A Comparison of Semantic-Based Initialization Methods for Genetic Programming

Hammad Ahmad
Washington and Lee University
Lexington, Virginia, USA
ahmadh19@mail.wlu.edu

Thomas Helmuth
Hamilton College
Clinton, New York, USA
thelmuth@hamilton.edu

ABSTRACT

During the initialization step, a genetic programming (GP) system traditionally creates a population of completely random programs to populate the initial population. These programs almost always perform poorly in terms of their total error—some might not even output the correct data type. In this paper, we present new methods for initialization that attempt to generate programs that are somewhat relevant to the problem being solved and/or increase the initial diversity (both error and behavioral diversity) of the population prior to the GP run. By seeding the population—and thereby eliminating worthless programs and increasing the initial diversity of the population—we hope to improve the performance of the GP system. Here, we present two novel techniques for initialization (Lexicase Seeding and Pareto Seeding) and compare them to a previous method (Enforced Diverse Populations) and traditional, non-seeded initialization. Surprisingly, we found that none of the initialization methods result in significant differences in problem-solving performance or population diversity across five program synthesis benchmark problems. We attribute this lack of difference to our use of lexicase selection, which seems to rapidly converge on similar populations regardless of initialization method.

CCS CONCEPTS

• **Computing methodologies → Genetic programming**;

KEYWORDS

genetic programming, initialization, diversity, program synthesis

ACM Reference Format:
Hammad Ahmad and Thomas Helmuth. 2018. A Comparison of Semantic-Based Initialization Methods for Genetic Programming. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3205651.3208218

1 INTRODUCTION

Many initialization methods in genetic programming (GP) concentrate on issues specifically related to tree generation. For example, ramped half-and-half and Sean Luke's initialization methods attempt to design a diverse range of trees [9, 10]. These methods are not applicable to GP that is not based on trees, such as the stack-based GP we use in this work.

On the other hand, semantic-based methods of initialization only take into account a program's behavior, not its structure when determining whether to allow it in the initial population. We define the *behavior* of a program to be the vector of its outputs produced when the program is run on each of the inputs. Then, the *behavioral diversity* of a population is the percent of distinct behaviors of individuals in the population. Similarly, an individual's *error vector* is computed by applying the error function to its behavior vector; a population's *error diversity* is the percent of distinct error vectors in the population. We hope that increasing initial population diversity will contribute to increases in problem solving.

In one example of semantic-based initialization, Jackson describes a method of iteratively creating new individuals until one is found with different behavior from those already produced, up to some set number of trials [8]. This approach, which tries to significantly increase behavioral diversity in the initial population, showed significant improvements in performance over a range of simple problems compared to ramped half-and-half. We use a variant of this method, Enforced Diverse Populations, in this work.

Here we present two novel population initialization methods and compare them to Enforced Diverse Populations and to traditional random initialization. Both of our new methods attempt to seed the population with individuals that are diverse but also perform relatively well.

In the following section, we describe each of the seeding methods we use in our experiments, followed by a description of the experiments. In Section 4, we give our results, which show that none of our seeding methods have much impact on problem solving performance. To explain this finding, we look at the average population diversity across our sets of runs, and find that while the methods start with wildly different populations, they quickly converge on similar levels of diversity within a few generations. We attribute this convergence to our use of lexicase parent selection, which has previously been shown to collapse populations following hyperselection events, where a single individual receives many of the selections in a generation [4]. We hypothesize that hyperselection events are causing populations to quickly converge in early generations, regardless of initialization method.

2 POPULATION SEEDING METHODS

We test three different methods of population seeding, each unique in the way it initializes the population for the GP runs. All methods attempt to filter out individuals with large errors and increase the error diversity of the initial population. The methods are described in further detail in the sub-sections below.

Each of these methods requires extra computation to evaluate individuals to decide which to include in the initial population. To be fair and comparable to regular non-seeded populations, we account for the extra evaluations made during the seeding process by decreasing the maximum number of generations allowed for the GP run. For example, with a population of 1000 individuals, we decrease the generations by one for every 1000 individuals we evaluate during initialization.

2.1 Lexicase Seeding

Our first method of population seeding is based on lexicase parent selection (see Section 3.3). The way this algorithm works is as follows. A random pool of 100 individuals is generated. A single individual is selected from this pool using lexicase selection and added to a "return list." This individual is removed from the pool (to prevent the same individual being selected multiple times) and the process is repeated until 10 different individuals have been selected. These individuals are chosen to be part of the initial population. The entire process is repeated $0.1 \times$ population-size times, and an initial population is returned. This initial population is then used as a starting point for the genetic programming run.

2.2 Enforced Diverse Populations

Enforced Diverse Populations creates an initial population that must have an error diversity of 1.0 (i.e., all individuals in this population have a unique error vector). A variant of this approach was first described by Jackson [8]. To accomplish this, we iteratively generate new individuals, and only add them to the pool if they do not have the same error vector as some individual already in the pool. The process is repeated until there is a complete population (with a 100 percent error diversity), which is then used for the initial population.

Note that this method will evaluate some number of individuals that cannot be predetermined; thus each run using it will have a dynamically calculated maximum number of generations. While this process could theoretically require an extremely high number of evaluations to find the initial population, in practice we found that it used at most around 5 to 10 generations worth of evaluations.

2.3 Pareto Seeding

Finally, we implement Pareto Seeding, which selects individuals using Pareto tournaments to seed the initial population. Given a set of fitness cases, an individual Pareto dominates another if the former does not perform worse than the latter in all fitness cases, and the former performs better in at least one fitness case [11]. In Pareto tournament selection, individuals on the Pareto front (i.e., individuals not dominated by others) are all selected as parents [12]. We adapt this idea to create a pool of random individuals, and use the Pareto dominance test to look for individuals that are not dominated in the pool. From each pool of 100 individuals, all of the non-dominated individuals are selected to be in the initial population. This process is repeated with a different pool of a random individuals, until a population is initialized. With this algorithm, each individual in the population is a non-dominated individual in the sub-population from which it was obtained.

3 EXPERIMENTAL METHODS

This section describes our experimental methods.

3.1 Push and PushGP

In our experiments, we use the stack-based language Push as our representation for the evolving programs. Push was designed specifically for use in genetic programming systems [13, 14]. Push uses one stack per data type, and takes arguments from stacks and returns results to stacks. Additionally, a Push program can manipulate its own code as it executes, which allows for various control structures including conditionals, loops, and recursion among more exotic possibilities.

While we use PushGP in our study, none of our experiments are predicated on the use of a particular representation. We believe the results are relevant to tree-based GP and any other GP representation. We chose to use PushGP because its multiple data types and control flow structures make it a good choice for the program synthesis problems we use as benchmarks.

3.2 Benchmark Problems

To determine the effects of our initialization methods, we conduct GP runs on five program synthesis benchmark problems taken from a recent benchmark suite [5]. This set of benchmark problems is designed to contain problems with a range of difficulties and requirements. Every problem is taken from an introductory computer science textbook, and thus requires GP to do the type of programming we expect of humans. From this suite we choose five representative problems: Count Odds, Double Letters, Replace Space with Newline, String Lengths Backwards, and Syllables.

3.3 Other parameters

Lexicase selection is a recent semantic-based parent selection method that is based on individual test cases and sequences of test cases, rather than an aggregate fitness as in most other parent selection methods [6]. During each lexicase selection event, the test cases are shuffled randomly. Then, they are considered one at a time. For each test case, we remove any individuals that do not have the best error value of any remaining individuals. This process is repeated until only a single individual remains, which is selected. Lexicase selection has led to improvements in problem-solving performance in a variety of systems and on a variety of problems [1, 5, 6]. One factor that likely contributes to lexicase selection's success is its ability to increase and maintain population diversity better than methods such as tournament selection [2, 3, 6].

Besides parent selection, we use fairly typical GP parameters, such as a population size of 1000 and a maximum generations of 300. Keep in mind that the population seeding techniques use evaluations, which we subtract from the 300 available generations to make comparisons equitable. We use PushGP's crossover and mutation operators, known as alternation and uniform mutation operators [7]. Here, we use 20% alternation, 20% uniform mutation, 10% uniform close mutation, and 50% alternation followed by uniform mutation.

Table 1: Success rates across problems. For problems, RSWN is Replace Space With Newline and SLB is String Lengths Backwards. For methods, *None* indicates no seeding (i.e. entirely random individuals), *Enforce* is Enforced Diverse Populations, and *Pareto* is Pareto Seeding.

Problem	None	Lexicase	Enforce	Pareto
Count Odds	8	6	10	4
Double Letters	6	3	5	4
RSWN	51	53	54	51
SLB	66	68	62	61
Syllables	18	20	19	18

4 RESULTS

The number of successful programs out of 100 is given in Table 1. A pairwise χ^2 test indicates that none of the differences in these success rates are significant at the 0.05 level, which is not surprising since they are all so close. Thus none of our initialization methods had any significant effect on the performance of the system.

In Figures 1 through 4, we plot error diversities of the GP runs with the problems (we left Count Odds, which looked largely similar to the others, off for space reasons). It appears that the diversity for the GP runs drops significantly during the first generations. We believe that due to hyperselection caused by lexicase selection, a small subset of the programs ends up dominating the whole population during the first generation, which causes a steep drop in diversity [4]. This is most evident in Enforced Diverse Population runs, which all start with a diversity of 1.0, yet drop to the same levels as the other methods in the first generation. Although the diversities eventually recover as the GP runs proceed, the initial drops tend to render the differences in initialization futile.

We also see that the diversity graphs for all methods run virtually parallel, with insignificant fluctuations between generations. This is reflected in the very similar success rates for the GP runs using different initialization methods shown in Table 1, all of which are comparable to the success rates for non-seeded populations. These signs point toward the differences between initializations being nullified after the first generation because of hyperselection. Figure 5 zooms in to show the drop in diversity for Double Letters and the gradual recovery from the drop as the GP run proceeds.

Our hypothesis is that the similarity in diversities can be largely attributed to the fact that lexicase tends to discard the semantic-based initialization in the first generation through the process of hyperselection. We believe that the initialization methods result in populations in which a small group of programs dominates the entire population. Since lexicase selection tends to select individuals that are very good on a subset of the fitness cases, even if they perform poorly on other fitness cases, such initialized populations consist of individuals that are heavily favored by lexicase [6]. The enormous drops in error diversity graphs for the GP runs are in line with our conjecture.

We interpret these results to mean that although behavioral and error diversity should, in theory, contribute to increases in problem solving in Genetic Programming, such diversity induced through

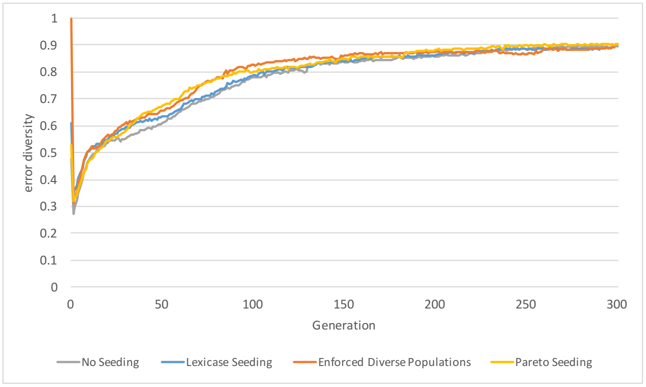

Figure 1: Error Diversity of RSWN

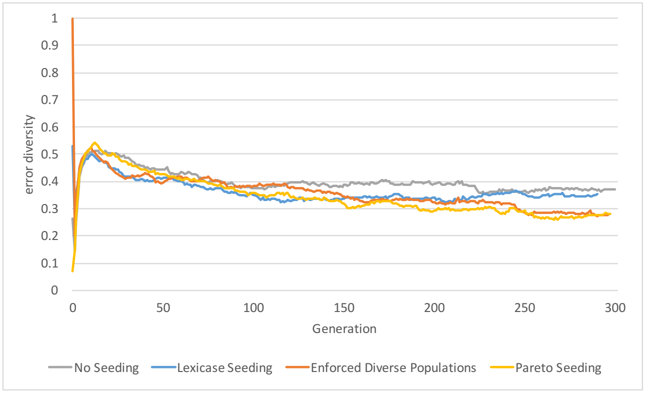

Figure 2: Error Diversity of String Lengths Backwards

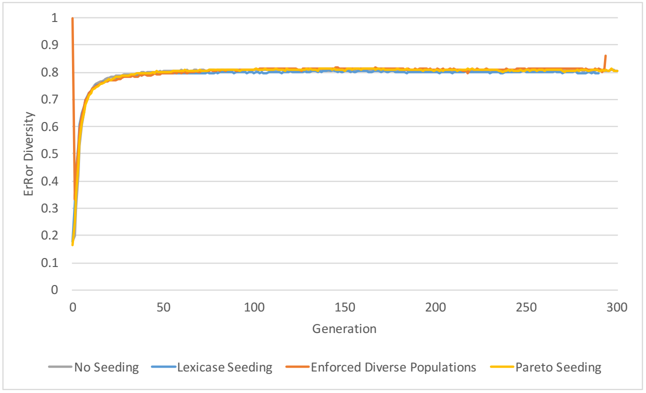

Figure 3: Error Diversity of Count Odds

semantic-based initialization methods is not of much use when lexicase is the parent selection method.

5 DISCUSSION AND CONCLUSIONS

Contrary to what we had hoped initially, the number of solutions found (with zero error on test set) is not significantly different from the original non-seeded populations. Any differences appear to

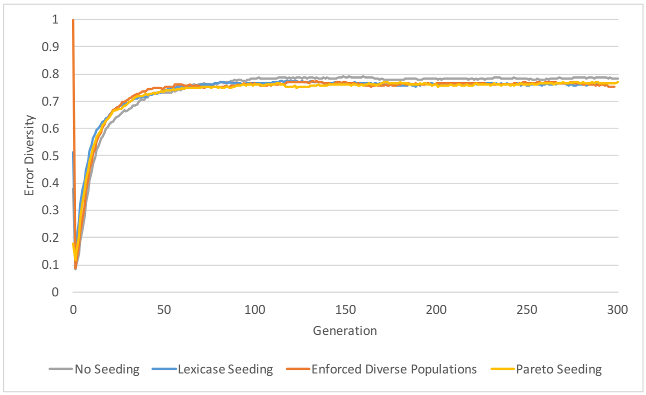

Figure 4: Error Diversity of Double Letters

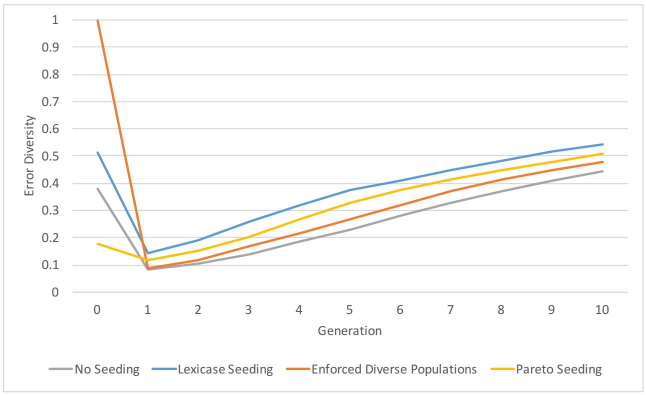

Figure 5: Diversity Drop shown in the first generation of Double Letters. This gives the first 10 generations of the same data as Figure 4.

be solely due to the random nature of the GP runs. As is clearly evident in the graphs, the error diversity for the GP runs is not significantly affected by any form of seeding. It appears as though lexicase "discards" any form of initial diversity after the population is initialized, and works for the first 10-20 generations on its own to build diversity that it finds useful. The results also show that lexicase results in hyperselection for the first few generations [4]. In other words, a small group of individuals completely dominate the population due to their low error values. This causes a steep drop in the error diversity. After this, certain genetic operators work together with lexicase to steer the population towards good diversity, completely discarding the initial seeding in the process.

There seems to be ample evidence that lexicase selection does not need population seeding to perform any better. We have seen two populations, one starting with an initial diversity of 100 percent and the other starting with an initial diversity of 7 percent, performing almost identically. Therefore, our theory of seeding the population to eliminate the worthless programs and increase the initial diversity, and thereby increase the number of solutions found, seems to not hold. Since all diversity graph lines flatten out at similar diversities, it appears that to lexicase, the initial error diversity does not matter. These results are a testament to lexicase selection's ability to increase and maintain population diversity.

However, we should not take these results to indicate futility in improving initialization *if one uses a different parent selection technique*. To this end, we think future work comparing these initialization methods on different problems with a different parent selection technique (such as tournament selection) could shed more light on the importance of initialization in different settings.

ACKNOWLEDGMENTS

We would like to thank Washington and Lee University for the Summer Research Scholar grant that funded this study. We would also like to thank Nicholas Freitag McPhee, Lee Spector, and Sarah Anne Troise for their feedback on and suggestions about this research.

REFERENCES

[1] 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.
[2] 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
[3] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2016. Effects of Lexicase and Tournament Selection on Diversity Recovery and Maintenance. In *GECCO '16 Companion: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM, Denver, Colorado, USA, 983–990. DOI: http://dx.doi.org/doi:10.1145/2908961.2931657
[4] 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
[5] 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
[6] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. DOI: http://dx.doi.org/doi:10.1109/TEVC.2014.2362729
[7] 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.
[8] David Jackson. 2010. Phenotypic Diversity in Initial Genetic Programming Populations. In *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010 (LNCS)*, Vol. 6021. Springer, Istanbul, 98–109. DOI: http://dx.doi.org/doi:10.1007/978-3-642-12148-7_9
[9] 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
[10] Sean Luke. 2000. Two Fast Tree-Creation Algorithms for Genetic Programming. *IEEE Transactions on Evolutionary Computation* 4, 3 (Sept. 2000), 274–283. http://ieeexplore.ieee.org/iel5/4235/18897/00873237.pdf
[11] 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).
[12] Michael Schmidt and Hod Lipson. 2010. Age-Fitness Pareto Optimization. In *Genetic Programming Theory and Practice VIII*, Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva (Eds.). Genetic and Evolutionary Computation, Vol. 8. Springer, Ann Arbor, USA, Chapter 8, 129–146. http://www.springer.com/computer/ai/book/978-1-4419-7746-5
[13] 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
[14] 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