcal{P}_i \cap \mathcal{P}_j = \emptyset \end{cases} \tag{3}$$

Starting from a single-class partitioning of $\mathcal{P}_o = \mathcal{D}$, for which the equation is equivalent to GI, we aim to progressively find a better partitioning together with a mapping m that associates each partition $\mathcal{P}_k$ with an evolved version of the initial software.

4 IMPLEMENTATION CONSIDERATIONS

The described approach is yet to be implemented and analysed.

Firstly, none of the available GI search processes can simultaneously operate on multiple target applications, even if those are known from the start of the search, the current only option being to duplicate the search on each application independently. This approach will then require a new, specifically made, search process. The need to manage a portfolio of software variants however suggests that population- and archive-based algorithms such as multi-objective local search or genetic programming may be considered as potential candidates. Then, there is no evidence that during the search there will be enough deviating data on the different instances to accurately classify them into separate applications. An unsatisfactory classification mechanism will fail to distinguish between instance classes, while a faulty one will needlessly create new targets. We expect limiting the number of partitions to be necessary, while no partitioning being found should simply mean a general-purpose software will be evolved. Finally, it can be expected that this mechanism will take some computational resources. Applying it too often, or too thoroughly, will necessarily result in slowing down an already computationally heavy search process. We claim, however, that our approach will be more efficient than running multiple independent GI processes for each individual application domain. We will compare our approaches to validate this claim.

5 CONCLUSIONS

Software specialisation requires manual identification of the target application domains. This must be achieved before any search-based technique is applied, and requires either prior expert knowledge or expensive instance feature analysis. We propose a novel approach, hybridising specialisation as an online mechanism of the GI search process. This approach does not rely on expert knowledge or instance feature selection, but rather on statistical differences in performance as learned during the GI search process.

ACKNOWLEDGMENTS

This work was funded by the UK EPSRC fellowship EP/P023991/1.

REFERENCES

[1] Aymeric Blot. 2018. *Reacting and Adapting to the Environment: Designing Autonomous Methods for Multi-Objective Combinatorial Optimisation.* Ph.D. Dissertation. University of Lille, France.
[2] Isabelle Guyon and André Elisseeff. 2003. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research* 3 (2003), 1157–1182.
[3] Saemundur O. Haraldsson and John R. Woodward. 2014. Automated design of algorithms and genetic improvement: contrast and commonalities. In *Genetic and Evolutionary Computation Conference (GECCO '14 Companion).* ACM, 1373–1380.
[4] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432.
[5] 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 *Genetic Programming (EuroGP 2014) (Lecture Notes in Computer Science)*, Vol. 8599. Springer, 137–149.
[6] 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.
[7] David Robert White, Andrea Arcuri, and John A. Clark. 2011. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation* 15, 4 (2011), 515–538.
[8] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2010. Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. In *AAAI Conference on Artificial Intelligence (AAAI 2010).* AAAI Press, 210–216.