Self-Focusing Genetic Programming for Software Optimisation

Brendan Cody-Kenny
Codykenb@scss.tcd.ie

Stephen Barrett
Stephen.Barrett@scss.tcd.ie

Distributed Systems Group
School of Computer Science and Statistics
Trinity College Dublin, Ireland

ABSTRACT

Approaches in the area of Search Based Software Engineering (SBSE) have seen Genetic Programming (GP) algorithms applied to the optimisation of software. While the potential of GP for this task has been demonstrated, the complexity of real-world software code bases poses a scalability problem for its serious application. To address this scalability problem, we inspect a form of GP which incorporates a mechanism to focus operators to relevant locations within a program code base. When creating offspring individuals, we introduce operator node selection bias by allocating values to nodes within an individual. Offspring values are inherited and updated when a difference in behaviour between offspring and parent is found. We argue that this approach may scale to find optimal solutions in more complex code bases under further development.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; D.1.2 [**Software**]: Automatic Programming

Keywords

Search Based Software Engineering; Genetic Programming; Scalability

1. INTRODUCTION

A general issue with the use of GP is that the size and complexity of individuals has an impact on the effectiveness of a GP algorithm [3, 2]. This is particularly true when considering the use of GP to modify software systems. Complexity in this context can be defined in many ways but generally contains a notion of counting interdependency and interactions of entities within a program. We seek to address complexity through the use of rule sets which influence node selection bias from information which is generated during the GP run. Our research question follows: Can node selection bias inferred by parent-offspring differences, impact the scalability of a GP algorithm for source code optimisation? Our experimental results found that the rules outlined here did affect scalability, albeit negatively.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
ACM 978-1-4503-1964-5/13/07.

2. RELATED WORK

White and Arcuri remark on the problem of scalability when applying GP to Software Optimisation but designate it out of scope in their work [8]. Weimer and Forrest address the scalability problem when applying GP to bug fixing [7] by biasing node selection before the GP run is started.

Although biasing the selection of nodes before applying GP has the advantage of adding little overhead during the GP run, as the GP run progresses and the population evolves further from the original program, this bias may become less accurate at recommending relevant locations. If the bias effect is dampened as the algorithm runs, this may limit the technique to finding only those solutions that are the result of a small number of modifications to the code. This may not be important as Weimer and Forrest's work [7] along with Langdon's [5] show that a large improvement in performance is possible with few modifications.

Node selection bias can be introduced by attaching values to program nodes. During the GP run, these values are updated with random noise when an offspring is created [1]. Values for all nodes in a program form a parameter tree of values in the same structure as the program. Through repeated generations, parameter tree's emerge which bias the application of operators to places within the tree which are generally good. The emergence of useful bias is dependent on the evolutionary pressure and contains the potential for a random bias update to be performed which is non-optimal.

3. GP SYSTEM

By taking a more direct approach in modifying node selection bias we hope to dampen the effects of any sub-optimal node selection and increase the node selection pressure or node bias. We therefore look for alternative ways to vary the parameter tree values other than randomly. By introducing specific rules for how parameter tree values are changed, we may be able to optimise node selection and reduce wasted computational effort. Improving node selection might then improve the scalability of the algorithm.

Our solution is based on attaching a value to each possible modification point in a program as described by Angeline [1]. Each value is used to determine the probability that the associated node is picked for the application of an operator. When an offspring individual is generated, the probability values from its parents are inherited. We inspect two different rule sets for changing this selection bias when an offspring program is created.

According to our first rule set, the values in the offspring parameter tree will be decreased if the offspring functionality score is different to the parents. When the offspring functionality is unchanged from the parents we increase the probability of change at the modified nodes.

The second rule set specifies the opposite. If we make a change in functionality by applying operators at a location in the program tree, then this location is relevant to the functionality and so should have a higher chance of being modified again. This rule set strives to focus the GP algorithm in areas of each individual which have some effect on the functionality of the individual. By doing this, it is expected that GP can ignore areas of a program that can be considered irrelevant bloat or introns from a GP perspective. By focusing on areas of the code that impact functionality, it is hoped that less effort is wasted in finding a solution.

We seed the GP algorithm with the program to be optimised by converting program source code to an Abstract Syntax Tree (AST) [6]. The initial population of programs is generated by repeatedly applying mutation to the original program. The fitness function is calculated using the number of test cases passed and number of instructions performed [4] by each program. To get a normalised measure of performance, the seed program is evaluated and used as a baseline. The seed program receives a runtime value of 1 while programs worse than the seed program receive values higher than 1. Programs which run with less instructions than the seed program and pass all test cases get a fitness value below 1.

Primitives for our GP system are gathered from the seed program which includes operators (infix, postfix), conditionals, variables, number literals, expressions, updaters and statements. We manually add the equal to operator ("==") and postfix decrement operator ("--"). The following GP setup is used: Population Size: 250, Generations: 60, Tournament selection: 2, Crossover Rate: 0.9, Mutation Rate: 0.3, Max Program Length: 20 Lines, Max Operator Applications: 100, Elitism: 5%.

4. EVALUATION

To inspect the scalability of GP on software we use a naive version of bubble sort as described by White, Arcuri [8] that requires at minimum 2 changes to find the obvious optimisation. Our control setup uses traditional random selection of nodes when applying operators. When rule sets are used to introduce node selection bias, tournament selection is used to select nodes.

Our analysis found that both rule sets reduced the algorithms ability to find the optimisation in the program. Standard GP found the optimisation 13.5% of the time, rule set one 11% and rule set two 9% of the time. We also analysed the different distributions of program size and fitness across the different mechanisms and did not find any variance in these distributions. Neither of these rule sets affect the distribution of individual sizes.

The average retry rate has remained the same across all mechanisms with an average retry of 1.1 for crossover and max number of retries of 5. The retry average for mutation is also 1.1 with the max number of retries at 6. This indicates that the difficulty in performing crossover and mutation is also not influenced by our approach. We deduce from our analysis that the rule sets change the behaviour of the GP algorithm with regard to the structure and the content of GP individuals only. The structure and content can change but only limited within the bounds of the distribution of program size.

5. CONCLUSION

It appears from these results that biasing node selection in response to changes in functionality is not beneficial to GP for the example bubble sort case. As our results are preliminary and negative, we seek to experiment with different bias allocation rules and extend the range of information that is gleaned from the generation of a new individual.

Further experimentation, which is currently underway, indicates that updating bias based on performance changes can benefit the GP algorithm in the desired way. Provided this development can be validated, we speculate that more complex rules, possibly including some domain knowledge, can improve the selection of nodes for the application of operators.

We have answered our research question by demonstrating how simple rules can be used to apply operators in such a way as to influence the pace at which GP finds solutions. Our general research direction is toward developing a form of "fuzzy" modularisation which exists and operates solely for the purpose of guiding GP. If Object Oriented Programming provides useful modularisation for human developers, we are interested in conceptually similar probability based techniques for GP algorithms.

6. REFERENCES

[1] P. Angeline. Two self-adaptive crossover operations for genetic programming. 1995.

[2] W. Banzhaf and J. Miller. The challenge of complexity. *Frontiers of Evolutionary Computation*, pages 243–260, 2004.

[3] E. de Jong, R. Watson, and D. Thierens. On the complexity of hierarchical problem solving. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1201–1208. ACM, 2005.

[4] M. Kuperberg, M. Krogmann, and R. Reussner. ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation , Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*, 2008.

[5] W. B. Langdon and M. Harman. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 19 Sept. 2012.

[6] The Eclipse Foundation. Java development tools. `http://www.eclipse.org/jdt/`, Nov. 2012.

[7] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[8] D. White, A. Arcuri, and J. Clark. Evolutionary improvement of programs. *Evolutionary Computation, IEEE Transactions on*, (99):1–24, 2011.