Genetic Improvement for Adaptive Software Engineering

Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo and Fan Wu
CREST Centre, University College London, UK

ABSTRACT

This paper presents a brief outline of an approach to online genetic improvement. We argue that existing progress in genetic improvement can be exploited to support adaptivity. We illustrate our proposed approach with a 'dreaming smart device' example that combines online and offline machine learning and optimisation.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Artificial Intelligence, Machine Learning, Genetic Improvement, Search Based Software Engineering

1. INTRODUCTION

Many Artificial Intelligence (AI) techniques are designed to cope with a world characterised by noisy, incomplete and inconsistent data. They can also cater for multiple conflicting and competing objectives; precisely the scenario faced by the practicing software engineer.

In this paper we outline how a Search Based Software Engineering approach called 'genetic improvement' could be extended to provide online adaptivity. Search Based Software Engineering (SBSE) [7, 9, 13, 16, 19] is one example of a form of 'Software Engineering AI' that has been widely applied across many software engineering domains, including requirements [42], management [8], design [35], testing [30], refactoring [31] and bug fixing [26].

SBSE applies computational search to optimise problems in software engineering. It might be thought of as merely ‘another engineering application of computational search and optimisation’. However, the virtual nature of software makes it a special case; the optimisation system and the artefact to be optimised are both constructed from the same engineering material [14]. This has profound consequences for the optimisation process and for the optimisations it is able to achieve.

A subfield within SBSE has emerged focusing on Dynamic Adaptive Search Based Software Engineering [15, 17], for which a blend of machine learning and optimisation is required. In dynamic adaptive SBSE, properties are adapted by reconfiguring or even rewriting the deployed software as it executes. The properties considered are typically operational properties such as computation time, bandwidth, throughput, memory use and power consumption.

Our goal is to develop this kind of iterative learning based adaptivity, so that optimisation techniques, developed for the SBSE research agenda, can be used to optimise learning and adaptivity. In particular, we see an important role for Genetic Programming (GP) [34] as a means of ‘program improvement’ that has come to be known as ‘genetic improvement’ [17, 18, 25] (and has also been referred to as ‘evolutionary improvement’ [40]).

2. GENETIC IMPROVEMENT

There has been a dramatic recent upsurge in interest and techniques for improving existing programs using genetic improvement [17]. Genetic improvement has demonstrated several recent advances, such as speeding up execution [24, 25, 32, 40] and fixing bugs [2, 26]. Furthermore, because genetic improvement modifies an existing system, rather than seeking to build a new system from scratch, it can speed up highly non-trivial real world programs [24, 25, 36, 37] and fix real world bugs [22, 27, 26].

Genetic improvement has also been used to port one system to a new version on a different platform and language [23] and to balance implementation objectives [41]. It can also synthesise and specialise different versions of a system [33]. Related work on similar performance-improving modifications to existing code has also reported dramatic speed ups on real-world systems [36, 37].

Genetic improvement starts from the premise that an existing program can be improved by searching for modifications in existing system [32, 40, 41], finding 'patches' from elsewhere in the system [1, 24, 25, 26, 39] or by transplanting code from one program to another [18, 33].
3. ONLINE GENETIC IMPROVEMENT

Recent breakthroughs in genetic improvement make it tempting to speculate on whether ‘online’ genetic improvement may now lie within our grasp. The remainder of this paper sketches possible ways in which this might be realised.

3.1 Exposing Implicit Parameters

We first propose to search for expressions over internal program variables that are sensitive to operational properties of interest so that we might expose these variables as ‘tuning parameters’ [6, 20]. For instance, suppose we seek to improve response time for a device, a property that can be measured as a function of the input vector to the program. We could search for arithmetic and logic expressions in the program (right hand sides of assignments, actual parameters, conditional and loop predicates etc). The expressions we seek are those which, when mutated by GP, will have an effect on response time.

Having identified these expressions, we can refactor the program to make the values of these expressions ‘tunable’. That is, we expose the internal program variables on which these expressions depend, so that they can become additional parameters to the program. The system will thereby acquire an extended input space that includes exposed ‘input’ parameters.

Observe that tuning may (often) change the semantics of the program and thereby cause the parameter choice to be rejected. The interesting problem is thus to find the choices of parameters and their values that yield a program that is acceptably faithful to the original’s semantics, while improving the non-functional property of interest. These values can be found (learned) by optimisation, as with other examples of parameter tuning using SBSE [28, 38].

Human programmers might reject changes made by any automated programming technique. If they feel that they have little understanding or control there will be a natural reluctance to ‘trust the optimiser’. Automated programming approaches such as ours thus raise the question ‘will the machine generated code be maintainable?’ [10]. Fortunately, since parameters already embody a degree of encapsulation, the human merely need understand the parameters exposed by our approach and can choose to accept some and reject others. Our approach could thus be thought of as a dialog between the human and machine about the design of the ‘adaptive interface’.

We envisage two distinct classes of user: programmers and endusers. Programmers will be concerned with maintaining the system and will use the explored parameters as a design dialog. We speculate that a variant of Aspect Oriented Programming (AOP) [21] might prove to be one way to introduce such tunable adaptivity into an existing software system. The traditional aim of AOP was to separate ‘cross-cutting’ concerns, but it can also be used to extract aspects from non-aspectual code [4] in much the way we propose here.

End users might also be able to exploit the advantages of parameter exposition. Clearly, we would require an intuitive way to communicate the options exposed. The user will need to be able to indicate which operational characteristics matter. They would want a simple, effective and always-available option to revert to the unoptimised version. These are well-known challenges for the software personalisation community, on which we might draw inspiration.

Figure 1: Our approach to offline learning and optimisation with online deployment for adaptivity.

Our overall approach is illustrated in Figure 1. The approach consists of offline learning and optimisation and subsequent online adaptive selection and deployment of learned optimisations.

3.2 Illustrative Example: Dreaming Devices

In some scenarios optimisation might be undertaken in real time. This would be particularly challenging due to the tight time bounds inherent in real-time adaption. However, in this paper we wish to focus on the many scenarios we envisage for dynamic adaptive SBSE and online genetic improvement in which there is a longer optimise-and-adapt cycle. We believe such longer optimised adaptivity cycles map well onto current and likely future use-cases. This has the significant technical advantage that it offers considerably more computational time to develop learned optimisation.

Imagine the scenario in which a device such as a smartphone has two modes of operation: ‘normal’ and ‘charging’. In normal operational mode, the device is not connected to a power source, while in the other, it is charging. The charging process might take place overnight, while the device’s user sleep or in other periods where the user is disengaged from the device. We use the term ‘overnight’ to make the scenario more concrete, but this period refers to any time during which the user is disengaged and power is available for learning optimisation. During an overnight charging cycle there is no need to optimise the power consumption, but such optimisation becomes more important during normal operational mode. This instantiation of our approach is depicted in Figure 2.

We propose to augment the device with a background process that runs during normal operational mode, to collect data that can subsequently be used in an ‘overnight learning cycle’. During the overnight cycle, the data collected during the daytime operation can be used as a source of training data to optimise performance for subsequent operation. Since the device is not in normal use during the overnight cycle and since there is plenty of power available during this cycle, this is surely the ideal opportunity for optimisation learning.
Humans may learn overnight during sleep cycles by replaying the day’s events; a process we call ‘dreaming’. In the same way, our device might also learn from the observations made during normal operational use. Allowing a little poetic license, we might even claim that we would have created a kind of ‘dreaming device’ that learns during its sleep cycle by replaying the events of the day. It might even prove interesting, entertaining (and perhaps even informative) to switch on the device’s screen and directly observe this dream state.

There are many challenges in achieving such a ‘dreaming device’ learning process: Data collection needs to be an unobtrusive background process. The overnight learning process might be designed specifically to reduce daytime power and therefore the costs of data collection would have to be amortised into the overall resource reduction achieved.

Such ‘dreaming’ is an offline learning technique that can subsequently be used to dynamically adapt the device as it is used. The natural cycle and rhythm of operation affords a considerable period for optimisation: typically something in the region of 6 to 8 hours during overnight charging cycle.

There are many other examples of systems where similar periods of time might be available for the learning and optimisation process, yet the overall approach would still offer online dynamic adaptive optimisation. Even an ‘old fashioned’ desktop application might be augmented with an overnight learning optimisation process. On-board systems on commercial and private vehicles might also be able to avail themselves of an overnight learning optimisation phase; the vehicle may be stationary for much of its life, offering opportunities for optimisation.

Web-based systems, providing services to customers, might also be able to find opportunities for optimisation during operation. These systems have no overnight mode. They are typically the subjects of a theoretical requirement for continual availability. However, using profiling, we might hope to find evidence for obvious exploitable periods of likely ‘quiet usage’, during which learning and optimisation can take place.

Of course, predictive models, such as these, that are able to determine when exploitable quiet periods are likely to occur would be very useful. Indeed, learning when the learning phase might best take place is also an example of the learning optimisation process.

![Figure 2: Our approach applied to dynamic adaptive optimisation of smart devices.](image-url)

Concepts of self-adaptive and autonomic computing are certainly not new [3, 12, 29]. Much work has also already been done on machine learning for adaptivity. For instance, the problem of trading disk idle time, power consumption and spin-up latency has been studied and developed over two decades [5]. Our vision of online genetic improvement, briefly set out here, is not without challenges, many of which will be familiar to researchers working on adaptivity in general.

It is also encouraging to note that the number of changes, being small, lies within the realms of ‘discoverability’: empirical analysis of nearly half a billion lines of code by Gabel and Su [11] showed that a programmer has to write at least six lines of code before what they create is original (up to variable renaming). This is to be contrasted with the observation that only 7 lines needed to be changed in order to produce a factor 70 speedup using genetic improvement in a recent case study [25].

Acknowledgement: Thanks to John Shawe-Taylor for comments on drafts of this paper. This work is part supported by the EPSRC DAASE and GISMO projects.

4. REFERENCES


C. Le Goues, S. Forrest, and W. Weimer. Current

W. B. Langdon and M. Harman. Optimising existing

M. Harman. Why the virtual nature of software makes it

M. Gabel and Z. Su. A study of the uniqueness of source


W. B. Langdon and M. Harman. Evolving a CUDA kernel

G. Kiczales and M. Mezini. Aspect-oriented programming


M. Harman. Why the virtual nature of software makes it ideal for search based optimization. In 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010), pages 1–12, Paphos, Cyprus, March 2010.


