Effective Simplification of Evolved Push Programs Using a Simple, Stochastic Hill-climber

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Thomas Helmuth
Computer Science
University of Massachusetts
Amherst, MA 01003
thelmuth@cs.umass.edu

ABSTRACT

Genetic programming systems often produce programs that include unnecessary code. This is undesirable for several reasons, including the burdens that overly-large programs put on end-users for program interpretation and maintenance. The problem is exacerbated by recently developed techniques, such as genetic programming with geometric semantic crossover, that tend to produce enormous programs. Methods for automatically simplifying evolved programs are therefore of interest, but automatic simplification is non-trivial in the context of traditional program representations with unconstrained function sets. Here we show how evolved programs expressed in the stack-based Push programming language can be automatically and reliably simplified using a simple, stochastic hill-climber. We demonstrate and quantitatively characterize this simplification process on programs evolved to solve four non-trivial genetic programming problems with qualitatively different function sets.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

Keywords

genetic programming, simplification, Push

Simplifying Evolved Programs

In some application areas the usefulness of genetic programming (GP) hinges on the conciseness of the programs that it produces. The issue of program size has recently become even more significant with the development of new GP approaches based on "geometric semantic" program transformations [2], which show promise in terms of problem solving power but may produce enormous programs.

GP systems that evolve programs in a minimal-syntax language permit particularly simple forms of automatic simplification that can nonetheless be highly effective, even with programs involving arbitrary types and function sets. We demonstrate this point by showing how programs evolved in the minimal-syntax Push programming language [5, 4] can be effectively and automatically simplified by means of a trivial hill-climbing simplification algorithm. The demonstrations here all begin with correct programs produced by successful genetic programming runs.

The simplification algorithm that we use here takes as input a program to simplify, a fitness function (which produces a vector of fitness values, one for each fitness case), and a number of simplification steps. It attempts to simplify the program step by step, for each step proposing a simplification to the current simplest program (which is initially the input program) and testing the new, simpler program's fitness. In 80% of steps we remove either one or two randomly selected "things" from the program, where single instructions, literals, and parenthesized sub-programs all count as single "things." In the remaining 20% of steps we flatten (remove all internal parentheses from) a randomly selected thing. Because there is essentially only one syntax rule in Push—that parentheses must be balanced—the programs that result from these steps will always be syntactically valid. If the new program's fitness vector is the same as the original fitness vector then the new program is retained as the current simplest program, but if the simplification results in a changed fitness vector then the new program is discarded. This process is repeated for the specified number of steps, and then the final simplest program is returned as the result of the simplification process.

On each of four problems—the 3-bit digital multiplier problem [6], the Pagie-1 symbolic regression problem [3], the wc (word count) problem [1], and the odd problem [5]—we gathered a number of evolved solutions of differing sizes, and used the simplification algorithm to reduce their sizes multiple times. In the results shown in Figures 1-4, gray triangles indicate evolved program sizes, sorted by size. Below each gray triangle is the median simplified program size from 100 independent simplification runs, each of which ran for 10,000 steps, on the corresponding program. Error bars mark the minimum and maximum simplified program sizes.

Overall, we found this technique to efficiently and reliably reduce the size of the evolved programs across four qualitatively different problems involving widely varying instruction sets and genetic programming parameters. It would be interesting to study the use of the simplification algorithm presented here during, rather than just after, evolution. In fact the Clojush system used for these experiments includes a built-in genetic operator that runs this simplification algo-

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
ACM 978-1-4503-2881-4/14/07.
http://dx.doi.org/10.1145/2598394.2598414.

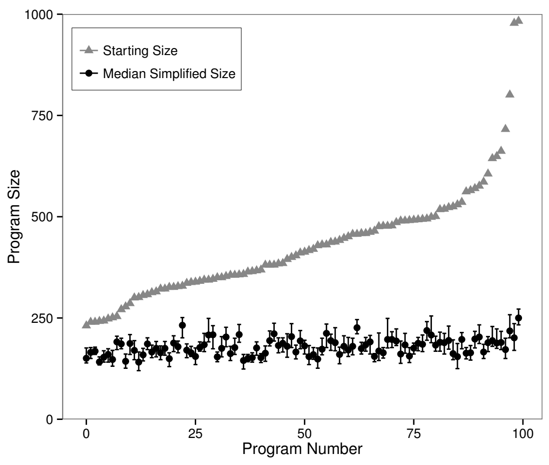

Figure 1: Results: 3-bit digital multiplier problem.

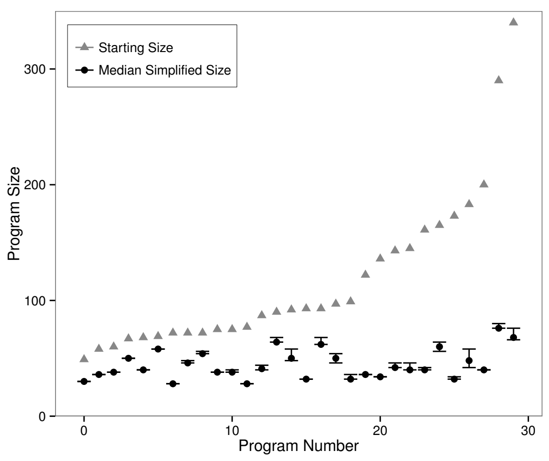

Figure 2: Results: Pagie-1 problem.

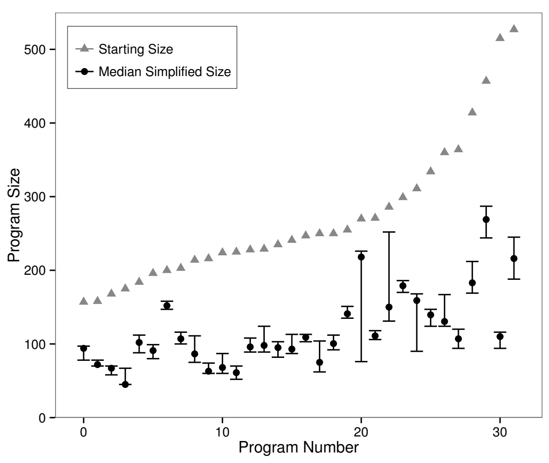

Figure 3: Results: wc (word count) problem.

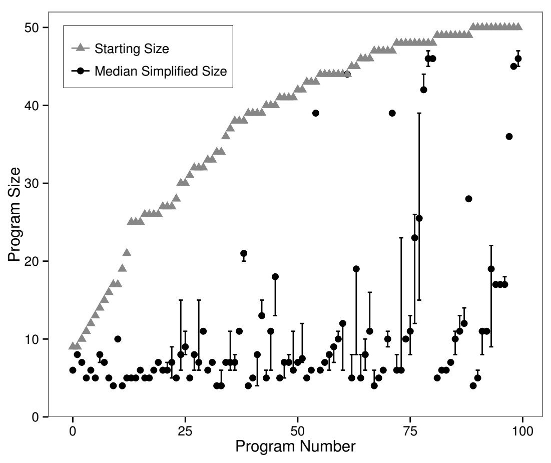

Figure 4: Results: odd problem.

rithm on selected parents to produce children. This feature has been used before in various contexts, but no systematic study has yet been conducted on the effects that this operator has on evolutionary dynamics or on problem-solving success.

1. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 1017817 and 1129139. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

2. REFERENCES

[1] T. Helmuth and L. Spector. Word count as a traditional programming benchmark problem for genetic programming. In *GECCO '14: Proceedings of the sixteenth international conference on Genetic and evolutionary computation conference*, Vancouver, 2014.

[2] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature, PPSN XII (part 1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31, Taormina, Italy, Sept. 1-5 2012. Springer.

[3] L. Pagie and P. Hogeweg. Evolutionary consequences of coevolving targets. *Evolutionary Computation*, 5(4):401–418, Winter 1997.

[4] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[5] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[6] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.