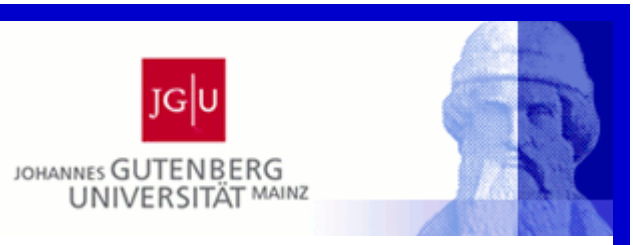

Gutenberg School of Management and Economics

& Research Unit “Interdisciplinary Public Policy”

Discussion Paper Series

A Comparison of Large Language Models and Genetic Programming for Program Synthesis

Dominik Sobania, Justyna Petke, Martin Briesch, and Franz Rothlauf

September 16, 2024

Discussion paper number 2414

Johannes Gutenberg University Mainz
Gutenberg School of Management and Economics
Jakob-Welder-Weg 9
55128 Mainz
Germany
https://wiwi.uni-mainz.de/

Contact details

Dominik Sobania
Johannes-Gutenberg University
Department of Law and Economics
55099 Mainz
Germany
dsobania@uni-mainz.de

Justyna Petke
University College London
Department of Computer Science
London
United Kingdom
j.petke@ucl.ac.uk

Martin Briesch
Johannes-Gutenberg University
Department of Law and Economics
55099 Mainz
Germany
briesch@uni-mainz.de

Franz Rothlauf
Johannes-Gutenberg University
Department of Law and Economics
55099 Mainz
Germany
rothlauf@uni-mainz.de

A Comparison of Large Language Models and Genetic Programming for Program Synthesis

Dominik Sobania, Justyna Petke, Martin Briesch, and Franz Rothlauf

***Abstract*—Large language models have recently become known for their ability to generate computer programs, especially through tools such as GitHub Copilot, a domain where genetic programming has been very successful so far. Although they require different inputs (free-text vs. input/output examples) their goal is the same – program synthesis. Therefore, in this work we compare how well GitHub Copilot and genetic programming perform on common program synthesis benchmark problems. We study the structure and diversity of the generated programs by using well-known software metrics. We find that GitHub Copilot and genetic programming solve a similar number of benchmark problems (85.2% vs. 77.8%, respectively). We find that GitHub Copilot generated smaller and less complex programs as genetic programming, while genetic programming is able to find new and unique problem solving strategies. This increase in diversity of solutions comes at a cost. When analyzing the success rates for 100 runs per problem, GitHub Copilot outperforms genetic programming on over 50% of the problems.**

***Index Terms*—Program Synthesis, Genetic Programming, Large Language Models, Codex, GitHub Copilot, Software Engineering.**

I. Introduction

SINCE the introduction of GitHub Copilot[1], a program synthesis approach based on large language models is available to support real-world software developers in their daily work. GitHub Copilot, which is available as an add-on for several common development environments, is based on the large language model Codex [1], which has been trained on large amounts of publicly available source code. The system gives the programmer both suggestions for completing existing code as well as suggestions based on natural language descriptions entered as comments.

Another approach that is known for its performance in automatic program synthesis is genetic programming (GP) [2], [3]. Instead of natural language descriptions, in GP-based program synthesis, the user's intent is usually expressed by a set of given input/output examples (test cases). Based on these input/output examples, GP employs an evolutionary process to search for programs that meet the user's intent.

To compare the performance of large language model based and evolutionary program synthesis, we applied in a recent conference paper [4] GitHub Copilot on the program synthesis benchmark problems introduced by Helmuth et al. [5], [6], which are commonly used to evaluate GP-based program synthesis approaches, and compared the results achieved by GitHub Copilot to those reported for GP in the literature. We found that, despite their differences in the definition of the user's intent, the overall number of benchmark problems solved by GitHub Copilot and GP is on the same level. However, in our previous conference paper [4], we only made a binary assessment (solved or not) for GitHub Copilot and GP on the benchmark problems, which does not provide any insight into the stability of the approaches. For instance, is a problem only solved by chance or is it solved on a regular basis? Furthermore, due to space limitations, we did not compare the structure of the programs suggested by GitHub Copilot and GP with common software metrics used in the program synthesis literature [7], [8], which could open up further research directions in program synthesis.

Therefore, we extend our previous conference paper [4] and carry out a more comprehensive comparison of GitHub Copilot and GP on common program synthesis benchmark problems. In addition to that, we analyze and compare the structure and the diversity of the generated programs using well-known software metrics. Furthermore, we identify different strengths and weaknesses of the two approaches and discuss potential future research directions.

To evaluate GitHub Copilot, we use the add-on[2] for the Visual Studio Code development environment. For every benchmark problem, we provide GitHub Copilot the function's signature and the textual problem description as a Python comment and evaluate Copilot's suggestions. Since Copilot returns only a maximum of 10 suggestions, we repeat this step several times. For GP, we take the results from the literature. More precisely, from recent GP papers reporting success rates achieved with PushGP [9] and grammar-guided GP [10] for the problems from the first (PSB1) [5] and/or the second program synthesis benchmark suite (PSB2) [6]. Both GP papers present recent and comprehensive results for their respective approaches. For comparison, we evaluate GitHub Copilot on the same benchmark problems.

To analyze and compare the structure and the diversity of the programs generated by GitHub Copilot and GP, we use common software metrics, e.g., the number of source lines of code (SLOC) and the cyclomatic complexity [11], which allow us to draw conclusions about the programs' source code. We perform the structure and diversity analysis on a representative subset of eight problems from PSB1 and PSB2. For GitHub Copilot, we use the source code generated in our experiments.

Dominik Sobania, Martin Briesch and Franz Rothlauf are with the Johannes Gutenberg University, Mainz, Germany (dsobania@uni-mainz.de, briesch@uni-mainz.de, rothlauf@uni-mainz.de).

Justyna Petke is with the University College London, London, United Kingdom (j.petke@ucl.ac.uk).

[1]https://github.com/features/copilot

[2]https://marketplace.visualstudio.com/items?itemName=GitHub.copilot

For GP, we use solutions generated with a grammar-guided GP approach provided by a recent paper [12].

We find that overall GitHub Copilot and GP can solve a similar number of benchmark problems (85.2% vs. up to 77.8%, respectively). However, in terms of the success rates GitHub Copilot significantly outperforms the GP approaches on over 50% of the benchmark problems. Further, we find that the structure of the generated programs strongly differs based on the used approach. The programs generated with the studied grammar-guided GP approach are on average significantly larger and more complex than the programs generated with GitHub Copilot. We observe also for the programs generated by GitHub Copilot a lower variance and diversity for the studied software metrics compared to those obtained by the studied GP approach. However, we should keep in mind that GitHub Copilot may have already seen many of the program synthesis benchmark problems during its training phase. Additionally, the training of such large models is computationally very intensive, while GP usually requires less compute and consumes a lower amount of energy. Which approach should be used depends more on the task at hand. Due to its integration into several code editors, GitHub Copilot is well suited as support system in software development. On the other hand, GP can be easily adapted to a wide variety of tasks, e.g., by adjusting the grammar or the function set. In addition, GP can discover novel programs which potentially make use of new solution strategies.

To sum up, our contributions are: **1)** A comparison of the performance of GitHub Copilot and different GP-based program synthesis approaches. **2)** An analysis of the structure and the diversity of the generated programs with common software metrics. **3)** The discussion of the different strengths and weaknesses of GitHub Copilot and GP as well as the identification of future research directions in program synthesis.

Following this introduction, Sect. II presents recent work on GP-based program synthesis as well as program synthesis using large language models. Section III describes our methodology before presenting the performance comparison of GitHub Copilot and GP in Sect. IV. In Sect. V, we discuss the results and identify some further research directions for GP-based program synthesis. Section VII concludes the paper.

II. Related Work

In this section, we present fundamental and recent work on program synthesis approaches based on large language models and GP.

A. Program Synthesis with Large Language Models

In the area of large language models, the problem of program synthesis is often treated as a natural language processing (NLP) task. For a given problem description in natural language, the corresponding source code required to solve the problem should be returned. NLP has made a lot of progress in recent years, which, in addition to higher computing power, can also be attributed to new deep learning architectures, like the transformer model [13].

The success of the transformer model is based on two key innovations: positional encoding and attention. Due to positional encoding, transformer models can be parallelized well, in contrast to gated recurrent unit (RNN) [14] and long short-term memory (LSTM) [15] networks, and can also be trained on large amounts of data. And due to attention mechanisms, transformers can refer to the relevant context. Figure 1 shows an overview of the elements of an encoder-decoder transformer architecture.

Well-known models for general purposes based on the transformer architecture are, e.g., the BERT or the GPT language model families. In addition to that, models specialized on the generation of source code have recently been released, such as CodeBERT [16], PyMT5 [17], AlphaCode [18], and Codex [1]. These specialized models are usually (pre-)trained on large amounts of freely available open source code. For example the Codex model, which powers GitHub Copilot, was first pre-trained with 159 GB of open source code and then fine-tuned with smaller data sets, e.g., from programming competition websites [1].

Beyond pure code generation, large language models can also be used to improve existing software, e.g., for automatic program repair [19] or refactoring [20].

However, the use of large language models also harbours risks, as they are prone to hallucinatory behaviour and may also provide incorrect answers [21], [22]. It is also challenging that a clear separation between training and test set, as in traditional machine learning, is difficult in practice, as it can be assumed that large language models have already seen many of the classic test problems during training [23].

B. Program Synthesis with GP

In GP, program synthesis is one of the major topics since the field's inception. The first GP paper by Cramer [2] already deals with code generation, and also Koza's first book on GP [3] describes the evolution of source code (e.g., using LISP S-expressions). Different to large language models, in GP usually input/output examples are used to define the user's intent. Based on this definition, GP searches for programs that meet the given requirements in an evolutionary process.

An important step towards comparability of different GP approaches was achieved with the program synthesis benchmark suites PSB1 and PSB2 by Helmuth et al. [5], [6]. In recent years, great progress has been made on these benchmark problems, for example with PushGP [24], [25], grammar-guided GP [10], [26], as well as other GP-based program synthesis approaches [27], [28]. These approaches differ mainly in the representation used and the way different data types are handled.

In PushGP, the programming language Push [29], [30], which was specially created for program synthesis with GP, is used for representation. This programming language separates data types by utilizing different stacks, one stack for each supported data type and one for the program's commands. In recent years, advances with PushGP have been made through the use of a novel mutation operator, uniform mutation by addition and deletion (UMAD) [24], and new selection

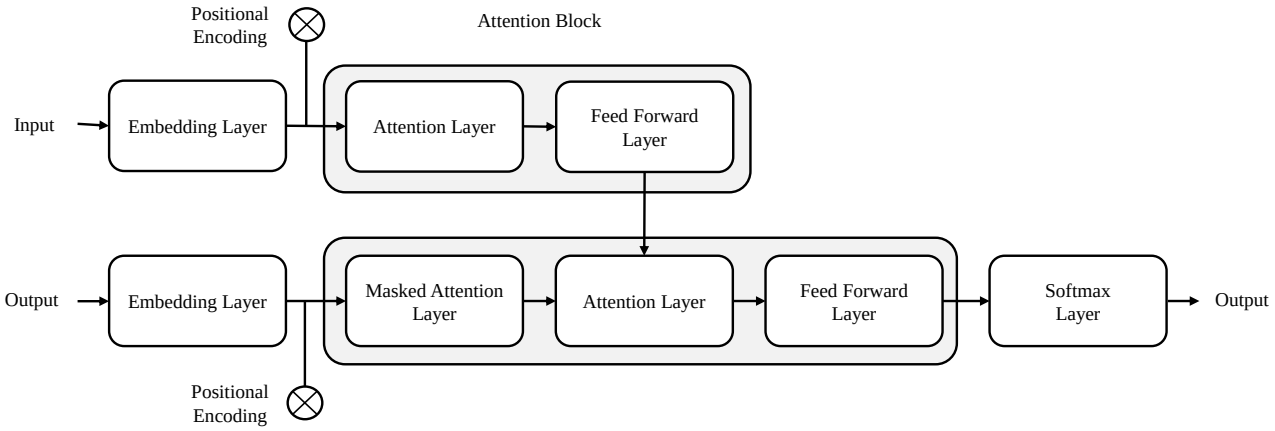

Fig. 1: Transformer model (taken from [4], based on the illustration by Vaswani et al. [13]).

methods (based on lexicase selection) [9], [12], [25], [31]. However, the Push language has not seen a wider uptake in real-world software development so far.

With grammar-guided GP approaches [32], however, also programs in programming languages like Python can be evolved [10], [26]. During evolution, a context-free grammar is used to comply with the rules of the programming language and to distinguish between different data types. Recent work on program synthesis with grammar-guided GP focuses, e.g., on the usage of domain-knowledge [7], [33], [34], generalization to unseen data [35]–[37], and the use of functional programming languages [38]. Furthermore, new selection methods are being studied for grammar-guided GP [12], [39], [40]. For further insights into the fundamentals and an overview of additional grammar-guided GP approaches, we refer to a survey by McKay et al. [41].

In addition to the benchmark problems from PSB1 and PSB2, Boolean logic problems are often studied in the literature on GP-based program synthesis [42], [43]. Of particular interest is, for example, recent work by Liskowski et al. [44] in which they investigate how to search the latent space of an autoencoder to find suitable programs.

A related field of research where similar methods as in GP-based program synthesis are studied is genetic improvement (GI) [45]. Instead of generating source code from scratch, GI methods improve existing software, improving its either functional or non-functional properties, e.g., to automatically fix bugs [46] or improve runtime [47], respectively. In recent work by Yuan et al. [27], ideas from GI even found their way back into program synthesis and helped to outperform even existing GP-based program synthesis approaches on some problems from PSB1.

III. Experimental Design

This work's experimental method can be divided into two parts. In the first part, we compare the performance of GitHub Copilot and GP on common benchmark problems with respect to the correctness of the code suggestions. In the second part, we analyze and compare programs suggested by GitHub Copilot and GP on a representative selection of problems using different code metrics. Additionally, we discuss the strengths and weaknesses of the two approaches and identify further research directions.

A. Performance Comparison of GitHub Copilot and GP

To collect suggestions from GitHub Copilot, we use the add-on for the Visual Studio Code development environment, to be as close as possible to the experience of a real software developer during the evaluation.[3] For each benchmark problem from PSB1 and PSB2, we provide GitHub Copilot the generic function signature, including the input parameters and types, and the textual problem description from either PSB1 or PSB2 as a Python comment. For some problems from PSB1 we have adjusted the problem description such that results are returned and not printed. This way, the problem descriptions are consistent across all benchmark problems and in line with the novel benchmark suite PSB2. Based on this input, we expect from GitHub Copilot suggestions for the completion of the function. The used textual problem descriptions as well as the code generated by GitHub Copilot in our experiments are available online.[4]

```
1 # The textual problem as defined in
2 # PSB1 or PSB2.
3 def myfunc(str1: str, str2: str):
4     # To be completed by GitHub Copilot.
```

Fig. 2: An abstract function definition as input for GitHub Copilot using the textual problem description from PSB1 or PSB2 as a Python comment.

Figure 2 shows an example abstract function definition that could be used as input for GitHub Copilot. The Python comment in lines 1-2 gives the textual problem description, The generic function signature is given in line 3, and beginning from line 4 we expect from GitHub Copilot suggestions for possible completions.

[3]All experiments presented in Section IV requesting GitHub Copilot were performed in June 2023.

[4]https://github.com/domsob/github-copilot-generated-programs-2023/.

For GP-based program synthesis we do not perform new experiments but use the results reported in the literature for PushGP and grammar-guided GP. For PushGP, we take the results from a recent paper [9] that reports success rates for the problems from PSB1 and PSB2. This paper is well suited for comparison, as almost all benchmark problems from PSB1 and PSB2 are covered and results for the two common selection methods currently studied in the area of program synthesis, namely standard lexicase [48] and down-sampled lexicase selection [28], are reported. For grammar-guided GP, we take the results from [10] which reports success rates for most of the benchmark problems from PSB1. Of the benchmark problems from PSB2, only a very small number of problems have been examined with grammar-guided GP so far in the literature. Therefore, for the problems from PSB2, we only compare GitHub Copilot with the results for PushGP.

As the literature on GP-based program synthesis typically studies 100 runs [10], [36], [49], we also analyze 100 suggestions from GitHub Copilot. However, the suggestions in the add-on for Visual Studio Code are limited to a maximum of 10 suggestions. Consequently, we repeat the request to GitHub Copilot until the required number of code suggestions is collected. This gives GitHub Copilot's code suggestions a logical order, which allows us to evaluate whether GitHub Copilot's first suggestion is already a correct solution. This reflects the use in practice, since a software developer would also start examining the first code suggestion.

In order to check the correctness of the suggested programs, we use $1,000$ test cases for each benchmark problem.[5] For the problems from PSB1, we use test cases based on the cases provided by the PonyGE2 framework [50] which follow the design principles suggested in the PSB1 paper [5]. For the PSB2 problems, we use the Python module accompanying the PSB2 paper [6] to collect the required test cases.[6]

B. Analysis of the Suggested Programs

In addition to the success rates, the structure of the generated programs is also important if the generated code should be used in real-world software maintained by human programmers. Therefore, we compare the source code generated by GitHub Copilot and GP using common software metrics known from the software engineering and GP literature as a proxy for complexity and readability [7], [8], [51].

The code metrics we use can either be calculated directly on the source code or are based on a tree representation of the given code snippet (a Python function). To calculate the tree-based metrics, we first transform every generated function into its abstract syntax tree (AST) representation using the Python module `astdump`.[7] The code metrics are defined as follows:

- **Source lines of code (SLOC)**: The number of code lines in the generated program without empty and commented lines.
- **Cyclomatic complexity**: The number of decision branches introduced by a generated program [11]. We calculate the cyclomatic complexity with the Python module `radon`.[8]
- **AST nodes**: The number of tree nodes in the program's AST representation.
- **AST depth**: The number of edges from the AST's root node to the deepest leaf node.

Each of the above code metrics measures unique characteristics of a given program's structure. SLOC is the common metric to measure the size of a program. The cyclomatic complexity can be seen as a proxy for the number of used control structures like loops and conditionals. Finally, the AST-based metrics provide information about the number of used elements (e.g., variables, function calls, etc.) and their nesting.

In addition to the structure, we measure and compare the diversity of the programs generated by GitHub Copilot and GP to check if always the same programs are generated in different runs or if new solution strategies are found. Therefore, we measure the number of unique programs generated per benchmark problem. However, since programs can be identical except for the use of different function and variable names, we also measure the number of unique ASTs, which abstract from the mentioned details.

For the evaluation (calculation of code metrics and diversity), we use a representative subset of eight benchmark problems from PSB1 and PSB2 for clarity and to save computational effort. For the analysis of GitHub Copilot, we use the solutions generated in our experiments (see Sect. III-A). For the analysis of GP, we use solutions generated with a grammar-guided GP approach provided by the supplementary material[9] of a recent paper [12]. We do not study the structure of programs generated with PushGP in this experiment because most of the software metrics would not give us any meaningful information for Push code. However, with the grammar-guided GP from [12], the programs were generated in Python and are therefore well suited for such an investigation.

IV. Comparison of GitHub Copilot and GP

In this section, we compare the code suggestions of GitHub Copilot and different GP variants and analyze their correctness on given test data as well as the structure and diversity of the generated code.

A. Performance Comparison

First, we compare the performance of GitHub Copilot and different GP variants on given test data by analyzing the achieved success rates on the benchmark problems from PSB1 and PSB2. The success rate indicates how many of the 100 solution candidates examined per approach work correctly on all of the used test cases. For GitHub Copilot, we also study whether the first code suggestion works correctly on all test cases, in order to be able to assess whether a working solution can be found quickly in practical software development or

[5] With the exception of the problems Sum of Squares and Wallis Pi, since only a smaller number of cases is defined in the literature for these benchmark problems.

[6] For the eight benchmark problems we use for structural code analysis (see Sect. III-B), we use the same test data as in [12] for better comparability.

[7] https://pypi.org/project/astdump/.

[8] https://pypi.org/project/radon/.

[9] https://gitlab.rlp.net/mbriesc/informed-down-sampled-lexicase-selection/.

whether many solution candidates have to be examined which can be quite time consuming for programmers.

Table I shows the success rates achieved by GitHub Copilot as well as PushGP with standard and down-sampled lexicase selection and grammar-guided GP (denoted as G3P) with standard lexicase selection on the 29 benchmark problems from PSB1. If GitHub Copilot's first code suggestion for a benchmark problem works correctly on all test cases, we indicated this with a check mark (✓). A cross (✗) indicates that the first suggestion is not correct. The reported PushGP results are taken from [9] and grammar-guided GP results are taken from [10]. For benchmark problems for which no GP results are reported in the papers, we have indicated this with a minus (-) sign. Best success rates are printed in **bold** font and significant improvements in comparison to the other approaches at the $\alpha = 0.05$ level are **underlined**. For statistical testing, we used a two-sided proportion z-test with Bonferroni correction.

Table II shows the same analysis for the 25 benchmark problems from PSB2. Again, the PushGP results are taken from [9]. Grammar-guided GP results are not reported as only a small number of benchmark problems from PSB2 have been examined with grammar-guided GP so far in the literature.

Overall, we see that the number of solved benchmark problems is on the same level for GitHub Copilot and PushGP. For PSB1, GitHub Copilot solved 26 and PushGP 25 (using down-sampled lexicase selection) out of 29 problems. Grammar-guided GP solved 17 benchmark problems from PSB1, a lower number than the other two approaches. For PSB2, GitHub Copilot solved 20 and PushGP 17 out of 25 problems.

However, if we study the success rates, we see significantly higher values for GitHub Copilot than for the GP variants on PSB1 as well as on PSB2. GitHub Copilot significantly outperforms the GP approaches on 13 benchmark problems from PSB1 and 15 benchmark problems from PSB2. For the GP approaches we only see significant improvements for the problems REPLACE SPACE WITH NEWLINE, X-WORD LINES (both PSB1), and COIN SUMS (PSB2) with PushGP. For the benchmark problems COLLATZ NUMBERS, WALLIS PI, and WORD STATS, no results are reported for PushGP in [9] and the success rates reported for grammar-guided GP are all 0 for these problems. However, a recent survey on GP-based program synthesis [52] finds that also no other GP approach has been successful on these problems so far. With GitHub Copilot, however, at least the COLLATZ NUMBERS problem can be solved (73 successful solutions).

For many benchmark problems, we see high success rates for GitHub Copilot. However, the success rates achieved by GitHub Copilot on PSB2 are on average lower compared to the results for PSB1. Furthermore, for five benchmark problems, GitHub Copilot did not find a single working solution (BOUNCING BALLS, BOWLING, DICE GAME, MASTERMIND, and SOLVE BOOLEAN) and for some other problems we observe very low success rates (e.g., CUT VECTOR, LUHN, and SNOW DAY). We expect that this is due to a low precision of the textual problem description for these benchmark problems. We will discuss this further in Sect. V.

If we look at the first code suggestions given by GitHub Copilot, we see that the first suggestion is not always correct for every benchmark problem that can be solved by GitHub Copilot in principle. For some problems, a programmer has to search through further suggestions to find a working solution. However, if we consider the problems from PSB1 and PSB2 combined, we see that for more than 50% of the problems solvable by GitHub Copilot, the first suggestion already works correctly on all test cases.

In summary, GitHub Copilot solves about the same number of benchmark problems as a recent PushGP approach (lower values observed for grammar-guided GP) while GitHub Copilot achieves significantly higher success rates than the studied GP variants on many problems. However, when comparing the results, we should keep in mind that although the inference time for large language models is usually relatively short, the computational effort required to train state-of-the-art large language models is much greater than the time required to execute a GP run.

B. Analysis of the Structure and the Diversity

Second, we analyze the structure and diversity of the source code synthesized with GitHub Copilot as well as with GP. Since source code should be of low complexity, easy to read, and maintain, we measure the structure of the generated source code with common software metrics. In order to study whether always the same source code is generated or whether novel solutions are found, we also analyze the number of unique code suggestions given by GitHub Copilot and GP.

Figures 3-6 show box-plots of common software metrics for the source code generated by GitHub Copilot and GP for a representative selection of eight benchmark problems from PSB1 and PSB2. For the GitHub Copilot results we measured the software metrics on the code generated in our experiments. For the GP results, we measured the metrics on code taken from the associated repository from a recent paper [12] using a grammar-guided GP approach. Specifically, the plots show comparisons for SLOC (Fig. 3), cyclomatic complexity (Fig. 4), AST nodes (Fig. 5), and AST depth (Fig. 6).

For most of the studied benchmark problems, we see that the code generated by the grammar-guided GP approach is larger (larger values for SLOC and AST nodes), more complex (higher values for the cyclomatic complexity), and more nested (large values for the AST depth). For example for the FIZZ BUZZ problem, we observe for the grammar-guided GP a median SLOC value of around 30 while GitHub Copilot only generated solutions with a median value of around 9.

Furthermore, the variance of the software metrics is larger for the solutions generated by the grammar-guided GP compared to the ones generated by GitHub Copilot for almost all studied benchmark problems. An exception is the SCRABBLE SCORE problem, where we observe larger variances and a significantly higher median value of the cyclomatic complexity for the GitHub Copilot solutions compared to the solutions generated by the grammar-guided GP. However, this is not surprising if we take a closer look at the generated solutions. Figure 7 shows a code example generated with GitHub Copilot

TABLE I: Success rates achieved by GitHub Copilot, PushGP, and grammar-guided GP on the benchmark problems from PSB1. For GitHub Copilot, we also report whether the first given suggestion passes all the test cases (denoted with ✓) or not (✗). The PushGP results are taken from a recent paper [9] comparing standard and down-sampled lexicase selection. The grammar-guided GP (denoted as G3P) results for standard lexicase selection are taken from [10]. Best success rates are printed in **bold** font. Significant improvements ($\alpha = 0.05$) in comparison to the other approaches are **underlined**.

Benchmark Problem	Large Language Model		PushGP [9]		G3P [10]
	GitHub Copilot (First)	GitHub Copilot	Standard Lexicase	Down-Sampled Lexicase	Standard Lexicase
CHECKSUM	✓	**89**	1	18	0
COLLATZ NUMBERS	✓	**73**	-	-	0
COMPARE STRING LENGTHS	✓	**70**	32	51	0
COUNT ODDS	✓	**98**	8	11	3
DIGITS	✗	0	19	**28**	0
DOUBLE LETTERS	✓	**88**	19	50	0
EVEN SQUARES	✗	**11**	0	2	0
FOR LOOP INDEX	✓	**72**	2	5	6
GRADE	✓	**84**	0	2	31
LAST INDEX OF ZERO	✓	61	62	**65**	44
MEDIAN	✓	**79**	55	69	59
MIRROR IMAGE	✓	70	**100**	99	25
NEGATIVE TO ZERO	✓	**99**	80	82	13
NUMBER IO	✓	93	98	**99**	83
PIG LATIN	✓	**54**	0	0	3
REPLACE SPACE WITH NEWLINE	✓	87	87	**100**	16
SCRABBLE SCORE	✗	**35**	13	31	1
SMALL OR LARGE	✗	**51**	7	22	9
SMALLEST	✓	66	**100**	98	73
STRING DIFFERENCES	✗	**9**	0	1	-
STRING LENGTHS BACKWARDS	✗	60	94	**95**	18
SUM OF SQUARES	✓	**90**	21	25	5
SUPER ANAGRAMS	✗	**55**	4	4	0
SYLLABLES	✓	**96**	38	64	39
VECTOR AVERAGE	✓	92	88	**97**	0
VECTORS SUMMED	✓	**87**	11	21	21
WALLIS PI	✗	0	-	-	0
WORD STATS	✗	0	-	-	0
X-WORD LINES	✗	1	61	**91**	0
Σ (Solved)	19	26	22	25	17

TABLE II: Success rates achieved by GitHub Copilot and PushGP on the benchmark problems from PSB2. We also report for GitHub Copilot whether the first given suggestion passes all the test cases (denoted with ✓) or not (✗). The PushGP results are taken from a recent paper [9] comparing standard and down-sampled lexicase selection for program synthesis. Best success rates are printed in **bold** font. Significant improvements ($\alpha = 0.05$) in comparison to the other approaches are **underlined**.

Benchmark Problem	Large Language Model		PushGP [9]	
	GitHub Copilot (First)	GitHub Copilot	Standard Lexicase	Down-Sampled Lexicase
BASEMENT	✓	**95**	1	2
BOUNCING BALLS	✗	0	0	**3**
BOWLING	✗	0	0	0
CAMEL CASE	✓	**31**	1	4
COIN SUMS	✗	12	2	**39**
CUT VECTOR	✗	**1**	0	0
DICE GAME	✗	0	0	**1**
FIND PAIR	✗	**41**	4	20
FIZZ BUZZ	✓	**89**	25	74
FUEL COST	✓	**97**	50	67
GCD	✓	**80**	8	20
INDICES OF SUBSTRING	✓	**82**	0	4
LEADERS	✓	**67**	0	0
LUHN	✗	**6**	0	0
MASTERMIND	✗	0	0	0
MIDDLE CHARACTER	✗	**98**	57	79
PAIRED DIGITS	✗	**88**	8	17
SHOPPING LIST	✓	**75**	0	0
SNOW DAY	✗	**10**	4	7
SOLVE BOOLEAN	✗	0	5	5
SPIN WORDS	✓	**96**	0	0
SQUARE DIGITS	✗	**55**	0	2
SUBSTITUTION CIPHER	✓	78	61	**86**
TWITTER	✓	**89**	31	52
VECTOR DISTANCE	✓	**79**	0	0
Σ (Solved)	12	20	13	17

and Fig. 8 an example generated with the grammar-guided GP for the SCRABBLE SCORE problem. We see that GitHub Copilot's solution (Fig. 7) contains many conditionals and logical operators which notably increase the cyclomatic complexity. The solution of the grammar-guided GP makes use of domain knowledge and higher-order functions (like `map()`) which is common in GP-based program synthesis [33], [53]. In the given example, domain knowledge is introduced via the variable `scrabblescore`, which contains the Scrabble scores for each letter, which simplifies the problem for the grammar-guided GP. It is also noticeable that GitHub Copilot's code is much better structured than the code generated by the grammar-guided GP, which is hard to read and bloated. However, it can be assumed that GitHub Copilot has knowledge about the Scrabble scores of individual letters due to the pre-training of its underlying model. This knowledge seems to be only expressed differently in the generated code and is less noticeable in comparison to the GP-generated code as no predefined variables are used. We will discuss this in more detail in Sect. V.

For the analysis of the diversity of the generated code, Table III shows the success rates as well as the number of unique solutions based on the source code and on the AST (which abstracts in our implementation from details like

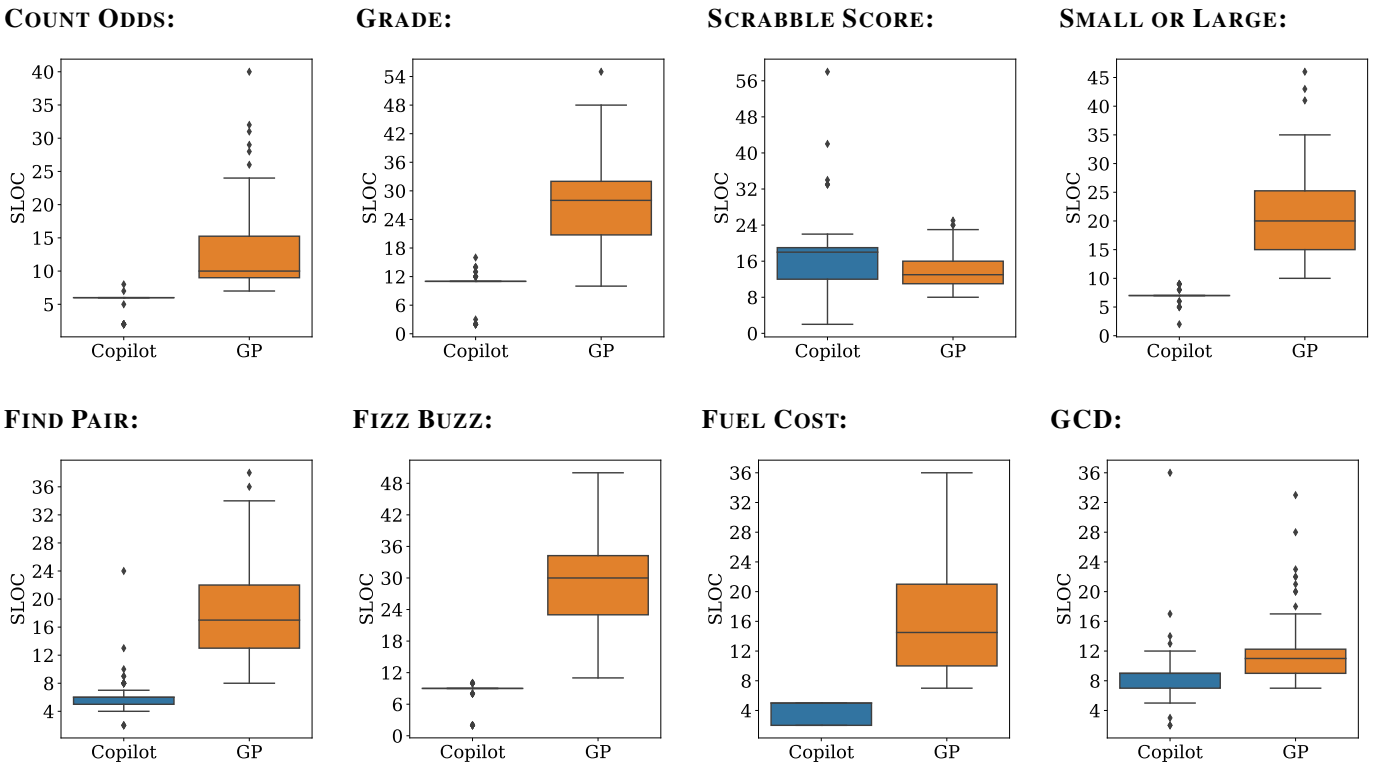

Fig. 3: Box-plots of SLOC for GitHub Copilot and GP for all considered benchmark problems.

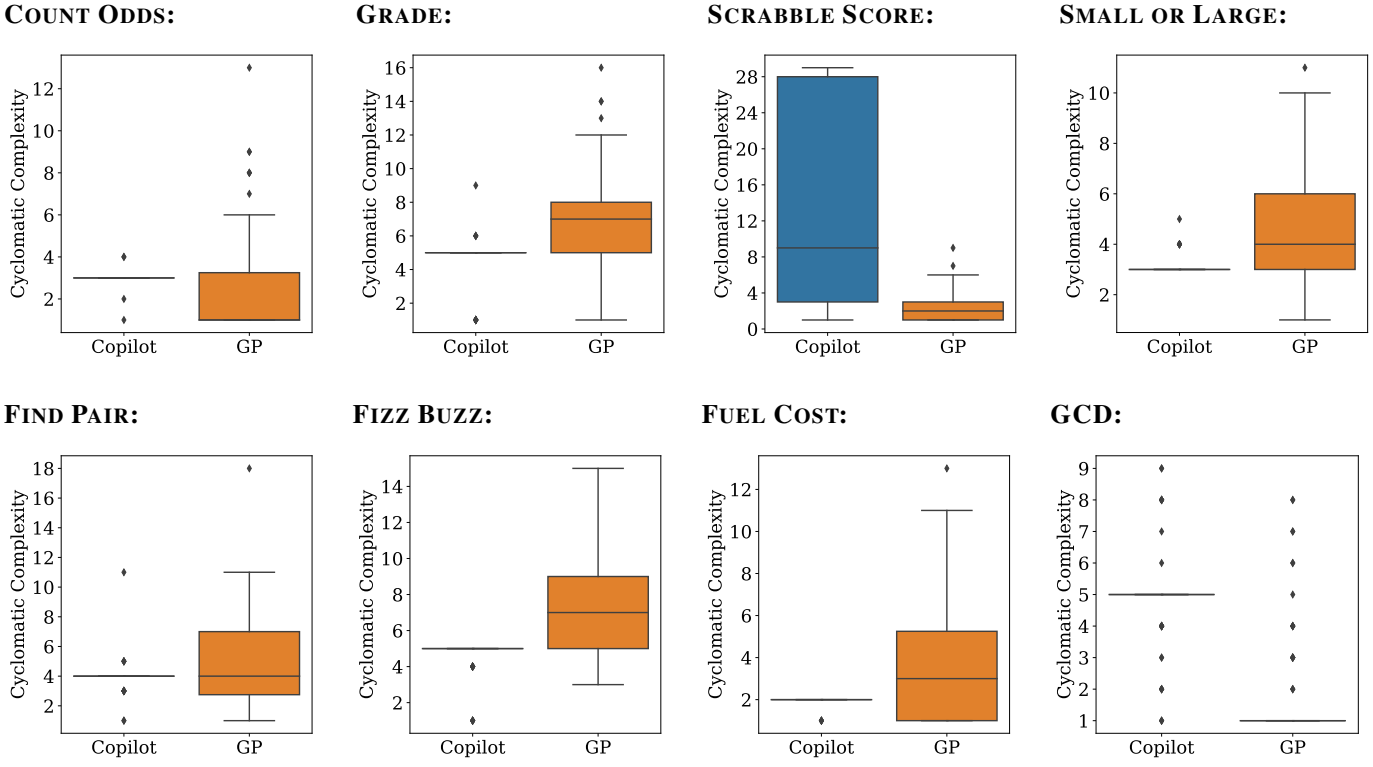

Fig. 4: Box-plots of the cyclomatic complexity for GitHub Copilot and GP for all considered benchmark problems.

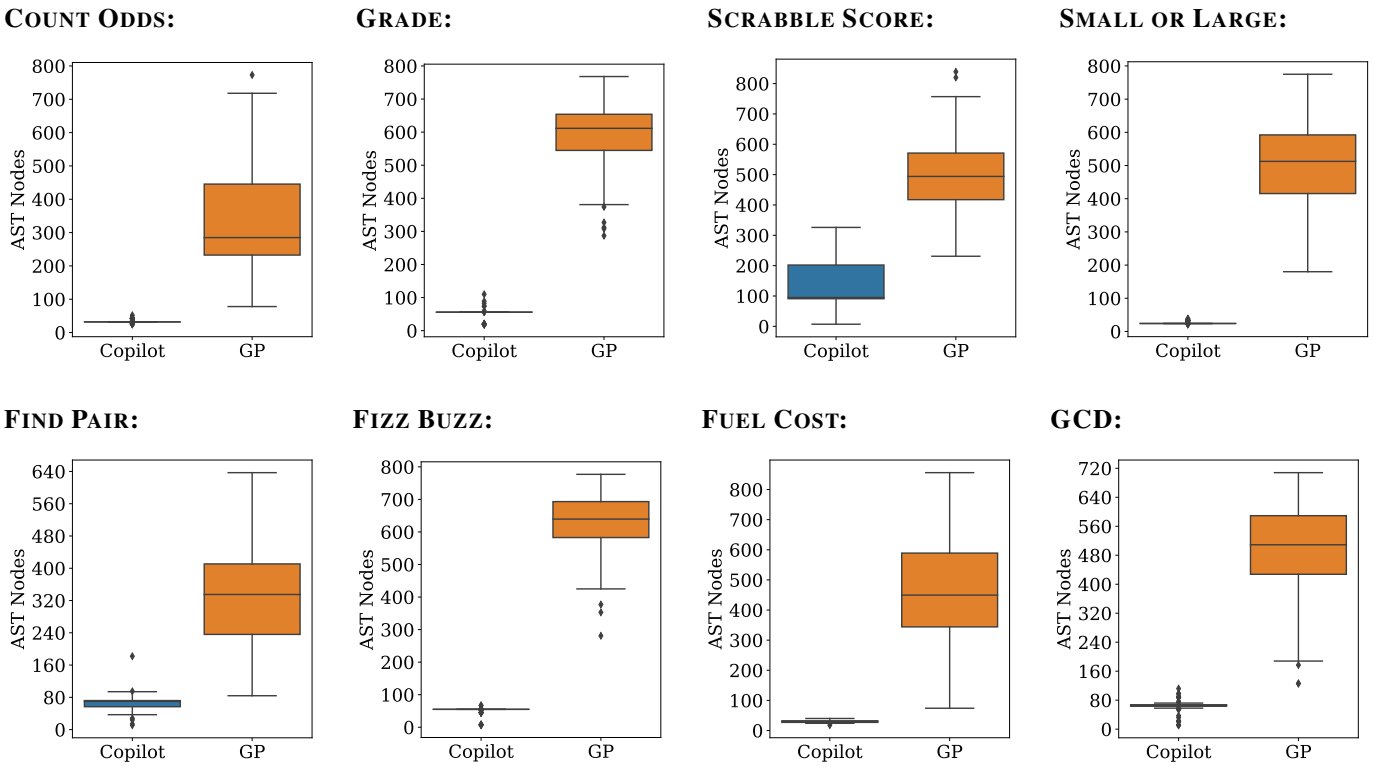

Fig. 5: Box-plots of the number of AST nodes for GitHub Copilot and GP for all considered benchmark problems.

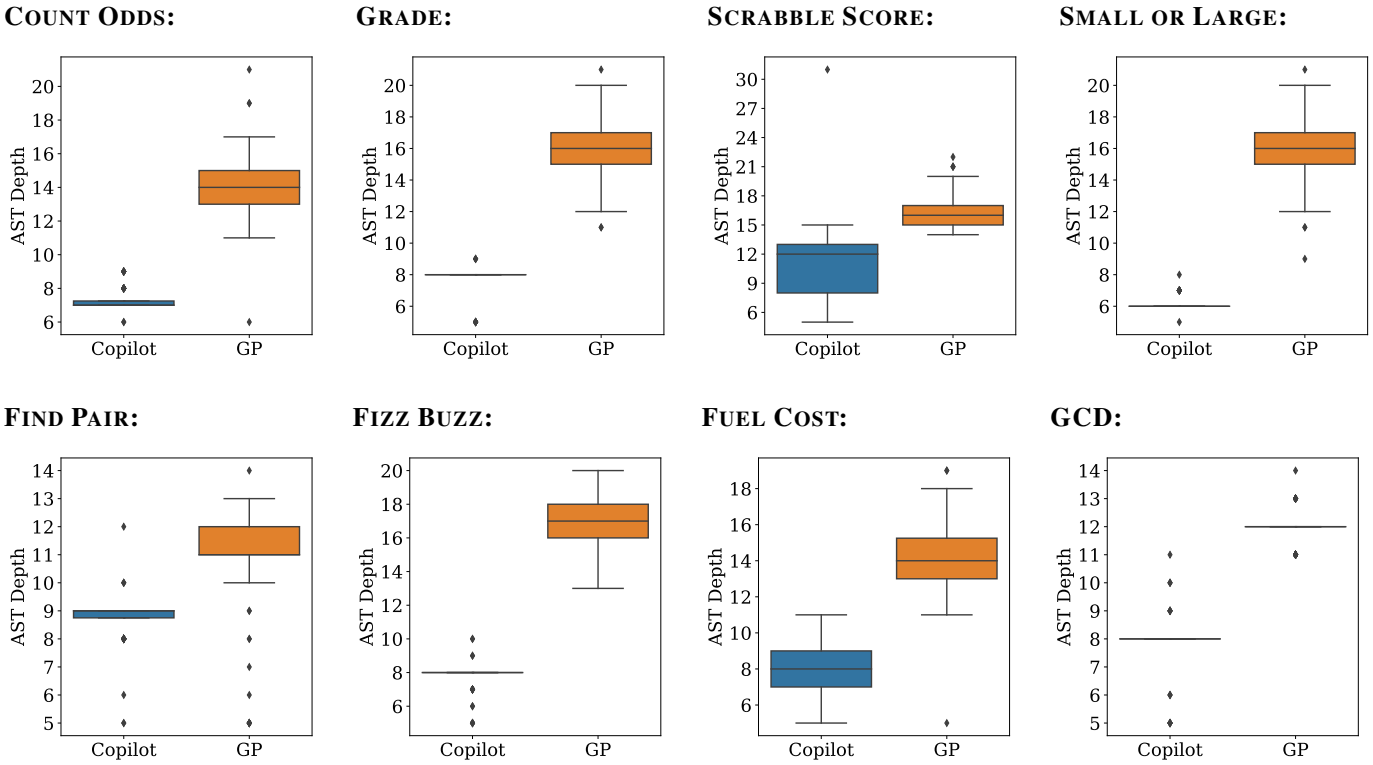

Fig. 6: Box-plots of the AST depth for GitHub Copilot and GP for all considered benchmark problems.

```python
def myfunc(str1: str):
    str1 = str1.upper()
    sum = 0
    for i in str1:
        if i == 'A' or i == 'E' or i == 'I' or i == 'L' or i == 'N' or i == 'O' \
            or i == 'R' or i == 'S' or i == 'T' or i == 'U':
            sum += 1
        elif i == 'D' or i == 'G':
            sum += 2
        elif i == 'B' or i == 'C' or i == 'M' or i == 'P':
            sum += 3
        elif i == 'F' or i == 'H' or i == 'V' or i == 'W' or i == 'Y':
            sum += 4
        elif i == 'K':
            sum += 5
        elif i == 'J' or i == 'X':
            sum += 8
        elif i == 'Q' or i == 'Z':
            sum += 10
        else:
            sum += 0
    return sum
```

Fig. 7: A code example given by GitHub Copilot for the SCRABBLE SCORE problem that passes all test cases. We added a line break for better display.

```python
def tempfunction(in0):
    i0 = int(); i1 = int(); i2 = int()
    b0 = bool(); b1 = bool(); b2 = bool()
    s0 = str(); s1 = str(); s2 = str()
    li0 = []; li1 = []; li2 = []
    res0 = int()
    i1 -= min(mod(i1,(res0 - divInt(getIndexIntList(li0, max(i2, i2)),abs((i2
        * i0 ))) )), max((abs(abs(max(i1, i0))) - (divInt(getIndexIntList(li0,
            res0),max(res0, i1)) - (divInt(i2,res0) * min(i0, res0) ) ) ),
                len(list(map(lambda x: mod(x,max(i0, i0)), list(map(lambda x:
                    divInt(x,res0), li1)))))))
    for i1 in list(map(lambda x: saveOrd(x), (getCharFromString(saveChr(saveOrd(s1)),
        i2).strip(s0.strip(in0).strip().upper()).rstrip() + (s2 + in0).lower().rstrip()
            .strip().lstrip()).strip(getCharFromString(s1, sum(list(map(lambda x:
                (x + i1), list(map(lambda x: len(x), s2))))[:divInt((res0 * res0),
                    sum(scrabblescore))])))).rstrip(getCharFromString(saveChr(res0)
                        .rstrip().capitalize(), max(max(getIndexIntList(scrabblescore,
                            i2), sum(li2)), getIndexIntList(li0, getIndexIntList(li2,
                                i1))))).strip().lower().rstrip()))):
        if abs(max(max(sum(li1), (i1 * i2)), min(abs(i1), getIndexIntList(li0,
            i0)))) not in scrabblescore[(res0 * int(8.0)):]:
            li1.insert(max(i0, res0),+i0)
        res0 += max(len(getCharFromString(s0, i0).strip(s0)), max(getIndexIntList(
            scrabblescore, i1), min(len(scrabblescore), len(s1))))
    return res0
```

Fig. 8: A code example given by a grammar-guided GP approach for the SCRABBLE SCORE problem that passes all test cases. Example taken from the code repository of [12]. We added several line breaks for better display.

variable and function names) achieved by GitHub Copilot and the studied grammar-guided GP approach (denoted as G3P in the table) for the studied benchmark problems from PSB1 and PSB2. As before, the results of the grammar-guided GP approach are based on the solutions taken from the associated code repository of a recent paper [12].

We see that the grammar-guided GP approach generated many more unique solutions compared to GitHub Copilot for all benchmark problems. For example for the GRADE, FIZZ BUZZ, and FUEL COST problems, every GP-generated solution is unique on the AST-level, while GitHub Copilot generated only 15, 13, and again 15 unique solutions on the AST-level, respectively.

In summary, we see that GitHub Copilot generates smaller and less complex solutions while the grammar-guided GP approach produces more diverse and unique solutions.

TABLE III: Success rates as well as the number of unique solutions based on the source code and on the AST for GitHub Copilot and the studied grammar-guided GP approach (denoted as G3P) for all considered benchmark problems from PSB1 and PSB2. The results of the grammar-guided GP approach are taken from a recent paper [12] and its associated repository.

Benchmark Problem	GitHub Copilot			G3P (with standard lexicase selection) [12]		
	Successes	Unique (Source)	Unique (AST)	Successes	Unique (Source)	Unique (AST)
COUNT ODDS	98	36	16	65	95	95
GRADE	84	24	15	36	100	100
SCRABBLE SCORE	35	83	46	6	91	90
SMALL OR LARGE	51	26	16	41	98	98
FIND PAIR	41	73	47	0	99	97
FIZZ BUZZ	89	31	13	62	100	100
FUEL COST	97	58	15	33	100	100
GCD	80	89	53	0	87	87

V. DISCUSSION AND FUTURE DIRECTIONS

During our work with GitHub Copilot and GP, we had many insights that could be relevant for future program synthesis research. Some of the examples we will discuss below are already mentioned in the previous conference paper [4], but are also mentioned in this paper to give a comprehensive view. However, many discussed examples have been added for this extended version.

One of the most obvious differences between GitHub Copilot and GP is the definition of the user's intent. GitHub Copilot is based on a textual problem description for the generation of programs, while GP, on the other hand, needs input/output examples as problem description. In practical software development, however, a textual description of the problem is usually more useful, since, e.g., for program synthesis with GP, a large number of input/output examples is necessary (up to 200 cases are used for training in the literature [5]). The manual generation of these input/output examples can be very time-consuming depending on the problem's complexity. However, if large data sets are available, e.g., from scientific experiments from nature, then GP is well suited to generate explainable solutions. But if we look at the code example in Fig. 8, then the efforts that exist in the area of symbolic regression to achieve interpretability [54], [55] must be extended to program synthesis as well. Future approaches could use, e.g., post-simplification as it is already implemented in PushGP [56] or domain knowledge incorporating human written source code during the search process (e.g., in the fitness function or during selection) [7]. However, the latter is challenging, as the results of Schweim et al. [34] show. Although it is possible to generate smaller and less complex source code by incorporating software metrics during search, this often comes at the price of the correctness of the solutions found. Thus, if GP should be applied in practical software development, future approaches should improve the interpretability of the generated source code and reduce the number of the input/output examples that are required for training (e.g., as in [37]). Furthermore, tools and frameworks are needed that can be used not only by researchers but are also accessible for practitioners (like for GitHub Copilot).

A huge drawback of large language models is that we do not always exactly know which data was used to train them. Thus, it is possible that solutions for the benchmark problems used for evaluation have already been seen by the model during training. One way to circumvent this issue is to use novel benchmark problems that were published online after the training of the analyzed large language model. For example, Tian et al. [23] studied a large language model's program synthesis performance on two benchmark sets collected from LeetCode[10]: One contained problems published on LeetCode before the large language model was trained (2016-2020) and the other contained problems uploaded to the platform after the large language model was trained (2022). The results show a notably lower correctness on the new, probably previously unseen, problems. Therefore, selection of new benchmark problems is a possible approach, but with each new generation of large language models, new benchmark problems must be collected. In addition, there are always fewer benchmark problems available from the time after the training of a new generation of large language models than from the time before. Another approach could be the automatic generation of new benchmark problems. However, this may come at the price of comparability with human-written code, as randomly generated source code may have different properties. Thus, there is still a need for research on novel approaches to evaluate large language models for general purposes as well as for program synthesis.

Even though it is a challenge for evaluation that large language models are trained on large amounts of publicly available resources, it is important for their performance that they have been trained on a sufficient amount of high-quality data. However, for many domains, only a limited amount of training data is available. For example, the results of Robinson

[10]https://leetcode.com/.

et al. [57] in the field of machine translation show that the studied GPT model performs significantly worse for low-resource languages. When it comes to the use of large language models in software development, the question arises how well new programming languages or even domain-specific languages can be supported? GP-based approaches, however, can be easily adapted to new requirements. Domain specific languages, e.g., used for synthesis of quantum computer programs [58], could be supported via a grammar or a function/terminal set.

```
# Given a vector of floats, calculate
# the average of those floats, round the
# result to 7 decimal places, multiply
# the result with 5 and return True if
# the result is greater than 10, otherwise
# return False.
def myfunc(list1: list):
    avg = sum(list1) / len(list1)
    result = round(avg, 7) * 5
    if result > 10:
        return True
    else:
        return False
```

Fig. 9: A code example given by GitHub Copilot for the VECTOR AVERAGE problem with a changed and extended textual problem description.

Nevertheless, during our work with GitHub Copilot, we had the experience that it works well as a programming support system for Python and that it can also generate programs for non-standard problems, as long as the given problem description explicitly defines the required steps. For demonstration, Fig. 9 shows GitHub Copilot's code suggestion for an extended version of the VECTOR AVERAGE problem. Although we cannot be sure that the large language model behind GitHub Copilot has not already seen this or similar programs, it is at least no longer one of the classic benchmark problems whose solutions can often be found on GitHub and other platforms. We changed not only the rounding precision but also notably expanded the problem description and changed the return type. Despite all the changes in the description, we get a correct code suggestion from GitHub Copilot. However, there are cases where GitHub Copilot could not find a solution, even without changing the textual problem description. An example of this is the DIGITS problem, which could only be solved by PushGP but not by GitHub Copilot (see Table I).

Large language models and GP could complement each other very well in the area of program synthesis. As for some of the solutions generated by GitHub Copilot during our experiments, we noticed that the source code contains meaningful comments. Figure 10 shows an example for comments generated by GitHub Copilot for the CHECKSUM problem. Large language models could be used to explain GP-generated code or to insert meaningful comments and variable names. Recent work already shows that large language models can be utilized to explain source code [59].

As previously mentioned, we have noticed that the success rates achieved by GitHub Copilot on the problems from PSB2 (see Table II) are lower than those for PSB1 (Table I).

```
# Given a string, convert each character
# in the string into its integer ASCII
# value, sum them, take the sum modulo
# 64, add the integer value of the space
# character, and then convert that
# integer back into its corresponding
# character (the checksum character).
def myfunc(str1: str):
    # Convert each character in the string
    # into its integer ASCII value
    # sum them
    sum = 0
    for i in str1:
        sum += ord(i)
    # take the sum modulo 64
    sum %= 64
    # add the integer value of the space
    # character
    sum += ord(' ')
    # convert that integer back into its
    # corresponding character
    # (the checksum character).
    return chr(sum)
```

Fig. 10: A code example given by GitHub Copilot for the CHECKSUM problem with meaningful comments. We added several line breaks for better display.

We believe that this is often related to imprecise textual problem descriptions. In order to test whether a more detailed

```
# Given a string where "T" means True,
# "F" means False, "|" is the OR operator,
# and "&" is the AND operator. Return
# the evaluation of the given string
# (str1) as Boolean value. The given
# string contains no spaces.
def myfunc(str1: str):
    return eval(
        str1.replace("T", "True")
            .replace("F", "False")
            .replace("|", " or ")
            .replace("&", " and ")
    )
```

Fig. 11: A code example given by GitHub Copilot for the SOLVE BOOLEAN problem. We added several line breaks for better display.

problem description is helpful for finding a correct solution, we expanded the description of the SOLVE BOOLEAN problem (e.g., added a description for the symbols used in the input string) where GitHub Copilot is not able to find a correct solution with the original description (see Table II). Figure 11 shows a correct code suggestion given by GitHub Copilot as well as the extended problem description (lines 1–6). Thus, the description of the problem seems to have a large influence on the solution quality. Therefore, it is not surprising that there is recent work on prompt engineering [60], [61] that attempts to further improve the quality of the output of large language models by systematically changing the input prompts.

Before using code suggestions generated by GitHub Copilot

or other large language models in the source code of productive software, further properties of the code should be checked in addition to correctness. During our experiments, we noticed that the evaluation of GitHub Copilot's code suggestions for the GCD problem on the test data takes significantly more time than for many other benchmark problems. Therefore, we measured the run-time more precisely. Figure 12 shows a histogram of the run-time in seconds for GitHub Copilot's code suggestions for the GCD problem passing all the test cases. We see that most of the generated programs take more

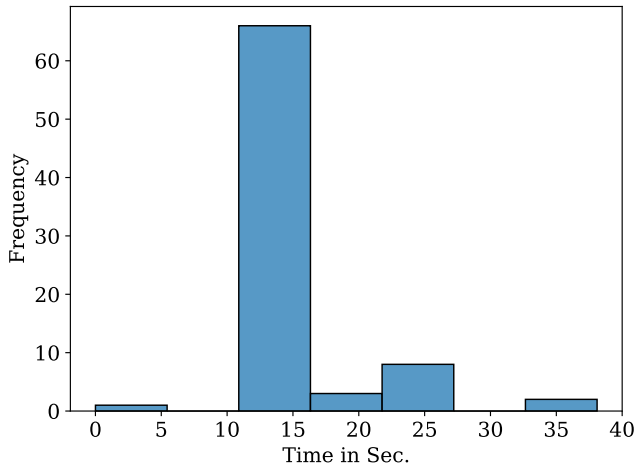

Fig. 12: A histogram of the run-time in seconds for all suggestions given by GitHub Copilot for the GCD problem solving all test cases correctly.

than 10 seconds to process all the test inputs. Some programs even take more then 30 seconds and only one solution takes less than 5 seconds. Thus, almost all suggested programs have poor run-time and should not be used in practice. However, it can get even worse, as code generated by large language models may have security issues [1], [62]. An example of this can be seen in Fig. 11. The `eval()` function is used in the suggested source code without checking the given input, which means that malicious code could be executed. This could be easily prevented in GP-based program synthesis by simply not including the `eval()` function in the used grammar. However, large language models rely on the data they are trained on and human-generated code is often error prone and not well optimized. Therefore, in future research, methods like genetic improvement (GI) [45] could be used for improving, e.g., the run-time or other non-functional properties of code generated by large language models.

Lastly, the training of large language models is computationally very intensive and consumes a lot of energy [63] while GP-based approaches can already be executed on more easily available hardware. However, the inference time of an already trained large language model is usually much faster than its training time. With GP, however, the evolution has to be repeated whenever a new program is required, which can take several hours or even days for a single program with current frameworks [6]. For symbolic regression, there are frameworks that can be accelerated with modern GPUs [64]. To improve the training times, frameworks like this are also necessary for GP-based program synthesis. Another interesting research direction for the acceleration of GP could be the incorporation of pre-trained models in the search process as studied by Reiter et al. [65].

VI. Limitations

We used GitHub Copilot via the Visual Studio Code extension to measure the experience of a software developer. However, GitHub Copilot and the large language model behind it works like a black box for researchers and the performance may change in the future due to further adjustments by the developers. Nevertheless, an analysis of its quality is important, as it can be assumed that GitHub Copilot and similar tools will influence many software developers.

Additionally, an internal threat to validity comes from not knowing whether the large language model underlying GitHub Copilot has not already been trained on problems from the PSB1 and PSB2 datasets. This threat concerns all work that compare with large language models [23]. However, these are the benchmarks against which program synthesis approaches have been compared thus far and are regarded as standard. Given we do not know what large language models have been trained on, selection of a new benchmark set would be a challenging task that goes beyond the scope of this paper.

VII. Conclusion

In this work, we carried out an in-depth comparison of GitHub Copilot and GP extending our previous conference paper [4]. Through our work with GitHub Copilot and by comparing the programs generated by Copilot and GP in terms of performance and structure, we were able to identify different strengths and weaknesses of each method as well as potential future research directions for the GP community in the area of program synthesis.

In our analysis of the performance on common program synthesis benchmark problems, we find that GitHub Copilot and GP can solve about the same number of benchmark problems. Specifically, on PSB1, GitHub Copilot is able to solve 89.7% of the benchmark problems, while PushGP solves 86.2% of the problems. With 58.6%, grammar-guided GP solves a notably lower number of benchmark problems from PSB1. On PSB2, GitHub Copilot solves 80% of the benchmark problems and PushGP solves 68% of the problems. However, when studying the success rates, we see that GitHub Copilot significantly outperforms the GP approaches on over 50% of the problems from PSB1 and PSB2. Although, when comparing the results, we must keep in mind that the large language model underlying GitHub Copilot may have already seen many of the benchmark problems tested during training. In addition, training large language models is very computationally intensive, whereas GP can be run on more easily available hardware.

Further, we find in our analysis of the structure and the diversity of the generated code, that for over 85% of the benchmark problems, the tested grammar-guided GP generates programs that are significantly larger with up to more than

three times more SLOC compared to the programs generated with GitHub Copilot. Additionally, we find that the grammar-guided GP approach generates more unique programs than GitHub Copilot (on average 95.9% vs. 27.6% unique solutions on the AST level, respectively) which indicates that GP is able to find novel solution strategies while GitHub Copilot suggests solutions that are shorter and less complex.

Overall, from a programmer's perspective, GitHub Copilot is currently better suited for daily work in software development. This is mainly because GitHub Copilot offers tools that can be integrated directly into the code editors used by the programmers, the generated code is usually easy to read and less bloated, and the given suggestions are often correct for common tasks. However, which method should ultimately be used – GitHub Copilot or GP – depends on the considered problem. For example, to generate a program that connects the observed data points of a scientific experiment, GP is certainly more suitable because the data points can simply be passed as input/output examples. GP can also be used when generating programs in domain-specific languages, as the used representation can be easily adapted via grammars or the function set. Furthermore, GP can find novel and innovative solutions through its evolutionary nature.

Nevertheless, for the GP community, several future research directions can be derived from our comparison of GitHub Copilot and GP. Therefore, we recommend researchers to focus more on increasing the readability and interpretability of the generated programs while preserving GP's unique ability to find diverse, novel, and innovative solutions. Furthermore, the community should focus more on providing fast and accessible tools and frameworks to make it easier for researchers as well as software developers to use GP-based approaches. In addition, large language models and GP could be combined. For example, large language models could be used to document solutions generated by GP, or GP could be used to automatically increase the performance of programs generated by large language models.

Acknowledgments & Copyright

This work was partially supported by UKRI EPSRC grant no. EP/P023991/1. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *proceedings of an International Conference on Genetic Algorithms and the Applications*, 1985, pp. 183–187.

[3] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[4] D. Sobania, M. Briesch, and F. Rothlauf, "Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2022, pp. 1019–1027.

[5] T. Helmuth and L. Spector, "General program synthesis benchmark suite," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 1039–1046.

[6] T. Helmuth and P. Kelly, "Psb2: the second program synthesis benchmark suite," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2021, pp. 785–794.

[7] D. Sobania and F. Rothlauf, "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*, 2019, pp. 1065–1074.

[8] ——, "Challenges of program synthesis with grammatical evolution," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2020, pp. 211–227.

[9] T. Helmuth and L. Spector, "Problem-solving benefits of down-sampled lexicase selection," *Artificial life*, vol. 27, no. 3–4, pp. 183–203, 2021.

[10] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Extending program synthesis grammars for grammar-guided genetic programming," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2018, pp. 197–208.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[12] R. Boldi, M. Briesch, D. Sobania, A. Lalejini, T. Helmuth, F. Rothlauf, C. Ofria, and L. Spector, "Informed down-sampled lexicase selection: Identifying productive training cases for efficient problem solving," *arXiv preprint arXiv:2301.01488*, 2023.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[14] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020, pp. 1536–1547.

[17] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: Multi-mode translation of natural language and python code with transformers," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 9052–9065.

[18] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[19] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2023, pp. 23–30.

[20] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.

[21] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[22] A. Borji, "A categorical archive of chatgpt failures," *arXiv preprint arXiv:2302.03494*, 2023.

[23] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant–how far is it?" *arXiv preprint arXiv:2304.11938*, 2023.

[24] T. Helmuth, N. F. McPhee, and L. Spector, "Program synthesis using uniform mutation by addition and deletion," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 1127–1134.

[25] T. Helmuth and L. Spector, "Explaining and exploiting the advantages of down-sampled lexicase selection," in *Artificial Life Conference Proceedings 32*. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info ..., 2020, pp. 341–349.

[26] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "A grammar design pattern for arbitrary program synthesis problems in genetic

programming," in *European Conference on Genetic Programming*. Springer, 2017, pp. 262–277.
[27] Y. Yuan and W. Banzhaf, "Iterative genetic improvement: Scaling stochastic program synthesis," *Artificial Intelligence*, vol. 322, p. 103962, 2023.
[28] 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*, 2019, pp. 2028–2031.
[29] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the Push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, 2002.
[30] L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control," in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, 2005, pp. 1689–1696.
[31] T. Helmuth and A. Abdelhady, "Benchmarking parent selection for program synthesis by genetic programming," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 237–238.
[32] P. A. Whigham *et al.*, "Grammatically-based genetic programming," in *Proceedings of the workshop on genetic programming: from theory to real-world applications*, vol. 16, no. 3, 1995, pp. 33–41.
[33] E. Hemberg, J. Kelly, and U.-M. O'Reilly, "On domain knowledge and novelty to improve program synthesis performance with grammatical evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 1039–1046.
[34] D. Schweim, E. Hemberg, D. Sobania, and U.-M. O'Reilly, "Exploiting knowledge from code to guide program search," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2022, pp. 262–277.
[35] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, "Towards understanding and refining the general program synthesis benchmark suite with genetic programming," in *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2018, pp. 1–6.
[36] D. Sobania and F. Rothlauf, "A generalizability measure for program synthesis with genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2021, pp. 822–829.
[37] D. Sobania, M. Briesch, P. Röchner, and F. Rothlauf, "MTGP: Combining metamorphic testing and genetic programming," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2023, pp. 324–338.
[38] F. Garrow, M. A. Lones, and R. Stewart, "Why functional program synthesis matters (in the realm of genetic programming)," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2022, pp. 1844–1853.
[39] D. Schweim, D. Sobania, and F. Rothlauf, "Effects of the training set size: A comparison of standard and down-sampled lexicase selection in program synthesis," in *2022 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2022, pp. 1–8.
[40] D. Sobania and F. Rothlauf, "Program synthesis with genetic programming: the influence of batch sizes," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2022, pp. 118–129.
[41] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill, "Grammar-based genetic programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, pp. 365–396, 2010.
[42] K. Krawiec, *Behavioral program synthesis with genetic programming*. Springer, 2016, vol. 618.
[43] T. P. Pawlak and K. Krawiec, "Competent geometric semantic genetic programming for symbolic regression and boolean function synthesis," *Evolutionary computation*, vol. 26, no. 2, pp. 177–212, 2018.
[44] P. Liskowski, K. Krawiec, N. E. Toklu, and J. Swan, "Program synthesis as latent continuous optimization: Evolutionary search in neural embeddings," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 2020, pp. 359–367.
[45] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, 2017.
[46] Z. Bian, A. Blot, and J. Petke, "Refining fitness functions for search-based program repair," in *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2021, pp. 1–8.
[47] A. Blot and J. Petke, "Comparing genetic programming approaches for non-functional genetic improvement: Case study: Improvement of minisat's running time," in *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 2020, pp. 68–83.
[48] L. Spector, "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*, 2012, pp. 401–408.
[49] T. Helmuth and P. Kelly, "Applying genetic programming to psb2: the next generation program synthesis benchmark suite," *Genetic Programming and Evolvable Machines*, vol. 23, no. 3, pp. 375–404, 2022.
[50] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, and M. O'Neill, "Ponyge2: Grammatical evolution in python," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1194–1201.
[51] S. Rizwan, M. S. Ali Sobuj, and M. R. Akhond, "A survey on software test case minimization," in *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing*, 2022, pp. 679–684.
[52] D. Sobania, D. Schweim, and F. Rothlauf, "A comprehensive survey on program synthesis with evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 1, pp. 82–97, 2022.
[53] M. C. Fernandes, F. O. De França, and E. Francesquini, "Hotgp - higher-order typed genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1091–1099. [Online]. Available: https://doi.org/10.1145/3583131.3590464
[54] M. Virgolin, A. De Lorenzo, E. Medvet, and F. Randone, "Learning a formula of interpretability to learn interpretable formulas," in *Parallel Problem Solving from Nature–PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II 16*. Springer, 2020, pp. 79–93.
[55] W. G. La Cava, P. C. Lee, I. Ajmal, X. Ding, P. Solanki, J. B. Cohen, J. H. Moore, and D. S. Herman, "A flexible symbolic regression method for constructing interpretable clinical prediction models," *NPJ Digital Medicine*, vol. 6, no. 1, p. 107, 2023.
[56] T. Helmuth, N. F. McPhee, E. Pantridge, and L. Spector, "Improving generalization of evolved programs through automatic simplification," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 937–944.
[57] N. Robinson, P. Ogayo, D. R. Mortensen, and G. Neubig, "ChatGPT MT: Competitive for high- (but not low-) resource languages," in *Proceedings of the Eighth Conference on Machine Translation*, P. Koehn, B. Haddow, T. Kocmi, and C. Monz, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 392–418. [Online]. Available: https://aclanthology.org/2023.wmt-1.40
[58] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy, "Quantum computing applications of genetic programming," *Advances in genetic programming*, vol. 3, pp. 135–160, 1999.
[59] D. Sobania, A. Geiger, J. Callan, A. Brownlee, C. Hanna, R. Moussa, M. Z. López, J. Petke, and F. Sarro, "Evaluating explanations for software patches generated by large language models," in *International Symposium on Search Based Software Engineering*. Springer, 2023, pp. 147–152.
[60] K. Zhou, J. Yang, C. C. Loy, and Z. Liu, "Learning to prompt for vision-language models," *International Journal of Computer Vision*, vol. 130, no. 9, pp. 2337–2348, 2022.
[61] T. Martins, J. M. Cunha, J. Correia, and P. Machado, "Towards the evolution of prompts with metaprompter," in *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*. Springer, 2023, pp. 180–195.
[62] M. O. F. Rokon, R. Islam, A. Darki, E. E. Papalexakis, and M. Faloutsos, "Sourcefinder: Finding malware source-code from publicly available repositories in github," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 149–163. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/omar
[63] A. S. Luccioni, S. Viguier, and A.-L. Ligozat, "Estimating the carbon footprint of bloom, a 176b parameter language model," *arXiv preprint arXiv:2211.02001*, 2022.
[64] F. Baeta, J. Correia, T. Martins, and P. Machado, "Tensorgp-genetic programming engine in tensorflow." in *EvoApplications*. Springer, 2021, pp. 763–778.
[65] J. Reiter, D. Schweim, and D. Wittenberg, "Pretraining reduces runtime in denoising autoencoder genetic programming by an order of magnitude," in *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, 2023, pp. 2382–2385.