Guiding Unconstrained Genetic Improvement

David R. White
UCL, London, UK
david.r.white@ucl.ac.uk

ABSTRACT

This paper argues that the potential for arbitrary transformation is what differentiates GI from other program transformation work. With great expressive power comes great responsibility, and GI has had mixed success finding effective program repairs and optimisations. The search must be better guided in order to improve solution quality.

Keywords

Genetic Improvement; Genetic Programming; SBSE

1. ARBITRARY TRANSFORMATIONS

What separates GI from other bug-fixing and program optimisation methods? It is often assumed that the answer to this question is "test-based evaluation", but in fact testing is ultimately relied upon for validation in much repair and optimisation work employing formal techniques and template-based transformations [4, 5]. Formal specifications and program annotations are not usually available for existing code, forcing automated methods to rely on test suites in most cases. GI is therefore not unique in this regard. Current formal approaches are limited in terms of the size and type of improvement required, providing higher quality results at the cost of generality and scalability, e.g. [4].

In truth what separates GI is that the search operators usually allow for *arbitrary transformations* of code: the transformations are not heavily constrained, for example, to simple template application. Ultimately, this is what makes the fields of Genetic Programming (GP) and GI exciting in their potential: they may produce any code that a programmer could consider. Unfortunately, another consequence of their expressive freedom is that they may produce code that a programmer would have good reason *not* to consider, and in conjunction with a reliance on test cases, this can lead to the problem of overfitting and bug-fixes that break more functionality than they improve [9]. Unconstrained optimisation is GI's greatest strength and yet also its greatest weakness.

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA
© 2016 ACM. ISBN 978-1-4503-4323-7/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2908961.2931688

Arbitrary transformations are excluded from most program repair and optimisation work for two reasons: first, they result in a vast space of possible program changes; second, most methods for increasing our confidence in the correctness of a change do not scale to support the creation of arbitrary code, i.e. a wide variety of changes in both type and size. Past experience of success in exploring vast combinatorial spaces tells us that the first problem may be solved approximately using search and other AI methods; the second problem appears to be more unique. Recently, the limitations of allowing arbitrary transforms have been underlined by close examination of patches generated by the popular GenProg tool [7]. If GI is to progress and be adopted by those outside the evolutionary computation community, the problem of low-quality patches and optimisations must be addressed.

2. WHITE-BOX GI

It appears likely that treating GI as a black box — or treating GP in the same way, for that matter — is insufficient to avoid overfitting. There are many programs that will satisfy any given set of test cases, but the correctness of a patch or a program lies in its generality, which cannot for most problems be evaluated simply by examining program output on a set of test cases; consider how we are to classify an unseen input as belonging to the set of positive or negative test cases when bug-fixing, for example. When creating software, we cannot reason via analogy with Machine Learning: programming is not a classification problem.

The fault-finding efficacy of test generation can be improved by employing metrics that describe how well a test suite exercises source code under test. Regardless, relying on testing alone is still post-hoc; we are reliant entirely on testing to identify any damage caused to useful program behaviour. Our testing procedure thus needs to be watertight, for arbitrary transformations. Witnessing the continued volume of bugs in deployed software should give us little hope that this can be achieved.

So we cannot rely on testing that results in high-quality solutions, yet we may not wish to abandon the arbitrary transformations that give GI its great potential. Without restricting the search space, how can we hope to bridge the gap between such an expressive tool and the demand for solutions that are at least *likely* to be correct? I propose that we *allow the generation of arbitrary transformations, but encourage the selection of high-quality ones.*

3. PATCH QUALITY HEURISTICS

Which possible heuristics should we consider? The most obvious heuristic, already employed in bug-fixing work, is to use Occam's Razor: to minimise the size of a patch. Unfortunately, this principle combined with test-based evaluation is akin to the converse of mutation testing: it invites the search process to find patches that minimise the number of changes to a program to improve some behaviour in the test suite, whilst also breaking any behaviour the test suite does not encapsulate. Insertion of dead code or removal of small amounts of code are good strategies for such a search, and indeed this behaviour has been observed [7]. A problem here is that we are considering *syntactic* change size.

More generally, the implementation of such a parsimony objective in GP is known to limit the exploration of the search space [6]. One solution is to minimise a patch after the run [3], but again this treats GI as a black box, and does not guide the search for high-quality solutions.

Given these observations, it is not too great a leap to conclude that only quality heuristics based on the static or dynamic analysis of program semantics will stand any chance of providing a patch quality heuristic (a PQH). I now discuss three such lines of attack.

3.1 PQH 1: Invariant Complexity

When a programmer writes code, they do not aim to solely pass a set of tests (even when working in TDD), rather they aim to produce a solution that is general. This generality of intent is what distinguishes programmers from test-based automated programming.

Through the application of automated invariant generation tools such as Daikon [1] to candidate solutions, invariants encapsulating their behaviour can be suggested. Similarly, invariants can be derived for the original program. Related work has guided repair by limiting the impact of program changes in terms of their changes to existing semantics [5], and such a method could also be applied to GI at a large scale if a tool such as Daikon is used: minimise the distance between invariant sets. Positive and negative behaviour may be treated separately when bug-fixing.

More speculatively, I propose that the *fragility* of the invariants that describe a candidate solution can be used as a heuristic to guide patch quality. This is a return to Occam's razor at the level of semantics: the most interesting programs are those that are encapsulated through simple invariants when compared to their neighbours in a program space. Ratcliff et al. [8] showed how the simplicity of an invariant was correlated with its ability to isolate a correct program in the space of mutated variants.

3.2 PQH 2: Information-Theoretic Measures

A position paper by Johnson and Woodward [2] suggested information theoretic measures of "progress" within a program as a suitable measure of proximity to a given goal. They succinctly describe this as "quantifying how much closer the program is to solving its task" at various points in a program. For example, they suggest using a compression distance measure to compare program state to a target. An optimisation may reduce this distance at a given point in the program. This is another example of white-box GI, using dynamic analysis to evaluate the quality of a given candidate solution. As with Daikon, they use the result of multiple runs to build a picture of program behaviour.

Information theoretic heuristics can be used to eliminate unwanted changes that do not contribute towards achieving a computational goal or modify the current behaviour of a program by too great an extent.

3.3 PQH 3: Ensemble Learning

By allowing GI to improve our program across multiple runs, we can use a form of ensemble learning [10] by partitioning test cases to produce multiple potential solutions and then perform post-processing to refine these solutions into a single patch. Partitioning the test cases across runs and then generalising from these results may prevent overfitting to any particular partition of test cases. In particular, a method similar to stacked generalisation [11] found in classification could be used to select modifications from the union of output patches across these multiple runs. This may at first appear to a high-level approach unrelated to PQHs 1 and 2, but the key observation is that these partitions be based on the white-box behaviour of the original program.

4. CONCLUSION

Arbitrary code changes are GI's greatest strength and weakness. I believe that we should *guide* rather than *constrain* GI, to retaining its scalability and creative potential, through the use of potentially expensive quality heuristics: high-quality arbitrary transformations come at a cost.

5. REFERENCES

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[2] C. G. Johnson and J. R. Woodward. Fitness as task-relevant information accumulation. In *GECCO Companion*, pages 855–856, 2015.

[3] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE Proc.*, 2012.

[4] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE Proc.*, 2015.

[5] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *ICSE Proc.*, 2013.

[6] R. Poli and N. F. McPhee. Parsimony Pressure Made Easy. In *GECCO Proc.*, 2008.

[7] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *ISSTA 2015*, 2015.

[8] S. Ratcliff, D. R. White, and J. A. Clark. Searching for invariants using genetic programming and mutation testing. In *GECCO Proc.*, 2011.

[9] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *ESEC/FSE*, 2015.

[10] K. Veeramachaneni, I. Arnaldo, O. Derby, and U.-M. O'Reilly. Flexgp. *Journal of Grid Computing*, 13(3):391–407, 2014.

[11] D. H. Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.