Program Synthesis with Genetic Programming: The Influence of Batch Sizes

Dominik Sobania(✉) and Franz Rothlauf

Johannes Gutenberg University, Mainz, Germany
{dsobania,rothlauf}@uni-mainz.de

Abstract. Genetic programming is a method to generate computer programs automatically for a given set of input/output examples that define the user's intent. In real-world software development this method could also be used, as a programmer could first define the input/output examples for a certain problem and then let genetic programming generate the functional source code. However, a prerequisite for using genetic programming as support system in real-world software development is a high performance and generalizability of the generated programs. For some program synthesis benchmark problems, however, the generalizability to previously unseen test cases is low especially when lexicase is used as parent selection method. Therefore, we combine in this paper lexicase selection with small batches of training cases and study the influence of different batch sizes on the program synthesis performance and the generalizability of programs generated with genetic programming. For evaluation, we use three common program synthesis benchmark problems. We find that the selection pressure can be reduced even when small batch sizes are used. Moreover, we find that, compared to standard lexicase selection, the obtained success rates on the test set are similar or even better when combining lexicase with small batches. Furthermore, also the generalizability of the found solutions can often be improved.

Keywords: Program synthesis · Genetic programming · Generalization

1 Introduction

Genetic programming (GP) [3,22] is a technique to automatically generate computer programs. For a given set of input/output examples (training cases) defining the requirements, GP searches in an evolutionary process for a program that completely fulfills these requirements. This procedure has similarities to the standard procedure in real-world software development, for example, as in test-driven development [2], where the test cases (e.g., unit tests) are defined first and after that the functional source code is written. GP, which in recent years has made some progress in automatic program synthesis [11], has the potential

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
E. Medvet et al. (Eds.): EuroGP 2022, LNCS 13223, pp. 118–129, 2022.
https://doi.org/10.1007/978-3-031-02056-8_8

to replace the second part of this process: writing the functional source code. However, this assumes that GP can find solutions for many everyday programming problems and that these solutions are generalizable which means that they also work on previously unseen test cases (as in production).

In recent work, the success rates (percentage of runs that find a correct solution) for standard program synthesis benchmark problems could be increased significantly [9,14]. This increase is strongly related to the use of lexicase selection [27], in which the training cases are evaluated individually instead of aggregating a program's performance on all training cases and selecting by this overall fitness value (as in tournament selection). However, considering the individual training cases during selection may lead to a strong overfitting on some benchmark problems [18,24]. Usually, such solutions generalize poorly to unseen test cases.

For classification problems, Aenugu and Spector [1] have shown that a variant of lexicase selection using batches combining a set of individual training cases usually leads to a better generalization. However, there are commonly many more training cases available in classification than in program synthesis, since in practice a programmer has to generate all the training cases manually (as there is no oracle function). So, due to the limited number of training cases, the choice of the batch size is also limited in program synthesis. Furthermore, it is still unclear how small batch sizes affect the program synthesis performance and generalization ability of GP.

Therefore, this work studies the influence of small batch sizes used during selection on the success rates and the generalizability of the programs generated by GP. For this analysis, we use three common problems from the general program synthesis benchmark suite [17].

For evaluation, we use a grammar-guided GP approach and use during selection batch sizes ranging from $\beta = 1$, which corresponds to standard lexicase selection, to $\beta = 100$. To analyze the influence of the batch sizes on the success rates as well as on the generalizability of the found solutions, we select three problems from the program synthesis benchmark suite [17] which are known in the literature for their generalization issues. We find in our experiments that using small batch sizes can lead to similar or even better success rates on the test set compared to standard lexicase selection ($\beta = 1$). Furthermore, best generalization rates are achieved with $\beta \geq 2$.

In Sect. 2 we give a brief introduction to lexicase selection and present the relevant work on GP-based program synthesis. Section 3 describes the used benchmark problems and the selection method. In Sect. 4 we present our experiments and discuss the results before concluding the paper in Sect. 5.

2 Lexicase Selection in GP-Based Program Synthesis

In the literature on GP-based program synthesis, variants of lexicase selection are often compared with other selection methods on a wide range of program synthesis benchmark problems and the lexicase variants usually outperform other selection methods like tournament selection, fitness-proportionate selection, or implicit fitness sharing [8,10,12,13,15–17,20,23–25].

Algorithm 1: Lexicase Selection

```
1  cases := shuffle(training_cases);
2  candidates := population;
3  while |cases| > 0 & |candidates| > 1 do
4  |   case := cases.pop(0);
5  |   candidates := best_individuals(candidates, case);
6  end
7  if |candidates| > 1 then
8  |   return choice(candidates);
9  end
10 return candidates[0];
```

Algorithm 1 shows the process of selecting an individual for the next generation with standard lexicase selection [16,27] as pseudo-code. First, the training cases are shuffled randomly (line 1) and all solutions from the population are considered as possible candidates for selection (line 2). In the next step, all candidates which do not have the exact lowest error on the first training case are discarded and the first training case is removed from the list (lines 4–5). This step is repeated until either all cases have been considered or only one candidate solution is left (`while` loop defined in line 3). Finally, either a random solution chosen from the remaining candidates will be returned (lines 7–9) or, if there is only a single solution left, the last remaining candidate solution will be returned (line 10).

Since lexicase selection is computationally intensive in comparison to other selection methods such as tournament or fitness-proportionate selection, de Melo et al. [4] suggested batch tournament selection, which combines the benefits of tournament and lexicase selection. Tested on a set of common regression problems, batch tournament selection achieves a solution quality similar to lexicase selection but is significantly faster. Another approach that is also based on batches of training cases is batch lexicase selection suggested by Aenugu and Spector [1]. For classification problems, they show that batch lexicase selection can improve generalization. In addition to batches, the authors introduce also a threshold parameter which allows individuals to survive the selection process even if they have a larger error than the best individual on the considered batch (depending on the defined threshold). So this parameter allows a further adjustment of the selection pressure (in addition to the selection of the batch size). A selection method based on similar principles that has been applied to program synthesis problems, but without analyzing the generalizability of the found solutions, is summed batch lexicase selection [5].

To prevent pre-mature convergence, Kelly et al. [21] suggested knobelty selection. Based on a defined novelty probability, an individual is selected either based on its novelty or its performance. For the performance-based selection, the authors use lexicase selection.

Recently, Hernandez et al. [19] suggested down-sampled lexicase selection, which operates on a different random subset of the training cases in each

generation. Although down-sampled lexicase selection consistently achieves better results than standard lexicase selection, it has not yet been shown that the solutions found generalize better to unseen test cases [18].

However, to our knowledge, there is no work so far studying the influence of small batch sizes on the success rates and the generalizability of programs generated with GP on program synthesis benchmark problems that are known for their low generalization rates.

3 Methodology

To analyze the influence of batch sizes on the performance and generalizability of GP-based program synthesis, we apply a grammar-guided GP approach to common program synthesis benchmark problems. In this section, we present the selected benchmark problems and describe the used grammars as well as the selection method.

3.1 Benchmark Problems

As we want to study if the use of small batches increases generalizability in the program synthesis domain, we selected three problems from the program synthesis benchmark suite [17] that are known for their generalization issues in the literature [24]. The selected problems are:

- **Compare String Lengths**: For three given strings, return true if the strings are sorted in ascending order according to their length. Otherwise, return false.
- **Grade**: For five given integer values, where the first four values define the minimum score required to achieve the grades "A", "B", "C", and "D", and the fifth value defines the score achieved by a student, return the grade for this student. Return an "F" if the achieved score is lower than the score defined by the fourth integer value.
- **Small Or Large**: For a given integer n, return "small" if $n < 1,000$, "large" if $n \geq 2,000$, and an empty string if $1,000 \leq n < 2,000$.

As defined by the benchmark suite, we use 100 training and 1,000 test cases for Compare String Lengths and Small Or Large, and 200 training and 2,000 test cases for the Grade problem.

3.2 Grammars

In our grammar-guided GP approach, we use context-free grammars supporting an expressive subset of the Python programming language including variable assignments, different data types, as well as conditionals. The used grammars are based on the grammars provided by the PonyGE2 framework [7] which follow the principle proposed by Forstenlechner et al. [8] which suggests that program

Table 1. Data types supported by the used grammars for each of the studied program synthesis benchmark problems.

Benchmark Problem	Integer	Boolean	String	Char
Compare String Lengths	✓	✓	✓	
Grade	✓	✓		✓
Small Or Large	✓	✓	✓	

synthesis grammars should, in addition to some basic data types, only support the required data types (e.g., the data types specified by a function's input and output). With this approach, the used grammars and consequently the resulting search space can be kept small.

Table 1 shows for each of the studied program synthesis benchmark problems the data types supported by the used grammars.[1] For all benchmark problems, the basic types Boolean and integer are supported. For Compare String Lengths and Small Or Large, we support in addition also strings. For the Grade problem, we support chars together with the required functions to process char values instead of strings as for this problem no complex string handling is necessary.

3.3 Selection Method

To study the influence of batch sizes in GP-based program synthesis, we extend the lexicase algorithm to include batches. Algorithm 2 shows this extended lexicase variant as pseudo-code.

Algorithm 2: Lexicase Selection with Batches

```
1  cases := shuffle(training_cases);
2  candidates := population;
3  batches := generate_batches(cases, β);
4  while |batches| > 0 & |candidates| > 1 do
5  |   batch := batches.pop(0);
6  |   candidates := best_individuals(candidates, batch);
7  end
8  if |candidates| > 1 then
9  |   return choice(candidates);
10 end
11 return candidates[0];
```

Basically, this method is similar to standard lexicase selection. The only difference is that, instead of individual training cases, batches of training cases of a pre-defined size β are generated (line 3). If β is a divisor of the number

[1] Grammars: https://gitlab.rlp.net/dsobania/progsys-grammars-2022-1.

of training cases, then all batches are of equal size. Otherwise, the last batch is smaller. After the batches are created, all candidates that do not have the exact lowest aggregated error/best fitness on the first batch are discarded and the first batch is removed (lines 5–6). As in standard lexicase selection, this step is repeated until either all batches have been considered or only one candidate solution is left (lines 4–7). Finally, a randomly chosen candidate of the list of remaining candidates (lines 8–10) or the last remaining one (line 11) is returned.

For $\beta = 1$, the described method works like standard lexicase selection. In general, the method is a version of batch lexicase selection [1] without the fitness threshold [as we discard all candidates that do not have the exact lowest error/fitness on a considered batch (line 6)] and consequently also similar to summed batch lexicase selection [5] as we aggregate in our experiments the fitness of a batch by calculating the sum of the errors on the contained training cases.

4 Experiments and Results

To study the influence of small batch sizes on the success rates and the generalizability of programs generated by GP we use in our experiments a grammar-guided GP implementation based on the PonyGE2 framework [7]. We set the population size to 1,000 and use position independent grow [6] as initialization method. We set the maximum initial tree depth (for initialization) to 10 and the maximum overall tree depth to 17. For variation, we use sub-tree crossover with a probability of 0.9 and sub-tree mutation with a probability of 0.05. A GP run is stopped after 300 generations.

As batch sizes, we study all divisors of 100, since for the majority of the considered benchmark problems 100 training cases are provided (this allows all batches to be equal in size). Finally, since the results in the program synthesis domain are often subject to high variance [26], we have doubled the number of runs used commonly in the literature (e.g., in [17] and [8]) and use 200 runs per configuration.

4.1 Influence on Selection Pressure

First, we study the influence of the batch sizes on the selection pressure. Therefore, we analyze the development of the average best fitness during a GP run for different batch sizes, where the fitness of an individual is the sum of its errors on the training cases. Furthermore, we analyze for all studied batch sizes the average generation in which a solution that correctly solves all training cases is found for the first time.

Figures 1, 2, 3, 4, 5 and 6 show the results for the benchmark problems considered in this study. The plots on the left (Figs. 1, 3, and 5) show the best fitness over generations for all studied batch sizes and benchmark problems. The results are averaged over 200 runs. The plots on the right (Figs. 2, 4, and 6) show the average generation of a first success on the training cases for all studied

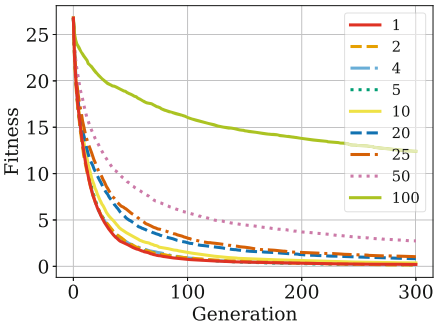

Fig. 1. Average best fitness over generations for the Compare String Lengths problem for all studied batch sizes.

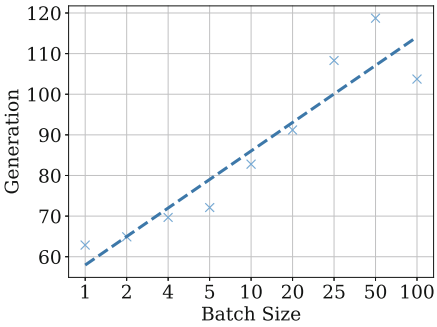

Fig. 2. Average generation of first success on training cases over batch sizes for the Compare String Lengths problem.

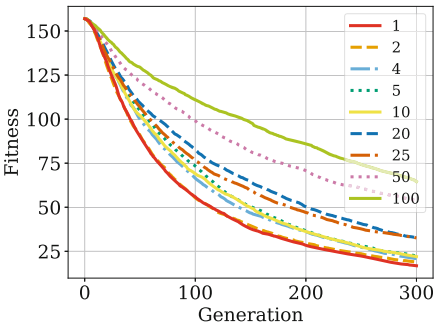

Fig. 3. Average best fitness over generations for the Grade problem for all studied batch sizes.

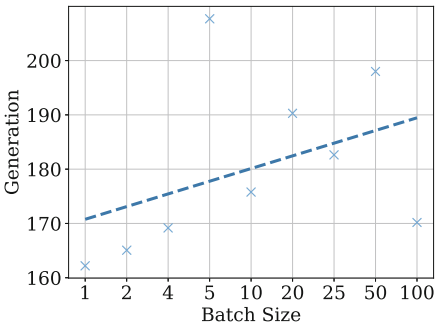

Fig. 4. Average generation of first success on training cases over batch sizes for the Grade problem.

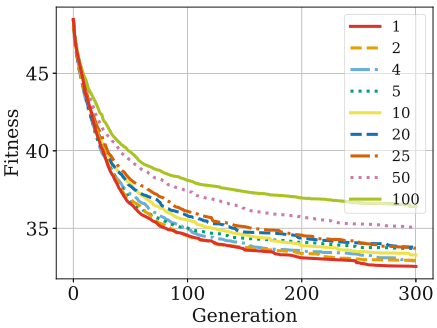

Fig. 5. Average best fitness over generations for the Small Or Large problem for all studied batch sizes.

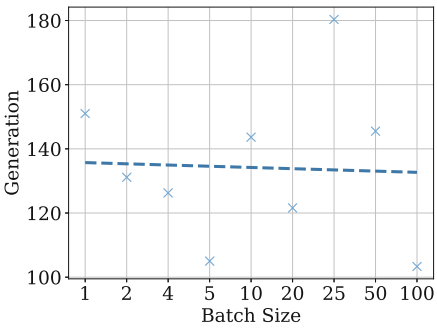

Fig. 6. Average generation of first success on training cases over batch sizes for the Small Or Large problem.

Table 2. Success rates on the training (s_{train}) and the test set (s_{test}) as well as the generalization rate g achieved by the grammar-guided GP approach for different batch sizes β for all studied program synthesis benchmark problems. Best values are printed in **bold** font.

Benchmark Problem	β	s_{train}	s_{test}	g
Compare String Lengths	1	**93.0**	10.0	0.11
	2	91.5	**15.5**	**0.17**
	4	91.5	8.5	0.09
	5	90.5	7.0	0.08
	10	80.0	4.5	0.06
	20	60.0	4.0	0.07
	25	57.5	5.5	0.1
	50	29.5	1.5	0.05
	100	5.5	0.5	0.09
Grade	1	34.0	8.5	0.25
	2	33.5	8.0	0.24
	4	31.0	10.5	0.34
	5	**36.0**	7.5	0.21
	10	25.0	**11.5**	**0.46**
	20	20.5	8.5	0.41
	25	16.5	6.5	0.39
	50	10.5	4.5	0.43
	100	5.5	2.5	0.45
Small Or Large	1	**9.0**	**3.5**	0.39
	2	4.5	3.0	0.67
	4	5.5	2.0	0.36
	5	2.5	1.5	0.6
	10	4.0	1.5	0.38
	20	5.0	**3.5**	0.7
	25	3.0	1.0	0.33
	50	1.0	0.5	0.5
	100	1.5	1.5	**1.0**

benchmark problems and training cases. The dashed regression line illustrates the development/trend for increasing batch sizes.

We see for all studied program synthesis benchmark problems that the fitness decreases more slowly over the generations for increasing batch sizes. The fastest

fitness reduction (minimization problem) is always achieved with batch size $\beta = 1$ (standard lexicase selection). Similarly, we observe the slowest fitness reduction for $\beta = 100$. E.g., for the Compare String Lengths problem (Fig. 1), for $\beta = 100$ the average best fitness is around 12 while for $\beta \leq 25$ the average best fitness is close to zero. Overall, the convergence speed is also reduced for small batch sizes ($4 \leq \beta < 20$).

For the generation of first success on the training cases we observe on average increasing values for increasing batch sizes. For the Compare String Lengths problem (Fig. 2) the success generation increases from around 60 for $\beta = 1$ to around 110 for $\beta > 50$ and for the Grade problem (Fig. 4) the generation increases from around 170 to 190. For the Small Or Large problem (Fig. 6), the regression line remains about constant (slight decrease) over the considered batch sizes. However, the results for this problem are based on only a smaller amount of data, compared to the other two benchmark problems, because the success rates on the training set are low for this problem (see Table 2).

In summary, with an increasing batch size β the selection pressure decreases. Additionally, the selection pressure can be reduced even with the use of small batch sizes.

4.2 Analysis of Success Rates and Generalization

To analyze the performance and the generalizability of the solutions found by GP with different batch sizes, Table 2 shows the success rates on the training (s_{train}) and the test set (s_{test}) as well as the generalization rate g achieved by the grammar-guided GP for different batch sizes β for the considered benchmark problems that are known in the literature for their poor generalization with lexicase selection [24]. As the results are based on 200 runs, we report s_{train} and s_{test} in percent. Best values are printed in **bold** font.

As expected, the success rate on the training set s_{train} decreases for an increasing batch size β. Also the success rates on the test set s_{test} are low on all considered benchmark problems for larger batch sizes ($\beta \geq 25$). Nevertheless, using small batch sizes ($2 \leq \beta \leq 10$) often leads to similar or even better success rates on the test set compared to standard lexicase selection ($\beta = 1$). E.g., for the Compare String Lengths problem we achieved a success rate of 15.5 with $\beta = 2$ compared to only 10 with standard lexicase selection.

Furthermore, for all considered benchmark problems, best generalization rates g are achieved with $\beta \geq 2$. Compared to standard lexicase selection, we see for the Grade problem and the Small Or Large problem on average notably larger generalization rates g for $\beta \geq 10$. From a practitioners perspective, a high generalization rate is even more important than a high success rate as it is essential that the found programs work also correctly on previously unseen test cases. An additional check with many test cases is usually not possible in practice as it is expensive to manually create a large test set. However, if a program synthesis method is known for producing generalizable solutions, a programmer can trust this method. If such a method has a low success rate but a high generalization

rate, then the search can be easily repeated if no successful solution is found in the first run.

Overall, we find that similar or even better success rates on the test set can be achieved when combining lexicase selection with small batches instead of using standard lexicase selection. In addition, best generalization rates are achieved with $\beta \geq 2$.

5 Conclusions

As GP is able to generate computer programs for a given set of input/output examples automatically, it has the potential to be used in real-world software development. Similar as in test-driven development [2], a programmer could define the input/output examples first and GP could then generate the functional source code. A prerequisite for GP as support system in software development is a good program synthesis performance and a high generalizability of the found programs. However, for some benchmark problems, GP generates programs that generalize poorly to unseen test cases especially when standard lexicase selection is used [24]. For classification problems, however, it has been shown that combining lexicase selection with batches of training cases can improve generalization [1]. Anyway, using batches in a program synthesis context is challenging as usually the number of input/output examples that can be used for training is low.

Therefore, we studied in this work the influence of small batch sizes during selection on the success rates and the generalizability of the programs generated by GP on common program synthesis benchmark problems.

We found that with an increasing batch size the selection pressure is reduced, which can be observed even for small batch sizes ($4 \leq \beta < 20$). Furthermore, we found that, compared to standard lexicase selection, the achieved success rates on the test set are either similar or even better when small batches are used. Overall, best generalization rates are obtained with a batch size $\beta \geq 2$.

So we suggest to use small batches with lexicase selection in GP-based program synthesis as the results are competitive or even better than with standard lexicase selection and also the generalizability of the found solutions can often be improved.

References

1. Aenugu, S., Spector, L.: Lexicase selection in learning classifier systems. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 356–364 (2019)
2. Beck, K.: Test-driven Development: by Example. Addison-Wesley Professional (2003)
3. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Proceedings of an International Conference on Genetic Algorithms and the Applications, pp. 183–187 (1985)

4. De Melo, V.V., Vargas, D.V., Banzhaf, W.: Batch tournament selection for genetic programming: the quality of lexicase, the speed of tournament. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 994–1002 (2019)
5. Deglman, J.: Summed batch lexicase selection on software synthesis problems. Scholarly Horizons: Univ. Minnesota, Morris Undergraduate J. **7**(1), 3 (2020)
6. Fagan, D., Fenton, M., O'Neill, M.: Exploring position independent initialisation in grammatical evolution. In: 2016 IEEE Congress on Evolutionary Computation (CEC), pp. 5060–5067. IEEE (2016)
7. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., O'Neill, M.: PonyGE2: Grammatical evolution in Python. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1194–1201 (2017)
8. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A grammar design pattern for arbitrary program synthesis problems in genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 262–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_17
9. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Extending program synthesis grammars for grammar-guided genetic programming. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (eds.) PPSN 2018. LNCS, vol. 11101, pp. 197–208. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99253-2_16
10. Helmuth, T., Abdelhady, A.: Benchmarking parent selection for program synthesis by genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, pp. 237–238 (2020)
11. Helmuth, T., Kelly, P.: PSB2: the second program synthesis benchmark suite. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 785–794 (2021)
12. Helmuth, T., McPhee, N.F., Spector, L.: The impact of hyperselection on lexicase selection. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp. 717–724 (2016)
13. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: a diversity analysis. In: Riolo, R., Worzel, B., Kotanchek, M., Kordon, A. (eds.) Genetic Programming Theory and Practice XIII. GEC, pp. 151–167. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34223-8_9
14. Helmuth, T., McPhee, N.F., Spector, L.: Program synthesis using uniform mutation by addition and deletion. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1127–1134 (2018)
15. Helmuth, T., Pantridge, E., Spector, L.: Lexicase selection of specialists. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1030–1038 (2019)
16. Helmuth, T., Pantridge, E., Spector, L.: On the importance of specialists for lexicase selection. Genetic Program. Evol. Mach. **21**(3), 349–373 (2020). https://doi.org/10.1007/s10710-020-09377-2
17. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1039–1046 (2015)
18. Helmuth, T., Spector, L.: Explaining and exploiting the advantages of down-sampled lexicase selection. In: Artificial Life Conference Proceedings, pp. 341–349. MIT Press One Rogers Street, Cambridge, MA 02142–1209 USA (2020)

19. Hernandez, J.G., Lalejini, A., Dolson, E., Ofria, C.: Random subsampling improves performance in lexicase selection. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 2028–2031 (2019)
20. Jundt, L., Helmuth, T.: Comparing and combining lexicase selection and novelty search. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1047–1055 (2019)
21. Kelly, J., Hemberg, E., O'Reilly, U.-M.: Improving genetic programming with novel exploration - exploitation control. In: Sekanina, L., Hu, T., Lourenço, N., Richter, H., García-Sánchez, P. (eds.) EuroGP 2019. LNCS, vol. 11451, pp. 64–80. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16670-0_5
22. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
23. Saini, A.K., Spector, L.: Effect of parent selection methods on modularity. In: Hu, T., Lourenço, N., Medvet, E., Divina, F. (eds.) Genetic Programming, pp. 184–194. Springer International Publishing, Cham (2020)
24. Sobania, D.: On the generalizability of programs synthesized by grammar-guided genetic programming. In: Hu, T., Lourenço, N., Medvet, E. (eds.) EuroGP 2021. LNCS, vol. 12691, pp. 130–145. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72812-0_9
25. Sobania, D., Rothlauf, F.: A generalizability measure for program synthesis with genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 822–829 (2021)
26. Sobania, D., Schweim, D., Rothlauf, F.: Recent developments in program synthesis with evolutionary algorithms. arXiv preprint arXiv:2108.12227 (2021)
27. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 401–408 (2012)