Adding Program Length Bias to the Lexicase and Tournament Selection Algorithms

Eva Moscovici
University of Massachusetts
140 Governors Drive
Amherst, MA 01003-9264
emoscovici@cs.umass.edu

ABSTRACT

Lexicase selection is a relatively new but promising algorithm for selecting parents to participate in evolving the next generation of programs. Tournament selection is a more established and commonly used algorithm that serves the same purpose. As is the case with many other genetic programming algorithms, the programs generated with involvement of the lexicase and tournament selection algorithms are often large and hard to understand. There has been prior work done related to parsimony and managing bloat in the programs generated using genetic programming. In this paper, I will discuss a way to reduce program size for lexicase and tournament selection algorithms, by incorporating the program length bias into the error for each test case used to test whether an individual is fit enough to participate in the evolution or not.

CCS Concepts

•**Computing methodologies** → **Genetic programming;** *Artificial intelligence;* •**Theory of computation** → *Design and analysis of algorithms;* •**Software and its engineering** → *Automatic programming;*

Keywords

Program synthesis; genetic programming; lexicase selection; tournament selection; parsimony; bloat

1. INTRODUCTION

Lexicase selection is an algorithm that was invented by Lee Spector[5], and studied by Helmuth et al[3]. The algorithm is designed for selecting parents that will participate in the evolution process to evolve next generation of the program. In his dissertation[2], Thomas Helmuth describes the lexicase selection algorithm in detail, and uses post-hoc Wilcoxon-Nemenyi-McDonald-Thompson test[4] to compare the performance of lexicase selection to the performance of IFS (implicit fitness sharing) and tournament selection algorithms. His calculations conclude that lexicase selection outperforms IFS and tournament selection algorithms with 0.05 significance level. *Tournament selection* is one of the standard algorithms used for parent selection. In the next section, I will discuss the key idea of both algorithms, describing lexicase selection in more detail because it is less known. I will also discuss my proposal of adding a bias based on the length of the program, which will decrease fitness of the larger programs. As a result, smaller programs will have a slightly higher chance of being selected as parents, and thus the final successful program the evolution generates would also be smaller.

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908961.2931668

2. LEXICASE AND TOURNAMENT SELECTIONS, AND PROGRAM LENGTH BIAS

The lexicase selection algorithm has two steps, initialization and the loop. In the initialization step, the candidate individuals (i.e. programs of the latest generation) are set to be the entire population of the programs in that generation. The test cases are set to be the list of all the test cases used for testing the programs. The test cases should be shuffled randomly. For the loop step, choose a subset of the remaining candidates (originally that will be all of the candidates) that perform best on the first test case in the list of the test cases. If there is only one individual in that subset of the candidates, return it. If there is only one test case left, return the random individual from the subset of candidates. Otherwise, remove the first case from the list of the test cases, and go back to the beginning of the loop, picking the next candidates from the subset of the remaining candidates. The number of parents chosen for the next generation depends on the parameter that represents the population size. The population size is an argument to the function that performs the algorithm that the programmer using the algorithm to evolve a program can choose. Usually operations such as crossover require 2 parents, and operations such as mutation require 1 parent. Imagine the evolution process where the population size is 1000, and you want 10% of the population (100 individuals) to be generated by mutation, and the rest (900 individuals) by crossover. In that case, you would need a total of $100 + 900 * 2 = 1900$ parents to generate a population of 1000.

The way tournament selection works is that a few individuals are chosen randomly from the population of programs. The number of competitors is an argument to the function that performs the algorithm, and can be chosen by the programmer. The individual who has the highest fitness score

among its competitors is the winner. The fitness score is determined by testing the program using all the test cases, determining the error magnitude based on how far from the correct answer the program was, and adding up the errors for all the test cases. The bigger the error, the less fit the program is to be a parent.

In this paper I am comparing and contrasting the performance of lexicase and tournament selection algorithms in order to determine which one works better. Both algorithms have certain advantages over each other, so it is not as much a question of which one is better, as which one works better for which situation. Analyzing the algorithms this way will deepen our knowledge of the algorithms, and will help programmers determine which one they should use depending on what they are trying to accomplish. It is possible, for example, that one algorithm will finish faster, while the other will take longer, but the one that takes longer, generates shorter and easier to understand programs. Then we would know that if we need the ability to generate programs fast, we will use the first method, while if it is more important for the code to be easy to understand and maintain, we would use the second algorithm. Finally, I am adding a modification to lexicase and tournament selection algorithms, and exploring the results.

As of now, both lexicase and tournament selections choose which individuals will become parents based on how well they perform on the test cases. The lower the error is, the better the performance of the program. Adding bias will help incorporate other criteria, besides how far any given individual is from the correct answer on the next test case in the list. I propose adding a bias based on the length of the program, thus giving slight preference to smaller programs. Similar approaches have been explored in the past, such as using both error and program length when determining which individuals are a best fit to become parents to reduce bloat[1]. The expectation is that the programs would be smaller on average, while the number of generations would increase. In many real-world situations, software developers would need to run the evolution only once, or a few times, to generate the program they want, but after that, they would just need to use the generated program itself. In those cases, it would be a worthwhile trade-off to spend more time generating a program that would, however, be shorter and easier to understand.

The two problems I used for testing the performance of lexicase and tournament selection algorithms are a symbolic regression problem and even 3 parity. The goal of the even 3 parity problem is to find a program that, given three boolean inputs, determines whether the number of true inputs is even. The goal of a symbolic regression problem is to find a formula (program) that produces the proper output for each input in a given set of input/output number pairs. For example, given the pairs (1, 2), (2, 5), (3, 10), (4, 17), etc, it will determine that the formula is $y = x^2 + 1$ or its equivalent. There is an infinite number of equivalent formulas ($y = x^2 + x \div x$, $y = x^2 + 1 - 1 + 1$, etc), and because genetic programming involves a lot of randomization, the program might generate any one of these equivalent formulas. The formula I am using for the tests is $y = x^2 + x + 1$. A symbolic regression problem can also be bimodal, where the formula that will be used will be different depending on the value of the input. In my experiments, for bimodal symbolic regression problem, I am using $y = x^2 + x + 1$ for $x > 0$, and $y = 7 * x$ for $x \leq 0$.

3. IMPLEMENTATION

Since the population selected for reproduction depends on how well the individual performs (i.e. how small is the error when the program is tested), one way to incorporate program length into the algorithm is to increase the error of each test case for larger programs. Every program in the population is tested using the error function for all the test cases. Each time a test case is solved correctly, the error function will still return 0, but if there is an error, I increase the error function's return value by size-of-program$\div$100. I am dividing the program size by 100 because I observed that almost every successful program for symbolic regression or even 3 parity is smaller than 100. I want the bias to be less than 1, so that the error is more important than the program size. The program size would be used mainly to break ties when comparing the fitness of the programs with similar errors.

4. EXPERIMENT

The symbolic regression problems and even 3 parity, with and without program length bias, using tournament and lexicase selection algorithms, were tested by running the code 100 times for each problem and configuration, and recording the average size of the evolved program, and the average number of generations until a solution was found. For the ones that do not always (or never) succeed, I also recorded the number of successes/failures, and the best (i.e. lowest) error achieved. If a run takes over 100 generations, I assume it failed and move on to the next run. I put N/A when an entry is not applicable. For example, if the program never succeeds during the 100 runs, the average number of generations entry is not applicable, since it will always fail after 100th generation and move on to the next run.

5. RESULTS

5.1 Lexicase even 3 parity problem

Without adding the program length bias, the average number of generations for the even 3 parity problem using lexicase selection algorithm, after 100 runs, was 6.29, and the average best program size was 33.76. When adding the program length bias, the average number of generations became 7.02, and the average best program size became 26.58. The results are statistically significant: for the difference in the average number of generations, $p = 0.00034$, so $p \leq 0.01$. For the difference in the average best program size, $p = 0.0001$, so $p \leq 0.01$ To calculate statistical significance, I use Wilcoxon Signed-Rank Test.

As expected, adding program length bias improved the even 3 parity problem by making the resultant programs smaller on average, but increased the average number of generations it takes to find the successful program. In many real-world situations, it would be a worthwhile trade-off to spend more time generating a program that would, however, be shorter and easier to understand, because software developers would need to run the evolution only once, or a few times, to generate the program they want, but after that, they would just need to use the generated program itself.

Table 1: Lexicase selection even 3 parity problem results

	Without bias	With bias
Average number of generations	6.29	7.02
Average best program size	33.76	26.58
Number of successful runs	100	100
Best error for failed runs	N/A	N/A

Table 2: Tournament selection even 3 parity problem results:

	Without bias	With bias
Average number of generations	18.78	N/A
Average best program size	77.22	N/A
Number of successful runs	95	0
Best error for failed runs	1	1.15

For lexicase even 3 parity problem, both with and without bias, each run was always successful. Out of 100 runs, the program succeeded in finding the solution 100 times, so the success rate is 100%.

5.2 Tournament even 3 parity problem

Without adding the bias, the average number of generations for the even 3 parity problem using tournament selection algorithm, after 100 runs, was 18.78, and the average best program size was 77.22. The program was successful 95 out of 100 times, and failed times, with the best error of 1. Therefore, success rate is 95%. All 5 times it failed, it was unable to succeed on only one test case. When adding the program length bias, the program was never successful anymore: 0% success rate. It was very close, though: on almost each run, the lowest and most commonly encountered best error was 1.15. Consider the way error is calculated: if the program solved the test case correctly, the current total error for the program is unchanged, but if the program solved the test case incorrectly, the error is increased by 1+program size÷100. Therefore, the best error of 1.15 (just like the best error of 1 for the failed runs in the unbiased version) means that only one test case was solved incorrectly. Perhaps quickly eliminating the programs that cannot solve the hardest test cases by giving more error weight to the hardest test cases would solve this issue, and the program will be successful. That, however, is not within the scope of this paper. The average program size was 20.869, much smaller than the average size of the program without adding the program length bias. If a way to achieve a successful result is found, error length bias would be a considerate improvement to the algorithm.

For the even 3 parity problem, lexicase selection performs better both with and without the program length bias. Without the bias, lexicase selection completed its runs in the average of 6.14 generations, with the best program size average of 32.54. In the meantime, tournament selection needed the average of 18.78 generations, and the average best program size average was 77.22. The results are shown in Tables 1 (lexicase) and 2 (tournament).

5.3 Lexicase symbolic regression problem

Without the bias, the average number of generations was 9.19, and the average best program size was 14.02. Every run resulted in a successful solution, so success rate is 100%. With the bias, however, the situation was different: 0% success rate. The best error was 1.51. One thing I observed was that most programs had the size of 3, sometimes 4. That is a very small program, so it is not surprising that the program never succeeded. To see if I could lessen the effect of the bias, I divided the size of the program by 10000, instead of 100, when adding it to the error measurement. Dividing the size of the program by 10000, instead of 100, would lessen the effect of the bias, which could make a difference in which program is selected. For example, imagine a situation where for a symbolic regression problem, for the last test case, individual1, of size 25, returned an error (before adding bias) of 1.1, and individual2, of size 50, returned an error of 1.01. If divisor is 100, then for individual1, $1.1 + 25 \div 100 = 1.35$ is the error after adding bias, and for individual2, $1.01 + 50 \div 100 = 1.51$, so individual2 has higher error if we consider the bias, and individual1 will be selected. However, if the divisor is 10000, then for individual1, $1.1 + 25 \div 10000 = 1.1025$, and for individual2, $1.01 + 50 \div 10000 = 1.015$, so individual1 has higher error if we consider the bias, and individual2 will be selected. In this scenario, magnitude of the divisor made a difference. Dividing by 10000 instead of 100, however, had little effect on the performance of the symbolic regression problem used for the experiments described in this paper: the program size would stabilize within the range of 3-7, which in many cases is still too small to be the correct solution. The successful solution was never found.

5.4 Tournament symbolic regression problem

Without the bias, the average number of generations for a symbolic regression problem using tournament selection algorithm was 7.3, and the average best program size was 10.69. The program was successful 92 out of 100 times, and failed 8 times, with the best error of 0.96. Therefore, success rate is 92%. With the bias, the program failed every single time, with the lowest best error of 1.48. The lowest non-zero error is higher than it is in the case of unbiased tournament selection runs, but not by much, especially if we remember that this error is slightly weighted by the program size as well. The smallest error in the unbiased case is out of 8, because only 8 runs failed, while the biased one is the smallest error out of 100, because all of the runs failed. We cannot compare them in meaningful way because of the number difference. Overall, for a regression problem, tournament selection performs better: both average number of generations and average best program size are lower. Lexicase selection performed better on one metric: its success rate was 100%, while that of tournament selection was 92%. With bias, both lexicase and tournament selections performed badly: 0% success rate. For lexicase selection the best error for failed runs was slightly higher (1.51) than for tournament selection (1.48), but the difference is too small to be statistically significant. The results are shown in Tables 3 (lexicase) and 4 (tournament).

5.5 Bimodal lexicase symbolic regression

Without the bias, the average number of generations was 35.25, and the average best program size was 80. Only 26 runs resulted in a successful solution, so success rate is 26%. This problem is more complex than the previous ones, so that is not surprising. The lowest best error was 0.17: it was very close to finding the correct solution. With the

Table 3: Lexicase selection symbolic regression results

	Without bias	With bias
Average number of generations	9.19	N/A
Average best program size	14.02	N/A
Number of successful runs	100	0
Best error for failed runs	N/A	1.51

Table 4: Tournament selection symbolic regression results

	Without bias	With bias
Average number of generations	7.3	N/A
Average best program size	10.69	N/A
Number of successful runs	92	0
Best error for failed runs	0.96	1.48

program length bias, the program never found the correct solution. It was not even close: the lowest non-zero error for failed runs is 9.6

5.6 Bimodal tournament symbolic regression

Without adding the program length bias, the average number of generations was 43.11, and the average best program size was 148.23. Only 10 runs resulted in finding a successful solution, so the success rate is 10%. This result is worse than that of lexicase selection for all the parameters: average number of generations and average best program size were higher, and the success rate was lower. The lowest best error was 0.22: it was very close to finding the correct solution, but slightly further than the program that used lexicase selection was. With the program length bias, both with lexicase and tournament selections the program never found the correct solution. However, the program that used tournament selection algorithm was much closer than the program that used lexicase selection algorithm was: the lowest non-zero error for failed runs is 3.7 with tournament selection, and 9.6 with lexicase. The results are shown in Tables 5 and 6.

6. CONCLUSIONS

The next step would be to explore the types of problems for which adding length bias would improve the performance, and for which it would not, or even make it worse. Also, we could explore other ways to bias the random selection that occurs in lexicase algorithm. Different approaches would hold different benefits when selecting the parents. The approach described in this paper was geared towards reducing the size of the program, but sacrifices speed with which the successful program is produced. If we want to speed up the program generation process, however, one approach yet to be explored would be to determine the difficulty of the test cases, and perform a biased shuffle, making the most difficult test cases appear at the beginning of the list of test cases. The benefit of solving the hardest test cases first is that this way we should eliminate the badly-performing programs faster, thus speeding up the parent selection process.

To improve the ability to solve a problem, a possible approach is to pick cases that approximately divide the population in two: half (or as close to half as possible) of the population solved the problem, and half did not. The next step is to put the test cases that divide the population in half to the front of the list of the test cases. If the effect of doing so is too strong, it could be reduced by instead adding bias to make those test cases more likely to appear at the front of the list, instead of always making them appear in the front. Prioritizing test cases that divide the population in two nearly equal parts would result in a more quick elimination of completely bad programs than randomly choosing test cases would, but would not eliminate the programs that may have some good content, but not good enough to take care of the hardest test cases. This would add more diversity, but still relentlessly eliminate the most unfit programs.

Table 5: Bimodal lexicase selection symbolic regression results

	Without bias	With bias
Average number of generations	35.25	N/A
Average best program size	80	N/A
Number of successful runs	26	0
Best error for failed runs	0.17	9.6

Table 6: Bimodal tournament selection symbolic regression results

	Without bias	With bias
Average number of generations	43.11	N/A
Average best program size	148.23	N/A
Number of successful runs	10	0
Best error for failed runs	0.22	3.7

7. ACKNOWLEDGMENTS

Thanks to everyone in the Hampshire College Computational Intelligence Lab, especially my advisor Lee Spector, for their advice related to this work. This material is based upon work supported by the National Science Foundation under Grants No. 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.

8. REFERENCES

[1] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective genetic programming: Reducing bloat using spea2. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 536–543. IEEE, 2001.

[2] T. Helmuth. General program synthesis from examples using genetic programming with parent selection based on random lexicographic orderings of test cases. 2015.

[3] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *Evolutionary Computation, IEEE Transactions on*, 19(5):630–643, 2015.

[4] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods*. John Wiley & Sons, 2013.

[5] L. Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report. *Genetic and evolutionary computation*, pages 401–408, March 2012.