A Quantitative Analysis of the Simplification Genetic Operator

Haoxi Zhan
Hampshire College
Amherst, MA, 01002
hz12@hampshire.edu

ABSTRACT

The simplification function was introduced to PushGP as a tool to reduce the sizes of evolved programs in final reports. While previous work suggests that simplification could reduce the sizes significantly, nothing has been done to study its impacts on the evolution of Push programs. In this paper, we show the impact of simplification as a genetic operator. By conducting test runs on the U.S. change problem, we show that using simplification operator with PushGP, lexicase selection and ULTRA could increase the possibility to find solutions in the short term while it might remove some useful genetic materials for the long term.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program modification

General Terms

Experimentation

Keywords

Genetic Programming; Genetic Operator; Push; Clojush; Simplification

1. INTRODUCTION

Programs produced by genetic programming systems are often filled with unused instructions. Some of them are useless while the others might provide some useful genetic materials for future generations. These redundant instructions increase the sizes of programs and make them complex. Not only might they prevent the genes from evolving, they slow down the evolutionary processes significantly.

Simplification was introduced to counteract the negative impacts of the redundancy. Usually, the simplification function is applied after the evolution to remove such instructions. A recent study has shown that the simplification function in PushGP could reduce the sizes of programs significantly[3]. However, simplifying programs after the evolution could only improve the readability of the final programs in the reports. In order to reduce the program sizes and to speed up the evolution, we need to apply simplification during the evolution. Although this approach has been studied in the past[6], its impact on PushGP is still unknown.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2881-4/14/07 ...$15.00.
http://dx.doi.org/10.1145/2598394.2605684.

Push is a stack-based programming language which is especially designed for evolutionary computation[1, 5]. In Push, there's a stack for each data type. The instructions take their inputs from relative stacks and then push the outputs to the specific stacks. If the input stack is empty when an instruction is being executed, the instruction will be "no-op"ed and it will do nothing. Thus, the only requirement on syntax is the balance of parenthesis, which could be easily fixed. As a result, we are able to do arbitrary modifications to Push programs without worrying about syntax.

PushGP is a Genetic Programming system based on Push. While mutation, crossover and tournament selection are still available, a few new methods has been developed in PushGP recently. Lexicase selection is a new selection method based on multiple fitness cases[2]. When selecting parents, all fitness cases will be randomly sorted. Individuals who performed the best in the first case will survive and join the competetion of the next case until there's only one candidate or one case. ULTRA, which stands for "Uniform Linear Transformation with Repair and Alternation", is a new genetic operator which travels the linearized parents, produces children and then repairs them[4].

In this paper, we analyze the impact of the simplification genetic operator on evolution which is based on PushGP, lexicase selection and ULTRA. By dividing the evolution into several phases, we examine the performances in each phase. Then we compare the success rates, program sizes, running times and fitness values with and without simplification.

2. METHODS

2.1 The Problem

The problem we chose in this research is the U.S. change problem. The input is a specific amount of money in the integer stack and the evolved programs are supposed to return the minimum number of U.S. coins to make a change.

2.2 Fitness Evaluation

For each individual generated during the evolution, a fitness test is performed. Due to the employment of lexicase selection[2], the fitness evaluations are based on 150 different test cases. In each test case, an error value is calculated and then stored in the error vector. The error vector is defined as:

$$\text{Vec}[\epsilon] = (\epsilon_1, \epsilon_2, \cdots, \epsilon_{150})$$

If a program fails to produce a number in test case n, an error score of 1000 is assigned. Thus, $\epsilon_n = 1000$. Otherwise, the error value is the difference between the solution and the output:

$$\epsilon_n = |\text{solution} - \text{output}|$$

2.3 Simplification

The simplification operator we used in this study is based on hill climbing algorithm. For each simplification step, it removes a pair of parenthesis or several instructions. There's a 20% probability to remove a pair of parenthesis while the probability to remove instructions is 80%. The maximum number of instructions to remove is 2. After each step, a fitness evaluation will be performed. Only if the new program's error vector is exactly the same as the error vector of the original program, the new program will be kept.

2.4 Parameters

To test the impacts of the simplification operator, we conducted 2 experiments on U.S. Change problem. In order to guarantee moderate success rates, the parameters we used is based on our previous experiments. Experiment NO-SIMP was solely based on ULTRA and lexicase selection. In experiment SIMP-1, there's a 10% probability to use the simplification operator while the probability of ULTRA was 90%. After several tests, we decided to use only 1 simplification step as it's already enough to counteract the code growth.

To ensure the fairness, we made the total number of fitness evaluations equal. As the use of simplification will increase the number of fitness tests per individual, the population size of SIMP-1 was reduced to 1818. The calculation of the moderate population size is based on probability.

$$1001 \times s + 1000 \times s \times 0.1 = 2000 \times 1001 \Rightarrow s \approx 1818$$

Notice that due to the existence of the initial generation, fitness tests will be performed to a total of 1001 generations.

The details of our parameters are listed in Table 1.

Table 1: The parameters we used for our experiments

Experiment	NO-SIMP	SIMP-1
Basic parameters		
Number of Runs	100	100
Max Generations	1000	1000
Population Size	2000	1818
Max Size	500	500
Selection	lexicase	lexicase
Genetic Operators		
ULTRA Probability	1	0.9
Simplification Probability	0	0.1
Simplification Step	N/A	1
Instruction Remove Limit	N/A	2

3. RESULTS AND DISCUSSIONS

We conducted 100 runs for each experiment. Table 2 presents the general performances of our experiments. Notice that the average sizes in the table are based on the program sizes of each run's last generation.

Table 2: general performances

Runs	NO-SIMP	SIMP-1
Success Rate	82%	75%
Average Sizes	338.2	31.13
Average Time(h)	15.305	3.238
Average Generations	548.63	541.78

The average size of NO-SIMP runs is 338.2 whereas SIMP-1 runs have an average size of only 31.3. The huge difference in average program size indicates that the simplification genetic operator successfully controlled the code growth. As expected, the introduction of simplification significantly reduced the running time. However, 82 runs succeeded in the NO-SIMP experiment while only 75 SIMP-1 runs succeeded ($p = .067$). The success rate of SIMP-1 runs is 7% lower than the one of NO-SIMP runs. Surprisingly, while SIMP-1 runs have a lower success rate, their average number of generations is still lower. This implies that successful SIMP-1 runs might tend to find the solution faster. Hence we divided the evolution process into several phases and analyzed the performances of each phase.

3.1 Performance by phase

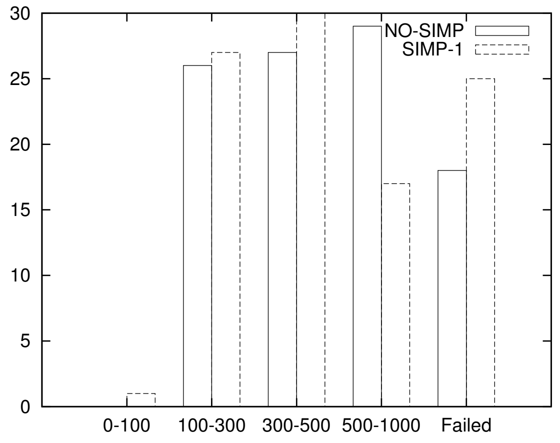

Figure 1: Performances by phases

As shown in Figure 1, SIMP-1 runs tend to find the solution faster. In the first 500 generations, 58% of SIMP-1 runs succeeded while only 53% of NO-SIMP runs found the solution. Among those who didn't terminate in the first 500 generations, 61.7% NO-SIMP runs returned the correct program in 1000 generations while only 40.5% SIMP-1 runs succeeded. The detailed success rates by phase are presented in Table 3. For each phase, the precentages reveal how many runs that didn't terminate before the phase succeeded in that specific phase.

Table 3: Success rates by phase

Phase	NO-SIMP	SIMP-1
0-100	0%	1%
100-300	26%	27.3%
300-500	36.5%	41.7%
500-1000	61.7%	40.5%

3.2 Average size by generation

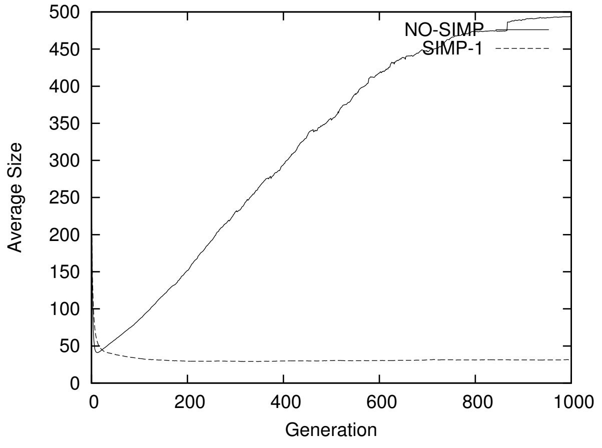

Figure 2: Average size by generation

Figure 2 presents the average program size by generation. As all initial programs were generated by the same function, both NO-SIMP runs and SIMP-1 runs started at around 250. Both experiments reveal a dramatic drop after the beginning of the evolution. Then the average size of NO-SIMP runs started to rise steadily, nearly reaching the maximum program size 500 in the last 100 generations. Meanwhile, the average size of SIMP-1 runs leveled off since the 200th generation.

3.3 Median error by generation

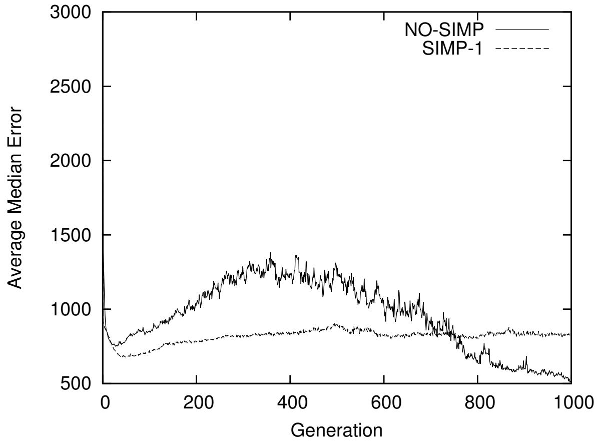

Figure 3: Median error by generation

Because average error is easily influenced by extreme individuals, median error is considered to be a better representation of the whole population's fitness. As shown in Figure 3, after a rapid decrease at the beginning, the median error of NO-SIMP runs grew steadily until the 400th generation. Then it gradually dropped. Finally it reached a low level at the end of the evolution. The change of SIMP-1 runs' median error is slightly different. It kept growing slowly for the remaining generations.

The information provided by Table 3 and Figure 3 is consistent. For instance, during the generation 300-500, NO-SIMP runs had a slightly higher median error. Meanwhile, it also has a lower success rate in that phase.

3.4 Best fitness by generation

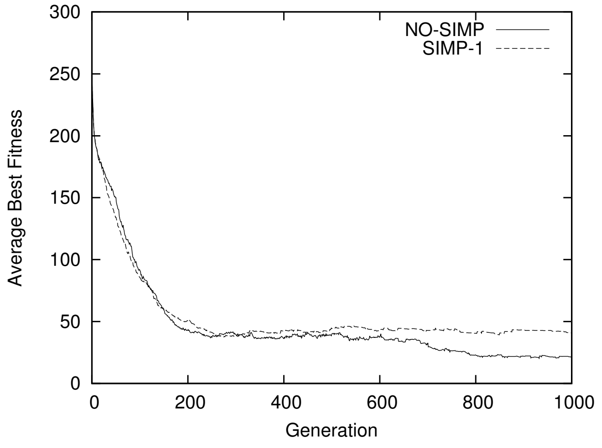

Figure 4: Best fitness by generation

While the median error represents the fitness of the whole population, the best fitness, which is shown in Figure 4, represents the elite group of each generation. Notice that as our fitness scores are represented by error values, the lower the value is, the better the program performs. For both the experiments, the best fitness dropped immediately after the beginning of evolution. Then SIMP-1 runs maintained a stable best fitness value, whereas for NO-SIMP runs the decrease continued. An unpaired t-test shows that $p = .084$ for the difference in best fitness.

Interestingly, SIMP-1 runs experienced no improvements on both median error and best fitness since the 200th generation. This implies that the evolution of the programs was kept stagnant due to the employment of simplification.

3.5 Number of unique programs by generation

One conjecture we had about the previous fact is that the diversity of programs is reduced by simplification. However, the number of unique programs, which is shown in Figure 5, disproved it. SIMP-1 runs show a fall in number of unique programs within the first 400 generations, followed by stable sizes around 1700. For NO-SIMP runs, most programs in the populations were unique in the first 400 generations. Then more and more duplicated programs were produced. In the last generation, only a bit more than 1300 programs were unique.

One possible reason for the rapid decrease experienced by NO-SIMPs runs is that the alternation of programs was limited by the size. When the parents had difficulty producing

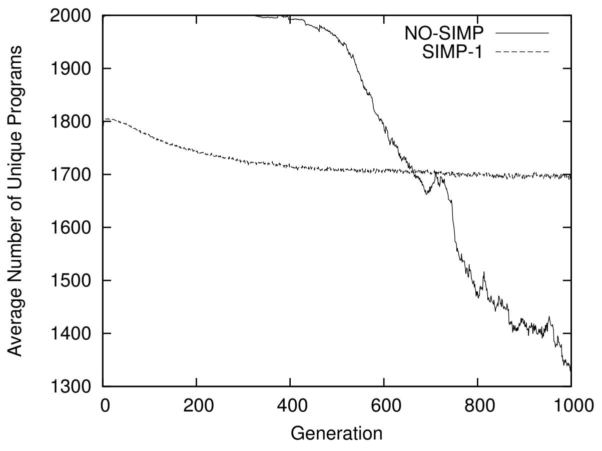

Figure 5: Number of unique programs by generation

children without exceeding the maximum size limit, one parent will be directly copied to the next generation. On the other hand, it's also possible that the population was gradually approaching the solution. This conjecture is supported by the data on median error and best fitness.

4. CONCLUSIONS, CONJECTURES AND FUTURE WORK

In this research we studied the impact of the simplification genetic operator in a PushGP environment with lexicase selection and ULTRA operator. The phase-based analysis revealed that the introduction of simplification operator will benefit the evolution in short term while it's harmful for the long run. By analyzing the data on median error and best fitness, we showed that in the runs with simplification, the evolution was actually almost stagnant. Then we used the diagram of the number of unique programs to prove that stagnancy was not caused by the lost of diversity.

Although we've got some interesting results, there are still many unknowns. Here we list the two major open problems: (1) How do the runs with simplification achieve a higher success rate in the short term? (2) Why is the simplification operator harmful in the long run?

Based on our data and analyzes, we have several conjectures on these problems. For problem 1, we conjecture that there are two reasons. Firstly, as the program sizes are successfully controlled by the simplification operator, the combinatorial search space is much smaller. Thus, the possibility to find a solution by luck is higher. Secondly, it's possible that the majority of the unused instructions will only be turned into useful ones in the long run. Thus, by simplifying the programs, most short-term-useless instructions are removed. For problem 2, we conjecture that many useful genetic materials are removed by simplification. While the populations of NO-SIMP runs are evolving towards the right direction, the evolution in runs with simplification relies more on randomness due to the lack of genetic materials.

In the future, we plan to develop a tracking system which is able to track the behavior of each individual in the population and conduct more experiments for a deeper research.

5. ACKNOWLEDGMENTS

We thank Lee Spector and Thomas Helmuth for creating Clojush, which is the PushGP implementation used in this research. We also appreciate Josiah Erikson for maintaining the Hampshire College Cluster Computing Facility. This research is based upon work supported by National Science Foundation under Grant No. 1017817 and 1129319. 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.

6. REFERENCES

[1] L. Spector. Autoconstructive evolution: Push, pushgp, and pushpop. In L. Spector, E. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, pages 137–146, San Francisco, CA, USA, July 2001. Morgan Kaufmann.

[2] L. Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In K. McClymont and E. Keedwell, editors, *1st workshop on Understanding Problems (GECCO-UP)*, pages 401–408, Philadelphia, Pennsylvania, USA, July 2012. ACM.

[3] L. Spector and T. Helmuth. Effective simplification of evolved push programs using a simple, stochastic hill-climber. In *Proceeding of the sixteenth annual conference companion on Genetic and evolutionary computation conference companion.*

[4] L. Spector and T. Helmuth. Uniform linear transformation with repair and alternation in genetic programming. In R. Riolo, J. H. Moore, and M. Kotanchek, editors, *Genetic Programming Theory and Practice XI.*

[5] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005)*, volume 2, pages 1689–1696, Washington DC, USA, June 2005. ACM Press.

[6] M. Zhang, P. Wong, and D. Qian. Online program simplification in genetic programming. In T.-D. Wang, X. Li, S.-H. Chen, X. Wang, H. A. Abbass, H. Iba, G. Chen, and X. Yao, editors, *Simulated Evolution and Learning, Proceedings 6th International Conference, SEAL 2006*, pages 592–600. Springer, October 2006.