Article

Enhancing Program Synthesis with Large Language Models Using Many-Objective Grammar-Guided Genetic Programming

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

[1] School of Computer Science, University College Dublin, D04 V1W8 Dublin, Ireland; ning.tao@ucdconnect.ie (N.T.); vivek.nallur@ucd.ie (V.N.)
[2] School of Computer Science and Statistics, Trinity College Dublin, D02 PN40 Dublin, Ireland; anthony.ventresque@tcd.ie
[3] School of Computer Science, University of Galway, H91 TK33 Galway, Ireland
* Correspondence: takfarinas.saber@universityofgalway.ie
† Current address: SFI Lero–the Irish Software Research Centre, V94 NYD3 Limerick, Ireland.

Abstract: The ability to automatically generate code, i.e., program synthesis, is one of the most important applications of artificial intelligence (AI). Currently, two AI techniques are leading the way: large language models (LLMs) and genetic programming (GP) methods—each with its strengths and weaknesses. While LLMs have shown success in program synthesis from a task description, they often struggle to generate the correct code due to ambiguity in task specifications, complex programming syntax, and lack of reliability in the generated code. Furthermore, their generative nature limits their ability to fix erroneous code with iterative LLM prompting. Grammar-guided genetic programming (G3P, i.e., one of the top GP methods) has been shown capable of evolving programs that fit a defined Backus–Naur-form (BNF) grammar based on a set of input/output tests that help guide the search process while ensuring that the generated code does not include calls to untrustworthy libraries or poorly structured snippets. However, G3P still faces issues generating code for complex tasks. A recent study attempting to combine both approaches (G3P and LLMs) by seeding an LLM-generated program into the initial population of the G3P has shown promising results. However, the approach rapidly loses the seeded information over the evolutionary process, which hinders its performance. In this work, we propose combining an LLM (specifically ChatGPT) with a many-objective G3P (MaOG3P) framework in two parts: (i) provide the LLM-generated code as a seed to the evolutionary process following a grammar-mapping phase that creates an avenue for program evolution and error correction; and (ii) leverage many-objective similarity measures towards the LLM-generated code to guide the search process throughout the evolution. The idea behind using the similarity measures is that the LLM-generated code is likely to be close to the correct fitting code. Our approach compels any generated program to adhere to the BNF grammar, ultimately mitigating security risks and improving code quality. Experiments on a well-known and widely used program synthesis dataset show that our approach successfully improves the synthesis of grammar-fitting code for several tasks.

Keywords: program synthesis; large language models; grammar-guided genetic programming; grammar; multi-objective

Citation: Tao, N.; Ventresque, A.; Nallur, V.; Saber, T. Enhancing Program Synthesis with Large Language Models Using Many-Objective Grammar-Guided Genetic Programming. *Algorithms* **2024**, *17*, 287. https://doi.org/10.3390/a17070287

Academic Editor: Massimiliano Caramia

Received: 27 March 2024
Revised: 23 June 2024
Accepted: 24 June 2024
Published: 1 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Program synthesis is the process that aims to simplify or automate programming jobs by providing a toolkit of methods that allow for code generation based on a high-level description of task intents (e.g., a textual task description, input/output examples, sketches). The fusion of AI and program synthesis holds immense promise: it can reduce development time, elevate the quality of programs, and reshape the way we build software.

In recent years, numerous algorithms have been proposed to automate programming in various programming languages and targeting diverse problems, from simple tasks (e.g.,

symbolic regression [1–7], string manipulation [8–11], and binary transformation [12–14]), to more complex challenges (e.g., robot path-finding [6,15], algebraic calculations [14,16–21], and intricate real-world programming problems). Particularly, Saha et al. [22] proposed an algorithm to generate a machine learning pipeline using a corpus and a human-written pipeline. Poliansky et al. [23] utilised genetic programming (GP) and context-oriented behavioural programming (COBP) in the tic-tac-toe game. Beltramelli introduced pix2code [24], which leverages convolutional neural networks (CNNs) to generate web development interface code (HTML/CSS) from screenshots of the graphical user interface. The AlphaCode developer team harnessed large-scale sampling and transformer language models to address previously unsolved competitive programming challenges [25]. Despite the wide range of approaches, GP continues to be a competitive method for addressing program synthesis challenges [26].

GP [27] devises a program using an evolutionary process. It commences with a population of unfit programs and gradually develops them into solutions customised for specific tasks—a process akin to natural genetic evolution. However, GP faces limitations in devising syntactically correct and semantically meaningful code. Grammar-guided genetic programming (G3P [28]) evolves programs that fit a predefined grammar, thus improving the synthesis performance by restricting/confining the search space. Furthermore, having code that obeys a grammar has several positive impacts, as the grammar can limit the structure of the code as well as the set of callable functions/methods/libraries. Therefore, having an impact on (i) security, e.g., using malicious functionalities, blacklisted libraries, vulnerable structures, etc.; (ii) computing environment, e.g., if the hardware is not able to handle some functions or if some functions are too costly in terms of memory or energy; and (iii) code quality, improving readability, reducing code smells, and following design patterns. G3P demonstrated its capability to evolve code in various programming languages to tackle a diverse range of problems. However, the system's reliance on randomly generated populations restricts its efficacy in addressing complicated and extensive scenarios.

Lately, large language models (LLMs) have shown to be successful at different software engineering tasks [29–33], including generating source code from textual problem descriptions. Although LLMs have shown success at program synthesis from a task description, they often struggle to generate correct code due to ambiguity in task specifications and complex programming syntax (e.g., generating known buggy code [34]). Moreover, there is a legitimate lack of trust in the generated code as it comes with documented risks (e.g., generating code with known and risky vulnerabilities [35,36]). The potential for LLM-generated bad code represents a large and growing risk to software and its stakeholders in general. Furthermore, LLMs' generative nature limits their ability to fix erroneous code with iterative prompting [37,38].

Recent research by Tao et al. [18] proposed an approach that combines the strengths of G3P and LLMs to evolve programs that fit predefined grammar. Specifically, they seeded the LLM-generated code into G3P's initial population by mapping the seed to fit the predefined grammar. Their results show that they successfully improved some LLM-generated programs to fit a predefined BNF grammar. However, leveraging LLM-generated code only as seeds for the initial population causes the algorithm to lose the seeded information quickly during the evolution process.

To restrict the search space and avoid programs that include bad coding practices/code smells, use unknown/untrustworthy libraries/functionalities, or do not follow a desired design pattern, we consider the definition of a "safe" grammar and ensure that any generated program obeys such grammar—ultimately mitigating security risks and improving code quality.

While the definition of "*safe*" grammars is a challenge on its own that warrants several research studies (i.e., to design and prove them for different tasks), in our work, we are concerned with the capability to generate correct grammar-fitting programs.

In our work, we make the following contributions:

- We show that while our considered LLM (i.e., ChatGPT) successfully generates correct programs for most tasks in a well-known program synthesis benchmark, it struggles to generate programs that obey the predefined grammars.
- We propose an evolutionary approach, i.e., Many-Objective Grammar-Guided Genetic Programming (MaOG3P), that exploits the LLM-generated program in two parts while guaranteeing that any evolved programs adhere to the predefined grammar:
  - Leveraging many-objective similarity measures towards the LLM-generated code to guide the search process throughout the evolution.
  - Mapping the LLM-generated program to another program that obeys the predefined grammar while maintaining as much information as possible and using it as a seed in the initial population.

Our MaOG3P algorithm considers four different similarity measures alongside the input–output error rate to strategically steer the search process towards more plausible program candidates and guide the evolutionary search towards solutions that exhibit structural similarities to the well-performing LLM-generated code. The idea behind using the similarity measures is that the LLM-generated code is likely to be close to the correct code. While MaOG3P is inspired by [20,21], such approaches only evaluate the ability to leverage similarity measures towards the correct program (obtained using an oracle) to guide the evolutionary process—-thus are not applicable in practice. Furthermore, as seeding is beneficial for evolutionary algorithms (both in single [39] and multi-objective [40] settings) by acting as a search accelerator/catalyser, there are various ways to source "*good*" quality seeds. For instance, Wick et al. [41] provide programs that solve tasks different from the one at hand (which could have previously been generated by a human or another evolutionary process) to the initial population (in addition to programs generated randomly). In our work, we follow a similar seeding principle (i.e., adding a program to the initial population). However, our seeded program is generated by an LLM on the same task, following its grammar mapping (to align with the grammar in the grammar-guided genetic programming).

Evaluating our approach on the general program synthesis benchmark suite 1 [42] demonstrates that MaOG3P effectively evolves the correct grammar-fitting programs for several tasks. Nevertheless, there are still improvements to be made as, in its default configuration, MaOG3P does not reach the success rate of the LLM when ignoring the grammar-fitting constraint.

The remainder of the paper is organised as follows: Section 2 summarises the relevant background and related work. In Section 3, we introduce our novel MaOG3P approach. Section 4 outlines the details of our experimental setup. The results and discussions from our experiments are presented in Section 5. Finally, Section 6 concludes this work and explores avenues for future studies.

2. Background and Related Work

In this section, we present the background and work related to our study in four areas: GP, G3P, LLMs, and program similarity measures.

2.1. Genetic Programming

GP is an evolutionary algorithm that produces computer programs by iteratively evaluating their fitness based on specific tasks. GP aims to create improved programs by evolving a population of individuals. These individuals start as randomly selected candidates, often not well suited for the intended purpose. GP employs genetic operators inspired by natural processes, such as crossover, mutation, and selection. Over time, various GP systems have been proposed, each possessing unique characteristics (e.g., GP [27], Cartesian GP [43], and linear GP [44]).

2.2. Grammar-Guided Genetic Programming

While numerous GP systems exist, G3P demonstrates outstanding efficacy. G3P, a variant of GP, utilises grammar as its representation. Notably, two well-known variants within this framework are grammatical evolution [45] and context-free grammar genetic programming (CFG-GP) by Whigham [46]. What sets G3P apart is its reliance on grammar as a guiding principle throughout program evolution, ensuring syntactically valid programs at each stage. Grammars offer adaptability and the ability to define search spaces externally to the GP system. Consequently, they find applications in diverse domains, including automated programming [47], transport system management [48], and wireless communications scheduling [49–53]. Forstenlechner et al. [28] proposed a G3P system featuring a composite and self-adaptive grammar to address synthesis problems. This innovation overcomes the limitation of customising or adjusting grammars for each problem. The approach involves predefined short grammars, each associated with a specific data type that describes the function or program to be evolved. By reusing these grammars across diverse problems, G3P reduces the computation costs while excluding data types that are irrelevant to the problem. Furthermore, subsequent enhancements in [54] extend the predefined grammar to include data types for characters instead of handling them as strings. It also added recursions to the grammar, enabling diverse program structures in the output program.

2.3. Large Language Models

LLMs are AI algorithms utilising deep learning technologies. These models can process and understand human languages using neural networks with large amounts of parameters. The concept underlying LLMs involves predicting the subsequent word in a sequence based on context. The widespread adoption of LLMs gained momentum around 2018 [55]. Since then, they have demonstrated impressive performance across a diverse range of tasks [29–33]. LLMs serve as the foundation for chatbots like OpenAI's ChatGPT [56] and Google's Gemini [57].

ChatGPT [56] is an AI-driven natural language processing tool developed by OpenAI. It can generate a human-like response based on the user prompt. Beyond answering users' questions, it can complete complex tasks like programming, composing essays, and editing emails. Due to its extensive training on a diverse text corpus, ChatGPT can generate content in various styles and formats.

To improve the programming performance of LLMs, Wang et al. [58] proposed grammar prompting to leverage BNF grammar as external knowledge and domain-specific constraints. They provided minimally sufficient grammar that can generate solutions for the input–output examples. For a given new input, LLM first predicts the grammar, and then, generates an answer according to the predicted grammar. In our work, we would like to use LLMs to generate code that obeys a predefined grammar instead of using LLMs to generate a grammar that can be used to generate better programs.

2.4. Approaches for Detecting Program Similarity

The assessment of code similarity serves multiple purposes, including the identification of repetitive code, plagiarism detection, and discovering similar bug fixes [59]. This research has selected the top four similarity metrics identified by Ragkhitwetsagul et al. [60] to evaluate the fitness in program synthesis within the MaOG3P framework.

2.4.1. FuzzyWuzzy

FuzzyWuzzy [61], an open-source Python library, is designed for string searching and constructed on top of the difflib library. Within its functionalities, FuzzyWuzzy offers various similarity functions, including the *TokenSortRatio* and *TokenSetRatio*. Interestingly, researchers discovered that this string-searching algorithm performs well in detecting code similarity [60]. The *TokenSortRatio* function tokenises the input string by removing punctuation, converting capitals to lowercase, and then, sorting the tokens alphabetically.

The resulting sorted tokens are concatenated to generate the similarity score. In contrast, the *TokenSetRatio* function removes common tokens without sorting them.

2.4.2. Cosine

In data science, the cosine similarity calculates the similarity between two vectors. It can be applied to calculate the similarity between two programs with tokenisation. The cosine similarity approach for program similarity calculation can be outlined through the following steps:

- Preprocessing and tokenisation: During this phase, we eliminate extraneous structural elements and tokenise the source code.
- Construct frequency vector: We compute the token frequencies and store them in a vector representing term frequencies.
- Similarity score calculation: As shown in Equation (1), we calculate the similarity score between two source programs by applying the cosine similarity function between two frequency vectors (denoted as vectors **A** and **B**) from the previous step.

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

2.4.3. CCFinder

Kamiya et al. [62] proposed CCFinder, a technique that identifies code clones within source code by leveraging a token-based approach. This approach identifies code clones using the following main steps: (i) Lexical analysis: this step creates token sequences from the input source code files by applying language-specific lexical rules. (ii) Transformation: the system standardises the program's structure by applying transformation rules to the token sequences. This standardisation enables the detection of code clones, even in code that displays diverse expressions. (iii) Clone matching: the suffix-tree matching algorithm is utilised in this step to determine the code clones. (iv) Formatting: this final step provides detailed information for each clone pair.

The CCFinder tool, initially intended for large-scale programs, has undergone adjustments and simplifications in light of the straightforward code structures used during our evaluation. The following modifications were made to enhance its suitability for our context.

- Given our goal of generating a similarity score for two code snippets, we divide the length of the cloned code by the length of the source code:

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

  where $Clone(a, b)$ represents the longest code clone between the code snippets a and b, while $Len(a)$ denotes the length of the code snippet a in terms of the number of characters.
- Instead of employing the suffix-tree matching algorithm, we calculate the length of the common tokens between two code snippets in a matrix form. Each token sequence corresponds to a dimension in this matrix.
- We also removed the reporting step of code clones. This decision was made because reporting the line number is no longer essential for our purposes.

2.4.4. SIM

SIM [63], a plagiarism detection tool, specifically caters to assignment plagiarism in programming courses. It leverages string alignment techniques on the token level to assess the structural similarity between two C programs. In this way, it identifies plagiarism with minor local modifications.

The detection algorithm involves the following essential parts: (i) generating tokens, and (ii) calculating similarity scores using alignment technology. A token sequence is extracted from the source program by applying lexical analysis. These tokens represent the fundamental building blocks of the code, such as keywords, identifiers, literals, and operators. The tokenisation process ensures that the code is broken down into meaningful units, making it easier to compare and analyse. The alignment starts by dividing the second token sequence into multiple sections. Each section is aligned with the first token sequence to get the similarity score. The advantage of such a technique is that it detects plagiarism by altering function order.

3. Proposed Approach

In this research, we aim to tackle program synthesis problems by generating syntactically correct programs that fit a BNF grammar, limiting the structure of the code as well as the set of callable functions/methods/libraries.

Our proposed system (i) prompts an arbitrary LLM (in our case, ChatGPT 3.5) to generate code based on a task description; (ii) maps the LLM-generated code to another program that adheres to a predefined BNF grammar; (iii) feeds the mapped program as a seed to our MaOG3P evolutionary process; then (iv) performs the MaOG3P evolutionary process, which uses similarity measures towards the LLM-generated code as a secondary objective to guide the search process. The data and algorithm used in this research, as well as the tools to run the experiment, are available online (https://github.com/TonBatbaatar/MaOG3P, accessed on 23 June 2024).

The overview of our proposed approach is shown in Figure 1. Our approach prompts an LLM (i.e., ChatGPT) to generate code that fulfils the task depicted in the textual description before mapping it into a predefined grammar. Our approach also extends the G3P system by (i) seeding the resulting grammar-mapped LLM-generated code into the initial population of the MaOG3P algorithm, and (ii) utilising the error rate (based on a given test suite) as the primary objective, alongside various code similarity measures as secondary objectives, to guide the search process.

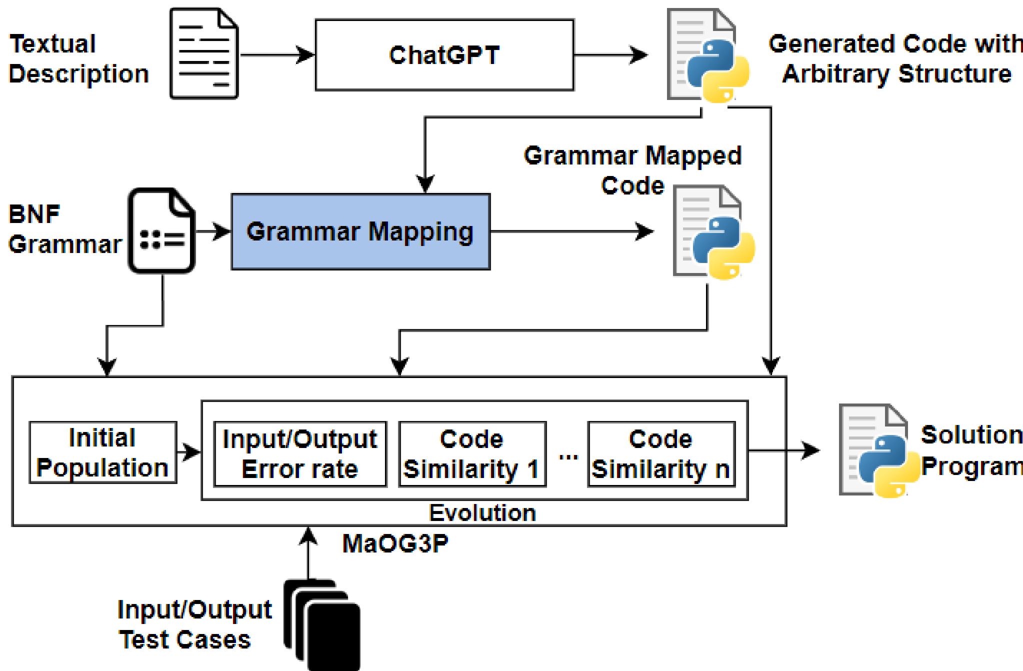

Figure 1. Overview of our MaOG3P system.

3.1. LLM Code Generation

LLMs exhibit remarkable capabilities in code generation based on prompts, which are user-provided inputs. In our initial research phase, we employed ChatGPT 3.5 to create code snippets by supplying textual task descriptions. We observed a high success rate in generating code for the considered benchmark problems and decided to query the LLM once for each problem with default temperature.

A recent study emphasises the significant influence of input prompt quality on the effectiveness of LLMs [64,65]. To ensure the quality of the output code, we structure the task description as a prompt template, as illustrated below.

Prompt Template

Main task: Generate a Python function to solve the task described below.
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).

In *Main task*, we outline the general goal of the query. Subsequently, in the *Task description*, we provide a detailed description of each task, and in *Output program format*, we add additional structural information to format the output program for our experiment.

3.2. Grammar-Mapping of LLM-Generated Code

In G3P, all programs in the population must adhere to the same grammar rules for genetic operators, such as crossover, mutation, and selection. Hence, without fitting it to the predefined grammar, we cannot seed the LLM-generated code into the G3P system.

Initially, we attempted to augment the prompt template with additional grammar information (as shown below), aiming to guide LLMs into generating code that adheres to the same grammar as the randomly generated population.

```
Main task: <Generate a Python function...>
           The function should obey the provided BNF grammar.
Task description: {input_task_description}
Output program format: <Function parameter ...>
Output program grammar: {input_task_BNFgrammar}
```

Unfortunately, specifying the grammar in the LLM prompt decreased the success rate of generating correct programs (the LLM tends to generate code that does not address the problem description) and often includes code snippets outside the defined grammar. In response, we developed a grammar-mapping algorithm to transform programs into a format that adheres to a predefined grammar file. In our evolution, we used the automatic grammar file selection approach proposed in the previous work by Forstenlechner et al. [28,54].

In the context of the mapping algorithm, LLMs generated code conflicts with the predefined grammar, primarily in three different aspects. First, certain functions used in the source code lacked support within the BNF grammar, or their function parameters did not align with those specified by the grammar. For example, in most programming languages, functions have multiple optional parameters with default values, while in G3P, the predefined grammar only keeps core parameters (that can be evolved with the genetic algorithm) for certain functions. Second, while the available variable names are fixed in the grammar file, the naming scheme for variables in the source code is more flexible. Third, the structural composition of the source code did not conform to the patterns established by the target grammar. For example, there were often short expressions for certain complex programming structures, while the BNF grammar in G3P does not support these short expressions.

The grammar-mapping strategy is described in Algorithm 1. Specifically, we generate an abstract syntax tree (AST) T of the provided program p. An AST, as described by

Baxter [66], is a tree-like data structure to represent the hierarchical structure of a program or code snippet. This representation can reconstruct code functionally equivalent to the original, regardless of the target language. We map the AST T of our provided program by having its root node R iterate through its child nodes and constructing the output program q recursively using *Node_Iterator*.

Algorithm 1: Grammar Mapping Algorithm.

Data: Program p from arbitrary source, Grammar G
Result: Grammar-mapped program q
Build Abstract Syntax Tree T from p;
$R \leftarrow$ Get Root of the T
$q \leftarrow$ Node_Iterator(R, G)
return q

A general structure of an AST is shown in Figure 2 using an example Python program "a = 1; b = a + 2". It contains a root node "*Module*", indicating the entry of a program, and a series of non-terminal and terminal nodes. For example, the "*Assign*" node represents an assign expression in Python with its attributes describing its structure, whereas the "*Name*" node represents a variable. Non-terminal nodes (node "*Assign*" in the example) also have child nodes, each with a sub-tree representing its sub-elements.

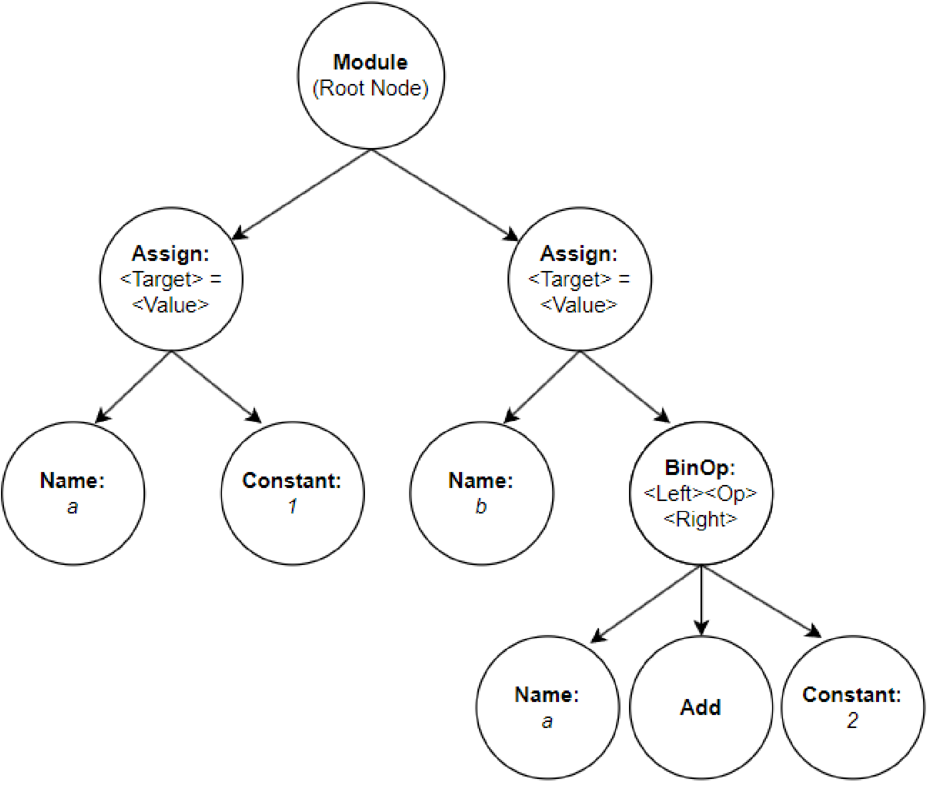

Figure 2. AST of an example program "a = 1; b = a + 2".

Algorithm 2 details the recursion process. We begin by iterating through each child node n of the given node R. For each child node n, the behaviour changes depending on whether it is a terminal or a non-terminal node.

We examine whether the value and parameters conflict with the grammar G for non-terminal nodes. A grammar conflict normally occurs when a certain function in the given program is not supported in G or does not have the specified number/type of parameters. When there is no grammar conflict, we construct the output program q using the value of n in a top-down manner and continue with recursively calling *Node_Iterator* on node n. We

build q with a dummy expression if there is a grammar conflict. Specifically, we replace the whole expression with an *assign expression* that assigns a variable to itself, which can be further improved by the genetic programming process during evolution.

Algorithm 2: Node_Iterator.

```
Data: Abstract Syntax Tree node R, Grammar G
Result: Grammar-mapped program q
for node n ∈ Children(R) do
    if n is non-terminal node then
        Check value and parameters of non-terminal node n for grammar conflict
         in G
        if no grammar conflict then
            Construct q with the value of non-terminal node n
            Node_Iterator(n, G)
        else
            Build q with a dummy expression
        end
    else
        if n is a variable then
            Map variable name in G
        else if n is a constant then
            Map constant to closest constant in G
        else
            Map operator name in G
        end
    end
end
return q
```

For terminal nodes, we construct q based on the node type. When the node is a variable "name", we check if it has previously been mapped, in which case, we map it to the same one. If the variable "name" has not been previously mapped, we map it in G using the first unused variable name with the same data type. For the "constant" type, we map it to the closest constant in G. For the remaining types (i.e., "operators"), we map the operator in G.

3.3. Many-Objective G3P with Seeding

As the likelihood of having a correct program as a result of the mapping phase is low, we devised a many-objective G3P approach (i.e., MaOG3P) to evolve the grammar-mapped program while maintaining its similarity with the LLM-generated code.

Figure 3 shows the overview of MaOG3P. We seed the grammar-mapped program into MaOG3P's initial population (generated using the selected grammar associated with the relevant data type for the given problem). The seeding process can initialise the evolution with a starting point that is closer to the solution of the task, potentially enhancing the algorithm's search process. Furthermore, expanding on the input–output error rate fitness evaluation, our approach incorporates an additional layer of sophistication by integrating code similarity measures towards the (unmapped) LLM-generated code as secondary objectives to guide the evolution process.

We considered four different similarity measures, described in Section 2, for our system. However, our innovation 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.

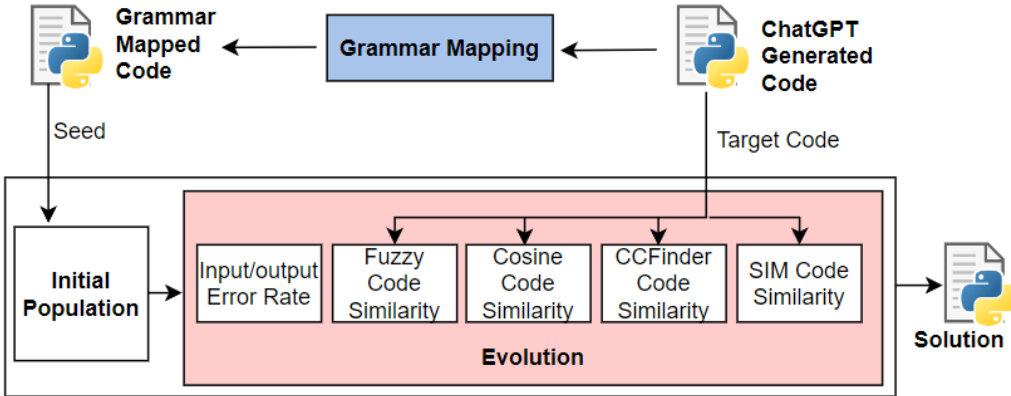

Figure 3. Overview of the MaOG3P.

While MaOG3P employs various objectives to evolve the population programs during the evolution process, we consider that an individual in the population solved the task only based on the primary objective. In other words, it must accurately pass all test cases for the given task.

In our study, we modified the tournament selector to accommodate multiple objectives, enabling the algorithm to evolve programs with different criteria. When selecting the parents for the next generation, we perform the same as the standard G3P algorithm for half of the individuals (select parents using input/output error rate). For the other half of the population, we select one with the primary objective (error rate) and the other with a secondary objective (code similarity). Considering we have four different similarity measures as secondary objectives, we iterate with four measures evenly for selecting the parents.

Determining which individuals can survive in evolution is a critical step for our algorithm. Similarly to the mechanism for selecting parents, we consider all objectives in this step. In this scenario, the primary objective decides which individual can survive for half of the population, while four similarity measures decide the remaining half.

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 is ChatGPT at program synthesis?
- RQ2: Could we fit ChatGPT-generated code to predefined grammars?
- RQ3: Could we improve the performance of ChatGPT at synthesising programs that fit predefined grammars using MaOG3P?

4.2. Benchmark Dataset

In our experiment, we use a well-known and widely used benchmark dataset for program synthesis created by Helmuth and Spector [42,67]. This benchmark suite contains 29 problems selected from introductory-level programming courses. Each problem in the benchmark comes with a detailed natural language problem description as well as training/testing input/output test sets. Table 1 shows four problems with their respective task descriptions. Table 2 indicates the number of train and test cases for each problem. In our experiments, we evaluated our model and ChatGPT with 28 problems. We have excluded the problem "String Differences" from our experiment as in previous work by Forstenlechner [28]. 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 our experiment, we use the same grammar as the one

defined by Forstenlechner et al. [28] (available at: https://github.com/t-h-e/HeuristicLab.CFGGP/tree/master/HeuristicLab.Problems.Instances.CFG/GrammarConstruction, accessed on 23 June 2024).

Note that, as we are using LLMs, there is a risk that our benchmark dataset has already leaked to the ChatGPT training data. However, this is not an issue in our case as, even if the problems and their respective code have been leaked, the used grammars are unlikely to be leaked—particularly since we notice in our experiments that most of the LLM-generated codes do not fit the predefined grammars. Therefore, while the generated code is often interesting, it is not correct from a grammar point of view.

Table 1. Description of four example problems in the dataset.

Problem Name	Description
Even Squares	Given an integer n, print all of the positive even perfect squares less than n on separate lines.
Last Index of Zero	Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.
Scrabble Score	Given a string of visible ASCII characters, return the Scrabble score for that string. Each letter has a corresponding value according to normal Scrabble rules, and non-letter characters are worth zero.
Wallis Pi	John Wallis gave the following infinite product that converges to $\pi/4$: (2/3) * (4/3) * (4/5) * (6/5) * (6/7) * (8/7) * (8/9) * (10/9) * . . . Given an integer input n, compute an approximation of this product out to n terms. Results are rounded to 5 decimal places.

Table 2. Number of train and test cases for each problem in the dataset.

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

4.3. Generating Programs Using ChatGPT

The fitness value of each program is computed by transforming it into a Python function. This function is then executed with all test inputs, and the resulting output values are compared to the expected outputs. However, the generated code from ChatGPT tends to be snippets (rather than complete functions) when we provide original problem descriptions as prompts. It includes `print` statements instead of reporting the result as a return value of a function. To address this challenge, we replaced the keyword "print" with "return" in the task descriptions. Specifically, we requested ChatGPT to devise a function instead of a code snippet, ensuring that we can seamlessly evaluate the code with the function's return value.

4.4. Parameter Settings

We use ChatGPT 3.5 with the default temperature and use the standard G3P parameter settings as used in previous studies [28]. 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 MaOG3P systems are indicated in Table 3.

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

[a] Using 200 generations for "Median", "Number IO", and "Smallest" as in [42].

5. Results

5.1. Effectiveness of ChatGPT at Program Synthesis (RQ1)

In this subsection, we contrast the performance of ChatGPT against G3P with tournament selection. Table 4 compares the program generation capabilities between ChatGPT and G3P in our benchmark problems. For G3P, we denote the problem with a checkmark (✔) when the evolution process successfully solves the task at least once across a hundred runs. In contrast, we use a symbol cross (✖) where the evolution fails to solve the task across a hundred runs. For ChatGPT, the checkmark (✔) indicates a correct program generated using the predefined grammar, the circle checkmark (✅) notes the problem was solved violating the predefined grammar, while a cross symbol (✖) means it failed to generate a correct program.

ChatGPT exhibited impressive capabilities in tackling program synthesis tasks, finding solutions for 26 out of the 28 considered problems. For the two failed tasks, it produced programs that closely approached the solution of the task. In particular, for one of the unsuccessful tasks, namely, "Wallis Pi", ChatGPT failed to understand the natural language prompt fully. The task requested the result for $\frac{\pi}{4}$, but ChatGPT provided a solution for calculating the value of $\frac{\pi}{2}$. For another problem, named "Digits", the task asked for a program to split the digits of a given number. ChatGPT encountered problems when handling negative numbers. However, we do not know if the problems in our dataset have already leaked into the ChatGPT training data. Considering that most generated programs do not obey the defined grammar, we can say that the grammar has not leaked. Therefore, while the generated code is often interesting, it is not correct from a grammar point of view. Detailed programs obtained by our prompting of ChatGPT are available online (available at: https://github.com/TonBatbaatar/MaOG3P/tree/main/HeuristicLab.Algorithms.CFG.MultiObjective/PSB1_Solution/json/ChatGPT, accessed on 23 June 2024).

Table 4. Comparison of ChatGPT and G3P on benchmark problems.

Benchmark Problem	G3P	ChatGPT
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 [a]

[a] Total of 24 problems solved with violating the predefined grammar.

5.2. Fitting ChatGPT-Generated Code to Predefined Grammars (RQ2)

Our proposed grammar-mapping algorithm successfully mapped all the code snippets produced by ChatGPT into programs conforming to the MaOG3P grammar. However, in certain cases, some structural details were ignored in the mapped program due to significant grammar conflicts. Based on the amount of information we successfully mapped, we categorised the mapping status into three distinct levels, as shown in Table 5. When the algorithm can map LLM-generated code without grammar conflict, we note it as "completely mapped" (denoted as ✔). We consider the mapping status as "partially mapped" (denoted as (✔)) when most information is successfully mapped in a code that fits the predefined grammar. Otherwise, we note the mapping status as "poorly mapped" (denoted as ✖), which indicates the grammar-mapped code contains minimal information from the source code. For 13 problems, the LLM-generated code was successfully mapped to programs that adhere to the BNF grammar without any loss of information (i.e., completely mapped). The LLM-generated code was mapped to nearly solved programs in four cases, demonstrating close alignment with the desired solutions (i.e., partially mapped). Unfortunately, for 10 problems, large grammar conflicts prevented successful mapping. These cases are categorised as poorly mapped.

Table 5. Status of the grammar-mapping process for LLM-generated code.

Problem	Grammar Mapping Status	Problem	Grammar Mapping Status
NumberIO	✔	Last Index of Zero	✅
Small Or Large	✔	Vector Average	✖
For Loop Index	✔	Count Odds	✔
Compare String Lengths	✔	Mirror Image	✔
Double Letters	✔	Super Anagrams	✖
Collatz Numbers	✔	Sum of Squares	✖
Replace Space with Newline	✅	Vectors Summed	✖
Even Squares	✅	X-Word Lines	✖
Wallis Pi	✔	Pig Latin	✖
String Lengths Backwards	✖	Negative To Zero	✖
Digits	✅	Scrabble Score	✖
Grade	✔	Word Stats	✖
Median	✔	Checksum	✖
Smallest	✔	Syllables	✔

5.2.1. Partially Mapped Code Analysis

In this subsection, we analyse the four grammar-mapped programs categorised as "partially mapped" from Table 5 (i.e., "Replace Space with Newline", "Even Squares", "Last Index of Zero", and "Digits"). Our analysis examines the portions of code that could not be accurately mapped.

- Replace Space with Newline: This task involves counting the number of newline characters in a given string. LLM-generated code counts the number of newlines by using an advanced Python grammar "*List Comprehension*". However, this particular grammar rule is not included in the predefined BNF grammar. Consequently, the grammar-mapping algorithm replaced it with a dummy expression.
- Even Squares: In this task, ChatGPT used a `range` function with three parameters to iterate through the loop. However, in our grammar, the `range` function contains only two parameters. Therefore, when mapping the `range` function, the algorithm ignored the extra parameter (i.e., the `step` parameter). Another grammar conflict occurred when mapping the `is_integer` function because the predefined grammar does not support such a function. This type of grammar conflict is replaced with a dummy expression for further evolutionary improvements.
- Last Index of Zero: The same parameter conflict for mapping the `range` function occurred in this task. In addition to this similar conflict, ChatGPT used the `break` statement to terminate the loop while the `break` statement is not defined in the BNF grammar.
- Digits: ChatGPT solved this task using multiple `return` statements with different outputs, while the MaOG3P grammar only allows the program to use the `return` statement once to return the output variable. The mapping algorithm handled this conflict by replacing the extra `return` statements with dummy expressions.

5.2.2. Poorly Mapped Programs Analysis

In this subsection, we analysed ten "poorly mapped" problems in four categories based on the type of grammar conflicts.

- "*String Lengths Backwards*", "*Sum of Squares*", *Vectors Summed*", and "*Negative To Zero*": ChatGPT solved these problems with one line "*list comprehension*" functions while such expressions are not defined in our grammar.

- “*Checksum*”, *Vector Average*”: When mapping the program for these tasks, significant grammatical conflicts adversely affected the mapping performance.
- “*X-Word Lines*”, *Pig Latin*”, *Word Stats*”: ChatGPT used the `split` function to solve these string manipulation tasks. However, our BNF grammar defined a `split` function integrated with a loop that cannot be used separately. This raised a grammar conflict that the mapping algorithm could not address. ChatGPT also builds the output string for the task by using the `join` function, which is not defined in our grammar.
- “*Super Anagrams*”, *Scrabble Score*”: The choice of data structure is limited in our BNF grammar. ChatGPT used a `dict` to solve these problems while the `dict` data structure is not defined in the grammar.

5.3. Performance of Our Proposed Approach (RQ3)

We evaluated our proposed approach by comparing the performance of its components when taken separately. We particularly compare our approach against ChatGPT, G3P, G3P with a seeded grammar-mapped ChatGPT-generated code, and MaOG3P using ChatGPT-generated code as the target code for similarity calculations. In our experiments, each system underwent 30 runs on every problem. We report the number of successful runs in Table 6. A run was considered successful if it found at least one correct solution that passed all training and test cases.

Table 6. Performance of ChatGPT, G3P, seeded G3P, MaOG3P without seeding, and MaOG3P with seeding, measured by number of times out of 30 runs a correct program is found. For ChatGPT, we consider it solved the problem if it generated the correct solution by obeying the predefined grammar. When ChatGPT found the solution but violated the grammar, we noted it as partially solved (✅). When It generated an incorrect solution, we noted it as failed (✖).

Benchmark Problem	ChatGPT	G3P	G3P ChatGPT Seeding	MaOG3P No Seeding	MaOG3P ChatGPT Seeding
NumberIO	✔	17	30	20	30
Small Or Large	✅	0	30	0	30
For Loop Index	✅	0	30	0	30
Compare String Lengths	✅	0	30	0	30
Double Letters	✅	0	30	0	30
Collatz Numbers	✅	0	30	0	30
Replace Space with Newline	✅	1	10	1	12
Even Squares	✅	0	0	0	0
Wallis Pi	✖	0	0	0	0
String Lengths Backwards	✅	3	1	2	4
Last Index of Zero	✅	6	30	5	30
Vector Average	✅	0	0	0	0
Count Odds	✔	0	30	0	30
Mirror Image	✅	20	30	21	30
Super Anagrams	✅	0	0	0	0
Sum of Squares	✅	0	0	0	0
Vectors Summed	✅	0	0	0	0
X-Word Lines	✅	0	0	0	0
Pig Latin	✅	0	0	0	0
Negative To Zero	✅	1	2	0	3
Scrabble Score	✅	0	0	0	0
Word Stats	✅	0	0	0	0
Checksum	✅	0	0	0	0
Digits	✖	0	0	0	0
Grade	✅	0	30	0	30
Median	✅	12	30	14	30
Smallest	✅	28	30	30	30
Syllables	✅	0	30	0	30
Number of Problems Solved	2	8	16	7	16

Overall, our proposed approach outperformed all the compared approaches. Specifically, our MaOG3P system outperformed ChatGPT, the G3P system, and the MaOG3P system using ChatGPT-generated code as a target code for similarity calculation in terms of the number of solved tasks using the predefined grammar in the benchmark suite. Moreover, our approach significantly improved the success rate of solved problems. Compared to G3P with seeded grammar-mapped programs, our proposed MaOG3P algorithm demonstrated better success rates across all problems.

While our approach may not surpass ChatGPT in generating correct code for unspecified grammars, it excels in refining LLM-generated programs to adhere to predefined grammars, having fixed a total of 14 programs that were violating the predefined grammar. However, it failed to address two unsolved problems with ChatGPT in its default configuration. By seeding a program resembling the solution code, our proposed approach doubled the number of problems solved compared to the G3P algorithm. This highlights the significant enhancement in the search process facilitated by the seeding approach. Furthermore, employing multi-objective optimisation with MaOG3P improved the success rate compared to G3P with seeding, demonstrating the efficacy of similarity measures in guiding the search process towards the correct solution.

We measure if there is a significant improvement in performance by calculating the p-value of the Wilcoxon rank-sum test on the best test fitness value (error-rated fitness value for MaOG3P) of each run from G3P and the proposed approach comparison experiment. The results are shown in Table 7. We use 0.05 as the significance level and highlight the value in bold for significant improvements. Overall, our proposed approach significantly improved the performance regarding the best fitness value of each run for 15 problems. We significantly improved all problems that were seeded with "completely mapped" code, except for the problem "Smallest" as G3P performed well in solving this easy problem. For problems seeded with "partially mapped" code, we achieved significantly better fitness values for all problems. Interestingly, we achieved significantly better fitness values for the problems "Even Squares" and "Digits", even though we did not evolve the correct solution. We did not achieve significantly better fitness value for any of the problems seeded with "poorly mapped" programs. Therefore, a large effort needs to be made in the future to devise advanced and more effective grammar-mapping techniques.

Table 7. The p-value for the Wilcoxon rank-sum test using best fitness of each run comparing G3P and proposed MaOG3P.

Problem	p-Value	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	0.739364
Replace Space with Newline	**0.001609**	Vectors Summed	0.393048
Even Squares	**0.040793**	X-Word Lines	0.795936
Wallis Pi	0.357296	Pig Latin	0.325468
String Lengths Backwards	**0.037062**	Negative To Zero	0.265681
Digits	**0.000064**	Scrabble Score	0.421242
Grade	**0.000064**	Word Stats	0.279861
Median	**0.005943**	Checksum	0.545199
Smallest	0.168078	Syllables	**0.000064**

A significant different is highlighted in bold.

Next, we examine the ability of the proposed approach to improve incomplete seeded codes. In two specific tasks, namely, “Replace Space with Newline” and “Last Index of Zero”, where partially mapped code was initially seeded to the MaOG3P system, our approach successfully rectified the code and demonstrated an enhanced success rate. Nevertheless, for the rest of the problems in the “partially mapped” category (i.e., “Even Square” and “Digits”), MaOG3P struggled to rectify incomplete code despite its proximity to the correct solution. Still, our approach successfully fixed the code for problems “String Lengths Backwards” and “Negative To Zero”, where the seeded grammar-mapped code contained minimal information from the original ChatGPT-generated code.

6. Conclusions and Future Work

In this paper, we proposed combining LLMs (specifically ChatGPT) with MaOG3P to tackle program synthesis tasks by generating programs that are not only correct but fit a predefined grammar with a limited code structure and callable functions/methods/libraries, with a large potential to reduce security threats and improve code quality.

We utilise the programs generated by ChatGPT as seeds in MaOG3P’s initial population following a grammar-mapping process. Furthermore, ChatGPT-generated programs are also used in MaOG3P’s calculation of secondary objectives by serving as a target code for code similarity assessments.

We conducted an in-depth experimental evaluation using a program synthesis benchmark. The experimental analysis showed that our approach enhanced the program synthesis ability of G3P for several problems, while alleviating the shortcomings and risks associated with LLM-generated code. Our approach facilitates the integration of diverse system strengths with evolutionary methods, allowing adaptation to a broader spectrum of problems across various programming languages and grammatical structures.

In our future research, we intend to enhance the grammar-mapping algorithm to accommodate a broader range of data structures, thereby maximising the utilisation of grammar-mapped code within the evolutionary process. We also endeavour to study the quality of the code generated by our approach in future work. Moreover, we plan to investigate the effect of using LLMs with different accuracy levels on the efficiency of our MaOG3P, particularly open-source LLMs that have been fine-tuned for code generation tasks, as well as investigating the performance of different multiple-objective optimisation algorithms as part of MaOG3P.

Author Contributions: Conceptualization: N.T. and T.S.; Methodology: N.T., A.V., V.N., and T.S.; Software: N.T.; Validation: N.T., A.V., V.N., and T.S.; Formal Analysis: N.T., A.V., V.N., and T.S.; Investigation: N.T. and T.S.; Resources: N.T.; Data Curation: N.T.; Writing—original draft preparation: N.T. and T.S.; Writing—review and editing: N.T. and T.S.; Methodology: N.T., A.V., V.N., and T.S.; Visualization: N.T.; Supervision: A.V., V.N., and T.S.; Project Administration: A.V., V.N., and T.S. All authors have read and agreed to the published version of the manuscript.

Funding: Partially supported by Science Foundation Ireland grant 13/RC/2094_P2 to Lero.

Data Availability Statement: The original data presented in the study are openly available in GitHub at https://github.com/TonBatbaatar/MaOG3P (accessed on 23 June 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hara, A.; Kushida, J.I.; Tanabe, S.; Takahama, T. Parallel Ant Programming using genetic operators. In Proceedings of the IEEE IWCIA, Hiroshima, Japan, 13 July 2013; pp. 75–80.
2. Masood, N.; Seyed, A.M.; Emadaldin, M.; Amir, H.G. Introduction of ABCEP as an automatic programming method. *Inf. Sci.* **2021**, *545*, 575–594.
3. Hosseini Amini, S.M.H.; Abdollahi, M.; Amir Haeri, M. Rule-centred genetic programming (RCGP): An imperialist competitive approach. *Appl. Intell.* **2020**, *50*, 2589–2609. [CrossRef]
4. Kim, H.T.; Kang, H.K.; Ahn, C.W. A conditional dependency based probabilistic model building grammatical evolution. *IEICE Trans. Inf. Syst.* **2016**, *99*, 1937–1940. [CrossRef]

5. Mahanipour, A.; Nezamabadi-Pour, H. GSP: An automatic programming technique with gravitational search algorithm. *Appl. Intell.* **2019**, *49*, 1502–1516. [CrossRef]
6. Lopes, R.L.; Costa, E. GEARNet: Grammatical Evolution with Artificial Regulatory Networks. In Proceedings of the GECCO, Amsterdam, The Netherlands, 6–10 July 2013; pp. 973–980.
7. Bowers, M.; Olausson, T.X.; Wong, L.; Grand, G.; Tenenbaum, J.B.; Ellis, K.; Solar-Lezama, A. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Languages* **2023**, *7*, 41. [CrossRef]
8. Lee, W.; Heo, K.; Alur, R.; Naik, M. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In Proceedings of the PLDI, Philadelphia, PA, USA, 18–22 June 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 436–449.
9. Ameen, S.; Lelis, L.H. Program synthesis with best-first bottom-up search. *J. Artif. Intell. Res.* **2023**, *77*, 1275–1310. [CrossRef]
10. Guria, S.N.; Foster, J.S.; Van Horn, D. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Languages* **2023**, *7*, 171. [CrossRef]
11. Yuan, Y.; Banzhaf, W. Iterative genetic improvement: Scaling stochastic program synthesis. *Artif. Intell.* **2023**, *322*, 103962. [CrossRef]
12. Miltner, A.; Fisher, K.; Pierce, B.C.; Walker, D.; Zdancewic, S. Synthesizing Bijective Lenses. *Proc. ACM Program. Languages* **2017**, *2*, 1. [CrossRef]
13. Valizadeh, M.; Berger, M. Search-Based Regular Expression Inference on a GPU. *Proc. ACM Program. Languages* **2023**, *7*, 160. [CrossRef]
14. Helmuth, T.; Frazier, J.G.; Shi, Y.; Abdelrehim, A.F. Human-Driven Genetic Programming for Program Synthesis: A Prototype. In Proceedings of the GECCO, Lisbon, Portugal, 15–19 July 2023; pp. 1981–1989.
15. Cropper, A.; Dumancic, S. Learning large logic programs by going beyond entailment. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20), Yokohama, Japan, 7–15 January 2021; pp. 2073–2079.
16. Arcuri, A.; Yao, X. Co-evolutionary automatic programming for software development. *Inf. Sci.* **2014**, *259*, 412–432. [CrossRef]
17. Botelho Guerra, H.; Ferreira, J.A.F.; Costa Seco, J.A. Hoogle: Constants and λ-abstractions in Petri-net-based Synthesis using Symbolic Execution. In Proceedings of the ECOOP, Seattle, WA, USA, 17–21 July 2023; Volume 263, pp. 4:1–4:28.
18. Tao, N.; Ventresque, A.; Saber, T. Program synthesis with generative pre-trained transformers and grammar-guided genetic programming grammar. In Proceedings of the LA-CCI, Recife-Pe, Brazil, 29 October–1 November 2023; pp. 1–6.
19. Tao, N.; Ventresque, A.; Saber, T. Assessing similarity-based grammar-guided genetic programming approaches for program synthesis. In Proceedings of the OLA, Sicilia, Italy, 18–20 July 2022; Springer: Cham, Switzerland, 2022.
20. Tao, N.; Ventresque, A.; Saber, T. Many-objective Grammar-guided Genetic Programming with Code Similarity Measurement for Program Synthesis. In Proceedings of the IEEE LA-CCI, Recife-Pe, Brazil, 29 October–1 November 2023.
21. Tao, N.; Ventresque, A.; Saber, T. Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis. In Proceedings of the IEEE CEC, Padua, Italy, 18–23 July 2022.
22. Saha, R.K.; Ura, A.; Mahajan, S.; Zhu, C.; Li, L.; Hu, Y.; Yoshida, H.; Khurshid, S.; Prasad, M.R. SapientML: Synthesizing Machine Learning Pipelines by Learning from Human-Writen Solutions. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; pp. 1932–1944.
23. Poliansky, R.; Sipper, M.; Elyasaf, A. From Requirements to Source Code: Evolution of Behavioral Programs. *Appl. Sci.* **2022**, *12*, 1587. [CrossRef]
24. Beltramelli, T. pix2code: Generating code from a graphical user interface screenshot. In Proceedings of the ACM SIGCHI, Paris, France, 19–22 June 2018.
25. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-Level Code Generation with AlphaCode, 2022. Available online: https://doi.org/10.1126/science.abq1158 (accessed on 27 March 2024)
26. Sobania, D.; Briesch, M.; Rothlauf, F. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. In Proceedings of the GECCO, Boston, MA, USA, 9–13 July 2022.
27. Koza, J.R. *Genetic Programming II: Automatic Discovery of Reusable Programs*; MIT Press: Cambridge, MA, USA, 1994.
28. Forstenlechner, S.; Fagan, D.; Nicolau, M.; O'Neill, M. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In Proceedings of the Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, 19–21 April 2017; Proceedings 20; Springer: Berlin/Heidelberg, Germany, 2017; pp. 262–277.
29. Li, T.O.; Zong, W.; Wang, Y.; Tian, H.; Wang, Y.; Cheung, S.C.; Kramer, J. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In Proceedings of the IEEE/ACM ASE, Luxembourg, 11–15 September 2023; pp. 14–26.
30. Ma, W.; Liu, S.; Wenhan, W.; Hu, Q.; Liu, Y.; Zhang, C.; Nie, L.; Liu, Y. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. *arXiv* **2023**, arXiv:2305.12138.
31. Surameery, N.; Shakor, M. Use Chat GPT to Solve Programming Bugs. *Int. J. Inf. Technol. Comput. Eng.* **2023**, *3*, 17–22. [CrossRef]
32. Xie, Z.; Chen, Y.; Zhi, C.; Deng, S.; Yin, J. ChatUniTest: A ChatGPT-based automated unit test generation tool. *arXiv* **2023**, arXiv:2305.04764.
33. Minaee, S.; Mikolov, T.; Nikzad, N.; Chenaghlu, M.; Socher, R.; Amatriain, X.; Gao, J. Large Language Models: A Survey. *arXiv* **2024**, arXiv:2401.14423

34. Jesse, K.; Ahmed, T.; Devanbu, P.T.; Morgan, E. Large language models and simple, stupid bugs. In Proceedings of the IEEE/ACM MSR, Melbourne, Australia, 15–16 May 2023; pp. 563–575.
35. Asare, O.; Nagappan, M.; Asokan, N. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empir. Softw. Eng.* **2023**, *28*, 129. [CrossRef]
36. Schuster, R.; Song, C.; Tromer, E.; Shmatikov, V. You autocomplete me: Poisoning vulnerabilities in neural code completion. In Proceedings of the USENIX Security 21, Virtual, 11–13 August 2021; pp. 1559–1575.
37. Stechly, K.; Marquez, M.; Kambhampati, S. GPT-4 Doesn't Know It's Wrong: An Analysis of Iterative Prompting for Reasoning Problems. *arXiv* **2023**. Available online: https://openreview.net/forum?id=PMtZjDYB68 (accessed on 27 March 2024).
38. Krishna, S.; Agarwal, C.; Lakkaraju, H. Understanding the Effects of Iterative Prompting on Truthfulness. *arXiv* **2024**, arXiv:2402.06625v1
39. Fraser, G.; Arcuri, A. The seed is strong: Seeding strategies in search-based software testing. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, 17–21 April 2012; pp. 121–130.
40. Saber, T.; Brevet, D.; Botterweck, G.; Ventresque, A. Is seeding a good strategy in multi-objective feature selection when feature models evolve? *Inf. Softw. Technol.* **2018**, *95*, 266–280. [CrossRef]
41. Wick, J.; Hemberg, E.; O'Reilly, U.M. Getting a head start on program synthesis with genetic programming. In Proceedings of the Genetic Programming: 24th European Conference, EuroGP 2021, Held as Part of EvoStar 2021, Virtual Event, 7–9 April 2021; Proceedings 24; Springer: Berlin/Heidelberg, Germany, 2021; pp. 263–279.
42. Helmuth, T.; Spector, L. General program synthesis benchmark suite. In Proceedings of the GECCO, Madrid, Spain, 11–15 July 2015.
43. Miller, J.F.; Harding, S.L. Cartesian genetic programming. In Proceedings of the GECCO, Atlanta, GA, USA, 12–16 July 2008.
44. Brameier, M.; Banzhaf, W.; Banzhaf, W. *Linear Genetic Programming*; Springer: Berlin/Heidelberg, Germany, 2007.
45. O'Neill, M.; Ryan, C. Volume 4 of Genetic programming. In *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*; Kluwer Academic Publishers: Norwell, MA, USA, 2003.
46. Whigham, P.A. Grammatical Bias for Evolutionary Learning. Ph.D. Thesis, University College, Australian Defence Force Academy, University of New South Wales, Canberra, Campbell, ACT, Australia, 1997.
47. O'Neill, M.; Nicolau, M.; Agapitos, A. Experiments in program synthesis with grammatical evolution: A focus on integer sorting. In Proceedings of the IEEE CEC, Beijing, China, 6–11 July 2014.
48. Saber, T.; Wang, S. Evolving better rerouting surrogate travel costs with grammar-guided genetic programming. In Proceedings of the IEEE CEC, Glasgow, UK, 19–24 July 2020.
49. Lynch, D.; Saber, T.; Kucera, S.; Claussen, H.; O'Neill, M. Evolutionary learning of link allocation algorithms for 5G heterogeneous wireless communications networks. In Proceedings of the GECCO, Prague, Czech Republic, 13–17 July 2019.
50. Saber, T.; Fagan, D.; Lynch, D.; Kucera, S.; Claussen, H.; O'Neill, M. Multi-level Grammar Genetic Programming for Scheduling in Heterogeneous Networks. In Proceedings of the EuroGP, Parma, Italy, 4–6 April 2018.
51. Saber, T.; Fagan, D.; Lynch, D.; Kucera, S.; Claussen, H.; O'Neill, M. A multi-level grammar approach to grammar-guided genetic programming: the case of scheduling in heterogeneous networks. *Genet. Program. Evol. Mach.* **2019**, *20*, 245–283. [CrossRef]
52. Saber, T.; Fagan, D.; Lynch, D.; Kucera, S.; Claussen, H.; O'Neill, M. Hierarchical Grammar-Guided Genetic Programming Techniques for Scheduling in Heterogeneous Networks. In Proceedings of the IEEE CEC, Glasgow, UK, 19–24 July 2020.
53. Saber, T.; Fagan, D.; Lynch, D.; Kucera, S.; Claussen, H.; O'Neill, M. A Hierarchical Approach to Grammar-Guided Genetic Programming The case of Scheduling in Heterogeneous Networks. In Proceedings of the TPNC, Dublin, Ireland, 12–14 December 2018.
54. Forstenlechner, S.; Fagan, D.; Nicolau, M.; O'Neill, M. Extending program synthesis grammars for grammar-guided genetic programming. In Proceedings of the PPSN, Coimbra, Portugal, 8–12 September 2018; Springer: Berlin/Heidelberg, Germany, 2018.
55. Manning, C.D. Human language understanding & reasoning. *Daedalus* **2022**, *151*, 127–138._a_01905. [CrossRef]
56. OpenAI. *GPT-4 Technical Report*; OpenAI: San Francisco, CA, USA, 2023.
57. Manyika, J.; Hsiao, S. An overview of Bard: An early experiment with generative AI. *AI Google Static Doc.* **2023**. Available online: https://ai.google/static/documents/google-about-bard.pdf (accessed on 27 March 2024).
58. Wang, B.; Wang, Z.; Wang, X.; Cao, Y.; A Saurous, R.; Kim, Y. Grammar prompting for domain-specific language generation with large language models. *Adv. Neural Inf. Process. Syst.* **2024**. Available online: https://dl.acm.org/doi/10.5555/3666122.3668959 (accessed on 27 March 2024).
59. Hartmann, B.; MacDougall, D.; Brandt, J.; Klemmer, S.R. What would other programmers do: Suggesting solutions to error messages. In Proceedings of the SIGCHI, Atlanta, GA, USA, 5–10 April 2010.
60. Ragkhitwetsagul, C.; Krinke, J.; Clark, D. A comparison of code similarity analysers. *Empir. Software Eng.* **2018**, *23*, 2464–2519. [CrossRef]
61. Cohen, A. FuzzyWuzzy: Fuzzy String Matching in Python, 2011. Available online: https://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/ (accessed on 27 March 2024).
62. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.* **2002**, *28*, 654–670. [CrossRef]
63. Gitchell, D.; Tran, N. Sim: A utility for detecting similarity in computer programs. *ACM Sigcse Bull.* **1999**, *31*, 266–270. [CrossRef]
64. Gao, T.; Fisch, A.; Chen, D. Making pre-trained language models better few-shot learners. *arXiv* **2020**, arXiv:2012.15723.

65. White, J.; Fu, Q.; Hays, S.; Sandborn, M.; Olea, C.; Gilbert, H.; Elnashar, A.; Spencer-Smith, J.; Schmidt, D.C. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv* **2023**, arXiv:2302.11382.
66. Baxter, I.D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone detection using abstract syntax trees. In Proceedings of the ICSME, Bethesda, MD, USA, 20 November 1998.
67. Helmuth, T.; Spector, L. *Detailed Problem Descriptions for General Program Synthesis Benchmark Suite*; University of Massachusetts Amherst: Amherst, MA, USA, 2015.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.