Choose Your Programming Copilot

A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming

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

Martin Briesch*
Johannes Gutenberg University
Mainz, Germany
briesch@uni-mainz.de

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

ABSTRACT

GitHub Copilot, an extension for the Visual Studio Code development environment powered by the large-scale language model Codex, makes automatic program synthesis available for software developers. This model has been extensively studied in the field of deep learning, however, a comparison to genetic programming, which is also known for its performance in automatic program synthesis, has not yet been carried out. In this paper, we evaluate GitHub Copilot on standard program synthesis benchmark problems and compare the achieved results with those from the genetic programming literature. In addition, we discuss the performance of both approaches. We find that the performance of the two approaches on the benchmark problems is quite similar, however, in comparison to GitHub Copilot, the program synthesis approaches based on genetic programming are not yet mature enough to support programmers in practical software development. Genetic programming usually needs a huge amount of expensive hand-labeled training cases and takes too much time to generate solutions. Furthermore, source code generated by genetic programming approaches is often bloated and difficult to understand. For future work on program synthesis with genetic programming, we suggest researchers to focus on improving the execution time, readability, and usability.

CCS CONCEPTS

• **Software and its engineering** → *Search-based software engineering*; **Genetic programming**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

Program Synthesis, Genetic Programming, Large-Scale Language Models, Codex, GitHub Copilot, Software Engineering

1 INTRODUCTION

In software development, programmers today usually have support from tools such as automatic code completion or comprehensive online resources, which greatly accelerate their daily work. Automatic program synthesis, in which source code is generated based on a given definition, e.g., in the form of a natural language description or input/output examples [14], has the potential to become another standard tool in software development.

A tool that has recently drawn some attention is GitHub Copilot[1], an extension for the Visual Studio Code development environment that offers suggestions for extending a programmer's source code based on problem descriptions and existing code. Based on the

large-scale language model Codex [6], which was trained on a large amount of source code, GitHub Copilot is more than a standard code completion tool (which often only suggests variable or function names) as it recommends the source code of complete functions and even suggests useful test cases for existing functions.

Genetic programming (GP) [9, 24] is another approach that has made great progress in the field of automatic program synthesis in recent years. Starting with a random population of programs, GP applies an evolutionary process to gradually improve the programs and to finally come up with solutions that meet the requirements. To define the functionality of the desired programs and to evaluate the generated programs during evolution, usually input/output examples are used.

To make different program synthesis approaches comparable, Helmuth et al. [16, 19] recently curated two benchmark collections containing a wide range of program synthesis benchmark problems with different complexity. In addition to a description of the problems, the benchmark suites define also how the training and test data should be defined. However, the problems of the benchmark suites have so far mainly been used to test and compare different GP-based program synthesis approaches [33]. A comparison with program synthesis approaches based on large-scale language models has not yet been carried out.

Therefore, in this work, we evaluate GitHub Copilot on the common program synthesis benchmark problems suggested by Helmuth et al. [16, 19] and compare the obtained results with those reported in the GP literature. In addition, we discuss the performance of the GP-based approaches and GitHub Copilot and identify future research directions.

To evaluate GitHub Copilot on the benchmark problems, we use the Copilot Extension[2] for Visual Studio Code and provide for every problem the function's signature and the textual problem description as comment. After that, we evaluate Copilot's suggestion. If the suggested program is not fully correct, we let Copilot generate further alternatives (a maximum of ten) and check these alternative programs for correctness. For GP, we distinguish between the two benchmark sets. For the older program synthesis benchmark suite PSB1, which was published in 2015 [19], we took the GP performance results from a recent survey on program synthesis [33]; for the new benchark suite PSB2, which was published in 2021 [16], we report GP results from all publications dealing with PSB2.

Following this introduction, we present in Sect. 2 recent work on GP-based program synthesis and briefly introduce large-scale language models as used by GitHub Copilot. Section 3 presents the results of our comparison. In Sect. 4, we discuss our findings and identify future research directions. Section 5 concludes the paper.

---

*Both authors contributed equally.
[1]https://copilot.github.com/

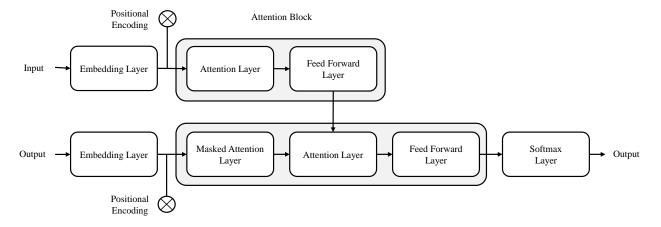[2]https://marketplace.visualstudio.com/items?itemName=GitHub.copilot

Figure 1: Encoder-Decoder architecture for a typical transformer model. After an embedding layer the input gets fed into multiple attention blocks of the encoder. The output of the encoder, as well as the current output of the whole model, is then processed by multiple attention blocks in the decoder to produce the final output (based on Vaswani et al. [36]).

2 RELATED WORK

In this section, we present recent work on GP-based program synthesis and briefly introduce relevant approaches. Furthermore, we describe the foundations of large-scale language models and present work where such models are used for program synthesis.

2.1 GP-Based Program Synthesis

The publication of PSB1 [19] in 2015 revived and consolidated the field of general program synthesis in GP. Different papers became comparable through a common base of benchmark problems, since in addition to the problems, the training and test data (input/output examples) are also defined in the benchmark suite. The benchmark suite PSB2 [16] from 2021 extends the collection of common problems. For solving these benchmark problems, three major categories of methods emerged from the GP field: stack-based GP, grammar-guided GP, and linear GP.

In order to be able to represent and process programs, a GP approach has to deal with different data types. Stack-based GP approaches provide stacks to differentiate between the supported data types [35]. The stack-based approach mainly used in the literature is PushGP [34, 35], which is based on the stack-based programming language Push. With the introduction of the Uniform Mutation by Addition and Deletion (UMAD) operator [18] and the usage of different selection methods [15], the success rates of PushGP in program synthesis could be increased significantly.

A disadvantage of approaches like PushGP is that stack-based programming languages are not used in real-world software development. Thus, the generated source code cannot be used directly in existing software projects. With grammar-guided GP approaches it is possible to represent common programming languages like Python [12, 13]. Using a context-free grammar allows, e.g., the usage of different data types, loops, and conditionals. To improve grammar-guided GP approaches, Hemberg et al. [21] used not only the GP-typical input/output examples, but also the textual problem descriptions of the benchmark problems. Other grammar-guided

GP work used source code mined from GitHub to improve the code quality of the programs generated by GP [30].

In linear GP, the provided functions usually operate on data registers similar to those known from common Assembler languages [3]. To reference these registers, Lalejini and Ofria [25] suggested tag-based memory to make the program representation more stable against changes by variation operators.

For a broader introduction to the relevant GP approaches for program synthesis and an overview of their success on the common benchmarks, we refer the reader to a recent literature survey [33].

2.2 Large-Scale Language Models

As an alternative to providing input/output examples to specify the program, it is also possible to formulate the desired behaviour with a natural language description [14]. Natural language processing has seen a variety of improvement in recent years. Much of those improvements can be attributed to the availability of larger data sets, more compute and novel deep learning architectures [4, 10, 27, 36].

One of the highly influential deep learning architectures that can be used for natural language processing is the Transformer model [36]. In contrast to prior architectures like Long Short-Term Memory networks [22] and Gated Recurrent Unit networks [7], Transformer models do not rely on recurrence and the sequential modelling of recurrent architectures. Instead they solely use attention mechanisms [2] to model dependencies within a sequence. This enables models to be trained more efficiently and on longer sequences while still achieving state-of-the-art results [36].

Attention mechanisms allow a model to identify the relevant context by comparing a query of an element at position i in a sequence with every other element of that sequence, also called self-attention. This is done using an attention function which parameters are learned during training. The most common attention functions are additive attention [2] and dot-product attention [26].

Figure 1 shows the general building blocks of a Transformer model using an Encoder-Decoder architecture. The inputs get encoded with an embedding layer and a positional encoding and are

then fed into the attention blocks of the encoder. The encoder consists of multiple stacked attention blocks, each with an attention layer followed by a feed forward layer. The decoder takes the output of the encoder as well as the preceding output of the model as input and also consists of multiple attention blocks. To prevent the model from looking at subsequent input tokens during training, the first attention layer uses masking. Following the last attention block of the decoder the model returns the probabilities for the next token using a softmax layer [36].

The attention layers utilize self-attention and a scaled dot-product attention function which is defined as

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V,$$

where Q is the matrix of query vectors q_i, K is the matrix of key vectors k_i, and V is a matrix of value vectors v_i. The vectors q_i, k_i, and v_i are learned linear projections of the input position i of the previous layer. $\sqrt{d_k}$ is a scaling factor for easier training. Transformers typically use multiple attention functions per layer with different parameters, allowing the network to focus on different aspects of context [36].

Since the training of these Transformers can be well parallelized on modern hardware, it is possible to train them efficiently with large data sets. While large labeled data sets are typically expensive to obtain for specific tasks, it is also possible to pre-train those models in an unsupervised fashion and fine-tune them for a specific task with a much smaller data set [23, 27]. The use of pre-trained models that have a high capacity and are trained with huge data sets, led to a large performance increase across many natural language processing tasks [4].

Given the large amount of freely available open source code, there is much data available to (pre-)train those large-scale language models for program synthesis. Trained models such as CodeBERT [11], PyMT5 [8] and Codex [6] take a description of the desired program behaviour as natural language specification and sample a corresponding Python program. Codex is a model based on the GPT-3 [4] architecture with up to 12 billion parameters. This model is first pre-trained with 159 GB of code samples from sources like GitHub in an unsupervised manner. Afterwards, the authors fine-tuned the model with a smaller supervised data set containing functions from competitive programming websites and curated code repositories. The final model achieves promising results [6]. GitHub Copilot is a special production version of Codex, which we use for our analysis in the following sections.

3 PERFORMANCE COMPARISON

In order to be able to compare the quality of GitHub Copilot with the program synthesis performance of GP, we apply GitHub Copilot to the common program synthesis benchmark problems and compare the achieved results to those from the GP-based program synthesis literature. In this section, we describe our test procedure for GitHub Copilot and present our comparison including all benchmark problems from PSB1 and PSB2.

3.1 Methodology

GitHub Copilot is accessible via an extension for the Visual Studio Code development environment. The extension can do both, make suggestions for extending existing code (e.g., to complete a single line of code) and suggest complete functions based on a given description. We use the problem descriptions from the PSB1 [19] and PSB2 [16] papers and let GitHub Copilot suggest a function to solve these problems. Therefore, we insert a benchmark problem's textual description as Python comment and define the function's signature adjusted to the considered problem (defining input parameters). Then, we let GitHub Copilot make a suggestion for the function's body (based on the problem's textual description).

```
1    # Given a string, convert each character in
2    # the string into its integer ASCII value,
3    # sum them, take the sum modulo 64, add the
4    # integer value of the space character, and
5    # then convert that integer back into its
6    # corresponding character (the checksum
7    # character). The program must print
8    # "Check sum is X", where X is replaced
9    # by the correct checksum character.
10   def myfunc(str1: str):
11       # To be completed by GitHub Copilot
```

Figure 2: An example definition as input for GitHub Copilot using the textual description for the Checksum problem. The problem description was taken from the PSB1 paper [19].

Figure 2 shows an example definition for the Checksum problem. The code lines 1-9 contain the problem description as Python comment and line 10 defines the function's signature. Beginning from line 11, we expect from GitHub Copilot to make a suggestion.

We assume that GitHub Copilot makes a correct suggestion (is successful) if either the first suggestion or one of the given alternatives correctly solves the problem at hand. In our experiments, we allowed a maximum of ten alternatives. For the GP-based approaches, we do not carry out any new experiments, but instead use the results from the literature.

3.2 Results

First, we study the results for the benchmark problems suggested in PSB1. We compare the results obtained by GitHub Copilot in our experiments with the results obtained by stack-based GP, grammar-guided GP, as well as linear GP approaches from the literature. The results for the GP approaches are taken from [33]. This recent literature review found 54 papers which study program synthesis using problems from PSB1. Out of these, 30 papers reported success on some benchmark problems for stack-based GP, 11 for grammar-guided GP, and 4 for linear GP. A test problem is solved by a GP approach, if there is at least one paper that finds a successful solution for the problem at hand in at least one run (independently of the overall number of runs).

Table 1 shows the results obtained by GitHub Copilot compared to the results reported in the literature for stack-based GP, grammar-guided GP, and linear GP on the benchmark problems from PSB1.

Table 1: Performance of GitHub Copilot compared to the results reported in the literature for stack-based GP, grammar-guided GP, and linear GP on the benchmark problems from PSB1. A checkmark (✓) indicates that the first returned solution is correct (for GitHub Copilot) or there is at least one run reported in the literature returning a successful solution (for GP). A checkmark with an asterisk (✓*) indicates that not the first solution suggested by GitHub Copilot is correct, but a correct solution can be found in the first ten suggestions. A cross (✗) indicates that none of the suggested programs were successful (Copilot) or the literature reports no found successful solution (GP approaches).

Benchmark Problem	GitHub Copilot	Stack-based GP	Grammar-Guided GP	Linear GP
Number IO	✓	✓	✓	✓
Small or Large	✓	✓	✓	✓
For Loop Index	✓	✓	✓	✓
Compare String Lengths	✓	✓	✓	✓
Double Letters	✓*	✓	✓	✗
Collatz Numbers	✓*	✗	✗	✗
Replace Space with Newline	✓*	✓	✓	✗
String Differences	✓*	✓	✗	✗
Even Squares	✓	✓	✓	✗
Wallis Pi	✗	✗	✗	✗
String Lengths Backwards	✓	✓	✓	✗
Last Index of Zero	✓*	✓	✓	✗
Vector Average	✓	✓	✓	✗
Count Odds	✓	✓	✓	✗
Mirror Image	✓	✓	✓	✗
Super Anagrams	✗	✓	✓	✗
Sum of Squares	✓	✓	✓	✗
Vectors Summed	✓	✓	✓	✗
X-Word Lines	✓	✓	✓	✗
Pig Latin	✗	✗	✓	✗
Negative To Zero	✓	✓	✓	✗
Scrabble Score	✓*	✓	✓	✗
Word Stats	✗	✗	✗	✗
Checksum	✓*	✓	✓	✗
Digits	✗	✓	✓	✗
Grade	✓*	✓	✓	✓
Median	✓*	✓	✓	✓
Smallest	✓*	✓	✓	✓
Syllables	✓	✓	✓	✗
Σ (Solved)	**24**	**25**	**25**	**7**

A checkmark (✓) indicates that a successful solution has been found with the first suggestion (for GitHub Copilot) or has been reported in the literature (for the GP approaches). A checkmark with an asterisk (✓*) indicates that a correct solution was not found with GitHub Copilot's first suggestion but in one of the suggested alternatives (a maximum of ten). By making this distinction, we take into account the standard use of GitHub Copilot (and other code completion tools) by a programmer, because a programmer would usually prefer using the first suggestion before studying the alternatives. A cross (✗) indicates that none of the suggested programs were successful (Copilot) or the literature reports no found successful solution (GP approaches).

```
1  def myfunc(str1: str):
2      sum = 0
3      for i in range(len(str1)):
4          sum += ord(str1[i])
5      sum = sum % 64
6      sum += ord(' ')
7      return chr(sum)
```

Figure 3: First suggestion of GitHub Copilot in our experiments for the Checksum problem. The result is not printed but returned.

```
1  def myfunc(str1: str):
2      sum = 0
3      for char in str1:
4          sum += ord(char)
5      checksum = chr(sum % 64 + ord(" "))
6      print("Check sum is {}".format(checksum))
```

Figure 4: One of the alternatives suggested by GitHub Copilot for the Checksum problem. The result is printed in the correct format.

Overall, the results for GitHub Copilot and the GP approaches are quite similar. GitHub Copilot finds solutions for 24 benchmark problems and the GP approaches found up to 25 (stack-based and grammar-guided GP). For linear GP, the literature reports successful solutions for only 7 benchmark problems. However, this does not necessarily mean that linear GP generally performs worse than the stack-based and grammar-guided approaches, as there are just a lower number of papers studying linear GP for these benchmark problems, which means that the coverage of the problems is not as high as for stack-based and grammar-guided GP. So it is not surprising that the number of successes is lower for the linear GP approaches.

From the 29 problems from PSB1, GitHub Copilot does not find successful solutions for 5 problems: Wallis Pi, Super Anagrams, Pig Latin, Word Stats, and Digits. However, benchmark problems like Wallis Pi and Word Stats seem to be very complex as also there is not one GP approach in the literature that can solve these problems. The reader should be aware that in standard GP papers usually 100 runs are performed and the problem is counted as solved if at least one of the hundred runs returns a successful solution (i.a., in [19], [12], and [31]). This gives GP approaches a higher chance to find successful solutions.

For some benchmark problems, GitHub Copilot does not find a fully correct solution with the first suggestion but one of the alternative solutions is correct (marked by ✓*). However, often the solution suggested first by Copilot is very close to a correct solution. For example, PSB1 requires for some benchmark problems that the result should be printed and not simply returned [19].

Figure 3 shows such an example suggestion for the Checksum problem where the result is returned and not printed as required. Nevertheless, for a programmer this is still a helpful solution as the `return` statement (line 7) can be easily replaced with a `print()`

Table 2: Performance of GitHub Copilot compared to GP for the benchmark problems from PSB2. A checkmark (✓) indicates that the first returned solution is correct (for GitHub Copilot) or there is at least one GP approach in the literature where at least one GP run returns a successful solution (for GP). A checkmark with an asterisk (✓*) indicates that not the first solution suggested by GitHub Copilot is correct, but a correct solution can be found in the first ten suggestions. A cross (✗) indicates that none of the ten suggested programs were successful (for Copilot) or the literature reports no found successful solution (for GP).

Benchmark problem	GitHub Copilot	GP
Basement	✓	✓
Bouncing Balls	✗	✓
Bowling	✗	✗
Camel Case	✗	✓
Coin Sums	✓	✓
Cut Vector	✗	✗
Dice Game	✗	✓
Find Pair	✓	✓
Fizz Buzz	✓	✓
Fuel Cost	✓	✓
GCD	✓	✓
Indices of Substring	✗	✓
Leaders	✗	✗
Luhn	✓*	✗
Mastermind	✗	✗
Middle Character	✓	✓
Paired Digits	✓	✓
Shopping List	✓	✗
Snow Day	✗	✓
Solve Boolean	✗	✓
Spin Words	✓	✗
Square Digits	✓*	✓
Substitute Cipher	✓	✓
Twitter	✓	✓
Vector Distance	✓*	✗
Σ (Solved)	**15**	**17**

function. Furthermore, in the suggested alternatives there is often a suggestion that completely fulfills the requirements. An example is Figure 4, which shows one of the suggestions of GitHub Copilot for the Checksum problem where the result is printed in the correct format.

In addition to the benchmark problems from PBS1, we study also the performance of GitHub Copilot on the problems from PSB2.

Again, we compare the obtained results to the results available in the GP literature. This means, we took the results from the two available publications, namely the PSB2 paper [16] as well as Helmuth and Spector [20]. Since there are so far only two papers that use the problems from PSB2, we do not differentiate this time between the GP method used.

Table 2 shows the results obtained by GitHub Copilot in our experiments compared to the results reported in the GP literature on the benchmark problems from PSB2. To mark the results, we use the same notation as in Table 1.

Again, the overall results are quite similar for GitHub Copilot and GP. GitHub Copilot finds a working solution for 15 and GP for 17 benchmark problems. The benchmark problems for which a successful solution is found are distributed similarly for GitHub Copilot and GP. Only for Luhn, Shopping List, Spin Words, and the Vector Distance problem, GitHub Copilot finds a working solution where GP fails. Conversely, GP finds solutions for Bouncing Balls, Camel Case, Dice Game, Indices of Substring, Snow Day, and the Solve Boolean problem where none of the analyzed suggestions of GitHub Copilot are fully correct. However, on those problems GP only finds a solution in less than 10% of the runs.

4 DISCUSSION

To better understand our findings, we exemplarily analyze and extend some of the solutions suggested by GitHub Copilot and discuss the implications for GP-based program synthesis.

4.1 Analyzing Suggested Solutions

Generalization is a general challenge for machine learning approaches. For a real-world usage, machine learning methods must also produce high quality results for previously unseen inputs; simply memorizing the training data is not sufficient. Low generalization is a relevant issue in large-scale language models [5], and also GP-based approaches suffer from poor generalization for some of the program synthesis benchmark problems [17, 29].

As the problems from the program synthesis benchmark suites are common exercises for programming beginners as well as coding challenges [16, 19], it can be expected that solutions to some of these problems, albeit often in slightly different versions, can be found in open source repositories and that GitHub Copilot has seen this data during the training process.

During our experiments we noticed for some benchmark problems that GitHub Copilot has some knowledge about some of the problems. E.g., for the Coin Sums problem, when just entering the first words of the problem description ("Given a number of cents"), GitHub Copilot suggests a continuation of the description as well as a function that has some similarities to the benchmark problem from PSB2. Figure 5 shows the completed problem description and the suggested function.[3] The suggested function is not a solution for the problem from PSB2, but it illustrates that GitHub Copilot not only has some knowledge about US-Dollar coins (see line 4, Fig. 5), but also a bias in the suggestions as the description does not mention that US-Dollar coins are wanted (e.g., Euro coins could be also possible). When giving GitHub Copilot the complete problem

---

[3]For all code examples in this section, we added additional line breaks for better readability.

```
1   # Given a number of cents, return the minimum
2   # number of coins needed to make change.
3   def func(cents):
4       coins = [25, 10, 5, 1]
5       num_coins = 0
6       for coin in coins:
7           num_coins += cents // coin
8           cents %= coin
9       return num_coins
```

Figure 5: An example code where we entered only the first five words of the description of the Coin Sums problem and GitHub Copilot already suggested the continuation of the problem description and the associated function.

```
1   # Given a number of cents, find the fewest
2   # needed number of US coins (pennies, nickels,
3   # dimes, and quarters) to make that amount,
4   # and return the number of each type of count
5   # as a separate output.
6   def myfunc(int1: int):
7       return (int1 // 25, int1 % 25 // 10,
8           int1 % 25 % 10 // 5,
9           int1 % 25 % 10 % 5)
```

Figure 6: Solution suggested by GitHub Copilot for the Coin Sums problem with the complete textual description from PSB2 [16].

description from the PSB2 paper (which is more specific and also defines the coins), GitHub Copilot suggests a correct solution which is shown in Fig. 6. However, this benchmark problem could also have been included in GitHub Copilot's training data in the exact same form. Thus, can only solutions be suggested for problems that are already known?

To examine if it is possible to create novel functionality with GitHub Copilot, we tested several inputs ranging from descriptions of arithmetic problems to specific graphical plots. In most cases, GitHub Copilot was able to make useful suggestions. Even if the suggestions were not completely correct, they often contain the relevant libraries and function calls a programmer would otherwise look up in online resources.

Exemplarily, Figure 7 shows GitHub Copilot's suggestion for the Checksum problem with a changed problem description (original problem description from PSB1 [19] given in Fig. 2). All of our changes are correctly implemented in Copilot's suggested function: the correct value is used for the modulo of the sum (line 16), the ASCII values are calculated for the correct letters (lines 17-18), and the intermediary result is multiplied with the correct value (line 19). So GitHub Copilot recognizes relationships between the problem description and the corresponding code lines, even for problems previously unknown. It is also noticeable, that the source code generated by GitHub Copilot could be significantly shorter (e.g., by using an existing sum() function and doing several steps in a single line), but it is still easily readable and understandable. This is

```
 1   # Given a string, convert each character in
 2   # the string into its integer ASCII value,
 3   # sum them, take the sum modulo 55, add the
 4   # integer value of the "A" character, then
 5   # add the integer value of the "Z" character,
 6   # multiply it by 5, and then convert that
 7   # integer back into its corresponding
 8   # character (the checksum character). The
 9   # program must print "Check sum is X", where
10   # X is replaced by the correct checksum
11   # character.
12   def myfunc(str1: str):
13       sum = 0
14       for i in str1:
15           sum += ord(i)
16       sum = sum % 55
17       sum += ord("A")
18       sum += ord("Z")
19       sum *= 5
20       sum = chr(sum)
21       print("Check sum is ", sum)
```

Figure 7: Solution suggested by GitHub Copilot for the Checksum problem with changed problem description from PSB1 [19] (line breaks adjusted).

```
 1   # Given a string where "T" means True, "F"
 2   # means false, "|" is the or operator, and "&"
 3   # is the and operator, return the evaluation
 4   # of the given string as Boolean value.
 5   def myfunc(str1: str):
 6       str1 = str1.replace("&", " and ")
 7       str1 = str1.replace("|", " or ")
 8       str1 = str1.replace("T", "True")
 9       str1 = str1.replace("F", "False")
10       return eval(str1)
```

Figure 8: Suggested solution for an alternative problem description with additional information for the Solve Boolean problem.

often different in GP-based program synthesis as GP may produce uncommon solution strategies and bloated code [30, 31].

For benchmark problems from PSB2, we noticed that often the textual problem descriptions are not specific enough to explicitly solve the problem (even for human programmers). This is not an issue for GP-based approaches as they are usually based on the given input/output examples but GitHub Copilot requires an adequate problem description. However, with an adjusted description, a solution can often be found with Copilot. Fig. 8 shows the suggested solution for the Solve Boolean problem if we extend the existing PSB2 problem description (we added an explanation for every operator and terminal). While GitHub Copilot was not able to find a correct solution in our experiments with the original description (see Table 2), a correct solution can be found using a more specific problem description.

Again, the generated source code is easy to read and understand. However, the code snippet reveals an issue of language models trained on publicly available source code which may contain insecure source code [6, 28]. The example function uses an eval() function (line 10, Fig. 8) without checking the given input string. Such a function in a productive system could pose potential security hazards. In GP-based program synthesis, this is not an issue as the choices for GP are limited/controlled by a grammar or a function and terminal set. So functions like eval() or exec() can be excluded easily.

4.2 Different Approaches, Different Strengths

In our comparison, GitHub Copilot and the GP-based approaches achieved similar results on the benchmark problems. However, the approaches strongly differ from a user's perspective in the specification of the user's intent. GitHub Copilot usually gets a textual problem description while a GP run uses input/output examples. In a real world use case, it is easier for programmers to define their intent with a textual description than with a large number of input/output examples, as it is expensive to manually generate a set consisting of 100 or more input/output examples [32]. For small problems it would be faster to program the function than to create the examples manually.

On the other hand, when using input/output examples, edge cases can be more easily addressed (like with unit tests in real-world software development). However, if there are many input/output examples available and it is difficult to define the problem with a short description text, then GP is currently the better alternative. This can also be seen in our results, since GP can also solve some benchmark problems which GitHub Copilot cannot solve as the problem description is not sufficient. However, we should keep in mind, that the shown results for GP are an aggregation of the results reported in a variety of different papers. In practice, a programmer would have to choose one of these approaches, which then may have a lower performance on the benchmark problems.

Additionally, the response time of a program synthesis approach is important for practical use. Even if the initial training of a large-scale language model is very time consuming and computationally expensive [6], GitHub Copilot can suggest solutions in just a few seconds. On the other hand, in GP-based program synthesis, the training process is less expensive, but whenever a new sample of input/output cases is given, the training process has to be repeated. As the current GP-based program synthesis frameworks still may take days to synthesize a solution [16], GP is not yet ready for a real-world usage. However, there is still room for improvement, as hardware-based acceleration (e.g., with GPUs) could significantly reduce the execution time of GP-based program synthesis as shown, e.g., for symbolic regression with GP [1].

5 CONCLUSIONS

The automatic code completion tool GitHub Copilot has recently drawn some attention for its program synthesis performance. However, a comparison with GP, which is also known for its success in automatic program synthesis, has not yet been carried out.

Consequently, we evaluated in this work GitHub Copilot on common program synthesis benchmark problems [16, 19] and compared the obtained results with those from the GP literature. Furthermore,

we discussed the performance of the GP-based approaches and GitHub Copilot.

We found that GitHub Copilot and GP perform similar on the studied benchmark problems. Overall, GP can solve more problems, but this comes at the price of practical usage, as GP usually needs many expensive hand-labeled training cases and takes too much time to generate a solution. Furthermore, the suggestions of GitHub Copilot are usually human readable while source code generated by GP is often bloated and difficult to understand. However, GitHub Copilot must be viewed as a black-box, as the user has no information about the exact training data used to generate the model. The generated code could potentially be malicious, biased, or insecure. On the other hand, with GP it is easily possible to control the training data as well as the choices GP can make during evolution (by designing appropriate grammars or function sets).

For future program synthesis research with GP, we suggest researchers to focus on improving the execution time, the readability of the generated code, and reducing the amount of required training data. Furthermore, GP must be integrated into the standard software development infrastructure (like GitHub Copilot) in order to be accessible for end-users.

ACKNOWLEDGMENTS

REFERENCES

[1] Francisco Baeta, João Correia, Tiago Martins, and Penousal Machado. 2021. TensorGP-Genetic Programming Engine in TensorFlow.. In *EvoApplications*. Springer, 763–778.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[3] Markus F Brameier and Wolfgang Banzhaf. 2007. *Linear genetic programming.* Springer Science & Business Media.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[5] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2633–2650. https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. 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)*. 1724–1734.

[8] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. 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)*. 9052–9065.

[9] Nichael Lynn Cramer. 1985. A representation for the adaptive generation of simple sequential programs. In *proceedings of an International Conference on Genetic Algorithms and the Applications*. 183–187.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*. 4171–4186.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. 1536–1547.

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

[13] 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.

[14] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 13–24.

[15] Thomas Helmuth and Amr Abdelhady. 2020. Benchmarking parent selection for program synthesis by genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. 237–238.

[16] Thomas Helmuth and Peter Kelly. 2021. PSB2: the second program synthesis benchmark suite. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 785–794.

[17] 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.

[18] 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.

[19] 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.

[20] Thomas Helmuth and Lee Spector. 2021. Problem-solving benefits of down-sampled lexicase selection. *arXiv preprint arXiv:2106.06085* (2021).

[21] 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.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 328–339.

[24] John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

[25] Alexander Lalejini and Charles Ofria. 2019. Tag-accessed memory for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 346–347.

[26] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 1412–1421.

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[28] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos. 2020. SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 149–163. https://www.usenix.org/conference/raid2020/presentation/omar

[29] Dominik Sobania. 2021. On the generalizability of programs synthesized by grammar-guided genetic programming. In *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 130–145.

[30] 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.

[31] Dominik Sobania and Franz Rothlauf. 2020. Challenges of program synthesis with grammatical evolution. In *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 211–227.

[32] Dominik Sobania and Franz Rothlauf. 2021. A generalizability measure for program synthesis with genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 822–829.

[33] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2021. Recent Developments in Program Synthesis with Evolutionary Algorithms. arXiv:2108.12227 [cs.NE]

[34] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The push3 execution stack and the evolution of control. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. 1689–1696.

[35] 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.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.