6-4-2020

Summed Batch Lexicase Selection on Software Synthesis Problems

Joseph Deglman
University of Minnesota - Morris, deglm006@morris.umn.edu

Recommended Citation

Deglman, Joseph, "Summed Batch Lexicase Selection on Software Synthesis Problems" (2020). *Student Research, Papers, and Creative Works*. 7.
https://digitalcommons.morris.umn.edu/student_research/7

Summed Batch Lexicase Selection on Software Synthesis Problems

Joseph Deglman
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
deglm006@morris.umn.edu

June 4, 2020

ABSTRACT

Lexicase selection is one of the most successful parent selection methods in evolutionary computation. However, it has the drawback of being a more computationally involved process and thus taking more time compared to other selection methods, such as tournament selection. Here, we study a version of lexicase selection where test cases are combined into several composite errors, called summed batch lexicase selection; the hope being faster but still reasonable success. Runs on some software synthesis problems show that a larger batch size tends to reduce the success rate of runs, but the results are not very conclusive as the number of software synthesis problems tested was small.

1. INTRODUCTION

Evolutionary Computation (EC) is a field of Computer Science which utilizes concepts from evolutionary biology including: mutation, reproduction, and survival of the fittest. These tools can make searching for solutions more efficient than other methods, like random search and hill climbing. [5]

One application of EC is evolving software programs that would be difficult to develop by hand. To do this, we generate a population of randomly generated candidate programs. We then run the programs on a given series of test cases. For each test case, we calculate an error value measuring how much the output of the program differed from the desired output of the program for that case. We use these errors to determine which programs perform well and should be used as parents for the next generation. This process is continued until a solution is found or until a specified number of generations occur (the later case being an unsuccessful run).

In Section 2 we'll get an overview of lexicase selection. In Section 3 we'll talk about related work. In Section 4 we'll talk about the selection method used in this study. Section 5 will discuss software synthesis problems and describe the specific ones used in this study. Section 6 will describe the setup of the study and discuss the results. Finally, Section 7 will provide a conclusion.

2. LEXICASE SELECTION

Lexicase selection is a parent selection algorithm that works by randomly ordering the set of test cases (Fig. 1, line 2). Each individual of the current generation will then be evaluated against the first case in this random ordering (Fig. 1, line 3). If multiple candidates perform equally well on the first test case, they then move on to the next round to be evaluated against the second test case (Fig. 1, line 4). This goes on until we are left with only one candidate for our parent or until we have gone through all test cases. If the latter occurs then all remaining individuals have identical error values, and a random candidate among those remaining will be selected (Fig. 1, line 6).

3. RELATED WORK

J. Hernandez, A. Lalejini, E. Dolson, and C. Ofria [4] proposed two variants of lexicase selection: down-sampled lexicase selection and cohort lexicase selection. Down-sampled lexicase selection works by only using a random subset of the test cases each generation. Cohort lexicase selection divides the population each generation into cohorts that each are evaluated on a subset of all the test cases. Test cases are distributed so that all of them are used in exactly one cohort each generation. They show that random subsampling can be used to increase the success rate over a given amount of computation time.

V. V. de Melo, D. V. Vargas, and W. Banzhaf [2] proposed batch tournament selection (BTS) as a selection method. Their implementation ordered the test cases decreasing by difficulty and split them into groups of equal sizes. Tournament selection was then performed, picking the individual with the best mean error on that batch of cases. They tested this method on a selection of regression problems. They found that BTS behaved quite similarly to lexicase selection in terms of success rate and genetic diversity while also taking significantly less computation time.

S. Aenugu and L. Spector [1] used batch lexicase selection with learning classifier systems (LCS). The selection method used here creates batches of test cases then loops through them. Candidates progress to the next round of batch lexicase if their fitness for the round (defined as a proportion of correct matches on the current batch of cases) is above a threshold given as an input parameter. This process continues until only one individual remains or until all batches are exhausted; in the former case the remaining individual

	A	B	C	D	E	
(1)	{20, 6, 5, 2}	{6, 5, 7, 4}	{7, 6, 13, 2}	{7, 6, 4, 13}	{7, 6, 4, 2}	initial order
(2)	{2, 6, 20, 5}	{4, 5, 6, 7}	{2, 6, 7, 13}	{13, 6, 7, 4}	{2, 6, 7, 4}	randomized order (4,2,1,3)
(3)	{**2**, 6, 20, 5}	~~{4, 5, 6, 7}~~	{**2**, 6, 7, 13}	~~{13, 6, 7, 4}~~	{**2**, 6, 7, 4}	first round
(4)	{2, **6**, 20, 5}	~~{4, 5, 6, 7}~~	{2, **6**, 7, 13}	~~{13, 6, 7, 4}~~	{2, **6**, 7, 4}	second round
(5)	~~{2, 6, 20, 5}~~	~~{4, 5, 6, 7}~~	{2, 6, **7**, 13}	~~{13, 6, 7, 4}~~	{2, 6, **7**, 4}	third round
(6)	~~{2, 6, 20, 5}~~	~~{4, 5, 6, 7}~~	~~{2, 6, 7, 13}~~	~~{13, 6, 7, 4}~~	{2, 6, 7, **4**}	final round

Figure 1: Process of lexicase selection algorithm on a small set of programs. We have programs A,B,C,D, and E, each with four error values. In each round the best errors are bolded. If a given program doesn't have the best error in a given round they get crossed out in that round and also in the subsequent rounds, indicating that in has been eliminated from consideration. E gets selected in this case.

	A	B	C	D	E	
(1)	{20, 6, 5, 2}	{6, 5, 7, 4}	{7, 6, 13, 2}	{7, 6, 4, 13}	{7, 6, 4, 2}	initial order
(2)	{2, 6, 20, 5}	{4, 5, 6, 7}	{2, 6, 7, 13}	{13, 6, 7, 4}	{2, 6, 7, 4}	randomized order
(3)	{(2, 6), (20, 5)}	{(4, 5), (6, 7)}	{(2, 6), (7, 13)}	{(13, 6), (7, 4)}	{(2, 6), (7, 4)}	partitioned
(4)	{8, 25}	{9, 13}	{8, 20}	{19, 11}	{8, 11}	sums
(5)	{**8**, 25}	~~{9, 13}~~	{**8**, 20}	~~{19, 11}~~	{**8**, 11}	first round
(6)	~~{8, 25}~~	~~{9, 13}~~	~~{8, 20}~~	~~{19, 11}~~	{8, **11**}	final round

Figure 2: Process of the summed batch lexicase selection algorithm on the same programs as Figure 1 with the same randomization. Program E gets selected here also.

	A	B	C′	D	E	
(1)	{20, 6, 5, 2}	{6, 5, 7, 4}	{8, 6, 1, 2}	{7, 6, 4, 13}	{7, 6, 4, 2}	initial order
(2)	{2, 6, 20, 5}	{4, 5, 6, 7}	{2, 6, 8, 1}	{13, 6, 7, 4}	{2, 6, 7, 4}	randomized order
(3)	{**2**, 6, 20, 5}	~~{4, 5, 6, 7}~~	{**2**, 6, 8, 1}	~~{13, 6, 7, 4}~~	{**2**, 6, 7, 4}	first round
(4)	{2, **6**, 20, 5}	~~{4, 5, 6, 7}~~	{2, **6**, 8, 1}	~~{13, 6, 7, 4}~~	{2, **6**, 7, 4}	second round
(5)	~~{2, 6, 20, 5}~~	~~{4, 5, 6, 7}~~	~~{2, 6, 8, 1}~~	~~{13, 6, 7, 4}~~	{2, 6, **7**, 4}	final round

Figure 3: Here we use lexicase selection again on the programs A, B, D, and E from above, in addition to program C′, which is program C from above with different first and third errors. Here we end on the third round since E was the only program left at the end of that round.

	A	B	C′	D	E	
(1)	{20, 6, 5, 2}	{6, 5, 7, 4}	{8, 6, 1, 2}	{7, 6, 4, 13}	{7, 6, 4, 2}	initial order
(2)	{2, 6, 20, 5}	{4, 5, 6, 7}	{2, 6, 8, 1}	{13, 6, 7, 4}	{2, 6, 7, 4}	randomized order
(3)	{(2, 6), (20, 5)}	{(4, 5), (6, 7)}	{(2, 6), (8, 1)}	{(13, 6), (7, 4)}	{(2, 6), (7, 4)}	partitioned
(4)	{8, 25}	{9, 13}	{8, 9}	{19, 11}	{8, 11}	sums
(5)	{**8**, 25}	~~{9, 13}~~	{**8**, 9}	~~{19, 11}~~	{**8**, 11}	first round
(6)	~~{8, 25}~~	~~{9, 13}~~	{8, **9**}	~~{19, 11}~~	~~{8, 11}~~	final round

Figure 4: Summed batch lexicase on the same programs as in Figure 3. Here program C′ is selected even though it was not selected in Figure 3.

is chosen as a parent and in the latter case an individual is chosen at random of those remaining to be a parent. This selection method was shown to create more general rules for LCS than the standard lexicase selection method did.

4. A NEW SELECTION METHOD

Summed batch lexicase selection works like lexicase selection, but test cases are grouped together and composite scores from these are used following the same method as lexicase selection. The implementation of summed batch lexicase selection used in this study works in the following manner, as shown in Figure 2:

1. Test cases are randomized (Fig. 2 line 2)
2. During a selection test cases are grouped into partitions of equal sizes (except possibly the final partition). (Fig. 2 line 3)
3. The errors of the test cases in each partition are summed and placed in a new vector. (Fig. 2 line 4)
4. Lexicase selection is performed with the new vector of composite errors. (Fig. 2 lines 5 and 6)

The batch size determines how many test cases are grouped together into one error value. A batch size of one is equivalent to lexicase selection and a batch size of two means that pairs of the original error values are used to make up the new error vector.

Figures 3 and 4 together demonstrate that this selection method does have a different behavior than lexicase selection in some cases as they select different individuals.

5. SOFTWARE SYNTHESIS PROBLEMS

Software synthesis is the automated creation of software programs to fulfill a certain task. Sometimes, we have problems that we want to solve, but we do not know how to solve them. To do this we need to develop some notion of correct behavior for the program. This can be done by producing test cases that describe the expected behavior for given inputs. And then using some metric to determine how close to the expected behavior a program is on each case, we produce a set of error values.

Helmuth and Spector present a collection of software synthesis problems based on introductory computer science problems. [3] We'll use two here: **smallest** and **median**.

Smallest The goal of this problem is to produce a program that returns the smallest of four input integers.

Median The goal of this problem is to produce a program that returns the median of three input integers.

Both of these software synthesis problems require conditionals in a solution that satisfies the specification given in the problem. They also are similar in that they both return one of the inputs. Both of these problems also only have boolean errors for each of their test cases, so the only information given is whether the program got that test case correct or not, without any notion of partial correctness. They both use 100 test cases. **Smallest** has 5 cases that are given by hand and the other cases are semi-randomly generated (making sure to have some cases where there are duplicates in the inputs). Median only uses semi-randomly generated cases (also includes some cases chosen with duplicates in the inputs).

		Batch Size			
		1	2	4	10
Problem	Median	90	84	72	44
	Smallest	100	100	100	100

Table 1: Success rate (in percentages out of 50 runs) for median and smallest for different batch sizes. Smallest succeeded in all cases. The median problem had some unsuccessful runs for all batch sizes and succeeded less frequently as batch size increases.

6. EXPERIMENTAL DESIGN AND RESULTS

The Clojush[1] implementation of PushGP [6] was used in this study. PushGP is an evolutionary computation system designed for evolving programs. It uses stacks to control data-flow which provides a robust representation for programs, allowing for programs to changed slightly and still be able to execute.

This study looked at the two previously described software synthesis problems: **smallest** and **median**. These problems were attempted using four different batch sizes of summed batch lexicase selection: 1 (the same as standard lexicase selection), 2, 4, and 10. Fifty runs of each combination of problem and batch size were done.

Table 1 shows that all runs of the **smallest** problem were successful, but the **median** problem had some unsuccessful runs and had less successes as the batch-size was increased. As **smallest** did not have any unsuccessful runs, the success rate is not helpful in measuring the effect of batch size on success, so we will look at success generation (defined as the generation in which a run first finds a successful program or 200 in the case of an unsuccessful run).

Figure 5 shows that the mean success generation from the sample is higher for all runs of **smallest** that used batching compared to the run that used standard lexicase selection. Another interesting thing to note is that the variance in success generation also seems to be greater for the batched runs.

Figure 6 shows that the mean success generation on batch sizes 1, 2, and 4 with the **median** problem are about the same. The success generation on a batch of ten is much higher than the other batch sizes. With a batch size of 2 the mean success generation is lower than with a batch size of 1; this could possibly be an anomaly that would disappear with more runs.

A pairwise Wilcox test (shown in Table 2) was done to determine if there was any statistically significant difference between the number of successes of each set of runs. The test showed very high confidence that the two problems had different mean success generations. This is because the **smallest** problem is easier than the **median** problem. The test also showed that for the median problem the batch size of ten had a mean success generation that was significantly different from the other batch sizes (which from Fig. 6 we know is significantly worse) and also shows that there is no significant difference observed in the mean success for the batch sizes of 1,2, and 4 of the median problem.

[1] This repository includes an implementation of summed batch lexicase selection: `https://github.com/deglm006/Clojush`

The batch size seems to have had little effect on the success of the **median** problem for smaller batches and had a much greater negative effect at higher batch sizes. The **smallest** problem on the other hand seemed to display more of an effect of the batch size on for smaller sizes, but we see very little difference between 4 and 10.

7. CONCLUSION

Batch size seems to affect the two test problems differently, although the differing difficulties could be making it seem this way. Many of the runs for the **median** problem were unable to produce a successful program especially for a batch size of ten. Overall, the results are inconclusive. A larger batch size does seem to have the effect of making it take more generations to find a successful program, although it is not clear to what extent this is the case. Further study could look at what improvements this makes upon run time, although it seems like it would not show much improvement if any, as all test cases still need to be computed for all individuals using this method. Another area of potential study would be to look at using some other way of turning batches into a single value. Median, maximum, minimum, and weighted averages are possible choices.

Acknowledgments

Thanks to the University of Minnesota Undergraduate Research Opportunities Program (UROP) for funding this project. I would like to thank my advisor on this project, Nic McPhee, for his guidance and feedback.

8. REFERENCES

[1] S. Aenugu and L. Spector. Lexicase selection in learning classifier systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 356–364, New York, NY, USA, 2019. Association for Computing Machinery.

[2] V. V. de Melo, D. V. Vargas, and W. Banzhaf. Batch tournament selection for genetic programming: The quality of lexicase, the speed of tournament. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 994–1002, New York, NY, USA, 2019. Association for Computing Machinery.

[3] T. Helmuth and L. Spector. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, page 1039–1046, New York, NY, USA, 2015. Association for Computing Machinery.

[4] J. G. Hernandez, A. Lalejini, E. Dolson, and C. Ofria. Random subsampling improves performance in lexicase selection. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 2028–2031, New York, NY, USA, 2019. Association for Computing Machinery.

[5] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[6] L. Spector and N. F. McPhee. Expressive genetic programming: Concepts and applications. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, GECCO '16 Companion, page 589–608, New York, NY, USA, 2016. Association for Computing Machinery.

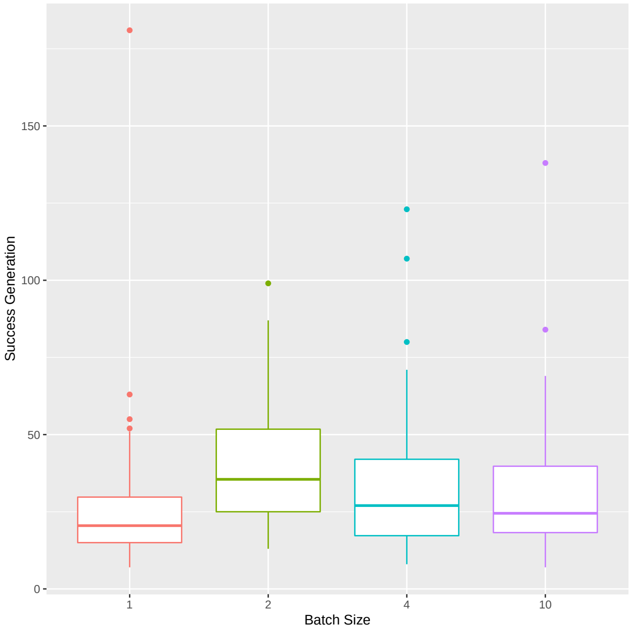

Figure 5: Box plot for smallest problem. This shows how many generations it took to find a successful program for different batch sizes. Increasing the batch size does seem to increase the number of generations needed to find a solution, but there doesn't seem to much difference between batch sizes of 4 and 10.

	1.median	2.median	4.median	10.median	1.smallest	2.smallest	4.smallest
2.median	1.0000	-	-	-	-	-	-
4.median	1.0000	1.0000	-	-	-	-	-
10.median	6.0e-05	1.0e-05	0.0031	-	-	-	-
1.smallest	1.8e-13	1.1e-11	3.2e-12	6.8e-16	-	-	-
2.smallest	6.5e-10	3.1e-07	4.4e-08	4.3e-15	8.9e-05	-	-
4.smallest	6.3e-11	5.2e-09	1.7e-09	2.3e-15	0.1877	0.2514	-
10.smallest	7.5e-12	7.9e-10	2.3e-10	1.3e-15	0.2962	0.0769	1.0000

Table 2: Pairwise comparisons of mean success generation using wilcoxon rank sum test. Labels along the top and side refer to particular combinations of batch sizes and problems (1.median would be the runs of the median problem done with a batch size of 1). Entries give p-values with the null hypothesis of the two configurations having the same mean success generation. This shows that median and smallest have a difference in success generation. It also shows the mean success generation for batch sizes of 1 and 2 of smallest are significantly different.

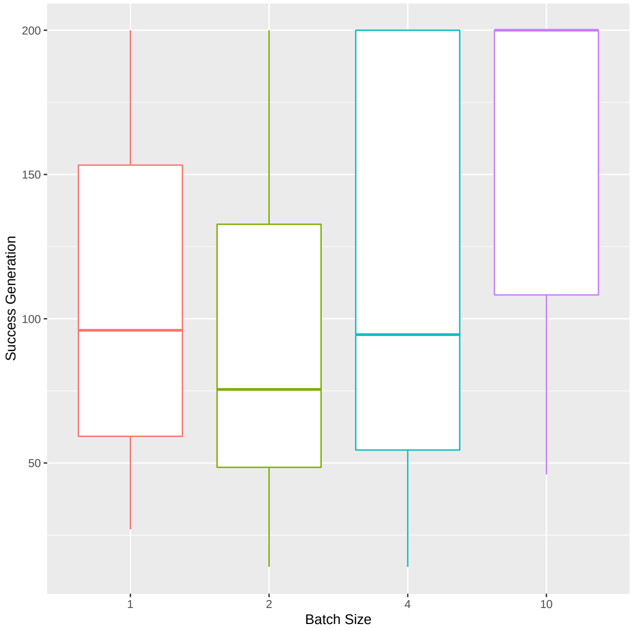

Figure 6: Box plot for median problem. Unsuccessful runs have a success generation of 200. The case with batch size of 10 is interesting. Most of the runs for this case did not succeed within the 200 generation limit and the mean generation of success is significantly worse for this case.