Improving Generalization of Evolved Programs through Automatic Simplification

Thomas Helmuth
Washington and Lee University
Lexington, Virginia, USA
helmutht@wlu.edu

Nicholas Freitag McPhee
University of Minnesota, Morris
Morris, Minnesota, USA
mcphee@morris.umn.edu

Edward Pantridge
MassMutual Financial Group
Amherst, Massachusetts, USA
epantridge@massmutual.com

Lee Spector
Hampshire College
Amherst, Massachusetts, USA
lspector@hampshire.edu

ABSTRACT

Programs evolved by genetic programming unfortunately often do not generalize to unseen data. Reliable synthesis of programs that generalize to unseen data is therefore an important open problem. We present evidence that smaller programs evolved using the PushGP system tend to generalize better over a range of program synthesis problems. Like in many genetic programming systems, programs evolved by PushGP usually have pieces that can be removed without changing the behavior of the program. We describe methods for automatically simplifying evolved programs to make them smaller and potentially improve their generalization. We present five simplification methods and analyze their strengths and weaknesses on a suite of general program synthesis benchmark problems. All of our methods use a straightforward hill-climbing procedure to remove pieces of a program while ensuring that the resulting program gives the same errors on the training data as did the original program. We show that automatic simplification, previously used both for post-run analysis and as a genetic operator, can significantly improve the generalization rates of evolved programs.

CCS CONCEPTS

•**Computing methodologies → Genetic programming;**

KEYWORDS

genetic programming, generalization, overfitting, automatic simplification, Push

ACM Reference format:
Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving Generalization of Evolved Programs through Automatic Simplification. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071330

GECCO '17, Berlin, Germany

978-1-4503-4920-8/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3071178.3071330

1 INTRODUCTION

Supervised machine learning algorithms are trained on data for which correct answers (e.g. classes, outputs, predictions) are known in advance, with the goal of producing a system that will subsequently also give the correct answers for new, unseen data. In the context of genetic programming (GP) [20], this means that we seek to evolve a program that will produce correct outputs when run on any valid inputs, including inputs not encountered during the evolutionary process. That is, we seek programs that *generalize* to *unseen test data*. While the challenges of learning solutions that generalize have long been recognized and studied, both in GP (e.g. [4, 8, 10, 32, 33]) and other machine learning algorithms (e.g. [5]), the challenges in this area have not yet been fully met, and significant problems remain open.

Another set of long-standing, open research questions in GP concerns the growth of programs over evolutionary time, and the presence of unnecessary code in evolved programs—see, for example, the survey [27]. Such code growth can consume computational resources, slow down the GP search process, and produce programs that are difficult for users to understand or apply.

In this paper we report on work stemming from steps taken to deal with long programs, which then unexpectedly provided substantial and robust benefits with respect to generalization. Specifically, we explore the idea of automatically simplifying evolved programs, originally developed to aid in their analysis and application [29]. When using automatic simplification in experiments, we noticed instances in which programs that did not generalize prior to simplification *did* generalize *after* simplification [11]. In this paper we systematically study the effects of automatic simplification on generalization.

The study presented here was conducted entirely on evolved Push programs, evolved with the PushGP genetic programming system [31]. Push has an unusually permissive syntax, in which any tokens of the language may appear in any order, and all possible programs produce interpretable results. This means that it is particularly easy to automatically simplify Push programs while maintaining program behavior on the training data. To do so, one can remove random tokens from programs, interpret the resulting programs, and compare program behavior before and after such modifications. All of the automatic simplification methods discussed in this paper work this way.

Automatic simplification was introduced to Push as a tool for making evolved programs easier to understand [25, 31]. Smaller solutions have other benefits as well, such as faster running times. While it can be applied to any Push program, it has typically been used only after the completion of GP runs, to make solution programs easier to understand without changing their behavior [29]. It has also been tried as a growth-control genetic operator during GP runs, with mixed success [35]. Here we consider only post-run simplification, and we focus only on its effects on code size and program generalization.

In the following section we first briefly describe related work on automatic simplification and generalization in GP. We next discuss relevant aspects of the Push programming language and our automatic simplification methods. We then describe our experimental design and results, and discuss our conclusions.

2 RELATED WORK

Simplification of evolved programs has been used for various reasons in GP, though as far as we know, never explicitly to improve generalization of an evolved solution. Some of the primary uses include making evolved programs easier to understand [20] and controlling bloat during a run [3, 6, 19, 34]. Of particular interest is work that uses simplification during a run to reduce overfitting [17, 19]. Additionally, algebraic simplification has been used to combat exponential code growth in geometric semantic genetic programming [23]. Unlike our work, most of these methods use algebraic techniques for simplification that cannot change the output of a program on any input, and therefore cannot affect the generalization to unseen data.

Various work has been done to reduce the problems of overfitting in GP [4, 8–10, 33]. In other work, an ensemble of symbolic regression datasets were created using bootstrapping [1]. Here, the variance of a individual's error on these data sets was used in combination with the error on the original data set to create a new fitness function. When this new fitness function is used to drive GP, it was shown to produce better generalizing programs. In [2] a measure of each individual's variance of output values (referred to as smoothness) is used in a modified version of tournament selection. This scheme was shown to produce better generalizing programs on symbolic regression problems.

Relations between program size and and generalization have been studied previously. Many of these studies have been conducted in the context of tree based genetic programming, which has an idiosyncratic tendency to produce program "bloat" [32]. The genetic representations and variation operators that we use here do not share these tendencies [15], so the relevance of the prior work in this area is unclear. Finally, our findings here may provide additional perspectives on discussions about the relations between model simplicity and generality, or the lack thereof, in the neural network and data mining research communities [5].

3 PUSH, PLUSH, AND PUSHGP

The Push programming language was developed specifically to express programs that evolve in GP systems [28, 30, 31]. Push is a stack-based language that runs on a virtual machine with separate stacks for each data type. It provides support for standard types, such as numbers, characters, and collections, and also for Push program code that can be dynamically executed.

Push programs have a nested structure, appearing superficially like Lisp programs, although the execution model is different. Early PushGP systems operated on this nested structure directly, mutating and recombining programs similarly to tree-based GP systems, by replacing and exchanging sub-expressions. More recently, a linear representation called Plush (the "l" is for linear) has been developed for genomes, to which uniform linear operations can be applied [15]. When Plush is being used, the individuals in the GP population are created, mutated, and recombined as linear Plush genomes, but are then translated into nested Push programs for execution (and therefore for error testing). Plush genes carry *epigenetic markers*, including a `silent` marker that prevents the translation of that gene to the Push program [21] and structural markers that close nested blocks. These aspects of the representation are important for the work described here because automatic simplification methods can be designed to operate on either Push programs or Plush genomes; for example, they could operate by removing either literals and sub-expressions in Push programs, or by removing or silencing genes in the linear Plush representation. See [15] for additional details of Push and Plush.

Algorithm 1: Automatic Simplification

```
Input: individual ind, number of simplification steps, method
       for simplification step takeSimplificationStep
errorVector ← computeErrorVector(ind) ;
for s ← 0 to steps do
    newInd ← takeSimplificationStep(ind) ;
    newErrorVector ← computeErrorVector(newInd) ;
    if newErrorVector = errorVector then
        ind ← newInd
    end
end
return ind
```

4 AUTOMATIC SIMPLIFICATION IN PUSH

Automatic simplification is a simple hill-climbing algorithm that tries to make a program smaller without changing the program's behavior on the training cases. At each step, we first make small changes to the program to make it smaller. We then check whether the smaller program produces the same error vector (sequence of errors for all test cases) as the original program. If so, we continue with the new program; otherwise, we revert to the previous program. This process repeats for a set number of simplification steps, and then returns the resulting simplified program. The algorithm is presented in detail in Algorithm 1.

In this paper we explore five different methods of automatic simplification. Each method follows the same general algorithm; they only differ in the particular steps taken to make programs smaller. Below is a description of each method:

Program simplification, which has been used in Push since its inception [25], acts on an individual's Push program, as opposed to its linear Plush genome. At each simplification step, it has an 80% chance of removing one or two random "code points" from

the program, where a *code point* is an instruction, a constant, or a parenthesized block of code. The other 20% of the time it randomly removes one set of parentheses from a code block in the program, flattening the code inside the parentheses into the level that the code block occupied.

The other four simplification methods act on an individual's linear Plush genome prior to its translation into a Push program. There are three basic steps that are used across these genome methods: silencing random genes, unsilencing random genes, and NOOPing random genes. Silencing a gene activates its "silent" epigenetic marker, preventing that instruction from appearing in the resulting Push program, as well as not opening or ending any parenthesized code blocks. When picking a random gene to silence, we never select a gene that is already silent.

Note that by silencing genes instead of removing them entirely, it becomes possible to backtrack by unsilencing previously silenced genes. We hope that the backtracking enabled by unsilencing will allow simplification to escape some program size local minima that it might otherwise be impossible to escape, resulting in more reliable simplification to smaller programs. We never unsilence a gene unless it is currently silent or NOOPed.

Finally, some methods "NOOP" genes by replacing their instructions with NOOP instructions that have no effect when executed. Some Push programs require an instruction to be present in a location, with no particular concern for that instruction's details. It is not uncommon, for example, for programs to use the number of instructions on the `exec` stack as a numeric constant without ever actually executing those instructions. We can replace these instructions with NOOPs without affecting the semantics of the program, potentially making the program more general without actually making it smaller, or possibly enabling further simplifications. Genes that are already NOOPed cannot be selected by a NOOP step of simplification.

The four genome methods of simplification are listed in Table 1. Each method lists the types of steps that can be taken and the probability of each step being chosen. **Genome** simplification, the most basic of the methods, only allows for the silencing of 1 to 4 random genes in a single step. **Genome-Backtracking** and **Genome-Backtracking-Noop** occasionally unsilence a silent gene while silencing another. **Genome-Noop** and **Genome-Backtracking-Noop** use NOOPing to turn intsructions into NOOPs without removing them entirely. We chose the probabilities for each method mostly arbitrarily, and do not believe that the precise values have much affect on our results.

Note that while the Program simplification method can remove unnecessary parentheses from a program, none of the genome methods have this ability. Since the genome methods change the genomes themselves, which have no parentheses, they cannot remove the parentheses that appear in the resulting translated program. They can silence an entire gene, which will omit its instruction and any open parentheses it requires, but they cannot simply remove a pair of extraneous parentheses.

Table 1: Genome simplification methods. Each method lists the possible single simplification steps it can take during simplification, and the probability of taking each step.

Method	Step	Prob.
Genome	Silence 1 (gene)	0.50
	Silence 2	0.30
	Silence 3	0.10
	Silence 4	0.10
Genome-Backtracking	Silence 1	0.40
	Silence 2	0.25
	Silence 3	0.10
	Silence 4	0.05
	Silence 1, Unsilence 1	0.05
	Silence 2, Unsilence 1	0.10
	Silence 3, Unsilence 1	0.05
Genome-Noop	Silence 1	0.40
	Silence 2	0.25
	Silence 3	0.10
	Silence 4	0.05
	Noop 1	0.10
	Noop 2	0.10
Genome-Backtracking-Noop	Silence 1	0.30
	Silence 2	0.20
	Silence 3	0.10
	Silence 1, Unsilence 1	0.05
	Silence 2, Unsilence 1	0.10
	Silence 3, Unsilence 1	0.05
	Noop 1	0.10
	Noop 2	0.10

5 EXPERIMENTAL DESIGN

In our experiments, we aim to answer several key questions. First, does automatic simplification effectively improve the generalization of evolved programs that pass every training case? Further, which of the simplification methods described in the previous section produces the smallest programs, which generalizes the best, and is there correlation between getting smaller and generalizing better?

To examine these questions, we conducted simplification experiments using evolved solution programs from a suite of general program synthesis benchmark problems [14]. This benchmark suite is composed of 29 general programming problems taken from introductory computer science homework assignments. These problems require solution programs to utilize a wide range of programming constructs, such as multiple data types, control flow structures, and multiple inputs/outputs. After the original publication of these benchmark problems [14], they have seen use in a range of studies using PushGP (eg. [11–13, 22]), as well as work considering a general program synthesis grammar for grammar guided genetic programming [7].

For these problems, we are primarily interested in whether we can find a program that passes every training case and unseen test case, since programs that do not achieve at least this level of generalization do not represent true solutions to the problems [14]. We will call a program that passes all of the training set a "solution", and a program that additionally passes all of the unseen test set a "generalizing solution".

We took our evolved solution programs, which we will henceforth call "unchanged" programs to distinguish them from their simplified versions, from the original set of benchmarking runs using the benchmark suite [14]. In those original runs, each problem was attempted 100 times each for several selection methods; in this work, we only use the 100 runs that used lexicase selection, as it consistently had the highest success rates and provided solutions to the most problems. In these runs, PushGP created at least one generalizing solution to 22 of the problems. In addition, we include one problem (Super Anagrams) for which 14 programs were evolved that passed the training set but did not pass the unseen test set. We also include the Checksum problem, which was not solved in the runs for the original publication, but was subsequently solved with some additions to the training set.

In earlier work, it was shown that simplifying the same program many times can result in a range of resulting program sizes [29]. To account for this variance in the simplified programs, we performed multiple simplifications of each unchanged program. For each of our five simplification methods, we conducted 100 separate simplification trials of each unchanged program, recording the size and generalization of each simplified program. Push program sizes are the sum of the number of literals and nested blocks they contain. Some of the unchanged programs also entirely passed the unseen test set (i.e. generalized), and some did not (i.e. did not generalize). When running GP on real-world problems, one will not know whether an evolved solution program generalizes or not before using simplification. Thus we are interested in not only what happens to evolved programs that do not generalize, but also to those that already generalize. For example, it would not be beneficial to improve generalization of some programs while breaking generalization of many more.

For each simplification trial, we simplified the unchanged program using 10,000 steps of the simplification algorithm (see Algorithm 1). While this number of steps is often far more than necessary to effectively simplify most programs, the simplification process is not prohibitively expensive. While every simplification step requires reevaluating the program, 10,000 steps is equivalent to the number of evaluations used in 10 generations of GP with a population of 1,000. Since we are advocating for one simplification at the end of a GP run, this computational effort seems reasonable compared to GP runs of hundreds of generations.

The benchmark suite we used prescribes using randomly generated training and test cases for most of the input and output data. Since automatic simplification requires use of training data, we use the training and test sets that were used for the original run for every simplification of the solution program from that run.

6 RESULTS

We first consider the the impact of simplification on the size of each solution program. Figure 1a presents the average program size of the unchanged solution programs on each problem as a horizontal black line. It then gives the average simplified size of the simplified solution programs, with 100 trials per solution, for each of our five simplification methods. It is immediately clear that every simplification method has a large impact on the average size of the solution programs, with most shrinking to half their original size or smaller. In fact, the size differences resulting from the various simplification methods are all quite small when compared to the difference between the simplified sizes and the unchanged programs' sizes.

Table 2: The average rank in size for each simplification method across the data in Figure 1a, where lower rank means smaller programs. "Unchanged" is the rank of the evolved programs without any simplification.

Method	Average Rank
Program	1.87
Genome-Backtracking-Noop	2.13
Genome-Backtracking	2.79
Genome-Noop	3.60
Genome	4.60
Unchanged	6.00

In Table 2 we examine aggregate performance of the simplification methods by calculating the average rank of program size for each method across the 24 problems, with rank 1 indicating the smallest average program size and rank 6 having largest. Note that we include the unchanged programs as one "method", which is why we have 6 possible ranks. Program simplification achieves the smallest programs on average, with both of the Genome backtracking methods coming soon after. The genome methods without backtracking have worse average ranks, with the unchanged programs always having worst rank. Since parentheses are counted as part of the size of a program, and since the Genome methods cannot remove extraneous parentheses (as mentioned in Section 4), we hypothesize that Program simplification is able to achieve smaller sizes largely based on removing such parentheses.

The Friedman test on the data in Table 2 gives a p-value < 0.001, indicating that at least one method performs significantly differently from the others. We then use a post-hoc Wilcoxon-Nemenyi-McDonald-Thompson test [16] to give the significance in the differences in ranking between each pair of methods at the 0.05 significance level. Every simplification method besides Genome significantly outranks the Unchanged programs. Program also outranks Genome-Noop and Genome; and both Genome backtracking methods outrank Genome. Note that even though these results show significant differences in rank among the methods, most of the actual differences in average size are relatively small compared to the sizes of the unchanged programs.

Next, we consider the effect of simplification on the generalization of programs to unseen test data. In Figure 1b, we plot the percent of programs that generalize for each simplification method, as well as a horizontal bar representing the percent of unchanged programs that generalize. Every simplification method either improves generalization or has no effect on every problem besides Double Letters. Some of the increases in generalization appear substantial, improving as much as 25% in the case of Median.

Also note that the five problems with the worst generalization in Figure 1b are among the problems for which program sizes were largest post-simplification in Figure 1a. For whatever reason, programs that solve these problems tend to be large, unable to shrink,

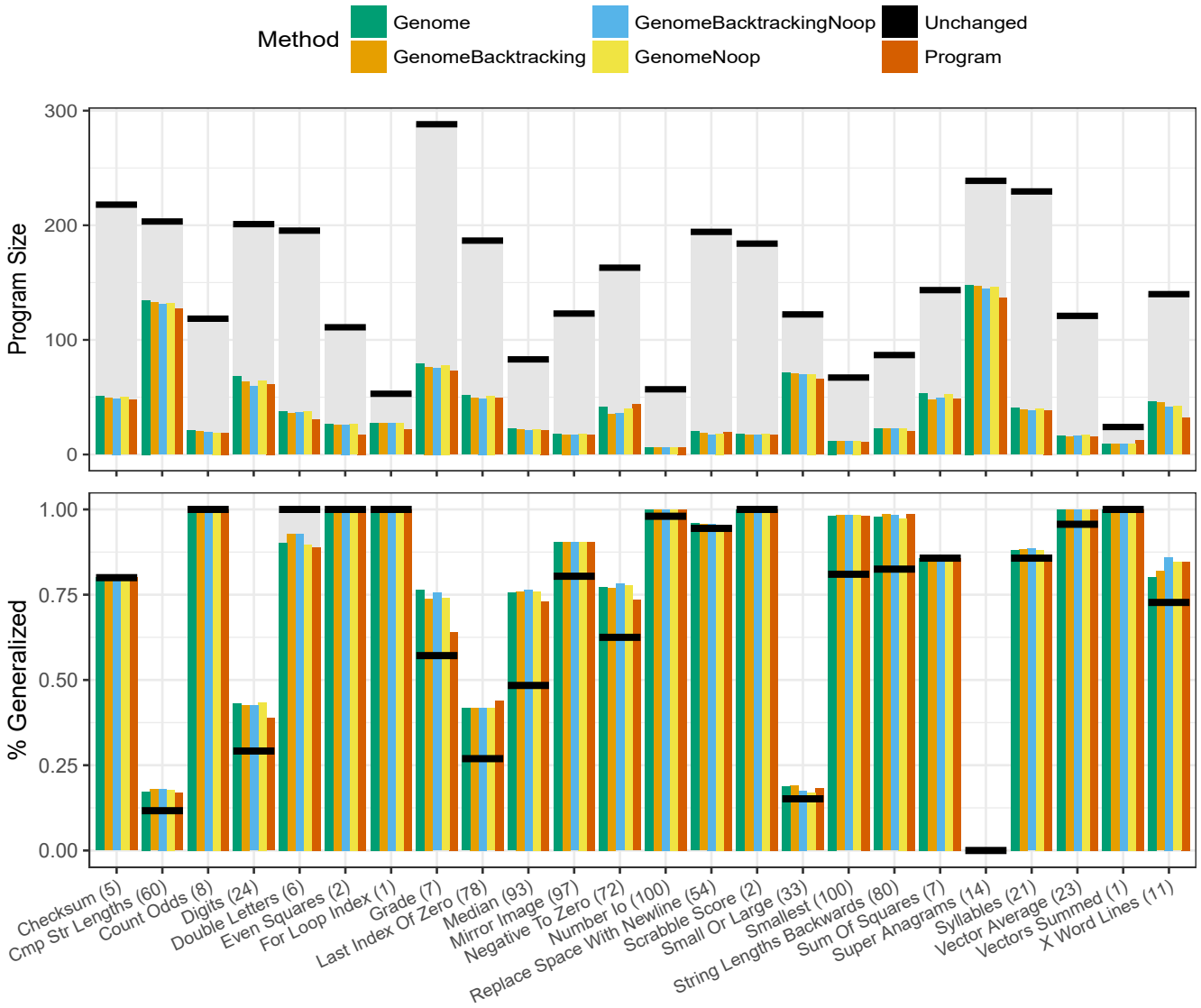

Figure 1: (a) The average program size of simplified programs and (b) proportion of simplified programs that generalize to unseen test data for each of our five simplification methods. The unchanged solution program averages are represented with horizontal black bars. Each simplification method performs repeated trials using the same set of unchanged programs, and therefore has the same opportunities for simplification as the other methods. The number of unchanged solution programs included for each problem is given in parentheses, representing the number of starting points for each problem.

and do not generalize well. We hypothesize that large programs such as these contain many ad-hoc components that overfit to individual test cases without really "learning" the underlying problem structure, and therefore do not generalize to unseen data.

As with average program sizes, there do not appear to be large differences between the simplification methods in terms of generalization. To see if the small differences do affect rank, we calculated the average rank of each method across all problems. Here, rank 1 signifies the best (highest) generalization, and rank 6 represents the worst generalization. These average ranks are presented in Table 3.

For generalization, the differences in rank for the simplification methods are not as pronounced as with simplified sizes. Note that Program simplification, while producing the smallest programs, has the worst generalization of the simplification methods; all of the Genome-based methods remain in the same order. The Friedman test on the data in Table 3 gives a p-value < 0.001, indicating that at least one method performs significantly differently from the others. We then use a post-hoc Wilcoxon-Nemenyi-McDonald-Thompson test [16] to give the significance in the differences in ranking between each pair of methods at the 0.05 significance

Table 3: The average rank in generalization for each simplification method across the problems in Figure 1b, where lower rank means better generalization. "Unchanged" is the rank of the evolved programs without any simplification.

Method	Average Rank
Genome-Backtracking-Noop	2.73
Genome-Backtracking	3.02
Genome-Noop	3.29
Genome	3.33
Program	3.67
Unchanged	4.96

Table 4: This table records every single simplified program across all problems. The top row corresponds to unchanged programs that did not generalize, where the bottom row is those that did generalize. The left column contains all simplified programs that did not generalize, and the right column has those that did. For example, of the programs that did not originally generalize, 53,787 of their simplifications do generalize while 102,617 do not. % Pre and % Post give the percent of programs that generalized pre- and post-simplification respectively.

	Generalizes Post-Simplification		
	No	Yes	% Pre
Generalizes Pre: No	102,617	53,787	35%
Generalizes Pre: Yes	3,735	289,265	65%
% Post	24%	76%	

level. Here, every simplification method significantly outranks Unchanged. This shows that any of these methods can be used to improve generalization. But, there is not a significant difference in generalization rank between any of the simplification methods. Thus the average rank of each method is approximately the same, which is unsurprising considering their similarities in Figure 1b.

Table 4 gives the total count of each combination of programs that did/did not generalize pre-simplification with did/did not generalize post-simplification. Considering the top row of unchanged programs that did not generalize, over 33% generalized after simplification. On the other hand, out of the starting programs that did generalize before simplification (the bottom row), only about 1.2% of the simplifications broke them so that they did not generalize. This gives strong evidence that simplification can be used to make evolved programs more likely to generalize, without a large risk of breaking programs that already generalize. In total, 65% of the unchanged programs generalized before simplification; that number rises to 76% after simplification, an increase in generalization of 11 percentage points.

We next want to explore how size after simplification corresponds to the generalization of simplified programs. Figure 2 plots the counts of the sizes of the simplified programs that started with unchanged programs that did not generalize, aggregated across

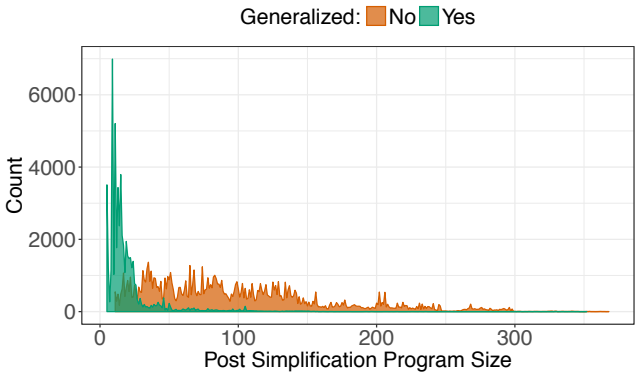

Figure 2: Counts of post-simplification program sizes for cases where the original program did not generalize. Orange (no) are programs that continued to not generalize after simplification; green (yes) are programs that do generalize after simplification. The programs here correspond to the top row in Table 4.

all problems. This figure clearly shows that when the resulting simplified program was relatively small (with size less than about 25), it was much more likely to change from ungeneralizing to generalizing, as seen by the early spike in the "yes" counts. On the other hand, programs that remained larger after simplification were more likely to remain ungeneralizing, shown by the "no" counts. Thus, most of the improvements in generalization that we described previously come from simplifications that achieve small program sizes. This clearly shows that there is a correlation, if not causation, between post-simplification size and the probability of generalization.

7 DISCUSSION

As noted earlier, all five simplification methods lead to substantial reductions in program size across all 24 test problems, as shown in Figure 1a, with the average simplified sizes often well under half the original sizes, and sometimes much smaller than that.

Further, all the simplification methods improved generalization rates for all but a few problems (see Figure 1b), but again with little difference among the simplification methods. In some cases, e.g., Count Odds, all the unsimplified programs already generalized, so the best simplification could do is to not make things worse. In other cases, such as Checksum, Double Letters, and Vectors Summed, there were very few solution programs generated in the initial 100 PushGP runs (5, 6, and 1, respectively). With so few data points to work with it's not surprising that in some cases (e.g., Checksum and Vectors Summed) there was no change in generalization. In the case of Double Letters, five of the six initial solutions were consistently simplified without affecting generalization. The sixth solution, however, frequently led to simplifications that no longer generalized. Only 37 of the 100 Program simplification results, for example, generalized, while only 45 of the 100 Genome simplifications generalized.

Our attempts at improving on simplification of Push programs by simplifying genomes, including with backtracking and NOOPs,

seems mostly for naught. Program simplification led to the smallest programs, though as mentioned, this might be due to its ability to remove extraneous parentheses as opposed to potentially overfit code. On the other hand, Genome-Backtracking-Noop simplification had the best rank on generalization, though not significantly so, and had the second smallest rank on simplified size of programs. The backtracking ability of this method may have allowed it to avoid some locally optimal simplifications. We have anecdotally noticed an example where Program simplification is not able to escape such a local optima, and Genome-Backtracking-Noop might have made improvements in such cases. Adding NOOPs seems to have a lesser effect, since the Genome-Noop method produced larger programs than Genome-Backtracking, yet both were smaller than Genome without either enhancement.

Despite all of these minor differences, all simplification methods performed rather similarly across the board, with all showing major improvements over not using simplification. Thus, the choice of using any simplification method seems like the more significant decision than which simplification method to use.

Figure 2 makes it clear that smaller post-simplification sizes were strongly correlated with the ability to generalize after simplification. This is consistent with a broad range of work that either argues theoretically (e.g., the minimum description length principle [18, 36]) and/or empirically that smaller programs will be more likely to generalize [26]. While there have been contrary results that show that program size and generalization are not always correlated [27, 32], the correlation seems extremely strong in our data.

The relationship between program size and generalization is almost certainly driven at least in part by both the problem being solved and the representation being used for solutions. It's possible, therefore, that the strong correlation we see is in some way related to the kinds of software synthesis problems we're using as benchmarks, which clearly behave differently from other common application areas such as symbolic regression and classification.

It is also possible that design decisions in PushGP play a role here. Any attempt to control or reduce program size in GP is premised on the idea that there must be some parts of the program that aren't playing an important role and can thus be removed; such code has been called many things over the years, such as "introns" or "bloat". The fact that the evolved PushGP solutions were often substantially larger than necessary is arguably not attributable to "bloat" as it's typically been understood [24], since in PushGP we don't see a general tendency towards increasing program size over time. There is obviously "unnecessary" code in these evolved Push programs, but understanding the source and role of this removable Push code is complex, in part because there are potentially many categories of unused or unnecessary code in Push. Examples include:

- Instructions that do nothing because they require arguments that are not on the appropriate stacks when they are executed.
- Instructions that do *something*, but not something that has an impact on the final result because, e.g., they act on values on stacks that play no role in the key computation.
- Instructions whose *presence* is important (e.g., as a marker or to make sure the exec stack has the right depth), but whose details or *behavior* is not.

Note also that the "activity" of many of these instructions is highly dependent on context. An instruction that does nothing in a parent because the arguments it needs aren't available, might play a prominent role in a child if a change "upstream" causes those arguments to now be available.

8 CONCLUSIONS AND FUTURE WORK

The results of our experiments show, across a range of software synthesis benchmark problems, that smaller programs evolved by PushGP tend to generalize better than larger programs. The results also show that evolved programs can be automatically simplified using a variety of simple hill-climbing procedures, and that simplified programs tend to generalize better than unsimplified programs. Furthermore, programs that can be successfully simplified (that is, that the simplification procedure can make much smaller) are more likely to show improved generalization after simplification. The differences between our simplification methods were not significant, but significant improvements were produced by all of them.

The recommendation, therefore, for users of PushGP is to always perform automatic simplification on the results of evolution, using any of the methods described here. There is a reasonable chance that doing so will improve the generalization of evolved programs, and a much smaller chance that it will hurt it. If a program is substantially smaller after automatic simplification, then it will have an even better chance of generalizing.

Would the same advice apply to users of other types of GP systems? For some systems, such as tree-based GP with non-numeric functions, automatic simplification based on hill-climbing may be non-trivial. Our methods might be applied more easily to other forms of GP, including grammatical evolution, Cartesian GP, and linear GP systems, in which one can remove single instructions without breaking the program entirely.

An extension of the work presented here would be to run the same tests using different types of problems, such as classification and symbolic regression. Another area for future work is to improve the automatic simplification methods themselves. While we presented results with several simplification methods, all of them used a simple hill-climbing search procedure that can get stuck in local minima, even when using backtracking steps. A multi-start hillclimber, for example, might simplify programs more reliably.

ACKNOWLEDGMENTS

Thanks to the members of the Hampshire College Computational Intelligence Lab for discussions that helped shape this work and to Josiah Erikson for systems support. This material is based upon work supported by the National Science Foundation under Grants No. 1617087, 1129139 and 1331283. 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.

REFERENCES

[1] Alexandros Agapitos, Anthony Brabazon, and Michael O'Neill. 2012. Controlling Overfitting in Symbolic Regression Based on a Bias/Variance Error Decomposition. In *Parallel Problem Solving from Nature, PPSN XII (part 1) (Lecture Notes in Computer Science)*, Vol. 7491. Springer, Taormina, Italy, 438–447. DOI: http://dx.doi.org/doi:10.1007/978-3-642-32937-1_44

[2] R. Muhammad Atif Azad and Conor Ryan. 2011. Variance based selection to improve test set performance in genetic programming. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, Dublin, Ireland, 1315–1322. DOI: http://dx.doi.org/doi:10.1145/2001576.2001754

[3] Markus Brameier and Wolfgang Banzhaf. 2001. A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. *IEEE Transactions on Evolutionary Computation* 5, 1 (Feb. 2001), 17–26. http://web.cs.mun.ca/~banzhaf/papers/ieee_taec.pdf

[4] Mauro Castelli, Luca Manzoni, Sara Silva, and Leonardo Vanneschi. 2010. A comparison of the generalization ability of different genetic programming frameworks. In *IEEE Congress on Evolutionary Computation (CEC 2010)*. IEEE Press, Barcelona, Spain. DOI: http://dx.doi.org/doi:10.1109/CEC.2010.5585925

[5] Pedro Domingos. 2016. *Master Algorithm*. Penguin Books.

[6] Aniko Ekart. 2000. Shorter Fitness Preserving Genetic Programs. In *Artificial Evolution. 4th European Conference, AE'99, Selected Papers (LNCS)*, C. Fonlupt, J.-K. Hao, E. Lutton, E. Ronald, and M. Schoenauer (Eds.), Vol. 1829. Dunkerque, France, 73–83. http://www.sztaki.hu/~ekart/ea.ps

[7] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *20th European Conference on Genetic Programming*. In press.

[8] Ashley George and Malcolm I. Heywood. 2006. Improving GP classifier generalization using a cluster separation metric. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Vol. 1. ACM Press, Seattle, Washington, USA, 939–940. DOI: http://dx.doi.org/doi:10.1145/1143997.1144159

[9] Ivo Goncalves and Sara Silva. 2013. Balancing Learning and Overfitting in Genetic Programming with Interleaved Sampling of Training data. In *Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013 (LNCS)*, Vol. 7831. Springer Verlag, Vienna, Austria, 73–84. DOI: http://dx.doi.org/doi:10.1007/978-3-642-37207-0_7

[10] Ivo Goncalves, Sara Silva, and Carlos M. Fonseca. 2015. On the Generalization Ability of Geometric Semantic Genetic Programming. In *18th European Conference on Genetic Programming (LNCS)*, Vol. 9025. Springer, Copenhagen, 41–52. DOI: http://dx.doi.org/doi:10.1007/978-3-319-16501-1_4

[11] Thomas Helmuth. 2015. *General Program Synthesis from Examples Using Genetic Programming with Parent Selection Based on Random Lexicographic Orderings of Test Cases*. Ph.D. dissertation. University of Massachusetts, Amherst. http://scholarworks.umass.edu/dissertations_2/465/

[12] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2015. Lexicase Selection For Program Synthesis: A Diversity Analysis. In *Genetic Programming Theory and Practice XIII (Genetic and Evolutionary Computation)*. Springer, Ann Arbor, USA. DOI: http://dx.doi.org/doi:10.1007/978-3-319-34223-8

[13] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2016. The Impact of Hyperselection on Lexicase Selection. In *GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, Tobias Friedrich (Ed.). ACM, Denver, USA, 717–724. DOI: http://dx.doi.org/doi:10.1145/2908812.2908851

[14] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*. ACM, Madrid, Spain, 1039–1046. DOI: http://dx.doi.org/doi:10.1145/2739480.2754769

[15] Thomas Helmuth, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook. 2016. Linear Genomes for Structured Programs. In *Genetic Programming Theory and Practice XIV (Genetic and Evolutionary Computation)*. Springer, Ann Arbor, USA.

[16] M. Hollander and D.A. Wolfe. 1999. *Nonparametric Statistical Methods*. Wiley.

[17] Dale Hooper and Nicholas S. Flann. 1996. Improving the Accuracy and Robustness of Genetic Programming through Expression Simplification. In *Genetic Programming 1996: Proceedings of the First Annual Conference*. MIT Press, Stanford University, CA, USA, 428. http://digital.cs.usu.edu/~flann/gp.pdf

[18] Hitoshi Iba, Hugo De Garis, and Taisuke Sato. 1994. Genetic programming using a minimum description length principle. *Advances in genetic programming* 1 (1994), 265–284.

[19] David Kinzett, Mengjie Zhang, and Mark Johnston. 2010. Investigation of simplification threshold and noise level of input data in numerical simplification of genetic programs. In *IEEE Congress on Evolutionary Computation (CEC 2010)*. IEEE Press, Barcelona, Spain. DOI: http://dx.doi.org/doi:10.1109/CEC.2010.5586181

[20] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[21] William La Cava and Lee Spector. 2014. Inheritable Epigenetics in Genetic Programming. In *Genetic Programming Theory and Practice XII (Genetic and Evolutionary Computation)*. Springer, Ann Arbor, USA, 37–51. DOI: http://dx.doi.org/doi:10.1007/978-3-319-16030-6_3

[22] Nicholas Freitag McPhee, Mitchell Finzel, Maggie M. Casale, Thomas Helmuth, and Lee Spector. 2016. A detailed analysis of a PushGP run. In *Genetic Programming Theory and Practice XIV (Genetic and Evolutionary Computation)*. Springer, Ann Arbor, USA.

[23] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric Semantic Genetic Programming. In *Parallel Problem Solving from Nature, PPSN XII (part 1) (Lecture Notes in Computer Science)*, Vol. 7491. Springer, Taormina, Italy, 21–31. DOI: http://dx.doi.org/doi:10.1007/978-3-642-32937-1_3

[24] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. http://www.gp-field-guide.org.uk (With contributions by J. R. Koza).

[25] Alan Robinson. 2001. *Genetic Programming: Theory, Implementation, and the Evolution of Unconstrained Solutions*. Division III thesis. Hampshire College. http://hampshire.edu/lspector/robinson-div3.pdf

[26] Justinian Rosca. 1996. Generality Versus Size in Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*. MIT Press, Stanford University, CA, USA, 381–387. ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/96.gp.ps.gz

[27] Sara Silva, Stephen Dignum, and Leonardo Vanneschi. 2012. Operator equalisation for bloat free genetic programming and a survey of bloat control methods. *Genetic Programming and Evolvable Machines* 13, 2 (2012), 197–238.

[28] Lee Spector. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. Morgan Kaufmann, San Francisco, California, USA, 137–146. http://hampshire.edu/lspector/pubs/ace.pdf

[29] Lee Spector and Thomas Helmuth. 2014. Effective simplification of evolved push programs using a simple, stochastic hill-climber. In *GECCO Comp '14: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*. ACM, Vancouver, BC, Canada, 147–148. DOI: http://dx.doi.org/doi:10.1145/2598394.2598414

[30] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM Press, Washington DC, USA, 1689–1696. DOI: http://dx.doi.org/doi:10.1145/1068009.1068292

[31] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (March 2002), 7–40. DOI: http://dx.doi.org/doi:10.1023/A:1014538503543

[32] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2010. Measuring bloat, overfitting and functional complexity in genetic programming. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, Portland, Oregon, USA, 877–884. DOI: http://dx.doi.org/doi:10.1145/1830483.1830643

[33] Leonardo Vanneschi and Steven Gustafson. 2009. Using crossover based similarity measure to improve genetic programming generalization ability. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, Montreal, 1139–1146. DOI: http://dx.doi.org/doi:10.1145/1569901.1570054

[34] Phillip Wong and Mengjie Zhang. 2006. Algebraic simplification of GP programs during evolution. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Vol. 1. ACM Press, Seattle, Washington, USA, 927–934. DOI: http://dx.doi.org/doi:10.1145/1143997.1144156

[35] Haoxi Zhan. 2014. A quantitative analysis of the simplification genetic operator. In *GECCO 2014 student workshop*, Tea Tusar and Boris Naujoks (Eds.). ACM, Vancouver, BC, Canada, 1077–1080. DOI: http://dx.doi.org/doi:10.1145/2598394.2605684

[36] Byoung-Tak Zhang and Heinz Mühlenbein. 1995. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation* 3, 1 (1995), 17–38.