Lexicase Selection with Weighted Shuffle

Sarah Anne Troise and Thomas Helmuth

Abstract Semantic-aware methods in genetic programming take into account information about programs' performances across a set of test cases. Lexicase parent selection, a semantic-aware selection, randomly shuffles the list of test cases and places more emphasis on those test cases that randomly appear earlier in the ordering than those that appear later in the ordering. In this work, we explore methods for weighting this shuffling of test cases to give some test cases have more influence over selection than others. We design and test a variety of weighted shuffle algorithms and methods for weighting test cases. In experiments on two program synthesis benchmark problems, we find that none of these methods significantly outperform regular lexicase selection. We analyze these results by examining how each method affects population diversity, and find that those methods that perform much worse also have significantly lower diversity.

1 Introduction

Many different types of problems typically tackled by genetic programming (GP), including symbolic regression, classification, and program synthesis, require a program that performs well on a set of tests, which we will call *test cases*. On such problems, each program is evaluated on each test, producing an *error vector* that summarizes its performance on the tests. These error vectors typically provide all of the information used to determine which individuals in the population are selected to be parents of the next generation.

Sarah Anne Troise
Washington and Lee University, Lexington, Virginia e-mail: troises19@mail.wlu.edu

Thomas Helmuth
Hamilton College, Clinton, New York e-mail: thelmuth@hamilton.edu

In many parent selection methods, such as the pervasive tournament selection, each error vector is aggregated into a single fitness value that represents the performance of an individual on the problem. Such methods ignore a wide range of behavioral and semantic information that could potentially be used to more effectively guide search [22, 16]. Recently, researchers have started incorporating this information in their GP systems, such as in the case of geometric-semantic GP [23], behavioral programming [15], and other semantic-aware methods [19].

One recent semantic parent selection method, lexicase selection, has been shown to improve problem-solving performance on a range of problems compared to tournament selection [9, 8] and other semantic-based selection methods [19]. These encouraging results suggest not only that lexicase selection deserves careful analysis of how it contributes to these improves results, but also whether there are modifications that could be made in order to improve its performance further. In this study, we explore variants of lexicase selection in which we modify how it considers the test cases and their order.

An essential part of the lexicase selection algorithm consists of randomly shuffling the test cases. It then considers the test cases in the shuffled ordering, with test cases earlier in the ordering receiving more attention than those later in the ordering. Traditionally, this shuffling has been conducted in a uniform fashion, with each test case having equal probability of appearing at any position [26]. While many people have asked us personally if it would be useful to weight the shuffling so that some test cases are more likely to come earlier in the shuffling than others, to our knowledge this has not been tested in practice.

In this paper we explore the idea of weighting the shuffle of test cases in lexicase selection. One key question, that does not seem to have an obvious theoretical answer, is how should the test cases be weighted? Should easier test cases appear earlier in the ordering, or should harder cases appear earlier? We could imagine it being better for easier test cases to appear earlier, since this may allow evolution to make small steps to improve slowly over time. On the other hand, maybe it would be better to have harder test cases appear earlier, which could reward programs that perform well on test cases on which the rest of the population performs poorly. Since we do not know the best method for weighting shuffle, here we conduct an empirical investigation of a variety of methods, some of which place easier test cases earlier, some of which place harder test cases earlier, and some of which dictate order based on variance.

Our experiments on two program synthesis problems show a surprising result: while some of the weighting methods reduce the performance of lexicase selection, none of them significantly improve performance. To help explain this result, we examine how each method affects population diversity throughout each GP run. We find that many of the methods result in significant reductions in diversity, and none appear to increase diversity compared to regular lexicase selection. Since we believe that diversity maintenance is an important feature of lexicase selection, these results help explain the cases where shuffling methods perform much worse.

In the next section, we give a detailed description of lexicase selection and prior results that use it. In Section 3, we describe the weighted shuffling algorithms and

Algorithm 1 Lexicase Selection *(to select one parent)*

```
Inputs: candidates, the entire population; cases, a list of test cases
Shuffle cases into a random order
loop
    Set first be the first case in cases
    Set best be the best performance of any individual currently in candidates on first
    Set candidates to be the subset of candidates that have exactly best performance on first
    if |candidates| = 1 then
        Return the only individual in candidates
    end if
    if |cases| = 1 then
        Return a randomly selected individual from candidates
    end if
    Remove the first case from cases
end loop
```

our methods for weighting test cases. Next, we discuss the design of our experiments on weighted shuffle, and present results from those experiments. We finally give some examples of related parent selection techniques.

2 Lexicase Selection

Lexicase selection is defined in terms of *test cases*, i.e. the data points used to evaluate the performance of individuals in the population. While we treat test cases as input/output pairs of the form used in supervised learning, lexicase selection could work in any population-based search technique where individuals are evaluated on multiple metrics. Lexicase parent selection was motivated by the desire of having parent selection treat individual test cases separately, without ever comparing the results of programs on one test case with the results on another [9, 26].

Algorithm 1 presents the lexicase selection algorithm. During lexicase selection, we consider one test case at a time, whittling down the population by removing any individual that does not exhibit the very best performance on that case. Once a single individual remains, it is returned. If we iterate through every test case and multiple individuals remain, that means those individuals have identical error vectors, so we return one of them at random. In practice, we actually retain only one random individual per error vector prior to each lexicase selection, since this gives the exact same results and reduces the time required to filter the population at each step.

A key element of the lexicase selection algorithm is that the test cases are shuffled into a different order for selecting each parent. The test cases at the start of the shuffled list have the most impact on selection, since they have potential to filter out the most individuals from the pool. Many times, a test case near the end of the shuffled list will have no bearing on which individual is selected, if the set of candidates is whittled to a single individual before using every test case. In this way,

lexicase selection often selects *specialist* individuals that perform poorly on some cases as long as they perform very well on the cases at the start of the ordering [4].

Empirical studies have shown lexicase selection to increase and maintain much higher levels of behavioral diversity than tournament selection [5, 6]. These effects on diversity are thought to be a consequence of lexicase selection's emphasis on selecting different specialist individuals. In particular, since lexicase selection uses a different ordering of test cases for each selection, it is able to reward individuals that do well on different parts of a problem. Tournament selection, on the other hand, computes a single fitness value aggregating a program's performance across all test cases. No matter how this aggregation is performed (e.g. summed errors, implicit fitness sharing [20], etc.), it emphasizes the selection of *generalist* individuals that perform well across all test cases. An individual can achieve terrible fitness and low probability of being selected by tournament selection if it performs very poorly on a single test case, even if it has excellent performance on all other cases; such an individual would often be selected by lexicase selection.

Other variants of lexicase selection have made alterations to other parts of the algorithm. In the initial work describing lexicase selection, what is now considered standard lexicase selection was described as "global pool, uniform random sequence, elitist lexicase parent selection" [26]. Each of these areas suggests part of the algorithm that could be changed. For example, "elitist" refers to the fact that only those individuals with exactly the best error on a test case will continue. This constraint has been relaxed in epsilon lexicase selection, in which any individual with an error value within an "epsilon" of the best error value on a case will continue to the next step [18, 17]. This variant has proved very successful on continuous-error problems, for which lexicase selection had previously performed poorly.

Since the test cases at the start of the shuffled list of cases have the most impact on selection, every selection will treat some cases as more important than others, but those cases will be different in different selection events. As indicated by "uniform random sequence" above, most work has used a uniform shuffling of test cases, giving each case equal probability of appearing at any point in the shuffled order. Since the invention of lexicase selection, many researchers (the authors included) have speculated that there must be some better way to arrange the test cases than using completely uniform shuffling. In fact, Spector tested many ad-hoc methods of weighting the test case shuffle around the time lexicase selection was invented, but none of them proved superior in initial testing [25]. Burks and Punch describe a variant of lexicase selection that does not use uniform shuffling of test cases, which we discuss in more detail below and use as a comparison [1].

3 Weighted Shuffle

In our experiments, we consider three different methods for shuffling the test cases in a non-uniform manner for lexicase selection. Each of these shuffling methods requires a technique for weighting or ordering the test cases, which we will call the

bias metric. The bias metric, when applied to a test case, will produce the weight for that test case.

3.1 Shuffling Methods

Weighted shuffle first scores each test case by the chosen bias metric, assigning the result as the weight for the case. Then, a list of test cases is built by selecting cases one at a time, with higher weighted test cases having a greater chance of being selected at each step. The weighted selection can be modeled with a roulette wheel. If a test case has a higher weight on the bias metric, its slice of the roulette wheel is larger. We then randomly select a test case based on these slices. This process is repeated until we have a weighted ordering of the test cases. This Weighted shuffle is repeated for every parent selected during a generation, meaning that different orderings will occur during the generation, but they will all use the same weights when performing the shuffle.

This Weighted shuffle algorithm, as far as we can tell, is a standard method for performing weighted shuffle. For example, this is the weighted shuffle implemented in Haskell [2].

Ranked shuffle takes the test cases and ranks them by the selected bias metric. Ranked shuffle then selects a random integer upper bound, uniformly selected between 1 and the number of test cases inclusive. Next, another uniform random integer is selected between 1 and that upper bound, inclusive; this is the index of the chosen test case. The test case at this index becomes the first test case in the new shuffled order. This same process then repeats for the remaining cases, adding each selected case to the end of the list so far. With this method, the test cases with a better rank (ex. 1, 2, 3, ...) are more likely to be chosen at each step because they are more likely to be within the range from 1 to the selected upper bound. The motivation for Ranked shuffle is that the chance of being selected is based on rank, instead of weight, and thus will not be as skewed by large differences in weight.

During each step of the Ranked Shuffle process, we choose a case out of T test cases. The case with rank $t \in 1,...,T$ has probability of being selected of

$$P(t) = \frac{1}{T}\sum_{i=t}^{T}\frac{1}{i},$$

which can be seen because it will have $1/i$ chance of being chosen for each index $i \geq t$. This distribution is "a discretized version of the negative log distribution" [3], and for every integer $t \in 1,...,T$, is equivalent to

$$P(t) = \frac{-\log(t/T)}{T}.$$

Fixed-order lexicase-based tournament selection (**FOLBaT**) is what we will call a a variant of lexicase selection introduced by Burks and Punch that does not

use uniform shuffling of test cases [1]. In fact, they use a fixed ordering of the test cases for every selection in a generation, instead of shuffling the test cases at all. They base this ordering on how well the population performs on the test cases that generation, with more difficult test cases coming first. Since the test case ordering is fixed each generation, if the entire population were used in each selection, the same exact individual would be selected every time. Instead, this method only applies lexicase to a subset of the population, as in tournament selection. Thus we will call this method fixed-order lexicase-based tournament selection.

For each generation, the test cases are ordered deterministically for every selection (although ties are broken randomly for each selection). The original work using FOLBaT selection uses test case orderings sorted by two different bias metrics: the Number-of-Nonzeros and Average metrics described below [1]. We use tournament size of 7, and experiment with using other bias metrics as well.

3.2 Bias Metrics

Some of our bias metrics tend to order "easier" test cases earlier, some order "harder" test cases earlier, and some base the ordering on the variance of the population error values on the cases. The Number-of-Zeros metric counts the number of individuals in the population that achieve zero (i.e. perfect) error on the given test case. This means that easier test cases that the population tends to get correct more often are given more weight, and therefore tend to appear earlier when shuffled. The Number-of-Zeros-Inverse metric simply divides 1 by the Number-of-Zeros metric. Thus, the weights are inverted, and more difficult test cases will be more likely to appear earlier when shuffled.

Similarly, the Number-of-Nonzeros metric counts the number of individuals in the population that do not achieve zero error on the test case. Thus it orders harder cases earlier. Note that this weighting is not equivalent to the the Number-of-Zeros-Inverse weighting, since the relative weights will be different between test cases. As we will see below, this difference is not simply theoretical, since these methods give significantly different results in our empirical tests. We also try a Number-of-Nonzeros-Inverse metric that, as above, divides 1 by the Number-of-Nonzeros metric.

We could also imagine that there might be more information in the actual error values for each test case, not just whether an individual perfectly passes the case or not. Thus, we use a Median metric, which uses the median error in the population on a test case as its weight. In this setting, a higher median error will give more weight to the test case, so harder cases will come earlier. We also test a Median-Inverse metric, with which easier cases will come first. We also use an Average error metric, though we do worry that outliers may make some test cases dominate the weighting. The original FOLBaT paper used Average, so here it serves as a comparison metric [1]. Again, higher average error will give more weight, so harder cases will come earlier.

Finally, we also could imagine that it would be useful to have cases that differentiate more between individuals to come earlier; thus, we also try a Variance metric, which uses the variance of errors on a test case as its weight. Thus cases that have more varied errors will come earlier in the ordering. To be thorough, we also include a Variance-Inverse metric, where cases that have less divergent errors tend to come first.

4 Experimental Setup

We conducted experiments to compare our weighted shuffle lexicase selection variants to regular lexicase selection. Below we describe the experiments, including the problems and GP system we used.

4.1 Problems

For our experiments, we use two general program synthesis problems from a recent benchmark suite [8]. The problems in this suite, which are taken from introductory programming textbooks, require a range of data types and programming constructs to solve. We chose problems for which lexicase selection has performed well but has also showed room for improvement, so that we can expect important differences in performance to be visible. The first problem, Replace Space With Newline (RSWN), requires a program to take a string as input and print the string after replacing all spaces in the input with newline characters. It also requires the program to functionally return an integer representing the number of non-whitespace characters in the input. The second problem, Syllables, also gives a string as input. The program must count the number of vowels in the string, and then print that number as X in the string `"The number of syllables is X"`.

In each of our experiments, we report the number of successful programs out of 100 runs. Here, a program must pass both the test cases used during evolution as well as an unseen test set in order to be called a solution. We created the both data sets using the methods described with the benchmark suite [8]. We will also plot the median *behavioral diversity* of populations across sets of runs, which is defined as the proportion of distinct behavior vectors of individuals in the population [11]. Here, a behavior vector is the list of outputs of a program when run on the test cases.

4.2 Push and PushGP

For our experiments, we use the PushGP system, which has previously been used extensively on the benchmark problems we use here [8, 7, 21, 5, 4]. PushGP evolves

Table 1 PushGP parameters used in our experiments.

Parameter	**Value**
runs per problem/parameter combination	100
population size	1000
maximum generations	300
Genetic Operator	**Prob**
alternation	0.2
uniform mutation	0.2
uniform close mutation	0.1
alternation followed by uniform mutation	0.5

programs in the Push programming language, a stack-based language designed specifically for GP [24, 27]. Push has many features that make it well-suited for general-purpose program synthesis, such as the availability of many data types and control-flow constructs. Besides the language it evolves programs in, PushGP is otherwise a standard generational GP system. For this work, we use the Clojure implementation of PushGP, which is currently the most actively-developed implementation[1].

We give the PushGP parameters that we use in our experiments in Table 1. Our experiments use Plush genomes, the linear genome representation of Push programs [10]. The genetic operators in Table 1 act on these Plush genomes. Alternation is a crossover of two parents, and uniform mutation and uniform close mutation act on one parent; more details can be found in [10].

5 Results

We present the number of successful runs out of 100 for each setting on the Replace Space With Newline (RSWN) problem in Table 2 and the Syllables problem in Table 3. As a comparison, regular lexicase found 54 successful programs on RSWN and 22 successful programs on Syllables. The success results in these tables show that none of the combinations of shuffle methods with bias metrics significantly improve performance compared to regular lexicase selection. In fact, some give significantly worse results, using a pairwise chi-square test with Holm correction for multiple comparisons.

While we tried every bias metric with each shuffle method, some combinations seem more relevant to consider than others. For example, in the paper describing FOLBaT, the authors use the Number-of-Nonzeros and Average bias metrics [1]. Our results with FOLBaT are mixed for these metrics. In fact, we expected the Weighted and Ranked methods to perform poorly with the Average bias metric, since we imagined it could be heavily skewed by outliers. The results show that

[1] `https://github.com/lspector/Clojush`

Table 2 Number of successes out of 100 runs on the Replace Space With Newline problem. Underlined results are significantly worse than regular lexicase selection, which produced 54 successes. No results were significantly better than regular lexicase.

Type	Bias Metric	Weighted	Ranked	FOLBaT
Easy-First	Number-of-Zeros	13	40	6
	Number-of-Nonzeros-Inverse	54	43	7
	Median-Inverse	53	39	4
Hard-First	Number-of-Zeros-Inverse	52	49	35
	Number-of-Nonzeros	61	44	40
	Median	50	53	26
	Average	33	45	17
Variance-Based	Variance	30	57	11
	Variance-Inverse	52	53	30

Table 3 Number of successes out of 100 runs on the Syllables problem. Underlined results are significantly worse than regular lexicase selection, which produced 22 successes. No results were significantly better than regular lexicase.

Type	Bias Metric	Weighted	Ranked	FOLBaT
Easy-First	Number-of-Zeros	20	12	7
	Number-of-Nonzeros-Inverse	13	10	8
	Median-Inverse	19	12	8
Hard-First	Number-of-Zeros-Inverse	11	16	2
	Number-of-Nonzeros	17	14	3
	Median	20	17	6
	Average	14	15	5
Variance-Based	Variance	11	13	20
	Variance-Inverse	16	19	10

while neither performed exceptionally well with Average, neither did exceptionally poorly either.

With the Ranked shuffle and FOLBaT, two sets of two methods should produce equivalent rankings of cases and therefore comparable results. That is, Number-of-Zeros and Number-of-Nonzeros-Inverse should behave identically, since counting the number of zeros will produce the same ordering of test cases as taking the inverse of the number of nonzeros. Similarly, Number-of-Zeros-Inverse and Number-of-Nonzeros should also behave equivalently with Ranked shuffle. As expected, the numbers of successes for each of these combinations is not significantly different from one another. This equivalency does not hold for Weighted shuffle, where the relative differences in weight matter for the shuffle.

Figure 1 plots the average population behavioral diversity for each bias metric when using Weighted shuffle on the RSWN problem. This plot shows that Weighted shuffle is not able to produce significantly higher levels of behavioral diversity than regular lexicase selection, no matter what bias metric is used. While many of the bias metrics produce similar diversity to regular lexicase, a few result in significantly

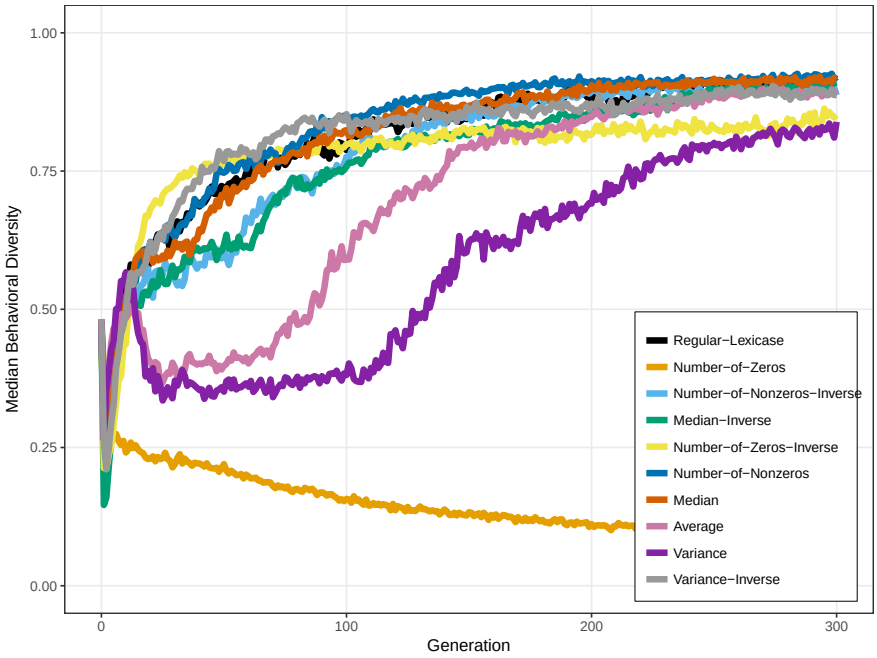

Fig. 1 For **Weighted shuffle**, the average population behavioral diversity of each bias metric plotted over the generations of each set of runs on the RSWN problem. Note that the black Regular-Lexicase line is mostly hidden behind the red Median line.

worse diversity. This is especially apparent with Number-of-Zeros, Variance, and Average, the three bias metrics that performed worst on this problem, showing a correlation between poor performance and poor diversity.

Figure 2 gives the same diversity plots, except for the Ranked shuffling method. Here, we see diversity more akin to that of regular lexicase, though many of the bias metrics have lower diversity in the first 150 generations of runs. There does not seem to be much correlation between diversity and success rate, with some of the metrics that create lower diversity still finding comparable numbers of solutions.

Finally, we give the diversity results for FOLBaT in Figure 3. Most of the bias metrics we used with FOLBaT do not exhibit the ability to increase and maintain diversity shown by regular lexicase selection. The two exceptions are with the Median and Average bias metrics; these metrics achieve high levels of diversity, although still lower than with regular lexicase. Interestingly, both of these metrics performed poorly in success rates, while some metrics with lower levels of diversity performed significantly better.

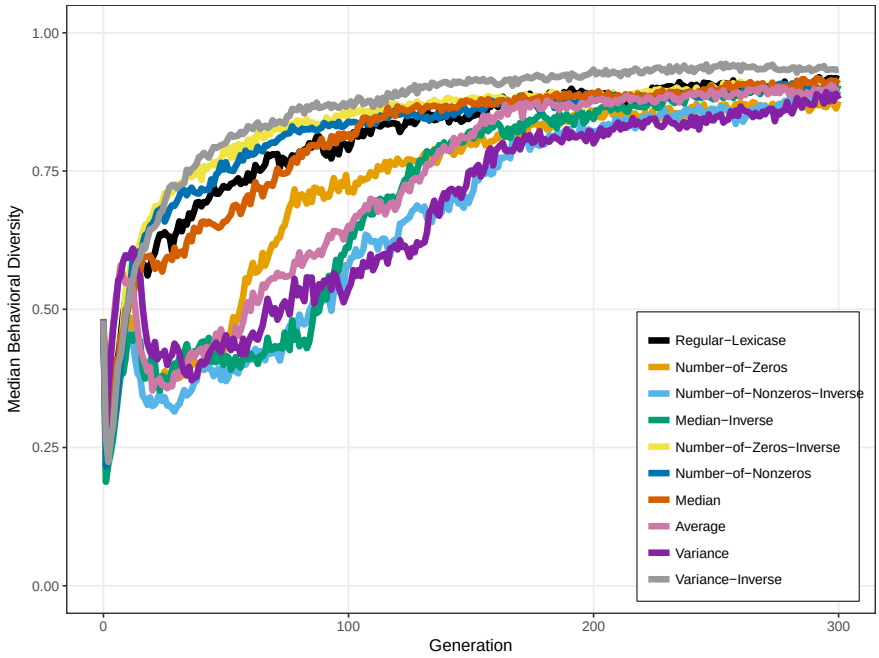

Fig. 2 For **Ranked shuffle**, the average population behavioral diversity of each bias metric plotted over the generations of each set of runs on the RSWN problem.

6 Discussion

Both Weighted shuffle and Ranked shuffle perform about as well as regular lexicase for most of the bias metrics. Weighted shuffle had slightly better results than Ranked shuffle with the bias metrics that gave the best results, but also much worse results with the worst bias metrics. FOLBaT, on the other hand, was often significantly worse than regular lexicase, with every bias metric except Variance-Inverse giving significantly worse results on one of the two problems.

As for the bias metrics, none stand out as particularly good or bad on these two problems, at least when only considering Weighted and Ranked shuffles. In fact, some of the bias metrics that give the worst results on one problem give the best results on the other problem.

The success rate results show that none of our combinations of shuffle methods with bias metrics resulted in significantly better results than with regular lexicase, which does not weight the shuffling of test cases. Thus it is difficult to recommend any of these shuffle methods or bias metrics over regular lexicase selection. Why, we must ask, does this intuitive idea of weighting the shuffling of test cases not lead to improvements?

Although we do not have any quantitative data, we have seen anecdotal evidence that suggests that with Weighted shuffle, some of the bias metrics that perform best

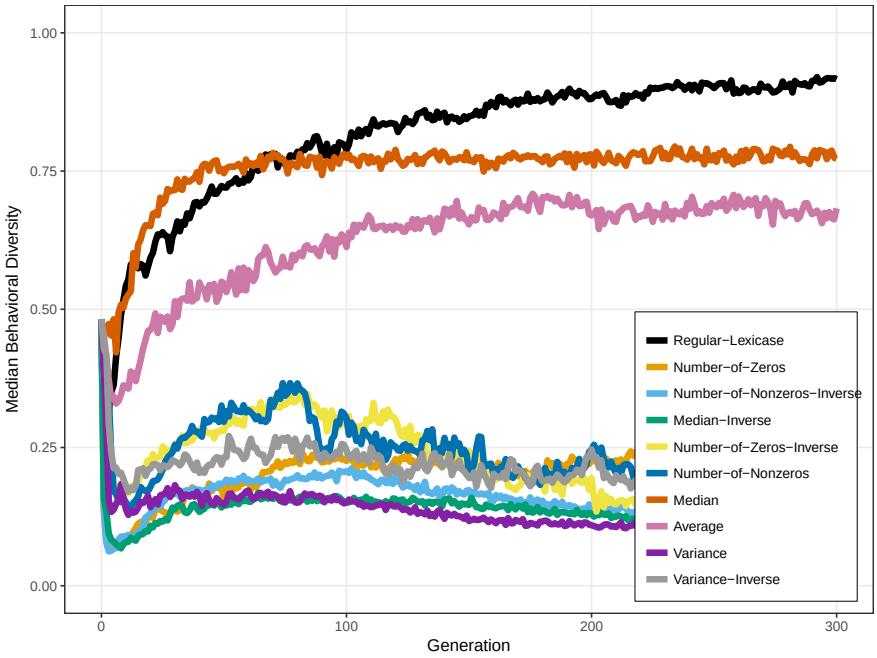

Fig. 3 For **FOLBaT**, the average population behavioral diversity of each bias metric plotted over the generations of each set of runs on the RSWN problem.

often use near-uniform distributions of test cases when shuffling. Test cases have similar or identical weights when most individuals in the population perform similarly on them. When most or all test cases have similar wights, Weighted shuffle acts very similarly to the uniform shuffle used by regular lexicase selection. Thus, it is not surprising that bias metrics that use near-uniform shuffling of test cases will give good performance results similar to regular lexicase selection. What is surprising is that many of the bias metrics that produce more weighting of test cases perform poorly, suggesting simply that weighting the shuffle leads to poor results. Note that these anecdotes only apply to Weighted shuffle, since the other shuffle methods result in different distributions of shuffles.

One possible explanation for the significantly worse results that we do see with some bias metrics with Weighted shuffle, and the slightly worse results for Ranked shuffle, is that they over-concentrate on some test cases while ignoring others. Such behavior may reduce the population diversity by limiting which test cases influence selection.

Turning attention to the diversity figures, we note that maintaining high diversity is somewhat correlated to better performance, but not always. In some cases, low diversity may play a large part in poor performance, especially with FOLBaT. But, this correlation does not always hold; for example, with Ranked shuffle, the Variance

bias metric had the highest number of success, yet exhibited some of the lowest diversity in the first half of runs.

Of the three shuffle methods, FOLBaT performed the worst and also the exhibited the lowest diversity. We believe this lack in diversity is caused by FOLBaT not using different orderings of test cases. This feature of lexicase selection allows it to emphasize different test cases with each selection, and therefore select many different individuals that perform well on different tests. Since FOLBaT doesn't shuffle the test cases at all, it always emphasizes the same cases within a generation. FOLBaT's reliance on a single sorting of the test cases gives a total ordering of the population, which can be seen as a scalar fitness value assigned to each individual based on rank. It only selects different individuals because it uses a tournament, and therefore the best individuals for each generation's ordering will not be present in every tournament. Avoiding a scalar fitness value is one of the key tenants of lexicase selection. This use of a scalar fitness value, a characteristic that it shares with tournament selection, may explain FOLBaT's inability to increase and maintain diversity, similar to tournament selection [6].

What does this teach us about lexicase selection? Lexicase selection's ability to emphasize different test cases with each selection seems paramount to its ability to maintain diversity, by selecting a wide range of individuals that specialize in different combinations of test cases. Using non-uniform shuffle decreases this uniformly random aspect of lexicase selection, which seems to have a neutral or negative impact on results.

Additionally, lexicase selection already places emphasis on individuals that uniquely perform well on single or multiple test cases, especially if such individuals also perform well on other cases. It appears to not be useful to place extra emphasis on some test cases; such favoring does not add utility to how often those cases appear early in the shuffle, and other times could lead to other cases being under-emphasized.

7 Related Work

The primary idea behind this work, that some test cases should be emphasized more than others based on how well the population performs on them, shares motivation with other parent selection methods. Each of the following methods uses tournament selection, but modifies fitness calculations in some way. In implicit fitness sharing (IFS), fitness is weighted so that test cases that are solved by fewer individuals receive more weight [20]. On problems such as those in this paper where test cases are non-binary, it is necessary to use a non-binary adaptation of IFS [14]. Earlier work showed that this non-binary IFS produced significantly worse results than lexicase selection on the two problems presented here, and on the benchmark suite that the problems come from more generally [8]; it has also been shown to produce lower levels of population diversity [5].

In "co-solvability" fitness, IFS has been extended to not weight individual cases, but instead pairs of test cases [13]. In this way, individuals that solve pairs of test cases not often solved by other individuals receive more reward. "Historically-assessed hardness" is another generalization of IFS to non-binary test cases, where fitness on each test case is scaled based on the performance of the population [12].

8 Conclusions and Future Work

This work demonstrates that using weighted shuffle of the test cases in lexicase selection does not increase the performance of GP. Over a wide array of methods for biasing the shuffle, including some that emphasize easy test cases and some that emphasize difficult test cases, we do not see any significant gains in performance compared to regular lexicase selection. We note a correlation between success rate and the ability to maintain diversity across some of our experimental results, in which methods that produced lower diversity were also those with the worst performance, though this correlation does not hold across the board.

These results, while discouraging with regard to improving performance, do suggest that it is not necessary to use any additional test case shuffling scheme in order to achieve good results with lexicase selection. Thus we recommend the continued use of lexicase selection with uniform random shuffling.

We hypothesize that one potential problem with the shuffle methods we present here is that they over-emphasize certain cases, which over-selects specific members of the population. In the future, we could consider whether there are ways to not over-emphasize specific test cases while still performing weighted shuffling. Such a scheme may be able to achieve better performance results while maintaining high levels of diversity. On the other hand, the resulting shuffles will be more similar to uniform shuffling, and therefore may simply behave more similarly to regular lexicase selection.

Acknowledgements Hammad Ahmad, Lee Spector, and Nicholas Freitag McPhee shared interesting disucssions that were very helpful in conducting this work.

References

1. Burks, A.R., Punch, W.F.: An investigation of hybrid structural and behavioral diversity methods in genetic programming. In: Genetic Programming Theory and Practice XIV, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA (2016)
2. Data.random.shuffle.weighted. `https://hackage.haskell.org/package/random-extras-0.19/docs/Data-Random-Shuffle-Weighted.html`. Accessed: 2017-05-01
3. Drury, M.: Does this discrete distribution have a name? Cross Validated. URL `https://stats.stackexchange.com/q/152786`. Accessed: 2017-11-25

4. Helmuth, T.: 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 (2015). URL http://scholarworks.umass.edu/dissertations_2/465/
5. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: A diversity analysis. In: Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA (2015). DOI doi:10.1007/978-3-319-34223-8. URL http://cs.wlu.edu/~helmuth/Pubs/2015-GPTP-lexicase-diversity-analysis.pdf
6. Helmuth, T., McPhee, N.F., Spector, L.: Effects of lexicase and tournament selection on diversity recovery and maintenance. In: GECCO '16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation, pp. 983–990. ACM, Denver, Colorado, USA (2016). DOI doi:10.1145/2908961.2931657
7. Helmuth, T., McPhee, N.F., Spector, L.: The impact of hyperselection on lexicase selection. In: T. Friedrich (ed.) GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, pp. 717–724. ACM, Denver, USA (2016). DOI doi:10.1145/2908812.2908851
8. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM, Madrid, Spain (2015). DOI doi:10.1145/2739480.2754769. URL http://doi.acm.org/10.1145/2739480.2754769
9. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation **19**(5), 630–643 (2015). DOI doi:10.1109/TEVC.2014.2362729
10. Helmuth, T., Spector, L., McPhee, N.F., Shanabrook, S.: Linear genomes for structured programs. In: Genetic Programming Theory and Practice XIV, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA (2016)
11. Jackson, D.: Promoting phenotypic diversity in genetic programming. In: PPSN 2010 11th International Conference on Parallel Problem Solving From Nature, *Lecture Notes in Computer Science*, vol. 6239, pp. 472–481. Springer, Krakow, Poland (2010). DOI doi:10.1007/978-3-642-15871-1_48
12. Klein, J., Spector, L.: Genetic programming with historically assessed hardness. In: Genetic Programming Theory and Practice VI, Genetic and Evolutionary Computation, chap. 5, pp. 61–75. Springer, Ann Arbor (2008). DOI doi:10.1007/978-0-387-87623-8_5
13. Krawiec, K., Lichocki, P.: Using co-solvability to model and exploit synergetic effects in evolution. In: PPSN 2010 11th International Conference on Parallel Problem Solving From Nature, *Lecture Notes in Computer Science*, vol. 6239, pp. 492–501. Springer, Krakow, Poland (2010). DOI doi:10.1007/978-3-642-15871-1_50
14. Krawiec, K., Nawrocki, M.: Implicit fitness sharing for evolutionary synthesis of license plate detectors. In: Applications of Evolutionary Computing, EvoApplications 2012, *Lecture Notes in Computer Science*, vol. 7835, pp. 376–386. Springer, Vienna, Austria (2013). DOI doi:10.1007/978-3-642-37192-9_38
15. Krawiec, K., O'Reilly, U.M.: Behavioral programming: A broader and more detailed take on semantic gp. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO '14, pp. 935–942. ACM, New York, NY, USA (2014). DOI 10.1145/2576768.2598288. URL http://doi.acm.org/10.1145/2576768.2598288
16. Krawiec, K., Swan, J., O'Reilly, U.M.: Behavioral program synthesis: Insights and prospects. In: Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation. Springer (2015)
17. La Cava, W., Moore, J.: A general feature engineering wrapper for machine learning using epsilon-lexicase survival. In: M. Castelli, J. McDermott, L. Sekanina (eds.) EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming, *LNCS*, vol. 10196, pp. 80–95. Springer Verlag, Amsterdam (2017). DOI doi:10.1007/978-3-319-55696-3_6

18. La Cava, W., Spector, L., Danai, K.: Epsilon-lexicase selection for regression. In: T. Friedrich (ed.) GECCO '16: Proceedings of the 2016 Annual Conference on Genetic and Evolutionary Computation, pp. 741–748. ACM, Denver, USA (2016). DOI doi:10.1145/2908812.2908898
19. Liskowski, P., Krawiec, K., Helmuth, T., Spector, L.: Comparison of semantic-aware selection methods in genetic programming. In: C. Johnson, K. Krawiec, A. Moraglio, M. O'Neill (eds.) GECCO 2015 Semantic Methods in Genetic Programming (SMGP'15) Workshop, pp. 1301–1307. ACM, Madrid, Spain (2015). DOI doi:10.1145/2739482.2768505. URL http://doi.acm.org/10.1145/2739482.2768505
20. McKay, R.I.: Fitness sharing in genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), pp. 435–442. Morgan Kaufmann, Las Vegas, Nevada, USA (2000)
21. McPhee, N.F., Finzel, M., Casale, M.M., Helmuth, T., Spector, L.: A detailed analysis of a PushGP run. In: Genetic Programming Theory and Practice XIV, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA (2016)
22. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. In: Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008, *Lecture Notes in Computer Science*, vol. 4971, pp. 134–145. Springer, Naples (2008)
23. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Parallel Problem Solving from Nature, PPSN XII (part 1), *Lecture Notes in Computer Science*, vol. 7491, pp. 21–31. Springer, Taormina, Italy (2012)
24. Spector, L.: Autoconstructive evolution: Push, PushGP, and Pushpop. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pp. 137–146. Morgan Kaufmann, San Francisco, California, USA (2001). URL http://hampshire.edu/lspector/pubs/ace.pdf
25. Spector, L.: personal communication (2012)
26. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion, GECCO Companion '12, pp. 401–408. ACM, New York, NY, USA (2012). DOI 10.1145/2330784.2330846
27. Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1689–1696. ACM Press, Washington DC, USA (2005). DOI doi:10.1145/1068009.1068292. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1689.pdf