Highlights

Grammar-Obeying Program Synthesis: A Novel Approach Using Large Language Models and Many-Objective Genetic Programming

Ning Tao,Anthony Ventresque,Vivek Nallur,Takfarinas Saber

- LLMs are effective at generating correct programs for program synthesis tasks; however, they often fail to produce programs that adhere to specified grammars.
- Five different LLMs were leveraged with a Similarity-Based Many-Objective G3P (SBMaOG3P) framework.
- The results on a well-known benchmark dataset demonstrate that SBMaOG3P is capable of finding correct programs that obey the defined grammars, outperforming LLMs and the state-of-the-art G3P.

Grammar-Obeying Program Synthesis: A Novel Approach Using Large Language Models and Many-Objective Genetic Programming

Ning Tao[a,*,1], Anthony Ventresque[b,2], Vivek Nallur[a,1] and Takfarinas Saber[c,3]

[a]*School of Computer Science, University College Dublin, Dublin, Ireland*
[b]*Lero, School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland*
[c]*Lero, School of Computer Science, University of Galway, Galway, Ireland*

ARTICLE INFO

Keywords:
Program Synthesis
Grammar
LLMs
Grammar Guided GP
Multi-objective

ABSTRACT

Program synthesis is an important challenge that has attracted significant research interest, especially in recent years with advancements in Large Language Models (LLMs). Although LLMs have demonstrated success in program synthesis, there remains a lack of trust in the generated code due to documented risks (e.g., code with known and risky vulnerabilities). Therefore, it is important to restrict the search space and avoid bad programs. In this work, pre-defined restricted Backus-Naur Form (BNF) grammars are utilised, which are considered 'safe', and the focus is on identifying the most effective technique for *grammar-obeying program synthesis*, where the generated code must be correct and conform to the predefined grammar. It is shown that while LLMs perform well in generating correct programs, they often fail to produce code that adheres to the grammar. To address this, a novel Similarity-Based Many-Objective Grammar Guided Genetic Programming (SBMaOG3P) approach is proposed, leveraging the programs generated by LLMs in two ways: (i) as seeds following a grammar mapping process and (ii) as targets for similarity measure objectives. Experiments on a well-known and widely used program synthesis dataset indicate that the proposed approach successfully improves the rate of grammar-obeying program synthesis compared to various LLMs and the state-of-the-art Grammar-Guided Genetic Programming. Additionally, the proposed approach significantly improved the solution in terms of the best fitness value of each run for 21 out of 28 problems compared to G3P.

1. Introduction

Program synthesis seeks to streamline or automate programming tasks by offering a set of techniques that enable code creation from a high-level description of task objectives (such as textual descriptions, input/output examples, or sketches).

In recent years, Large Language Models (LLMs) have demonstrated success in executing a variety of software engineering tasks [42, 15, 17, 32, 41, 20], including program synthesis from natural language prompt. However, the code generated by LLMs often lacks trustworthiness and might include documented risks. (e.g., code contains known and risky vulnerabilities [1, 30]). The possibility of faulty LLM-generated code poses a substantial and escalating threat to software and its stakeholders. Additionally, the inherent generative nature of LLMs limits their effectiveness in correcting errors through iterative prompting [31, 14].

To limit the search space and exclude programs that exhibit poor coding practices, use unreliable libraries or functionalities, or deviate from the desired design pattern, the definition of *"safe"* grammars is considered. This ensures that any generated program adheres to these grammars, thereby mitigating security risks and enhancing code quality.

Defining *"safe"* grammars is a complex challenge by itself that requires extensive research to design and validate them for various tasks. However, our focus in this work is on **grammar-obeying program synthesis**, which is the ability to generate correct programs that obey the specified grammar.

In this work, it is demonstrated that although five well-known LLMs (i.e., ChatGPT, Gemma, LLaMa, Mistral, and Zephyr) are effective at generating correct programs for program synthesis tasks, they frequently fail to produce programs that adhere to specified grammars.

A study by Tao et al. [37] has shown that it is possible to help improve LLMs' performance at grammar-obeying program synthesis by mapping their code to the predefined grammar and evolving it using Grammar-Guided Genetic Programming (G3P). While this approach brings some improvements, it only leverages one LLM-generated program at a time as a seed–thus losing an important amount of information.

Other works by Tao et al. [36, 35, 34] have shown that G3P can be further improved by leveraging the similarity of evolved programs to an "ideal" target code alongside the input-output error rate. However, these studies were considered correct target codes obtained from a fictive oracle–making them inapplicable in practice.

Tao et al. [33] demonstrated in their recent research that integrating ChatGPT solutions with a grammar-mapping process into a Many-objective G3P system can enhance the performance of grammar-obeying program synthesis tasks. However, relying solely on ChatGPT solutions as the initial seed restricts the diversity of code structures.

In this work, a Similarity-Based Many-Objective Grammar-Guided Genetic Programming (SBMaOG3P) approach is proposed, which leverages programs generated by multiple LLMs in two ways:

ning.tao@ucdconnect.ie (N. Tao); anthony.ventresque@tcd.ie (A. Ventresque); vivek.nallur@ucd.ie (V. Nallur); takfarinas.saber@universityofgalway.ie (T. Saber)
ORCID(s): 0000-0002-8154-547X (N. Tao)

- Leveraging the similarity towards the programs generated by 5 LLMs (i.e., ChatGPT, Gemma, LLaMa, Mistral and Zephyr) in a many-objective approach.
- Leveraging the programs generated by the LLMs as seeds to the evolutionary process (following a mapping to the predefined grammar).

The effectiveness of the proposed method has been assessed by benchmarking it against the traditional G3P algorithm introduced by Forstenlechner et al. [6], and the latest algorithm for grammar-obeying program synthesis developed by Tao et al. [33]. The results show that our proposed approach found solutions for the majority of tasks considered and outperformed the state-of-the-art algorithm. However, in the default configuration, SBMaOG3P did not outperform the LLM at generating solutions when grammar constraints were ignored.

The structure of the paper is as follows: Section 2 provides a summary of the relevant background and related work. Section 3 presents our novel SBMaOG3P approach. The details of our experimental setup are described in Section 4. Section 5 discusses the results of our experiments and provides an analysis. Finally, Section 6 concludes the paper and suggests directions for future research.

2. Background and Related Work

2.1. Genetic Programming

Genetic Programming (GP) is a type of evolutionary algorithm designed to automate creation and optimisation by repeatedly assessing their effectiveness in performing specific tasks. GP aims to develop programs through the evolution of a population of individuals. These individuals initially comprise randomly chosen candidates, typically ill-suited for the intended function. GP utilises genetic operators inspired by natural processes, such as crossover, mutation, and selection. Over time, a variety of GP systems have been introduced, each with distinct characteristics (e.g., GP [13], Cartesian GP [19], and Linear GP [3]).

2.2. Grammar-Guided Genetic Programming

G3P is a variant of Genetic Programming (GP) that leverages BNF grammar to define the search space. Grammatical Evolution [23] and Context-Free Grammar Genetic Programming (CFG-GP) by Whigham [39] are two well-known representations of G3P algorithms. By introducing grammar rules into the evolution process, the G3P algorithm ensures the reliability of the generated solutions, which are syntactically correct. This BNF grammar file is often predefined with a problem set that is suitable for various applications. It is widely utilised in automated programming [21], transport system management [29], and wireless communications scheduling [16, 26, 28, 27, 25]. However, designing grammar for each problem limits the scalability of the G3P algorithm.

In response, Forstenlechner et al. [5] designed an automatic grammar construction approach for G3P, where one BNF grammar design can be used for various problems. The mechanism behind this automation involves designing short grammars for each data type. The algorithm automatically selects the appropriate grammar as long as the user provides the data type for the problem. Another benefit of designing short grammar for each data type is reducing the search space by removing irrelevant grammar and reducing computation costs and execution time. The authors [6] further extended the grammar construction algorithm to include character data types, which were previously handled as strings. In the updated approach, they also introduced recursions, enriching the diversity of grammar choices to tackle a wider range of problems.

2.3. Large Language Models

Large Language Models (LLMs) are AI algorithms that leverage extensive parameters and deep learning architecture to understand and generate human language. They use transformer-based neural networks to capture long-range dependencies and contextual relationships within text. This architecture, combined with techniques such as self-attention mechanisms, has enabled LLMs to perform complex language tasks with high accuracy and fluency.

LLMs can effectively tackle a wide range of real-world problems, including programming, writing, and design [15, 17, 32, 41, 20]. Notable examples include AI chatbots like OpenAI's ChatGPT [22] and Google's Gemini [18], which use LLMs as their core technology.

ChatGPT [22] is one of the most powerful AI chatbots, capable of generating human-like responses based on user input. It can answer users' questions using a large knowledge corpus. Beyond simple queries, it can perform complex real-world tasks such as writing emails, analysing and summarising text, processing pictures, and even programming.

LLMs can generate programs based on user intent. However, they often produce inaccurate code due to ambiguous task descriptions and the complexity of programming tasks. Wang et al. [38] enhanced the programming capabilities of LLMs by leveraging BNF grammar as external knowledge and incorporating domain-specific constraints. In their approach, they included grammar in the prompt that could generate solutions for the given input-output examples while keeping the grammar size minimal. This method allows LLMs to generate programs more accurately by predicting the appropriate grammar for new tasks. In this research, we require LLMs to generate programs that adhere to predefined BNF grammar rather than allowing the model to predict the grammar to ensure accurate responses.

2.4. Program Similarity Assessment

Measuring code similarity is a critical task in programming. It enables the identification of repetitive code, the discovery of similar bugs in software development, and plagiarism detection in assignment assessments [9]. This research uses program similarity for fitness evaluation in the SBMaOG3P algorithm. Four similarity measures, rated highest in a survey by Ragkhitwetsagul et al. [24], were selected.

2.4.1. FuzzyWuzzy

FuzzyWuzzy [4] is developed based on the difflib library for string searching. This library provides two similarity functions for string matching named $TokenSortRatio$ and $TokenSetRatio$. The $TokenSortRatio$ function starts by preprocessing the string, which involves removing punctuation, converting the string to lowercase letters, and sorting the tokens. It then uses the sorted tokens to generate a similarity score. The $TokenSetRatio$ function differs by processing the tokens without sorting them. Although this library is designed for string matching, researchers have found it very efficient for program similarity detection [24].

2.4.2. Cosine

The cosine similarity function is designed to calculate the similarity between two vectors. This similarity measure can be applied to program similarity calculation by tokenising the programs into vectors. The steps for performing cosine similarity calculation between two programs are as follows:

- **Preprocessing and tokenisation**: The programme is split into tokens, removing irrelevant formatting symbols.
- **Construct Frequency Vector**: Iterate through the tokens generated in the previous step to create a token frequency vector for each program.
- **Similarity Score Calculation**: The similarity between two programmes is calculated using Eq. 1. In this equation, the token frequency vectors from the previous step are denoted as vectors **A** and **B**.

$$\cos(\mathbf{A},\mathbf{B}) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n}\mathbf{A}_i\mathbf{B}_i}{\sqrt{\sum_{i=1}^{n}(\mathbf{A}_i)^2}\sqrt{\sum_{i=1}^{n}(\mathbf{B}_i)^2}} \quad (1)$$

2.4.3. CCFinder

CCFinder is a token-level code clone detection tool designed by Kamiya et al. [12]. It processes the programme using the following steps to identify code clones: (i) Lexical Analysis: The programme is tokenised by applying language-specific lexical rules. (ii) Transformation: Each token from the previous step is transformed into a standard expression to enable the system to identify code clones with diverse expressions. (iii) Clone Matching: The main technique for identification leverages a suffix-tree matching algorithm. (iv) Formatting: Each clone pair is formatted and reported in this final step.

The tool is designed to identify code clones in large source codebases. Considering that the programs for similarity calculation in this research are fairly small, we applied the following adjustments to meet our requirements.

- Since CCFinder reports code clones rather than providing a similarity score, the following equation (Eq. 2) was introduced to calculate the similarity score of the identified clones. In this equation, the length of the cloned code is divided by the length of the source code to generate the similarity score.

$$Similarity(a,b) = \frac{Len(Clone(a,b))}{Max(Len(a),Len(b))} \quad (2)$$

  here $Clone(a,b)$ denotes the longest code clone between two programs a and b, while $Len(a)$ denotes the length (i.e., number of characters) of the program a.

- This research simplified the suffix-tree matching algorithm by calculating the common token length between two code snippets using a two-dimensional token matrix. Each dimension represents the token sequence of the program.
- The code clone reporting step is removed as only the similarity score is needed for this research.

2.4.4. SIM

Gitchell et al. [8] proposed a plagiarism detection tool called SIM, designed for detecting plagiarism in C programming course assignments. The tool utilises a string alignment algorithm at the token level to detect similar code structures. The advantage of the string alignment technique is that it can identify similar code even if the programme sequence has been locally modified.

The tool identifies plagiarism using the following steps: (i) splitting the programme into tokens and (ii) detecting similarity using an alignment algorithm. The tokenisation step involves lexical analysis to retain fundamental programme parts such as keywords, identifiers, literals, and operators while removing meaningless structural symbols. The alignment algorithm divides the second token sequence into multiple segments and uses each segment to align with the first token sequence to calculate the similarity score.

3. Proposed Approach

This study focuses on addressing grammar-obeying program synthesis problems by producing correct programs that adhere to a BNF grammar, thereby restricting the code's structure and the available functions, methods, and libraries.

We propose (i) prompting several LLMs to generate programs based on a task description, (ii) mapping the program generated by the LLMs to programs that adhere to a predefined BNF grammar, (iii) feeding the mapped programs as a seed to our SBMaOG3P evolutionary process, and (iv) performing the SBMaOG3P evolutionary process which uses similarity measures towards the initial programs generated by LLMs as secondary objectives to guide the search process. All the data and algorithms used in this research, as well as the tools to run the experiment, are available online[1].

The overview of our proposed approach is shown in Figure 1. Our approach involves prompting 5 LLMs (i.e.,

[1] https://github.com/TonBatbaatar/SBMaOG3P

ChatGPT, Gemma, LLaMa, Mistral and Zephyr) to generate programs based on the provided textual description, and then mapping each of these programs into the predefined grammar. Additionally, our approach expands the G3P algorithm through (i) the seeding of the grammar-mapped LLM-generated programs into SBMaOG3P's initial population and (ii) the utilisation of diverse code similarity measures (as secondary objectives) with input-output error rate (as the primary objective) to guide the evolution.

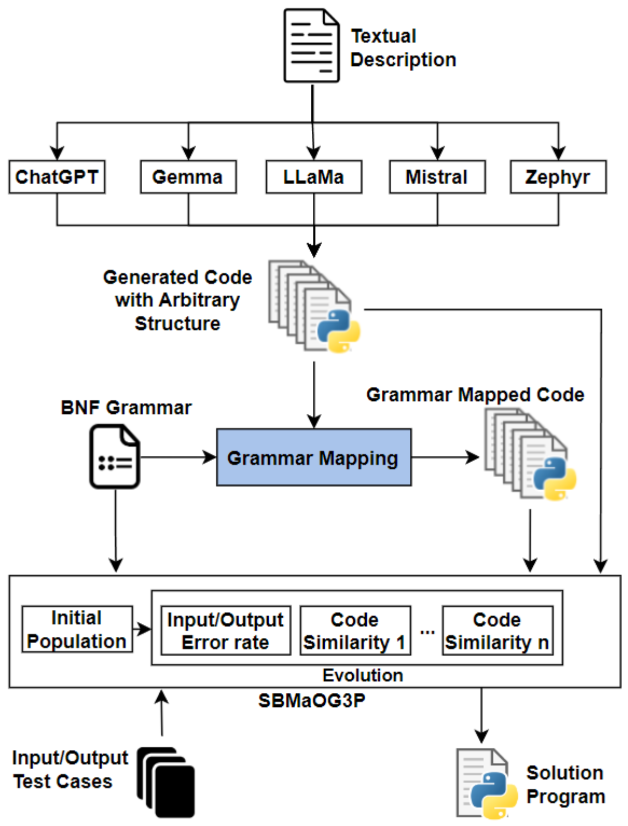

Figure 1: Overview of Our Approach

3.1. LLM Code Generation

LLMs demonstrate exceptional abilities in generating code based on user-provided prompts. Five popular LLMs (ChatGPT, Gemma, LLaMa, Mistral, and Zephyr) are utilised to generate code snippets in two different ways: First, by providing only textual task descriptions, and second, by including the task description along with the BNF grammar. Given the high success rate observed in generating code for benchmark problems, the LLMs were queried once for each problem using the default temperature settings.

A recent study highlights the substantial impact of input prompt quality on the effectiveness of LLMs [7, 40]. To ensure the quality of the generated code, two different prompt templates were defined, each following a structured prompt template, as shown below. Prompt Template 1 instructs the LLM to generate a program based on the task's textual description, whereas Prompt Template 2 provides the LLM with the required grammar alongside the task description.

Prompt Template 1: Task Only

Main task: Write a Python function without comment, explanation, and example usage.
Task description: {*input_task_description*}
Output program format: Function parameter name has to be in0, in1...(depends on how many parameters it needs), and return variable name has to be res0, res1...(depends on how many parameters it needs).

Prompt Template 2: Task with Specified Grammar

Main task: Write a Python function without comment, explanation, and example usage.
Task description: {*input_task_description*}
Output program format: Function parameter name has to be in0, in1...(depends on how many parameters it needs), and return variable name has to be res0, res1...(depends on how many parameters it needs).
Output program grammar: {*input_task_BNFgrammar*}

For Prompt Template 1, we start with the *Main task*, outlining the general goal of the query. Following this, the *Task description* section provides a detailed description of each task. Finally, the *Output program format* includes additional structural information to format the output program for our experiment. Building on Prompt Template 1, Prompt Template 2 incorporates the specified BNF grammar in the *Output program grammar* section.

Note that although two prompt templates were defined, based on the experimental analysis, which indicates the weakness of LLMs when specifying the required grammar in the prompt (as detailed below), Prompt Template 1 is used in the proposed approach.

3.2. Similarity-Based Many-Objective G3P

In this research, a similar Many-Objective G3P algorithm, proposed by Tao et al. [36], is utilised with the goal of guiding the search towards programs that are structurally similar to a target code. However, in our approach, instead of providing a unique target program provided by a fictive oracle for similarity calculation, we use programs generated by five distinct LLMs. The ability to leverage multiple target codes from diverse LLMs makes our algorithm scalable to a wider range of problems – taking advantage of the ability that some LLMs have at synthesising programs to some specific tasks while alleviating their limitations on others.

Figure 2 shows the overview of SBMaOG3P. In addition to evaluating input-output error rates, our approach incorporates code similarity measures with LLM-generated code as secondary objectives. We considered four different similarity measures described in Section 2 for our system. Our algorithm evaluates a program within the population

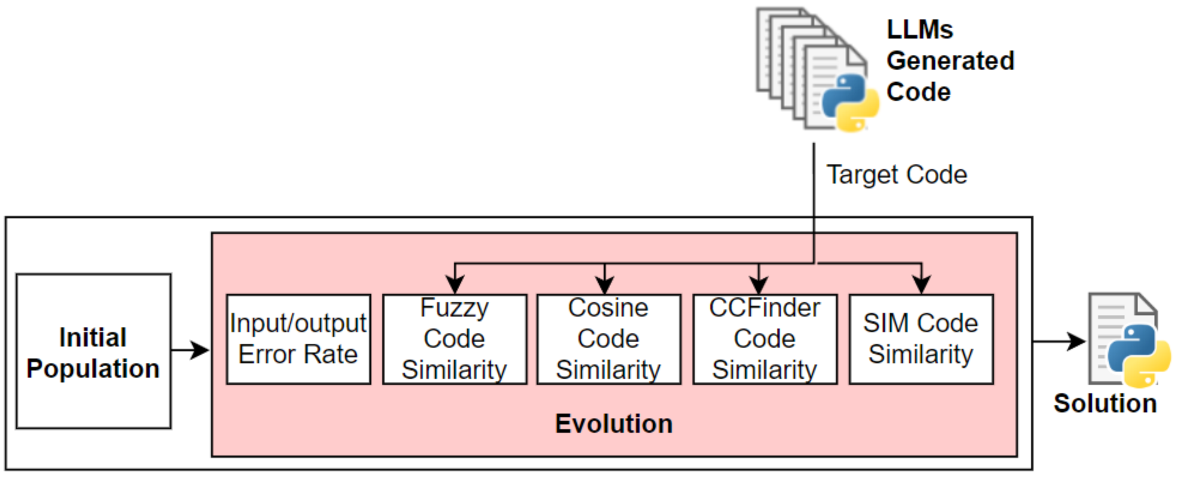

Figure 2: Overview of SBMaOG3P

using five fitness functions: one primary error-based objective and four similarity measures. The algorithm selects the highest similarity score value for each similarity measure by comparing it against the programs generated by the five different LLMs. However, our approach extends beyond mere correctness evaluation. By introducing code similarity measures as secondary objectives, our aim is not solely to identify the correct solution but, more importantly, to strategically steer the search process towards more plausible program candidates.

Although SBMaOG3P uses multiple objectives to evolve population programs, our method considers a task solved by an individual only if it meets the primary objective. Specifically, the individual must correctly pass all test cases for the assigned task.

In this research, the tournament selector was adapted to handle multiple objectives, enabling the algorithm to evolve programs based on diverse criteria. During the selection of parents for the next generation, the standard G3P algorithm is followed for half of the individuals, with parents chosen based on the input/output error rate. For the other half, one parent is selected using the primary objective (error rate) and the other using a secondary objective (code similarity). Since there are four different similarity measures used as secondary objectives, these measures are evenly rotated through when selecting parents.

Deciding which individuals will survive through evolution is a crucial aspect of our algorithm. Similar to the parent selection mechanism, we take all objectives into account during this phase. The primary objective determines the survival of half of the population, while the remaining half is decided based on the four similarity measures.

3.3. Seeding LLM Solutions to SBMaOG3P

In our SBMaOG3P approach, individual programs are represented as symbolic expression trees. Each node in the tree corresponds to a specific code segment, and the connections between nodes capture the structural information. The seeded programs must follow the same grammar rules and maintain the same tree structure to enable their evolution by SBMaOG3P (i.e., need by genetic operators such as crossover, mutation, and selection to function correctly). Therefore, to seed LLM-generated programs in SBMaOG3P, they must first be adapted to fit the predefined grammar.

Specifying the BNF grammar in the LLM prompt (using Prompt Template 2) reduced the success rate of LLMs at producing correct programs. The LLM often generated code that did not address the problem description and included snippets outside the defined grammar (see results below for details). In response, LLM-generated code was seeded using task-only prompts (i.e., Prompt Template 1). A grammar mapping algorithm, proposed in [37], was used to transform programs into ones that adhere to predefined grammars, enabling the structured evolution of LLM-generated code.

The grammar mapping algorithm starts by generating an Abstract Syntax Tree (AST) [2] of the provided program. Then, it maps the AST into a symbolic expression tree for SBMaOG3P, starting from the root node and iterating through its child nodes, and constructing the output program recursively. For each tree node, when there is no grammar conflict, the algorithm constructs the output tree. Otherwise, it builds output with a dummy expression, which can be further improved by the genetic programming process during evolution. It also maps the variables and constants. When the node is a variable “Name”, the mapping algorithm checks if it has previously been mapped, in which case, it maps it to the same one. If the variable “Name” has not been previously mapped, the algorithm maps it using the first unused variable name with the same data type.

During the mapping process, the LLMs-generated code becomes erroneous code due to grammar conflict. In the next step, we seed 5 grammar-mapped LLM-generated codes into the SBMaOG3P’s initial population for further fixing. The seeding process is demonstrated in the Figure 3.

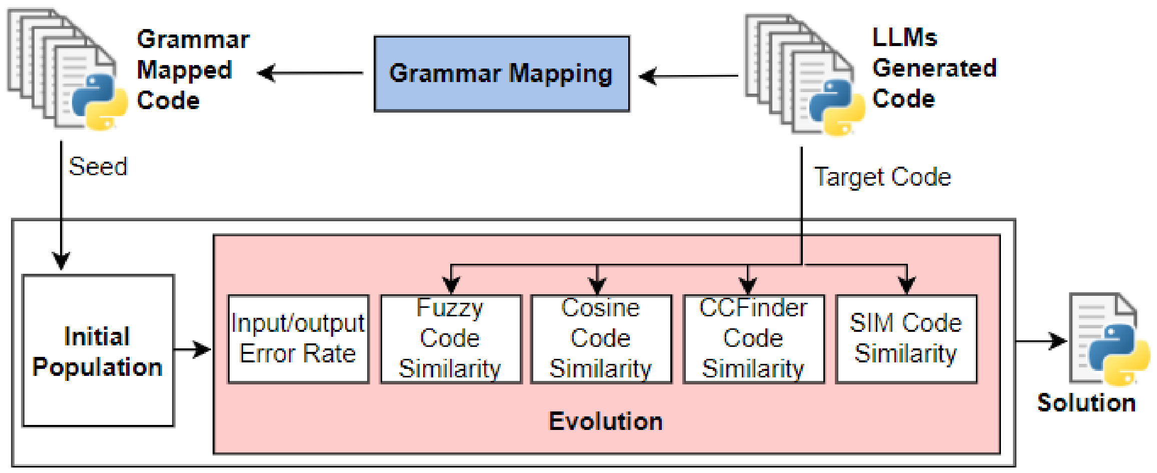

Figure 3: Overview of SBMaOG3P with Seeding

4. Experiment Setup

4.1. Research Questions

We evaluate the performance of our approach by attempting to answer the following Research Questions (RQs):

- **RQ1: How effective are LLMs at grammar-obeying program synthesis?**
- **RQ2: Could we improve the performance of LLMs at grammar-obeying program synthesis using SBMaOG3P?**

4.2. Benchmark Dataset

Helmuth and Spector [11, 10] created a benchmark suite for program synthesis problems, which contains 29 problems selected from introductory-level programming courses. This benchmark suite provides detailed natural language descriptions for each problem, as well as training and testing sets. Four problems with its task description are shown in Table 1. Table 2 indicates the number of train and test cases for each problem. In this research, our proposed system is evaluated with 28 problems from this benchmark suite. One problem named “String Differences” is removed from our experiment as in previous work by Forstenlechner [5]. The original benchmark program tested with PushGP often prints the result, whereas in G3P, the results are return values. Consequently, the “String Differences” problem is excluded because it requires multiple return values with different data types that our grammar cannot accommodate. In this experiment, the same grammar defined by Forstenlechner et al. [5][2] is used.

Note that, since LLMs are being used, there is a risk that the benchmark dataset may have already been included in the training data of these models. However, this does not pose a problem for our study. Even if the problems and their corresponding code have been leaked, it is unlikely that the grammars we use have also been leaked. This is evident from our experiments, where we observe that most of the LLM-generated code does not conform to the predefined grammars. Therefore, while the generated code is often interesting, it is not grammatically correct.

[2]Available at: https://github.com/t-h-e/HeuristicLab.CFGGP/tree/master/HeuristicLab.Problems.Instances.CFG/GrammarConstruction

Table 1
Description of Four Example Problems in the Dataset

Problem Name	Description
Double Letters	Given a string, print the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.
Collatz Numbers	Given an integer, find the number of terms in the Collatz (hailstone) sequence starting from that integer.
Replace Space with Newline	Given a string input, print the string, replacing spaces with newlines. Also, return the integer count of the non-whitespace characters. The input string will not have tabs or newlines.
String Lengths Backwards	Given a vector of strings, print the length of each string in the vector starting with the last and ending with the first.

4.3. Generating Programs Using LLMs

Unlike PushGP, each individual in our SBMaOG3P algorithm is a code snippet (i.e., a function) rather than a complete program with a console output. The SBMaOG3P algorithm evaluates each individual by calculating the fitness value based on the function’s return value compared to the expected output of the test case. However, the benchmark is designed to output results to the console, which is unsuitable for our experiment. Using such program descriptions as

Table 2
Number of Train and Test Cases for Each Problem in the Dataset

Name	Train	Test
Number IO	25	1000
Small Or Large	100	1000
For Loop Index	100	1000
Compare String Lengths	100	1000
Double Letters	100	1000
Collatz Numbers	200	2000
Replace Space with Newline	100	1000
Even Squares	100	1000
Wallis Pi	150	50
String Lengths Backwards	100	1000
Last Index of Zero	150	1000
Vector Average	100	1000
Count Odds	200	2000
Mirror Image	100	1000
Super Anagrams	200	2000
Sum of Squares	50	50
Vectors Summed	150	1500
X-Word Lines	150	2000
Pig Latin	200	1000
Negative To Zero	200	2000
Scrabble Score	200	1000
Word Stats File	100	1000
Checksum	100	1000
Digits	100	1000
Grade	200	2000
Median	100	1000
Smallest	100	1000
Syllables	100	1000

prompts for LLMs often results in programs with `print` statements instead of function returns. To adapt the problem descriptions to our needs, we modified them to specify that the program should return a value from a function instead of using a `print` statement. For example, we replaced the keyword "print" with "return" in the task descriptions. The modified problem descriptions are available online[3].

4.4. Parameter Settings

All five LLMs (ChatGPT, Zephyr[4], Mistral[5], Gemma[6] and LLaMa[7]) are used with the default temperature settings, and the version of each LLM is detailed in Table 3. we use the standard G3P parameter settings as used in previous studu [5]. We ran the evolution for each program synthesis task 30 times. The benchmark suite suggested using 300 generations for most tasks, while for straightforward synthesis tasks ("Median", "Number IO", and "Smallest"), 200 generations are enough. Other settings for our SBMaOG3P systems are indicated in Table 3.

[3]Available at: https://github.com/TonBatbaatar/SBMaOG3P/blob/main/Problem_Edited.csv

[4]Available at: https://huggingface.co/HuggingFaceH4/zephyr-orpo-141b-A35b-v0.1

[5]Available at: https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

[6]Available at: https://huggingface.co/google/gemma-1.1-7b-it

[7]Available at: https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct

Table 3
Experiment Parameter Settings

Parameter	Setting
Runs	30
Generation	300[a]
Population size	1000
Tournament size	7
Crossover probability	0.9
Mutation probability	0.05
Node limit	250
Variable per type	3
Max execution time (s)	1
ChatGPT version	GPT4
Zephyr version	zephyr-orpo-141b-A35b-v0.1
Mistral version	Mistral-7B-Instruct-v0.2
Gemma version	gemma-1.1-7b-it
LLaMa version	Meta-Llama-3-70B-Instruct

[a]200 generations for "Median", "Number IO", and "Smallest" as in [11]

5. Result

5.1. Effectiveness of LLM at Grammar-Obeying Program Synthesis (RQ1)

The effectiveness of LLMs in grammar-obeying program synthesis is analysed by comparing the selected LLMs and G3P with tournament selection on benchmark problems. The results are presented in Table 4. Each LLM is evaluated twice: once using the task-only prompt and once using the prompt with the task description and the specified BNF grammar. For LLMs' solutions, a regular checkmark (✔) indicates that the approach successfully solved the task using the BNF grammar. A circle checkmark (✅) signifies that the approach solved the task using a grammar not permitted by the BNF grammar. A cross symbol (✖) indicates that the approach failed to find a solution for the task. For the G3P algorithm, a problem is marked with a regular checkmark (✔) if at least one correct solution is found within 100 runs. Conversely, if the G3P algorithm fails to evolve a correct solution within 100 runs, it is marked unsuccessful (✖).

The experiment demonstrates that LLMs performed exceptionally well in the program synthesis task. At least one LLM found correct solutions for 27 out of the 28 problems considered. Only one problem, "Wallis Pi," remained unsolved, as the LLMs generated programs that calculate the value of $\frac{\pi}{2}$ or π instead of $\frac{\pi}{4}$.

Table 4
Comparison of LLMs and G3P on Benchmark Problems

Benchmark Problem	G3P	LLMs - Task Only					LLMs - Task with Specified Grammar				
		ChatGPT	Gemma	LLaMa	Mistral	Zephyr	ChatGPT	Gemma	LLaMa	Mistral	Zephyr
Number IO	✔	✔	✔	✔	✔	✔	✔	✖	✔	✖	✖
Small Or Large	✔	✅	✅	✔	✔	✅	✔	✅	✔	✖	✅
For Loop Index	✖	✅	✅	✔	✅	✔	✔	✖	✔	✖	✅
Compare String Lengths	✖	✅	✅	✅	✅	✅	✔	✅	✔	✖	✖
Double Letters	✖	✅	✅	✅	✅	✅	✔	✖	✅	✖	✅
Collatz Numbers	✖	✅	✔	✔	✅	✔	✔	✖	✖	✖	✖
Replace Space with Newline	✔	✅	✅	✅	✅	✅	✔	✖	✅	✔	✔
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	✔	✅	✔	✅	✅	✅	✔	✖	✅	✔	✅
Number of Problems Solved	12	2 (24) [a]	5 (19) [a]	5 (21) [a]	3 (20) [a]	4 (22) [a]	10 (15) [a]	1 (11) [a]	11 (14) [a]	3 (3) [a]	2 (14) [a]

[a] number in the bracket indicates the problem solved using a grammar not permitted by the BNF grammar

The performance of the LLMs we tested is quite comparable: ChatGPT, LLaMa, and Zephyr each solved 26 problems, Gemma solved 24 problems, and Mistral solved 23 problems. However, the performance of LLMs on grammar-obeying program synthesis varies significantly. All five LLMs, when prompted with only the task description, solved no more than five problems obeying the BNF grammar. The ability to generate code obeying the BNF grammar slightly improved for ChatGPT and LLaMa when using a prompting template that included both the task description and the specified grammar. We also observed that prompting with grammar reduced the code quality and the number of problems for which the LLMs generated correct solutions. Therefore, we decided to seed the LLMs' generated code by mapping the grammar for the subsequent SBMaOG3P experiments.

It is uncertain whether the impressive performance of LLMs in program synthesis is due to the benchmark datasets already being included in their training data. However, we can assert that the grammar of these problems was not leaked into the training set, considering that most LLMs generate programs using grammar rules not permitted by the predefined BNF grammar. Therefore, while LLMs are proficient at program synthesis, they struggle with grammar-obeying program synthesis. Solution programs obtained by LLMs are available online [8].

5.2. Performance of Proposed Approach (RQ2)

Since the proposed approach combines several components, its performance is assessed in two steps to identify the contribution of each: (i) the performance of the grammar-mapping algorithm and (ii) the performance of SBMaOG3P.

5.2.1. *Performance of the Grammar Mapping Algorithm*

In this subsection, the performance of the grammar mapping algorithm is analysed. All the code snippets generated by the LLMs were successfully mapped to code that adheres to the BNF grammar. However, some information was omitted due to grammar conflicts between the LLMs' generated code and the predefined BNF grammar. Table 5 summarises each solution's grammar mapping status for all considered LLMs. When the algorithm is able to map the LLMs' generated code without any grammar conflict, we mark the mapping process with a regular checkmark (✔). If there is minor information loss during the mapping process, we indicate the mapping status with a circle checkmark (✅). If the mapping retains minimal information from the LLMs' generated code due to significant grammar conflicts, we mark it with a cross (✖). The results show that the solutions generated by ChatGPT and Zephyr are mapped with relatively good information retention among the five considered LLMs. The mapping performance for solutions produced by Gemma is poor because it fails to meet the required format in terms of code structure and the naming scheme of input parameters. By analysing the grammar-mapped code, the grammar conflicts are summarised into the following main categories:

- **Wrong Code structure**: The code for seeding requires a fixed number of variables in the `return` statement instead of using a function or expression directly in the `return` statement.
- **Violates Naming Scheme**: The LLMs' generated code snippet must follow the naming scheme for the function's input parameters as specified in the prompt.
- **Function/Library Mismatch**: The LLMs' generated code often includes functions and libraries not defined in the BNF grammar. Even for functions that are defined in the BNF grammar, the number of parameters often mismatches.
- **Advanced Python Grammar**: The LLMs often solve the task using advanced Python grammar, such as `list comprehension`, which is not defined in the BNF grammar.
- **Data Type Conflict**: The data type that can be used for each problem is defined within the problem's grammar. When LLM-generated code uses data types that are not allowed for the current problem, the grammar mapping algorithm cannot map them.

5.2.2. *Performance of SBMaOG3P*

The proposed approach was evaluated by comparing the performance of three distinct G3P variants: (1) G3P with tournament selection without seeding proposed by Forstenlechner et al. [5, 6], (2) the state-of-the-art grammar-obeying program synthesis algorithm (named SBMaOG3P$_{ChatGPT}$) proposed by Tao et al. [33], and (3) our proposed algorithm (named SBMaOG3P$_{5LLMs}$).

SBMaOG3P$_{ChatGPT}$ and SBMaOG3P$_{5LLMs}$ both use grammar-mapped seeds in the initial population and utilise code similarity as secondary objectives. The main difference between these two approaches is that SBMaOG3P$_{ChatGPT}$ seeds only one grammar-mapped code generated by ChatGPT and calculates the code similarity using the ChatGPT-generated code as the target. In contrast, SBMaOG3P$_{5LLMs}$ seeds grammar-mapped programs generated by five LLMs and calculates the similarity score by choosing the maximum similarity value using the five target codes.

Each algorithm was run 30 times on the benchmark dataset, and the number of successful runs is reported in Table 6. A run is considered successful when the evolution finds at least one correct solution that passes all training and test cases.

Overall, the proposed SBMaOG3P$_{5LLMs}$ performed the best in both the number of solved problems and the success rate. Specifically, the proposed approach solved 21 problems while adhering to the predefined BNF grammar, which is 9 more problems solved compared to the G3P approach and 5 more problems compared to the SBMaOG3P$_{ChatGPT}$.

[8] Available at: `https://github.com/TonBatbaatar/SBMaOG3P`

Table 5
Grammar Mapping Status For LLM-generated code

Benchmark Problem	ChatGPT	Zephyr	Mistral	Gemma	LLaMa
NumberIO	✔	✔	✔	✖	✔
Small Or Large	✔	✔	✅	✖	✔
For Loop Index	✔	✅	✅	✖	✔
Compare String Lengths	✔	✖	✖	✖	✔
Double Letters	✔	✔	✅	✖	✔
Collatz Numbers	✔	✅	✖	✅	✔
Replace Space with Newline	✅	✔	✅	✖	✅
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	✔	✖	✖	✖	✖
Number of informative map [a]	18	20	10	3	10

[a] Sum of mapping without information loss or minor information loss

While the proposed approach did not evolve correct solutions for the program synthesis task as effectively as popular LLMs, it excelled in grammar-obeying program synthesis. Notably, by seeding grammar-mapped programs generated using prompts without specifying grammar, our approach successfully corrected LLM-generated solutions that violated the grammar for 14 problems. Even compared to the solutions generated by LLMs prompted with the task description and the BNF grammar, our approach evolved programs that fit the BNF grammar for 7 more problems. However, it failed to address the unsolved problem ("Wallis Pi") in its default configuration.

The proposed approach evolved correct solutions for 21 problems, leaving 7 problems unsolved with the current configuration. We further examined the reasons for each unsuccessful problem.

- **Wallis Pi:** Although the seeded grammar-mapped LLM code was close to the correct solution (generating code for calculating $\frac{\pi}{2}$ instead of $\frac{\pi}{4}$), the evolution did not correct the code with its default settings. We believe that increasing the computational power (i.e., increasing the number of generations) could enable the evolution process to fix the erroneous code.

- **Vector Average:** For this problem, our proposed algorithm did not gain helpful information from seeding since all five LLMs' generated codes utilised the `Sum` function, which is not supported by the BNF grammar.

- **SuperAnagrams:** Similar to the previous problem, all five LLMs' seeds utilised advanced Python grammar (i.e., list comprehension) to tackle this problem, which is not allowed in the BNF grammar.

- **X-word lines, Word Stats:** A large grammar conflict involving the use of data types prevented our algorithm from evolving a correct solution for these two string analysis problems. Specifically, all LLMs' solutions split the text using the `Split` function and stored it in a list variable. However, the list variable is not provided in the BNF grammar for these two problems, and the `Split` function is defined within a `for` loop that can not be used separately.

Table 6
Comparison of Proposed SBMaOG3P$_{5LLMs}$ Against State-of-the-Art System – SBMaOG3P$_{ChatGPT}$, and G3P

Benchmark Problem	G3P	SBMaOG3P		LLMs - Task Only					LLMs - Task with Specified Grammar				
		ChatGPT	5 LLMs	ChatGPT	Gemma	LLaMa	Mistral	Zephyr	ChatGPT	Gemma	LLaMa	Mistral	Zephyr
Number IO	17	30	30	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗
Small Or Large	0	30	30	✅	✅	✓	✓	✅	✓	✅	✓	✗	✅
For Loop Index	0	30	30	✅	✅	✓	✅	✓	✓	✗	✓	✗	✅
Compare String Lengths	0	30	30	✅	✅	✅	✅	✅	✓	✅	✓	✗	✗
Double Letters	0	30	30	✅	✅	✅	✅	✅	✓	✗	✅	✗	✅
Collatz Numbers	0	30	30	✅	✓	✓	✅	✓	✓	✗	✗	✗	✗
Replace Space with Newline	1	12	30	✅	✅	✅	✅	✅	✓	✗	✅	✓	✓
Even Squares	0	0	30	✅	✅	✅	✅	✅	✅	✅	✅	✗	✗
Wallis Pi	0	0	0	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
String Lengths Backwards	3	4	4	✅	✅	✅	✅	✅	✅	✗	✅	✗	✅
Last Index of Zero	6	30	30	✅	✅	✅	✅	✅	✅	✅	✅	✗	✗
Vector Average	0	0	0	✅	✅	✅	✅	✅	✅	✅	✅	✗	✅
Count Odds	0	30	30	✓	✓	✅	✅	✓	✅	✓	✓	✅	✅
Mirror Image	20	30	30	✅	✅	✅	✗	✅	✅	✅	✅	✗	✅
Super Anagrams	0	0	0	✅	✅	✅	✅	✅	✅	✅	✗	✅	✗
Sum of Squares	0	0	30	✅	✅	✅	✅	✅	✓	✗	✓	✗	✅
Vectors Summed	0	0	30	✅	✅	✅	✅	✅	✅	✗	✓	✗	✗
X-Word Lines	0	0	0	✅	✅	✅	✅	✅	✗	✗	✅	✗	✗
Pig Latin	0	0	30	✅	✗	✅	✅	✅	✅	✅	✅	✗	✅
Negative To Zero	1	3	30	✅	✅	✅	✅	✅	✓	✅	✓	✗	✗
Scrabble Score	0	0	0	✅	✗	✅	✗	✅	✅	✗	✓	✗	✅
Word Stats	0	0	0	✅	✗	✗	✗	✅	✅	✗	✅	✗	✅
Checksum	0	0	3	✅	✅	✅	✅	✅	✅	✗	✓	✓	✅
Digits	0	0	0	✗	✗	✅	✗	✗	✗	✗	✅	✗	✗
Grade	0	30	30	✅	✓	✓	✓	✅	✅	✅	✅	✗	✅
Median	12	30	30	✅	✅	✅	✅	✅	✅	✗	✅	✗	✗
Smallest	28	30	30	✅	✅	✅	✅	✅	✅	✅	✓	✅	✓
Syllables	0	30	30	✅	✓	✅	✅	✅	✓	✗	✅	✓	✅
Number of Problems Solved	8	16	21	2 (24) [a]	5 (18) [a]	5 (21) [a]	3 (20) [a]	4 (22) [a]	10 (15) [a]	1 (11) [a]	11 (14) [a]	3 (3) [a]	2 (14) [a]

[a] number in the bracket indicates the problem solved using a grammar not permitted by the BNF grammar

Table 7
The P-value for the Wilcoxon Rank Sum Test Using Best Fitness of Each Run Comparing G3P and Proposed SBMaOG3P$_{5LLMs}$

Problem	p-value
NumberIO	**0.034843**
Last Index of Zero	**0.000751**
Small Or Large	**0.000062**
Vector Average	0.143140
For Loop Index	**0.000064**
Count Odds	**0.000064**
Compare String Lengths	**0.000033**
Mirror Image	**0.077872**
Double Letters	**0.000055**
Super Anagrams	0.377587
Collatz Numbers	**0.000063**
Sum of Squares	**0.000064**
Replace Space with Newline	**0.000224**
Vectors Summed	**0.000064**
Even Squares	**0.000064**
X-Word Lines	0.795936
Wallis Pi	0.357296
Pig Latin	**0.000064**
String Lengths Backwards	**0.018616**
Negative To Zero	**0.000228**
Digits	**0.000064**
Scrabble Score	0.421242
Grade	**0.000064**
Word Stats	0.279861
Median	**0.005943**
Checksum	**0.019464**
Smallest	0.168078
Syllables	**0.000064**

- **Scrabble Score:** This problem assigns a score to each letter in a string, with the scores stored in a list according to the BNF grammar. Without providing the LLMs with the format of how such a scoreboard is defined in the grammar, their solutions can not generate a helpful seeding program to handle the scoreboard. This makes the seeding program inefficient in evaluating the correct solution.

- **Digits:** This problem requires splitting digits from a given number. Similar to the "Wallis Pi" problem, the LLMs' seeds were close to the correct solution. However, they failed to handle negative numbers, which requires assigning a negative sign to the least significant digit.

The Wilcoxon Rank Sum test was performed on the best test fitness value (error-rated fitness value for SBMaOG3P) from each run of G3P and the proposed approach to determine whether significant improvements in code generation performance exist. Table 7 shows the result of the Wilcoxon Rank Sum test. In the measurement, we used 0.05 as the threshold for the significance level, and significant results are highlighted in the table.

The proposed approach significantly improved the solution in terms of the best fitness value of each run for 21 problems compared to G3P with tournament selection. Surprisingly, we observed a significant improvement in the "Digits" problem, even though the problem remained unsolved with our approach. This further supports the idea that higher computational costs (i.e., more generations and larger population sizes) might improve the chances of solving the problem.

6. Conclusion and Future Work

This study proposed SBMaOG3P, which leverages code generated by five distinct LLMs (i.e., ChatGPT, Zephyr, Gemma, LLaMa and Mistral) to tackle grammar-obeying program synthesis tasks by evolving programs that are syntactically correct and adhere to a BNF grammar. By restricting the grammar rules of the program, it is ensured that the algorithm generates higher-quality programs without security threats.

The proposed approach utilised LLM-generated code in two ways: (i) leveraging the similarity to the programs generated by LLMs and (ii) using the programs generated by the LLMs as seeds for the evolutionary process, with the grammar mapped to fit the BNF grammar.

A comprehensive evaluation of the proposed method was conducted using the General Program Synthesis Benchmark Suite 1. The experimental results indicate that the approach successfully evolves accurate and grammar-compliant programs for various grammar-obeying program synthesis tasks, outperforming the state-of-the-art grammar-obeying program synthesis systems. However, there is still room for improvement, as in its default setup, SBMaOG3P does not achieve the success rate of LLMs when the grammar-fitting constraint is disregarded.

In our future research, we aim to improve the grammar mapping algorithm to support a wider variety of data structures, thereby maximising the use of grammar-mapped code in the evolutionary process. We also plan to evaluate the quality of the code generated by our approach. Additionally, we intend to explore the impact of employing various multi-objective optimisation algorithms within SBMaOG3P.

Acknowledgement

This work was supported by Science Foundation Ireland grant 13/RC/2094_P2 to Lero - the Science Foundation Ireland Research Centre for Software (www.lero.ie).

References

[1] O. Asare, M. Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering*, 28(6):129, 2023.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSME*. IEEE, 1998.

[3] M. Brameier, W. Banzhaf, and W. Banzhaf. *Linear genetic programming*. Springer, 2007.

[4] A. Cohen. Fuzzywuzzy: Fuzzy string matching in python, 2011.

[5] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In *Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings 20*, pages 262–277. Springer, 2017.

[6] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill. Extending program synthesis grammars for grammar-guided genetic programming. In *PPSN*. Springer, 2018.

[7] T. Gao, A. Fisch, and D. Chen. Making pre-trained language models better few-shot learners. *arXiv*, 2020.

[8] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. *ACM Sigcse Bulletin*, 1999.

[9] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *SIGCHI*, 2010.

[10] T. Helmuth and L. Spector. Detailed problem descriptions for general program synthesis benchmark suite. In *University of Massachusetts Amherst*, 2015.

[11] T. Helmuth and L. Spector. General program synthesis benchmark suite. In *GECCO*, 2015.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE*, 2002.

[13] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT press, 1994.

[14] S. Krishna, C. Agarwal, and H. Lakkaraju. Understanding the effects of iterative prompting on truthfulness. *arXiv*, 2024.

[15] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *IEEE/ACM ASE*, pages 14–26, 2023.

[16] D. Lynch, T. Saber, S. Kucera, H. Claussen, and M. O'Neill. Evolutionary learning of link allocation algorithms for 5g heterogeneous wireless communications networks. In *GECCO*, 2019.

[17] W. Ma, S. Liu, W. Wenhan, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu. The scope of chatgpt in software engineering: A thorough investigation. *arXiv*, 2023.

[18] J. Manyika and S. Hsiao. An overview of bard: an early experiment with generative ai. *AI. Google Static Documents*, 2, 2023.

[19] J. F. Miller and S. L. Harding. Cartesian genetic programming. In *GECCO*, 2008.

[20] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao. Large language models: A survey, 2024.

[21] M. O'Neill, M. Nicolau, and A. Agapitos. Experiments in program synthesis with grammatical evolution: A focus on integer sorting. In *IEEE CEC*, 2014.

[22] OpenAI. Gpt-4 technical report, 2023.

[23] M. O'Neill and C. Ryan. Grammatical evolution: Evolutionary automatic programming in a arbitrary language, volume 4 of genetic programming, 2003.

[24] C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *ESE*, 2018.

[25] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill. A hierarchical approach to grammar-guided genetic programming the case of scheduling in heterogeneous networks. In *TPNC*, 2018.

[26] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill. Multi-level grammar genetic programming for scheduling in heterogeneous networks. In *EuroGP*, 2018.

[27] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill. Hierarchical grammar-guided genetic programming techniques for scheduling in heterogeneous networks. In *IEEE CEC*, 2020.

[28] T. Saber, D. Fagan, D. Lynch, S. Kucera, H. Claussen, and M. O'Neill. A multi-level grammar approach to grammar-guided genetic programming: the case of scheduling in heterogeneous networks. *GPEM*, 2019.

[29] T. Saber and S. Wang. Evolving better rerouting surrogate travel costs with grammar-guided genetic programming. In *IEEE CEC*, 2020.

[30] R. Schuster, C. Song, E. Tromer, and V. Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security 21*, pages 1559–1575, 2021.

[31] K. Stechly, M. Marquez, and S. Kambhampati. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. *arXiv*, 2023.

[32] N. Surameery and M. Shakor. Use chat gpt to solve programming bugs. *IJITCE*, pages 17–22, 01 2023.

[33] N. Tao, A. Ventresque, V. Nallur, and T. Saber. Enhancing program synthesis with large language models using many-objective grammar-guided genetic programming. *Algorithms*, 17(7):287, 2024.

[34] N. Tao, A. Ventresque, and T. Saber. Assessing similarity-based grammar-guided genetic programming approaches for program synthesis. In *OLA*. Springer, 2022.

[35] N. Tao, A. Ventresque, and T. Saber. Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis. In *IEEE CEC*, 2022.

[36] N. Tao, A. Ventresque, and T. Saber. Many-objective grammar-guided genetic programming with code similarity measurement for program synthesis. In *IEEE LACCI*, 2023.

[37] N. Tao, A. Ventresque, and T. Saber. Program synthesis with generative pre-trained transformers and grammar-guided genetic programming grammar. In *LA-CCI*, pages 1–6. IEEE, 2023.

[38] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A Saurous, and Y. Kim. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[39] P. A. Whigham. Grammatical bias for evolutionary learning. *PhD Thesis, University College, Australian Defence Force Academy, University of New South Wales, Canberra*, 1997.

[40] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv*, 2023.

[41] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv*, 2023.

[42] Z. Zhang and T. Saber. Assessing the code clone detection capability of large language models. In *2024 4th International Conference on Code Quality (ICCQ)*, pages 75–83. IEEE, 2024.