A Generalizability Measure for Program Synthesis with Genetic Programming

Dominik Sobania
Johannes Gutenberg University
Mainz, Germany
dsobania@uni-mainz.de

Franz Rothlauf
Johannes Gutenberg University
Mainz, Germany
rothlauf@uni-mainz.de

ABSTRACT

The generalizability of programs synthesized by genetic programming (GP) to unseen test cases is one of the main challenges of GP-based program synthesis. Recent work showed that increasing the amount of training data improves the generalizability of the programs synthesized by GP. However, generating training data is usually an expensive task as the output value for every training case must be calculated manually by the user. Therefore, this work suggests an approximation of the expected generalization ability of solution candidates found by GP. To obtain candidate solutions that all solve the training cases, but are structurally different, a GP run is not stopped after the first solution is found that solves all training instances but search continues for more generations. For all found candidate solutions (solving all training cases), we calculate the behavioral vector for a set of randomly generated additional inputs. The proportion of the number of different found candidate solutions generating the same behavioral vector with highest frequency compared to all other found candidate solutions with different behavior can serve as an approximation for the generalizability of the found solutions. The paper presents experimental results for a number of standard program synthesis problems confirming the high prediction accuracy.

CCS CONCEPTS

• **Theory of computation** → **Design and analysis of algorithms**; • **Computing methodologies** → **Genetic programming**; • **Software and its engineering** → Search-based software engineering;

KEYWORDS

Program synthesis, Genetic programming, Generalization, Software engineering

ACM Reference Format:
Dominik Sobania and Franz Rothlauf. 2021. A Generalizability Measure for Program Synthesis with Genetic Programming. In *2021 Genetic and Evolutionary Computation Conference (GECCO '21), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3449639.3459305

1 INTRODUCTION

Program Synthesis is a major research topic in the field of genetic programming (GP). Based on a given set of input/output examples, GP searches for programs that map inputs to correct outputs, using an evolutionary process. With GP variants like strongly-typed genetic programming (STGP) [15] or grammatical evolution (GE) [17] even complex structures like conditionals, loops, and different data types can be represented in a program. As defining the functionality of a program using input/output examples is quite simple, GP-based program synthesis has not only the potential to support programmers in real-world software development but also could enable people without any programming knowledge to implement new functionality.

However, program synthesis with GP is still a challenging task. Especially the generalizability of the programs generated by GP is a major problem as the programs must not only work correctly on the given input/output examples that are used for training, but also on unseen test cases. In recent work, Forstenlechner et al. [5] found that increasing the number of training cases (input/output pairs) can improve the generalizability of the GP-generated solutions. Unfortunately, generating training data is usually quite expensive in a real-world context as the output value for every training case must be calculated and verified manually by the user. Thus, practitioners are interested in measures that help them to assess whether the generalizability of found solutions is high for previously unseen test cases. For such cases, where the measure indicates that the generalizability of found solutions is expected to be low, it can make sense to generate more training cases to achieve higher generalizability.

Therefore, this work suggests a measure to approximate the generalizability of the found programs which has a high prediction accuracy. To calculate the measure, we do not stop a GP run directly after a program is found that is correct on all training cases, but continue the run for an a-priori defined number of generations. This allows GP to find many more different candidate solutions that correctly solve all training cases. In a second step, we build an additional set of randomly generated input values (no expensive manual calculation of output data needed) for every benchmark problem and compute the behavioral vector (vector of outputs) for this new set of input values for every found candidate solution. In an ideal GP run (in the program synthesis domain), all candidate solutions would produce the same behavioral vector. However, in reality GP produces in a run many candidate solutions with different behavioral vectors although all candidate solutions correctly solve all training cases. The proportion of the number of different found candidate solutions generating the same behavioral vector with highest frequency compared to all other found candidate solutions with different behavioral vectors can serve as an approximation for the generalizability of the found solutions.

We present experimental results for a representative selection of seven problems from the general program synthesis benchmark suite by Helmuth and Spector [11, 12] and study GP runs using proportionate, tournament, and lexicase selection. We find that the suggested measure has high prediction accuracy for the generalizability of the found solutions. For benchmark problems where the synthesized programs generalize well to unseen cases, one behavioral vector is dominating (the vast majority of solution candidates produce this behavioral vector) and the other behavioral vectors different from the dominating one are represented by only a few candidate solutions. For benchmark problems where the synthesized programs do not generalize well to unseen data, there is more than one relevant behavioral vector. The experimental results confirm that the proportion between the number of solution candidates that represent the behavioral vector with highest frequency and all other candidate solutions can be used as a high-quality indicator to predict if a GP run produces generalizing solutions or if it needs more training data.

Section 2 presents recent work related to GP-based program synthesis and describes the selection methods we use in our experiments. In Sect. 3, we describe the selected benchmark problems and in Sect. 4 we present the grammar-guided GP approach we use in this work. In Sect. 5, we present our experiments and discuss the results before concluding the paper in Sect. 6. Finally, Sect. 7 describes the limitations of the work and mentions directions for future research.

2 RELATED WORK

In this section, we present recent work related to GP-based program synthesis and describe the selection methods used in our experiments.

2.1 Program Synthesis with GP

To evolve programs in modern high-level programming languages, structures like conditionals, loops, and above all typing constraints must be supported by the algorithm. Unfortunately, standard GP is not capable of doing this as all used functions must be compatible with each other. However, GP variants like STGP [15] or grammar-guided GP approaches (like GE [17]) are able to support such complex structures. Consequently, in the last years often grammar-guided GP approaches are used for program synthesis [4, 6, 7, 14, 19]. Instead of the classical function and terminal sets, grammar-guided GP uses a context-free grammar for the definition of the programming language. Furthermore, recent work often replaces the classical genotype-phenotype mapping used in standard GE with a tree-based representation as GE's mapping process may lead to many invalid individuals during evolution especially when using large grammars [20]. However, regardless of the mapping process, the success of grammar-guided GP approaches used for program synthesis is also determined by the size of the search space which increases with the size of the used grammar. Thus, to reduce the size of the grammar, Forstenlechner et al. [3] suggested to include in the grammar only the data types required to solve the given problem. Hemberg et al. [13] used in addition to input/output examples also the given textual problem descriptions (from the general program synthesis benchmark suite [11, 12]) to adjust the grammar.

Another approach for GP-based program synthesis is PushGP which uses the stack-based programming language Push proposed by Spector and Robinson [22]. In Push, a separate stack is provided for each data type to support the usage of typing constraints. Recent progress with PushGP has been made by introducing uniform mutation and deletion (UMAD) [10] as mutation operator for linear genomes. With UMAD, the success rate on many benchmark problems could be significantly improved.

The generalizability of programs synthesized with GP-based approaches has so far only been studied in a few papers. This applies to both grammar-guided GP and PushGP. Helmuth et al. [9] studied different program simplification methods for Push programs to improve generalizability. Forstenlechner et al. [5] found that increasing the number of training cases can improve the generalizability of the found solutions to unseen test cases. Recently, Sobania [18] showed that the generalizability often depends on the output cardinality of the considered program synthesis problem. If the output cardinality is low (e.g., for problems with a Boolean output), GP tends to generalize to a lower extend.

To our knowledge, no work so far studied the behavior of evolved programs that are successful on all training cases. Furthermore, there is no work that studies whether the number of training cases is high enough to allow a GP run to evolve generalizable programs.

2.2 Selection Methods

In our experiments, we use three selection methods which are commonly used by GP-based approaches. In the following, we briefly describe how an individual is selected for the next generation:

- With proportionate selection, the chance of an individual to be selected is proportional to an individual's fitness. The higher the fitness, the higher the chance of selection [8].
- In tournament selection, a sample of t individuals is chosen randomly from the population and the individual with best fitness is selected [16].
- Lexicase selection as proposed by Spector [21] uses for selection an individual's error on a set of training cases instead of a compressed fitness value. In a first step, a list containing all training cases is instantiated and shuffled randomly. After that, every individual in the population is evaluated with the first training case in the list. Every individual that has not the exactly lowest error on this training case is discarded and the first item in the list containing the training cases is removed. This step is repeated until there is either only one individual left or the list of training cases is empty. If there is only one individual left, this solution is selected for the next generation; otherwise, one individual is selected randomly from the remaining individuals.

3 BENCHMARK PROBLEMS

To make different program synthesis approaches comparable, Helmuth and Spector [11, 12] curated a list containing 29 benchmark problems of different complexity. In our experiments, we use a

Table 1: Number of training and test cases as well as the input and output types for each studied benchmark problem.

Benchmark problem	#Training cases	#Test cases	Input	Output
Compare String Lengths	100	1,000	`String, String, String`	`Boolean`
Count Odds	200	2,000	`Int[]`	`Int`
Grade	200	2,000	`Int, Int, Int, Int, Int`	`Char`
Median	100	1,000	`Int, Int, Int`	`Int`
Small Or Large	100	1,000	`Int`	`String`
Smallest	100	1,000	`Int, Int, Int, Int`	`Int`
Super Anagrams	200	2,000	`String, String`	`Boolean`

representative subset of seven problems from this benchmark suite which are defined as follows:

- **Compare String Lengths**: for the given strings str_1, str_2, and str_3, return true if $\text{len}(\text{str}_1) < \text{len}(\text{str}_2) < \text{len}(\text{str}_3)$ holds, return false otherwise.
- **Count Odds**: for a given integer list, return the number of odd values in this list.
- **Grade**: given five integers, where the first four integers define the required score to achieve the grades A, B, C, or D, and the last integer defines the score achieved by a student in an exam, return one of the grades A, B, C, D, or F, depending on the score achieved by the student.
- **Median**: given three integers, return the median 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 $1,000 \leq n < 2,000$ holds.
- **Smallest**: for four given integers, return the smallest value.
- **Super Anagrams**: given two strings str_1 and str_2, return true if all characters from str_1 also occur in str_2 with exactly the same number of copies, false otherwise.

Table 1 shows for each of the studied problems the number of training and test cases defined by the benchmark suite as well as the input and output types. For the Count Odds, Grade, and Super Anagrams problem, the benchmark suite defines 200 training and 2,000 test cases. For all other selected benchmark problems 100 training and 1,000 test cases are defined. Additionally, the selected problems cover a wide range of input and output types including strings, integers (denoted as `Int`), and integer lists (denoted as `Int[]`).

4 GRAMMAR-GUIDED GP APPROACH

For the grammar-guided GP approach, we use context-free grammars covering an expressive subset of the Python programming language. The grammars are all based on the program synthesis grammars provided by the PonyGE2 framework [2]. For all studied benchmark problems, the grammars support the important structures like conditionals, loops, and variable assignments together with the basic Python functions. However, to keep the grammars (and so the search space) small, as suggested by Forstenlechner et al. [3], the grammars support for every benchmark problem in addition to Booleans and integers only the types defined for input and output (see Table 1). For example, the grammar for the Count Odds problem supports Booleans, integers, and integer lists including the relevant functionality like list slicing. To ensure the reproducibility of our experiments, the grammars are available online[1].

The fitness function used for proportionate and tournament selection is defined as

$$f(J, C, p_{\text{err}}) = \begin{cases} p_{\text{err}} \, |C| & \text{if an error occurs} \\ \sum\limits_{c_i \in C} d(J, c_i) & \text{else} \end{cases}, \quad (1)$$

where J is the individual to be evaluated (a Python function generated by GP), C is a set containing all training cases, c_i is the ith training case from C, and p_{err} is a value used to penalize run-time errors. The function $d(J, c_i)$ returns 0 if the individual J performs correctly on c_i and 1 if J's output is wrong. So GP aims to minimize the fitness function f as the errors of an individual J on the training set C are counted. If a run-time error occurs during the evaluation, the individual's fitness equals p_{err} times the number of training cases $|C|$.

As lexicase selection is not based on a classical fitness function, we measure the error separately for every training case instead of adding up the error achieved on all training cases (see Sect. 2.2).

5 EXPERIMENTS AND RESULTS

In our experiments, we use a grammar-guided GP approach based on the widely used PonyGE2 framework [2]. We set the population size to 3,000 individuals and use position-independent grow [1] as initialization method with a maximum initial tree depth of 10. As variation operators, we use sub-tree crossover and sub-tree mutation with a probability of 0.9 and 0.05, respectively. As the grammar-guided GP may eventually evolve endless loops, we stop the evaluation of an individual after one second (using an AMD Opteron 6272 16x2.1GHz CPU). To penalize run-time errors, we set $p_{\text{err}} = 2$. Lastly, each run is stopped after 100 generations.

We perform runs for all selected benchmark problems using proportionate, tournament, and lexicase selection. We do not stop a run after a solution was found that correctly solves all training cases, but let the GP run for 100 generations. For each run, we measure how many structurally n_{struct} and behaviorally n_{behave} different solution candidates are found during the GP run that correctly solve all training cases. Two solution candidates are different in structure if their source code is different. To measure behavioral difference, we can neither use the training data, since the output vector is identical for all solution candidates, nor the test data, since these are not available for decisions in a real-world context. Hence,

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

we generate for every benchmark problem a set of 500 random input cases and calculate the behavioral vector (output vector) for every solution candidate. This set of random input cases can be easily generated as an expensive manual calculation of output data is not required. Finally, two solution candidates are different in behavior if their behavioral vectors (output vectors based on the randomly generated input) are different. Obviously, $n_{\text{behave}} \leq n_{\text{struct}}$. We denote a solution that is structurally different from another solution as a structural solution; a solution is a behavioral solution if its behavioral vector is different. Each behavioral solution is represented by one or many structural solutions.

5.1 Behavior of Found Solutions

We study how the candidate solutions evolved by the GP approach behave for the randomly generated input values resulting in a behavioral vector. During a GP run, n_{struct} structural solutions are evolved; all of them correctly solve all training cases. For our analysis, we present results for a representative set of easy and difficult problems. We assume that the Median and Smallest problem are easy problems, whereas Compare String Lengths and Super Anagrams are more difficult for GP approaches [5]. We also performed the experiments for the other test problems (not shown due to space limitations) which showed similar behavior.

We study how many of the structural solutions have the same behavioral vector. This means, we count the number of structural solutions that represent the different behavioral solutions found during a GP run. Figures 1 and 2 present box-plots showing the number of structural solutions that represent the same behavioral solution for the Median and the Smallest problem, respectively. The four box-plots are in descending order according to the number of structural solutions that represent the particular behavioral solution. Thus, the first box-plot of each figure shows the behavioral vector that is represented by the highest number of structural solutions. For example, a GP using lexicase selection solving the Median problem finds around 120,000 (median value) structurally different solutions that all show the same behavior on the randomly generated input cases. We only show results for the four behavioral solutions that are represented by the highest number of structural solutions. We should have in mind that all structural solutions are successful on all training cases and all structural solutions that represent the same behavioral solution show the identical behavior on the randomly generated input cases. All results are based on 100 GP runs.

The results for both benchmark problems (Figs. 1 and 2) are similar: the first box-plot (largest behavioral group) contains almost all candidate solutions while the median of the box-plots 2-4 is close to zero. Thus, most structural solutions found during a GP run show the same behavior on the random input cases. Consequently, when randomly selecting one of the found structural solutions there is a very high probability to select a candidate solution with the most frequent behavior.

Figures 3 and 4 present the same analysis for the more difficult problems Compare String Lengths and Super Anagrams, respectively. The results are quite similar to the Median and Smallest problem as most structural solutions represent one particular behavioral solution. However, in contrast to the more easy Median and Smallest problem, we see a higher variance in the most frequent behavioral solutions and the differences between the median of the most frequent and the other behavioral solutions are much lower. For example, for the Compare String Lengths problem, around 40,000 (median value) structural solutions represent the most frequent behavioral solution. The second most frequent behavioral solution is represented by a median value of around 6,000 different structural solutions. In contrast to the more easy Median and the Smallest problem, a lower percentage of the structural solutions show the same behavior on the random input cases. When randomly sampling from the found structural solutions, the diversity of the observed behaviors is higher.

We find that the different behavioral solutions found during a GP run do not occur uniformly but some behavioral solutions occur more often than others. We expect that if the occurrence of the different behavioral solutions becomes more uniformly (like for Compare String Lengths and Super Anagrams), the found structural solutions generalize worse as some of the found structural solutions represent different behavioral solutions. In such cases, generalization must suffer.

5.2 Defining a Measure for the Generalization Capability of Found Solutions

Consequently, we want to suggest a measure b which should capture the properties of the distribution of behaviorally different solutions. If there are many structural solutions that result in the same behavior, we want b to be large; if there is no dominant behavioral vector but each of the different behavioral vectors is represented by the same number of structural solutions, b should be small. Thus, we consider all n_{struct} structurally different candidate solutions found during a run (that correctly solve all training cases) and define L as the frequency of the behavioral vector that represents the highest number of structural solutions. In Figs. 1-4, the left box-plot shows the distribution of L for four different program synthesis test problems.

Then, b is a measure for the distribution of the frequencies of the different behaviors of the candidate solutions found during a GP run:

$$b = 1 - \frac{n_{\text{struct}} - L}{L} = 2 - \frac{n_{\text{struct}}}{L}. \tag{2}$$

The measure $b \in [2 - n_{\text{behave}}, 1]$ measures the proportion of the L different found candidate solutions generating the behavioral vector with highest frequency compared to all other $n_{\text{struct}} - L$ found solutions with different behavioral vectors. For high values of $b \approx 1.0$, most found structural solutions represent the same behavioral solution (compare Figs. 1 and 2). For minimal values of b (e.g., $b = -1$ for $n_{behave} = 3$ different equally distributed behaviors), each of the different behaviors is produced by the same number of different solutions.

5.3 Generalization of Found Solutions

We study whether b can be used as a proxy for generalization of solutions found by a GP run.

To analyze the generalizability of programs that are successful on all given training cases found during a GP run, we perform runs for all selected benchmark problems using proportionate, tournament,

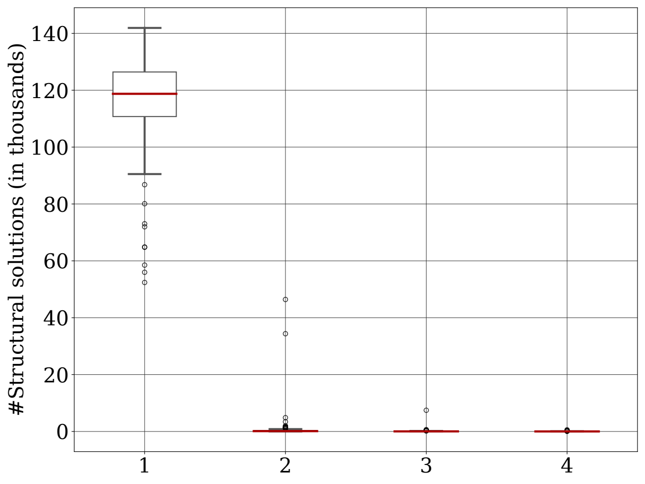

Figure 1: Median problem with lexicase selection: the box-plots show the number of structural solutions with identical behavior. We only show results for the four behavioral solutions that represent the highest number of structural solutions.

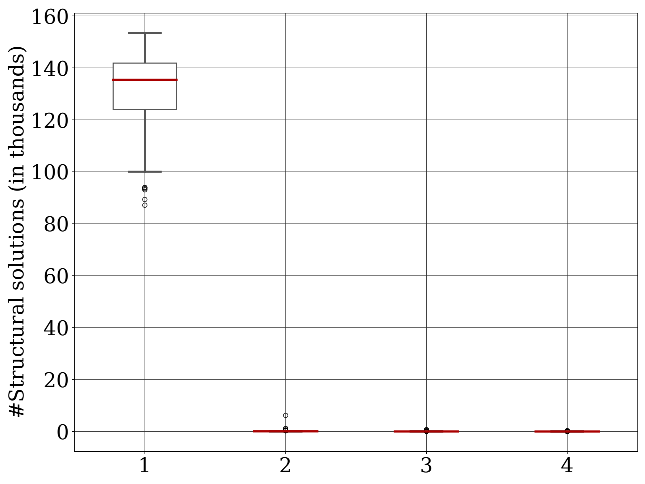

Figure 2: Smallest problem with lexicase selection: the box-plots show the number of structural solutions with identical behavior. We only show results for the four behavioral solutions that represent the highest number of structural solutions.

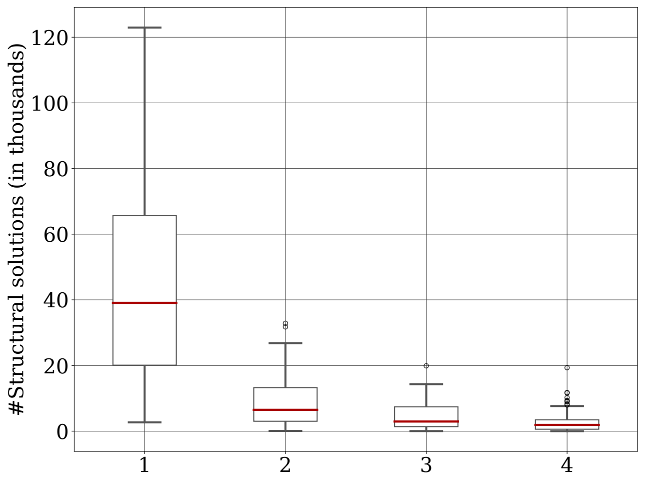

Figure 3: Compare String Lengths problem with lexicase selection: the box-plots show the number of structural solutions with identical behavior. We only show results for the four behavioral solutions that represent the highest number of structural solutions.

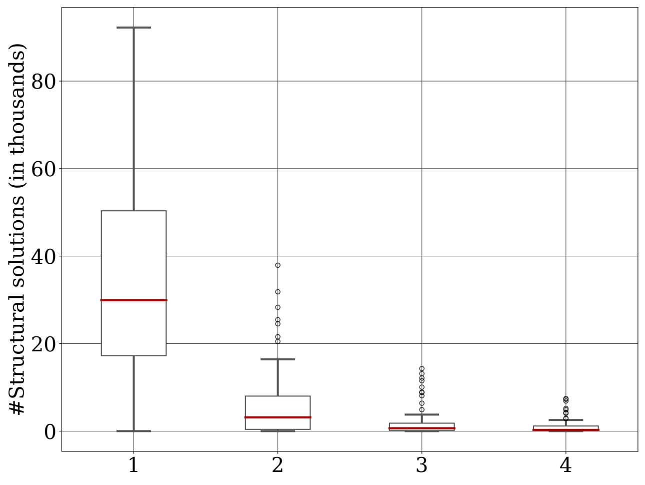

Figure 4: Super Anagrams problem with lexicase selection: the box-plots show the number of structural solutions with identical behavior. We only show results for the four behavioral solutions that represent the highest number of structural solutions.

and lexicase selection. To assess how well the programs generalize, we measure the success rates (percentage of runs in which a program could be found that solves all training/test instances) on the training and the test set. The generalizability is low for a benchmark problem, if the success rate on the unseen test set is significantly lower than on the training set.

For all studied benchmark problems and selection methods, Table 2[2] shows the number of runs s_{training} (training success) that find during the 100 generations at least one solution that correctly solves all training cases. As we stop each run after 100 generations (and not after the first solution is found that succeeds on all training

[2]The data of the GP runs analyzed in the paper were taken from [18].

Table 2: Success rates achieved on the training (s_{training}) and test set (s_{test}), generalization rate g (test/training success ratio), median number of unique structural ($\tilde{n}_{\text{struct}}$) and behavioral ($\tilde{n}_{\text{behave}}$) different solutions found during a run that solve all given training cases, and the measure $\tilde{b}$ which is the proportion of the number of solution candidates that represent the behavioral vector with highest frequency to all other candidate solutions for all studied benchmark problems and selection methods. Best success rates for a benchmark problem are printed in bold font. An asterisk (*) indicates a significantly ($p \leq 0.05$) lower success rate on the test set compared to the training set. For the statistical testing, we use a chi-square test or, in case at least one observation is lower than five, a Fisher's exact test. The results for the generalization rate g are printed in green (marked by 'h' for black and white printouts) if $g = 1.0$, in orange ('m') if $0.5 \leq g < 1.0$, and in red ('l') if $g < 0.5$. The results for the measure $\tilde{b}$ are printed in green ('h') if $\tilde{b} \geq 0.99$, in orange ('m') if $0.95 \geq \tilde{b} < 0.99$, and in red ('l') if $\tilde{b} < 0.95$.

	s_{training}	s_{test}	g	$\tilde{n}_{\text{struct}}$	$\tilde{n}_{\text{behave}}$	$\tilde{b}$
Compare String Lengths						
Proportionate	20	6*	0.3 l	1,907.5	6.0	0.978 m
Tournament ($t = 2$)	20	2*	0.1 l	50,439.0	74.5	0.837 l
Tournament ($t = 4$)	16	6*	0.375 l	35,562.0	45.5	0.839 l
Tournament ($t = 6$)	16	7	0.438 l	41,206.5	60.0	0.722 l
Lexicase	**97**	**18***	0.186 l	80,757.0	244.0	0.322 l
Count Odds						
Proportionate	1	1	1.0 h	5,735.0	28.0	0.992 h
Tournament ($t = 2$)	2	2	1.0 h	35,398.0	17.5	0.996 h
Tournament ($t = 4$)	0	0	-	-	-	-
Tournament ($t = 6$)	3	2	0.667 m	15,410.0	26.0	0.998 h
Lexicase	3	**3**	1.0 h	26,272.0	13.0	0.998 h
Grade						
Proportionate	0	0	-	-	-	-
Tournament ($t = 2$)	1	1	1.0 h	25,122.0	13.0	0.999 h
Tournament ($t = 4$)	0	0	-	-	-	-
Tournament ($t = 6$)	0	0	-	-	-	-
Lexicase	**35**	**10***	0.286 l	14,090.0	20.0	0.922 l
Median						
Proportionate	96	96	1.0 h	28,697.5	25.0	0.994 h
Tournament ($t = 2$)	93	93	1.0 h	109,322.0	32.0	0.997 h
Tournament ($t = 4$)	93	93	1.0 h	113,630.0	32.0	0.998 h
Tournament ($t = 6$)	94	94	1.0 h	115,565.5	35.0	0.997 h
Lexicase	**100**	**100**	1.0 h	120,371.5	41.0	0.997 h
Small Or Large						
Proportionate	0	0	-	-	-	-
Tournament ($t = 2$)	1	1	1.0 h	20,552.0	2.0	1.0 h
Tournament ($t = 4$)	3	3	1.0 h	22,168.0	1.0	1.0 h
Tournament ($t = 6$)	5	5	1.0 h	21,693.0	3.0	1.0 h
Lexicase	**15**	**10**	0.667 m	27,179.0	13.0	0.984 m
Smallest						
Proportionate	100	100	1.0 h	55,638.0	44.5	0.996 h
Tournament ($t = 2$)	100	100	1.0 h	131,854.5	49.5	0.998 h
Tournament ($t = 4$)	100	100	1.0 h	139,794.0	45.0	0.997 h
Tournament ($t = 6$)	100	100	1.0 h	140,786.5	44.5	0.997 h
Lexicase	100	100	1.0 h	135,803.5	31.0	0.999 h
Super Anagrams						
Proportionate	0	0	-	-	-	-
Tournament ($t = 2$)	0	0	-	-	-	-
Tournament ($t = 4$)	2	0	0.0 l	13,491.0	8.0	0.964 m
Tournament ($t = 6$)	0	0	-	-	-	-
Lexicase	**82**	**3***	0.037 l	48,938.5	25.5	0.886 l

cases), successful runs usually find more than one solution solving correctly all training cases. Sobania [18] studied how well the found solutions generalize and suggested to choose the solution with the lowest number of nodes in its abstract syntax tree[3] (AST) that correctly solves all training cases to maximize generalizability. If more than one candidate solution with the same lowest number of AST nodes was found during a GP run, Sobania chose the first solution with the minimum number of AST nodes. Choosing the solution with the lowest number of nodes in its AST leads on average to a higher generalizability in comparison to other approaches (e.g., choosing the first found solution that solves all training cases). Consequently, we report the number s_{test} of runs that correctly solve all test cases. Highest success rates for a benchmark problem are printed in **bold** font and an asterisk (*) indicates a significantly ($p \leq 0.05$) lower success rate on the test set compared to the training set. For statistical testing, we use a chi-square test or a Fisher's exact test in case one observation is lower than five.

For better readability, the table also presents the generalization rate g defined as

$$g = \frac{s_{\text{test}}}{s_{\text{training}}}. \tag{3}$$

The generalization rate g is rounded to three decimal places and printed in green (marked by 'h' for black and white printouts) if $g = 1.0$, in orange (marked by 'm') if $0.5 \leq g < 1.0$, and in red (marked by 'l') if $g < 0.5$.

Furthermore, the table shows the median number $\tilde{n}_{\text{struct}}$ and $\tilde{n}_{\text{behave}}$ of unique structural and behavioral solutions found during the 100 GP generations that correctly solve all given training cases ($n_{\text{struct}} \geq n_{\text{behave}}$). Solutions are structural different if they differ in their structure; they are behavioral different if they differ in their output for the randomly generated additional inputs.

Finally, we list the median of the measure $\tilde{b}$ over all 100 runs. It measures the proportion of the behavioral vector with highest frequency compared to all other behavioral vectors. The values of $\tilde{b}$ are rounded to three decimal places and printed in green ('h') if $\tilde{b} \geq 0.99$, in orange ('m') if $0.95 \geq \tilde{b} < 0.99$, and in red ('l') if $\tilde{b} < 0.95$.

For all studied benchmark problems, we confirm recent findings from the literature as lexicase selection performs best returning highest success rates [3]. This holds for both s_{training} and s_{test}. Additionally, lexicase selection finds also solutions to problems for which other selection methods can only find few or even no successful solutions. For example, for the Grade problem, lexicase selection achieves a success rate on the test set $s_{\text{test}} = 10$ while tournament selection ($t = 2$), which is the second best selection method on this problem, only achieves $s_{\text{test}} = 1$. The difference is even larger on the Super Anagrams problem, where GP achieves a success rate $s_{\text{test}} = 3$ with lexicase selection while GP does not find any single successful generalizing solution when using any of the other selection methods.

Nevertheless, for some benchmark problems the generalization rate is extremely low ($g < 0.5$). E.g., for the Compare String Lengths and the Super Anagrams problem, solutions that work correctly on the training set have much lower success rates on the associated test set independently of the used selection method. Especially for lexicase selection, the generalization rate is low for many problems as we observe a low generalization rate ($g < 0.3$) for the Compare String Lengths, Grade, and Super Anagrams problem. For the Small Or Large problem, where the few solutions found by the other studied selection methods generalize perfectly ($g = 1.0$), lexicase finds 15 solutions that solve all training cases but only 10 of them correctly solve all test cases ($g = 0.667$). Low generalization rates are problematic when using GP-based program synthesis approaches in real-world use cases (like software development), as users expect that the found programs behave on the test data analogously to the training data.

For all studied benchmark problems and selection methods, $\tilde{n}_{\text{struct}}$ is large meaning that GP finds many structurally different solutions that correctly solve all training instances. As expected, $\tilde{n}_{\text{struct}}$ is on average larger for problems with high success rates (easier problems). However, $\tilde{n}_{\text{struct}}$ is low when using proportionate selection. For tournament (independently of tournament sizes) and lexicase selection, the results are on the same level for most benchmark problems.

Studying $\tilde{b}$ for the different problems and selection methods reveals that $\tilde{b}$ is a good proxy for the generalizability g of the candidate solutions found by GP. For all combinations of problems and selection methods where $g = 1.0$, the measure b is greater or equal to 0.99. In contrast, b is always low ($b < 0.99$) for all combinations of problems and selection methods where the generalizability of the found solutions is also low (marked by 'l'). Thus, for all considered problems and selection methods, b can serve as a proxy for the expected generalizability of the solutions found by GP. The measure b has the nice property that no additional training or test cases are necessary but it can be calculated from random inputs which are usually easy to produce for the problem at hand.

6 CONCLUSIONS

Program synthesis is one of the major research topics in GP. However, program synthesis with GP is still a challenging task and especially the generalizability of the programs generated by GP is a major problem as the generated programs must not only work correctly on the given training cases but also on unseen test cases. As shown by Forstenlechner et al. [5], a larger number of training cases can improve the generalizability of programs generated by GP. Unfortunately, generating training data is usually an expensive task as the output value for every training case must be calculated manually by the user.

Therefore, we suggested in this work a measure b that accurately predicts the generalizability of candidate solutions (solving correctly all training cases) found during a GP run. The measure is based on the proportion of the number of solution candidates that represent the behavioral vector with highest frequency compared to all other candidate solutions. The definition of b is motivated by the finding that for benchmark problems where the synthesized programs generalize well to unseen cases, one behavioral vector is dominating (the vast majority of solution candidates have the same behavior) and the other behavioral vectors different from the dominating one are represented by only a few candidate solutions. On the other hand, for benchmark problems where the synthesized

[3]For generating the AST of a function, we used the Python module `astdump`. Project website: https://pypi.org/project/astdump/.

programs do not generalize well to unseen data, there are more than one relevant behavioral vectors.

We tested the novel measure b on candidate solutions generated by GP with proportionate, tournament, and lexicase selection on a representative set of seven program synthesis benchmark problems. We found that the measure makes high-quality predictions as $b \approx 1.0$ for benchmark problems where the generalization rate is high. For benchmark problems with a low generalization rate, we achieve lower values for b. For values of $b < 0.99$, we suggest researchers as well as practitioners to repeat the GP run with more training cases to give GP the chance to better learn the relevant properties necessary for synthesizing generalizing solutions.

7 LIMITATIONS AND FUTURE WORK

The proposed generalizability measure b performs well on all program synthesis benchmark problems studied in this work and we generally recommend to repeat a GP run with more training data (including all required edge cases) if $b < 0.99$. However, even if we did not observe this in our experiments, it could be possible that there is one dominating behavioral group that represents a non-generalizing solution. In such cases, the metric would wrongly predict that GP has produced generalizing solutions with high probability as $b \approx 1.0$.

Consequently, we will study in future work the proposed measure on additional benchmark problems and for GP runs using differently sized training sets. Furthermore, we will analyze different sample sizes for the vector of random inputs.

REFERENCES

[1] David Fagan, Michael Fenton, and Michael O'Neill. 2016. Exploring position independent initialisation in grammatical evolution. In *2016 IEEE Congress on Evolutionary Computation*. IEEE, 5060–5067.

[2] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. 2017. Ponyge2: grammatical evolution in Python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1194–1201.

[3] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In *Genetic Programming*. Springer International Publishing, Cham, 262–277.

[4] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Towards effective semantic operators for program synthesis in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1119–1126.

[5] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE.

[6] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. Semantics-based crossover for program synthesis in genetic programming. In *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 58–71.

[7] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Extending program synthesis grammars for grammar-guided genetic programming. In *International Conference on Parallel Problem Solving from Nature*. Springer, 197–208.

[8] David E Goldberg. 1989. *Genetic algorithms in search optimization and machine learning*. Addison-Wesley.

[9] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving generalization of evolved programs through automatic simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 937–944.

[10] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1127–1134.

[11] Thomas Helmuth and Lee Spector. 2015. Detailed problem descriptions for general program synthesis benchmark suite. *School of Computer Science, University of Massachusetts Amherst, Tech. Rep.* (2015).

[12] Thomas Helmuth and Lee Spector. 2015. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1039–1046.

[13] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1039–1046.

[14] Jonathan Kelly, Erik Hemberg, and Una-May O'Reilly. 2019. Improving genetic programming with novel exploration-exploitation control. In *Genetic Programming*. Springer International Publishing, Cham, 64–80.

[15] David J Montana. 1995. Strongly typed genetic programming. *Evolutionary computation* 3, 2 (1995), 199–230.

[16] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming*. Lulu.com.

[17] Conor Ryan, JJ Collins, and Michael O. Neill. 1998. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 83–96.

[18] Dominik Sobania. 2021. On the generalizability of programs synthesized by grammar-guided genetic programming. In *Genetic Programming*. Springer International Publishing, Cham, 130–145.

[19] Dominik Sobania and Franz Rothlauf. 2019. Teaching GP to program like a human software developer: using perplexity pressure to guide program synthesis approaches. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1065–1074.

[20] Dominik Sobania and Franz Rothlauf. 2020. Challenges of program synthesis with grammatical evolution. In *Genetic Programming*. Springer International Publishing, Cham, 211–227.

[21] Lee Spector. 2012. 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*. 401–408.

[22] Lee Spector and Alan Robinson. 2002. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines* 3, 1 (2002), 7–40.