Enhancing Large Language Models-Based Code Generation by Leveraging Genetic Improvement

Giovanni Pinna, Damiano Ravalico, Luigi Rovito(✉), Luca Manzoni, and Andrea De Lorenzo

University of Trieste, 34127 Trieste, TS, Italy
{giovanni.pinna,damiano.ravalico,luigi.rovito}@phd.units.it,
{lmanzoni,andrea.delorenzo}@units.it

Abstract. In recent years, the rapid advances in neural networks for Natural Language Processing (NLP) have led to the development of Large Language Models (LLMs), able to substantially improve the state-of-the-art in many NLP tasks, such as question answering and text summarization. Among them, one particularly interesting application is automatic code generation based only on the problem description. However, it has been shown that even the most effective LLMs available often fail to produce correct code. To address this issue, we propose an evolutionary-based approach using Genetic Improvement (GI) to improve the code generated by an LLM using a collection of user-provided test cases. Specifically, we employ Grammatical Evolution (GE) using a grammar that we automatically specialize—starting from a general one—for the output of the LLM. We test 25 different problems and 5 different LLMs, showing that the proposed method is able to improve in a statistically significant way the code generated by LLMs. This is a first step in showing that the combination of LLMs and evolutionary techniques can be a fruitful avenue of research.

Keywords: Evolutionary Computation · Evolutionary Algorithms · Large Language Models · Artificial Intelligence · Machine Learning · Neural Networks · Code Generation · Genetic Improvement · Grammatical Evolution · Genetic Programming

1 Introduction

Designing and developing software is a complex and demanding activity, which requires specific skills, knowledge, and expertise. Hence, tools to help the programmer in producing the code have been developed during the decades, with recent advances providing tools for automatic code generation, including LLM-based ones, like Copilot [46,51]. In general, the text generation abilities of LLMs

can be leveraged to help developers in generating code starting from a description of its expected behavior. However, when the description (i.e., prompt) of the problem is poorly-formulated and not well-defined or the problem itself is non-trivial, the generated code may be incorrect or incomplete. Even in that case, the code can still provide a useful starting point to provide a complete and correct solution.

In this paper, we define an evolutionary method—Genetic Improvement (GI)—that employs a collection of user-provided test cases (as input-output pairs) starting from the code generated by LLMs having only a textual prompt. The output of the LLM is also used to influence the method employed by GI. For it, we employ Grammatical Evolution (GE) with a grammar that is dynamically built for each problem based on the code provided by the LLM. This design choice enables us to automatically implement a grammar that, with high probability, is coherent with the problem without the drawbacks of either using a very large grammar or having to manually define a problem-specific one. The proposed method has a large applicability since it can be used to enhance any LLM-generated code, with the only requirement being a set of input-output pairs for each problem. The support of languages different from Python can be done by defining the grammar of the language and the language-specific parts of specializing the grammar.

We test 25 different problems from PSB2 [16,17] with 5 LLMs and we show that the proposed method is able to improve the correctness of code generated by LLMs in a statistically significant way. The research questions we aim to answer after our experimental analysis are detailed as follows:

RQ1 Do LLMs suffice to automatically generate code that is correct regardless of the complexity of the tackled problem?

RQ2 Is it possible to improve the correctness of the code generated by LLMs by employing a GI-based technique?

The main contributions of our work are, thus, the followings: (i) we implemented a dynamic grammar definition that can specialize a grammar for the specific problem, and (ii) we introduced an evolutionary method that improves the correctness of code generated by LLMs. While the proposed method is implemented for the Python language, the principles are not tied to it and are applicable to other programming languages.

In Sect. 2 we provide a literature review of this broad research field, in Sect. 3 we delve into our proposed method, in Sect. 4 we show the outcome of our experimental analysis, in Sect. 5 we try to give an answer to our research questions, and in Sect. 6 we do a brief recap with final conclusions.

2 Related Works

2.1 Large Language Models

The development of neural network-based architectures, particularly the Transformer model [52], marked a significant turning point in Natural Language Processing (NLP). This model, relying exclusively on attention mechanisms, paved the way for advancements in text generation and comprehension. Subsequent to this, various tools and models, such as BERT [10], have been developed, all trained on extensive text corpora to enhance their ability to infer missing text from input sequences.

In 2023, Meta AI released LLaMA [49], a family of pre-trained LLMs open-sourced for research and development. These models, with parameters ranging from 7 to 65 billion, were trained on vast quantities of textual data. A notable derivative, Alpaca [48], developed by Stanford University, is a fine-tuned version of LLaMA that employs the self-instruct method for instruction tuning [54]. Alpaca, specifically the 7B and 13B versions, was trained on instruction-following demonstrations, providing capabilities akin to other chatbots.

Additionally, LLaMA 2 [50], an enhanced version with up to 70 billion parameters, was trained on 40% more data than its predecessor, further advancing the field.

The contribution of OpenAI to LLMs, particularly with GPT3 [6] and its subsequent versions, GPT3.5 and GPT4 [37], showcases significant advancements in text generation and comprehension. These models, based on decoder-only Transformer architectures, vary in the number of parameters and scale of training data. GPT4, available through ChatGPT Plus and OpenAI APIs, continues to demonstrate remarkable capabilities in various text-based tasks, although limitations remain due to its predictive nature and the unavailability of its training data and procedures for public scrutiny.

2.2 Code Generation

The research field concerning code generation was born even before the first electronic computers were released [32], and has already found great interest in the scientific community since the second half of the Twentieth Century with program synthesis [4,14,29,30] and program transformation [55].

Recently, automatic code generation tools were proposed with the purpose of implementing design patterns [7], generating efficient code based on Event-B formal specifications [33], generating C code from mathematical models [42], generating code based on Generative Adversarial Networks (GANs) [47], generating code in Verilog Hardware Description Language (VHDL) based on Unified Modeling Language (UML) specifications [35] or high-level hardware descriptions [25], and generating code for web applications based on the Model-View-Controller (MVC) pattern [39].

Automatic code generation has found great interest even in the field of Evolutionary Computation (EC). Specifically, Genetic Programming (GP) [20] is

probably the most famous evolutionary algorithm that was initially developed with the purpose of evolving programs. Then, a GP method defined as Grammatical Evolution (GE) [36,43] was also introduced and gained popularity because of its ability to represent and evolve programs that are compliant with a given grammar.

GE was adopted to generate VHDL code representing digital circuits [19]. In [27], an evolutionary algorithm was implemented to generate code for Programmable Logic Controllers (PLCs) in controlling processes. In [56], an optimized hybrid evolutionary algorithm was employed to predict code running time and speed up automatic code optimization for Ansor [57], which is an auto-scheduling system that extends Apache TVM [9]. In [44], a hybrid Genetic Algorithm (GA) was leveraged to automate parallel code generation for regular control problems. Cartesian Genetic Programming (CGP) [34] was employed in [53] to evolve machine code for a simpler implementation of the MOVE processor. Multi-objective linear GP was used in [45] to generate assembly driver routines for devices belonging to micro-controller-based systems.

The recent advent of LLMs has additionally enlarged the scope of automatic code-generation techniques. Especially, LLaMA and ChatGPT, have expanded automatic code generation. ChatGPT has demonstrated proficiency in this domain but has limitations [3]. Studies on the effectiveness of LLMs include an analysis of Copilot for coding support [51], an empirical assessment of ChatGPT generated code [24], and a study on ChatGPT-like tools non-determinism impacting scientific validity [38].

Evaluating code generation is challenging, prompting the introduction of several benchmarks. In [2], the LLMs prior to LLaMA and ChatGPT were tested against two benchmark datasets to identify their limitations. The PSB2 benchmark suite [16,17], featuring 25 updated general program synthesis problems, assesses code generation methods. For instance, the effectiveness of Copilot was compared with evolutionary approaches like GP using PSB2 [46]. The HUMANEVAL set [8], designed to measure functional correctness in code synthesis from doc-strings, tested LLMs like Codex. EvalPlus [23] further extends the test cases of HUMANEVAL to rigorously evaluate LLM-synthesized code's functional correctness.

Although numerous code generation tools, including LLMs-based ones, have demonstrated impressive capabilities, their effectiveness in generating completely correct code varies with problem complexity and, for LLMs, prompt quality. Genetic Improvement (GI) [22], a GP method, optimizes existing code through evolutionary processes. GI has been applied to enhance auto-generated code in languages like C++ [40], C, Java [31], and Python [11]. Tools like [1,5] focus on bug removal and time efficiency optimization. Recent studies have explored integrating GI methods with LLMs, evolving code snippets generated and modified by LLMs [26]. For example, [41] used recent LLMs to create new hybrid swarm intelligence optimization algorithms.

Besides evolving the generated code, also the prompt itself can be evolved and refined. This is because writing an effective prompt is not trivial and, especially for an LLM, adopting a well-written and descriptive-enough prompt could yield totally different results. EvoPrompt [15] was developed as a framework for discrete prompt optimization in which LLMs are employed to evolve prompts. Promptbreeder [12] was presented as a general-purpose self-referential self-improvement technique that opportunely evolves and adapts prompts for a specific domain.

In this work, we propose a GI method based on GE in which we improve the correctness of code generated by the most recent and performing LLMs.

3 Proposed Approach

In this section, we introduce the technique used to improve the LLM output via GI. We can identify the following main steps:

- **Encoding of the LLM output.** The output of the LLM is processed to extract the code to be improved and encode it in a genome;
- **Dynamic Grammar Definition.** Both the prompt and the initial solution generated by the LLM is used to specialize a grammar to the specific problem;
- **Evolution.** GE is used to evolve the genome using the specialized grammar.

The second and third steps are what compose GI properly, with the first step being a necessary preprocessing phase. All these steps will be detailed below.

3.1 Encoding of the LLM Output

As a first step, the textual description of the problem is given as input to an LLM in order to obtain a textual answer that contains, among possibly additional text, the code that the LLM consider to be the solution to the problem.

We recall that, even when directed by textual instructions, the output of an LLM can contain text that is extraneous with respect to the actual code needed to solve the problem at hand. Hence, the first phase is the one of extracting the code from the textual answer. To this end, we employ specialized tokens called *tags* that delimit the starting and ending position of the code in the text.

Once the code has been obtained, a series of preprocessing steps are performed in order to allow its encoding in a way that is suitable for GI:

1. Removal of comments, since they are not necessary for actually executing the code. In the current implementation, this is performed via regular expressions matching all comments.
2. Once the comments have been removed, the procedure/function actually representing the solution is extracted. In this step, we need to ensure that the code is actually syntactically correct. As a simple heuristic, we start with the full body of the function, we check if it is syntactically correct and, if not, we remove one line at a time starting from the end until a syntactically

valid solution has been found. In the worst case, this step will produce an empty function body which is syntactically correct or can be made so in an automated way (e.g., adding a `pass` for Python).

3. With a syntactically valid code it is now possible to represent the code as an Abstract Syntax Tree (AST). In this representation the name of the variables is immaterial, so their names are replaced with v_i, with i ranging from 0 the number of variables minus one. This allows for the evolution process not having to use a grammar with specialized variable names. Since the name of the function actually containing the solution to the problem is immaterial, it is replaced by a default value which, in our implementation, is `evolve`.

The additional steps to be taken depend on the specifics of the implementation and the language since the AST should be then transformed in a way that is manageable by the GI technique employed. In particular, we may impose restrictions on the grammar used to produce the AST (e.g., having it represent only a subset of the entire language) in order to help the successive GI step. In the specific implementation used in this paper the restriction on the grammar is imposed by only parsing a subset of the Python language, and the additional preprocessing step consists of replacing indents and newlines with specific tags that are then used by the GE implemented with the ponyge2 library [11].

3.2 Dynamic Grammar Definition

Once the LLM-produced code is correctly encoded, it can be used, together with the textual description of the problem, to modify an existing (general) grammar to include problem-specific constants and words.

Let $\mathcal{G}$ be a grammar consisting of all the main constructs of the programming language used to express the solution to the problem. Notice that the grammar $\mathcal{G}$ might not be the standard formal grammar for a language, but will have some biases and changes necessary for the evolutionary process to work. First of all, $\mathcal{G}$ will be designed in such a way that, when used in a generative fashion, there is a bias towards small and actually manageable programs. This is generally obtained by giving additional derivations to expansion containing terminal instead of non-terminals, skewing a random selection toward shorter individuals, but without changing the recognized (or generated) language. In addition to that, it is possible to make the grammar generate code already including, for example, commonly used libraries and functions without having to generate "from scratch" their names.

The main idea is to add additional terminal symbols to the grammar $\mathcal{G}$ that are problem-specific. In particular, there are two main classes of symbols that can be useful to guide the search: constants and functions. Constants can be present in the problem description (e.g., a description containing "find all multiple of 37 less than 1000" has two constants) and finding them during the evolutionary process can be a difficult task while inserting them from the problem description makes easy to use in the resulting solution. Functions and libraries can help not "reinvent the wheel" by generating existing library code from scratch, but finding

which libraries can be useful is a complex task that requires some knowledge of the language ecosystem (and not only of its grammar). Being trained on a large corpus of text—including code—makes the LLM a useful source of information on which libraries can be useful. Hence, the following steps are taken to extract information from the problem description and the LLM output:

- Extraction of imported libraries, function names, string and numbers from the LLM output. This ensures that libraries—and the functions they offer—that are useful to construct the problem's solution can be used in the GI step;
- Use of the problem description to extract constants (either strings or numbers);
- Use of KeyBERT [13] to extract keywords from the problem description to be used as terminals in the grammar, always in the form of strings.

Hence the problem-specific grammar $\mathcal{G}_P$ is obtained by augmenting $\mathcal{G}$ with the previously extracted symbols. In addition to that, it is possible to change the probabilities of generating the extracted symbols (making their presence in a solution more probable) by replicating the added derivation multiple times.

One clear advantage of this approach is that the solutions found by the evolutionary process can more easily use the existing library functions and there is no need to evolve "from scratch" constants that can easily be inferred by the problem description. One drawback is, however, that the quality of the grammar is strongly dependent upon the quality of the description provided and the quality of the output produced by the LLM. An incorrect but generally reasonable output (e.g., containing calls to useful library functions) can be helpful in guiding the search. A completely incorrect output which is extremely far from a real solution can make the evolutionary process worse by increasing the probability of using constants and functions that are not useful.

3.3 Evolution

Since GE is employed as the evolutionary process to enhance existing solutions, it is necessary to define two main components: the creation of the initial population and the evaluation of individuals. The other steps of the evolutionary process, specifically selection, recombination, and mutation, are not specific to the proposed method and, therefore, will not be discussed in detail here.

The initial population is initialized by the solutions generated by the LLM once encoded as detailed before. Since the number of solutions generated in this way can be smaller than the population size, individuals are replicated until the desired population size has been reached. This step is necessary since LLM generation of solutions can be expensive in terms of computational resources.

Regarding the fitness function used, we employ a series of test cases (as input-output pairs) provided by the user in its definition. Formally, given a set

of n test cases $D_P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ for the problem P and a program M, the fitness function is defined as:

$$F_{D_P}(M) = \sum_{i=1}^{n} [\text{out}(M, x_i) \neq y_i] \tag{1}$$

where $\text{out}(M, x_i)$ represents the output of M on input x_i and $[\cdot]$ is the indicator function that is 1 if the enclosed proposition is true and 0 otherwise. By minimizing F_{D_P} we reduce the number of incorrect outputs across the set D_P of test cases.

4 Experimental Phase

In this section, we detail our experimental phase, in which we test the following LLMs: Alpaca 7B (`A7`), Alpaca 13B (`A13`), LLaMA 2 13B (`L2`), ChatGPT (`CG`), and GPT4 (`G4`).

4.1 Language and Grammar

While the proposed method has been described in a general way, in the experimental phase we focused on the Python language. The choice was mainly due to the fact that popular languages provide more training material to LLM, thus increasing their ability to generate meaningful solutions and the availability of libraries for parsing and analysis of the code.

The grammar we employed is actually a subset of the entire Python grammar whose structure is shortly defined here. First of all, the grammar always generates a series of imports followed by a call to the `evolve` function, which receives a certain number of parameters as input and contains some code as the body. The code can include one or more statements, which are divided into conditional statements (if, for, while) and non-conditional statements. Especially, a non-conditional statement can be either a return, an assignment, a print, a break keyword, a continue keyword, a pass keyword, a raise, or an instance method called on an object. A non-terminal representing a value can be either a problem parameter, a string, a number, a Boolean, a null, a logical or mathematical combination of two variables, a list, a tuple, a list comprehension, a tuple comprehension, or a function called on variables. The complete base grammar in Backus-Naur Form (BNF) for the subset of Python that we use can be found in our repository (see Sect. 4).

4.2 Problems

In order to evaluate our approach, we chose to utilize a widely-used problem dataset, PSB2 [16,17]. This dataset offers a comprehensive set of problems (Table 1), serving as a benchmark for validating automatic code generation algorithms. We performed an initial evaluation of the complexity of the coding problems from PSB2. This evaluation proceeds as follows: initially, for each problem,

we prompt the LLM to generate a solution. This step is independently repeated 10 times with the same prompt. If the LLM correctly solves the problem (i.e., the generated code is completely correct with respect to the examples given by the user) we do not consider it for further improvement. Such problems are denoted by ✓ in Table 2.

Table 1. List of PSB2 problems.

Name	Alias	Name	Alias
Basement	BS	Bouncing Balls	BB
Bowling	BW	Camel Case	CC
Coin Sums	CS	Cut Vector	CV
Dice Game	DG	Find Pair	FP
Fizz Buzz	FB	Fuel Cost	FC
Greatest Common Divisor	GD	Indices of Substring	IS
Leaders	LD	Luhn	LH
Mastermind	MM	Middle Character	MC
Paired Digits	PD	Shopping List	SL
Snow Day	SD	Solve Boolean	SB
Spin Words	SW	Square Digits	SQ
Substitution Cipher	SC	Twitter	TW
Vector Distance	VD		

In our context, the problem description is crucial since it is the main component of the prompt for the LLMs. Since multiple problem descriptions might be available[1], among them we adopt the official one available in the PSB2 paper itself [16,17].

4.3 Experimental Settings and Results

We request the LLM to solve each problem 10 times, producing 10 individuals that are then replicated 100 times each to meet the required population size. The fitness is computed by adopting 1000 test cases as training data and 1000 test cases as test data. For each combination of problem and LLM, we conduct 10 separate GI runs starting from the same initial population. In all our experiments, the test cases are obtained via the `psb2` library [16,17]. Regarding the parameters used for the evolution, we use 1000 as population size, which is

[1] The PSB2 paper also includes an external hyperlink to a file that contains the same problems but with different descriptions (e.g., FB description in the external table details that the output is printed, while the description in the paper itself details that the output is returned, which is more coherent with the original purpose of PSB2).

evolved for 100 generations. We set an initial maximum depth of 15 and a maximum depth of 30 for the AST.[2] We do not impose a wrap-around limit for the genomes and we set a maximum codon size of 200. The selection is performed using tournament selection with pressure 50, ensuring a population that remains syntactically close to the solution obtained from the LLM. Both crossover and mutation are sub-tree operators, applied with a probability of 0.80.

In order to test that GI is better than the self-correction ability of LLMs we implemented the following procedure, as described in [26]. For each LLM, we request it to solve a given problem (a method denoted as $\mathcal{L}$) and then we evaluate the correctness of the output code. When the code is incorrect we ask the LLM to self-correct its output (a method denoted as $\mathcal{L}^+$). During a self-correction run, we request the LLM to incrementally correct the output code 10 times. In each iteration, we supply 20 pairs of input and output examples that were not resolved in the last iteration. To provide these examples, we instruct the LLM with the prefix `Make sure that:` followed by the examples formatted as `{input} -> {output}`. We perform 2 self-correction runs and we store the maximum value among the median values retrieved from those runs.[3] In this way, we select the best result obtained by using self-correction of the LLM without employing GI. Our Python code is publicly available online.[4]

In Table 2 we show the median number (scaled in $[0, 1]$) of passed test cases for $\mathcal{L}$ and $\mathcal{L}^+$. Statistical significance is assessed by using a Wilcoxon-Mann-Whitney test [28] with $\alpha = 0.05$. For more than two comparisons a preliminary Kruskal-Wallis test [21] with $\alpha = 0.05$ is performed and then, if there is a statistically significant difference, a Holm-Bonferroni correction [18] with $\alpha = 0.05$ is employed. For $\mathcal{L}$ and GI, the sample size is 10. For $\mathcal{L}^+$, the sample size is 2.

For each LLM, we highlight, in the table, both the problems that are directly solved (✓) and the ones that are not parsed by the grammar (`np`), which is due to the fact that the employed grammar does not capture the entirety of the Python language. In principle, we could parse all the problems by defining an exhaustive grammar. Defining an exhaustive grammar leads to a larger search space that could slow down the evolutionary processes. However, the employed grammar defines a large enough subset of the Python language, giving a reasonable trad-off between expressiveness and complexity of the search space.

For `A7` and `A13`, both $\mathcal{L}$ and $\mathcal{L}^+$ lead to incorrect code and GI provides a significantly improved version of it for all the parsed problems. `L2`, on average, performs slightly better than `A7` and `A13`, and GI provides a significantly improved version of the code for almost all the parsed problems. As expected, `CG` and `G4` directly solve more problems than the smaller LLaMA-based models. Even in this case, GI obtains a significant improvement for almost all the parsed problems that are not directly solved by them. As regards `A7`, there is a high

[2] The BW problem has an initial maximum depth of 25 and maximum depth of 40 since the initial solution requires a depth greater than 15.
[3] The number of repetitions is constrained by the limitations of our available budget.
[4] https://github.com/dravalico/LLMGIpy.

Table 2. Median number (scaled in $[0, 1]$) of passed test cases for each P and LLM. We highlight in bold the methods that are statistically significant better for the same problem and LLM. Problems that are directly solved from $\mathcal{L}$ are indicated with ✓. Problems that are not parsed for the GI are indicated with `np` (not parsed). With a 0 we denote problems that do not pass any test cases.

Problem	A7			A13			L2			CG			G4		
	$\mathcal{L}$	$\mathcal{L}^+$	GI	$\mathcal{L}$	$\mathcal{L}^+$	GI	$\mathcal{L}$	$\mathcal{L}^+$	GI	$\mathcal{L}$	$\mathcal{L}^+$	GI	$\mathcal{L}$	$\mathcal{L}^+$	GI
BS	0.01	0	**0.58**	0	0	**0.20**	0.13	0.00	**0.20**	✓	—	—	✓	—	—
BB	0	0	np	0	0	np	0	0	np	0.00	0.00	**0.06**	0.00	0.04	**0.06**
BW	0	0	np	0	0	**0.03**	0	0	**0.06**	0.00	0.00	**0.03**	0.04	0.00	0.04
CC	0	0	**0.31**	0	0	np	0	0	**0.31**	✓	—	—	✓	—	—
CS	0	0	np	0	0	0	0	0	0	0	0	0	0	0	0
CV	0	0	np	0	0	0	0	0	np	0	0	0	0	0	0
DG	0	0	np	0	0	np	0	0.00	0.00	0.07	0.00	**0.17**	0.15	0.00	**0.17**
FP	0	0	np	0	0	np	0	0	0	0	0	0	0	0	0
FB	0	0	np	0.94	0	0.94	✓	—	—	✓	—	—	✓	—	—
FC	0	0	np	0	0	np	0.18	1.00	**1.00**	✓	—	—	✓	—	—
GD	0	0	np	0	0	np	0	0	**0.59**	✓	—	—	✓	—	—
IS	0	0	np	0.16	0	**0.67**	✓	—	—	✓	—	—	✓	—	—
LD	0	0	**0.09**	0.05	0	**0.09**	0	0	**0.09**	✓	—	—	✓	—	—
LH	0	0	**0.04**	0	0	**0.06**	0	0	**0.04**	0.52	0.00	**1.00**	0.00	0	0.08
MM	0	0	np	0	0	0	0	0	np	0	0	0	0	0	0
MC	**0.50**	0	np	0	0	**0.02**	✓	—	—	✓	—	—	✓	—	—
PD	0	0	**0.13**	0	0	**0.13**	0	0.01	**0.19**	✓	—	—	✓	—	—
SL	0	0	**0.00**	0	0	0.00	0.00	0	**0.00**	0.00	0	**0.01**	0.00	0	**0.01**
SD	0	0	np	0	0	np	**0.05**	0	np	0.05	0.00	**0.06**	0.04	0.00	**0.09**
SB	0.50	0	**0.78**	0	0	np	0.49	0.00	**0.55**	0.00	**0.42**	np	0	0.67	0.50
SW	0	0	**0.34**	0	0	**0.34**	0.56	0.00	0.56	✓	—	—	✓	—	—
SQ	0	0	**0.01**	0	0	**0.01**	0.00	0	**0.01**	✓	—	—	✓	—	—
SC	0	0	np	0	0	**0.05**	0.04	0	**0.05**	✓	—	—	✓	—	—
TW	**0.30**	0	np	**0.30**	0	np	**0.99**	0	np	✓	—	—	✓	—	—
VD	0	0	np	0	0	np	0	0	np	0.96	0.00	**1.00**	**0.95**	0.00	0.95

percentage of problems that are not parsed, due to the low quality of the solutions provided by the LLM, which are not even parsable by the used grammar. Differently from other problems, TW is directly solved by OpenAI models, but the code produced by the others LLMs is not parsable by the used grammar, hence this is the only case where GI cannot provide any improvement in all models. In general, the self-correction procedure seems to be poorly effective for all the models, except for `L2` on FC and both `CG` and `G4` on SB, where in the very latter case no statistical significance is recorded. Except for `G4` on VD where the LLM alone performs better, we can safely state that our GI method is never worse than both the LLM and the self-correction procedure. The general behavior is that GI is able to improve the code generated by the LLMs. However, the improvement leads to an optimal fitness value only in three cases, probably due to the size and complexity of the search space.

In Fig. 1, we show the fitness trend of GI. It is possible to observe two different behaviors, depending on the LLM used to generate the initial solutions. In particular, for smaller models (i.e., the LLaMA-based ones) there is an improve-

ment in the fitness with the number of generations, possibly due to the fact that the initial solution is incorrect but not in a way that requires a complete rewrite—i.e., only small edits are sufficient to gain correctness. On the other hand, larger models (i.e., the OpenAI-based ones) exhibit a bimodal behavior: either the initial solution is completely correct or irremediably wrong—i.e., the number of passed test cases is close to zero and improving the code requires substantial rewriting. In these cases, we record a slight improvement (except for LH and VD where GI from `CG` is able to reach an optimum).

5 Discussion

Finally, we try to provide a brief summary of the main findings and give proper answers to our research questions:

RQ1 Do LLMs suffice to automatically generate code that is correct regardless of the complexity of the tackled problem?
The most performing LLMs (i.e., the larger models, in our case the OpenAI ones) tend to generate better code than the smaller ones, which in our investigation are the LLaMA-based ones. However, even these models often produce code that is not correct for many of the investigated problems. When incorrect code is generated, we attempt to prompt the LLM again with a small set of failed test cases, asking to modify the program so that it passes these specific tests. Despite these efforts, our findings indicate that if the model initially generates incorrect code, it is unlikely to correct itself effectively through mere re-prompting. This outcome highlights the challenges faced by LLMs when generating accurate code for a specified problem, even when examples are provided. It suggests that integrating additional techniques may be necessary to achieve the desired level of correctness.

RQ2 Is it possible to improve the correctness of the code generated by LLMs by employing a GI-based technique?
For almost all the tested problems, GI is capable of providing an improvement of the code generated by LLMs. In fact, the resulting code is generally better than simply re-prompting the LLMs, showing that GI can, in fact, provide an advantage with respect to a full LLM-based solution. The reason for these performances can be identified in the use of a specialized grammar. Such a grammar can guide the search process of GE in specific areas of the search space where better solutions can potentially be found. This mechanism, however, is strongly dependent on the quality of the initial solution provided by the LLM. While related to the actual correctness of the solution, the quality needed to specialize the grammar is the ability for the LLM to provide a reasonable set of functions, libraries, and constants to be used by GE. This is generally an easier task for an LLM with respect to the production of correct code, thus making even smaller LLMs a good source of initial solutions for the GI-based step. The effect

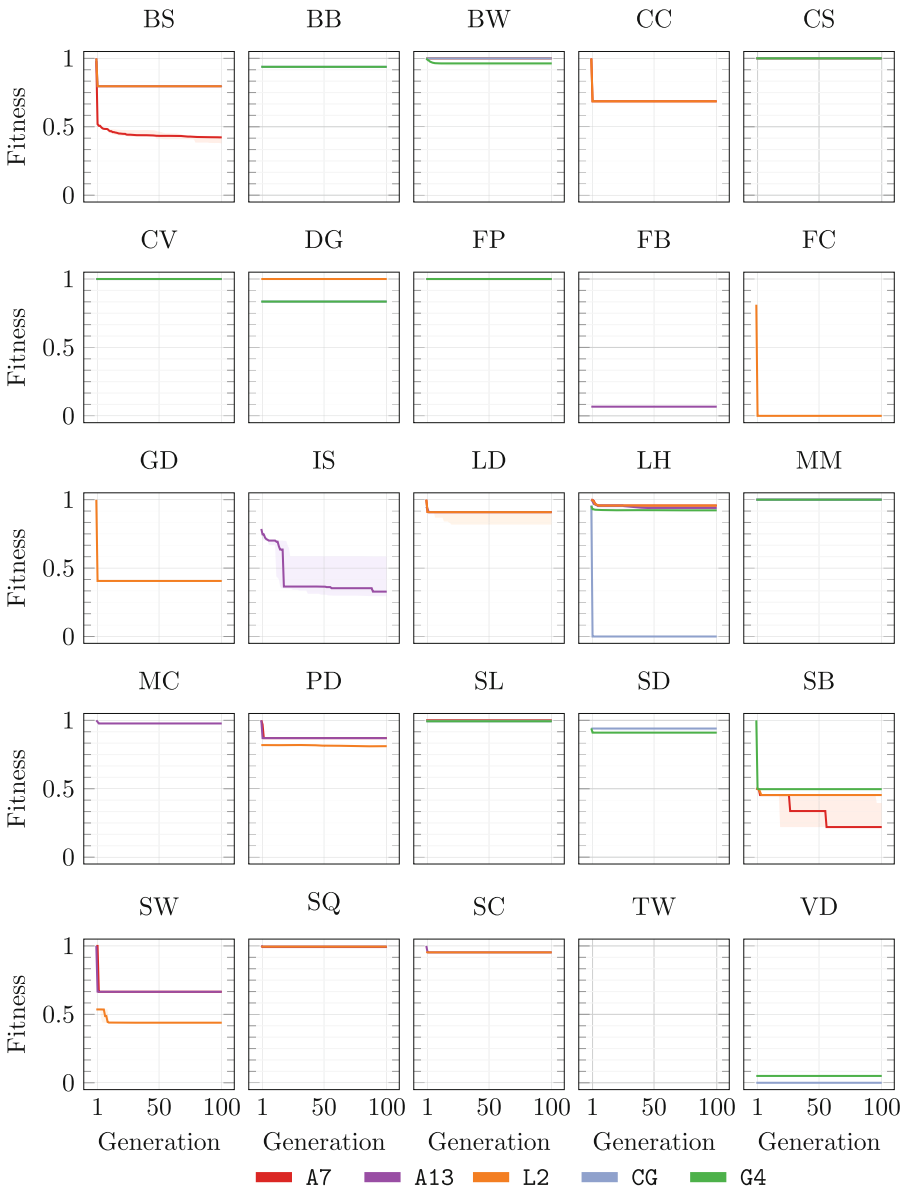

Fig. 1. Trend of the fitness (scaled in $[0, 1]$) across the generations of GI. Each line represents the median fitness across all runs. Shaded area represents the inter-quartile range. We omit the corresponding lines where no GI is performed.

is that the evolution is more effective for smaller models and simple problems (which are usually directly solved by larger models) and less evident for complex ones. A potential limitation could be the large number of test cases required to drive the evolution. Future works could focus on discovering the minimum number of test cases needed to achieve a meaningful improvement.

6 Conclusion

In this paper, we propose an evolutionary approach designed to improve the accuracy of code generated by state-of-the-art LLMs. For our approach we require the user to provide both a textual description of the problem and a set of test cases as input-output pairs. If the LLM fails to generate a solution that yields the correct outputs, we employ GI techniques to enhance the accuracy of the provided solution. To reduce the search space and enhance the effectiveness of the evolutionary process with GI, we have chosen to automatically specialize a grammar using the LLM output in order to tailor it to the specific problem. While the approach is generically applicable, in the experimental phase we focused on the Python language. For evaluating our approach, we conducted a comprehensive experimental campaign, involving 5 different LLMs and comparing GI with the self-correction ability of the LLMs. Our experimental analysis carried out on 25 problems sourced from the PSB2 dataset, showing the effectiveness of our method in enhancing the accuracy of Python code generated by the most advanced LLMs currently available.

As future research directions, we can study the effect of different specialization techniques, possibly also *reducing* the grammar according to the LLM outputs. This would allow to both use a large grammar for the initial parsing, thus increasing the number of correctly parsed solutions, and reducing the size of the search space for the GI phase. Finally, an analysis on the number of test cases needed to instruct these code correction methods would be beneficial to highlight the actual effort required by the user.

References

1. An, G., Blot, A., Petke, J., Yoo, S.: PyGGI 2.0: language independent genetic improvement framework. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1100–1104 (2019)
2. Austin, J., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
3. Bahrini, A., et al.: ChatGPT: applications, opportunities, and threats. In: 2023 Systems and Information Engineering Design Symposium (SIEDS), pp. 274–279 (2023)
4. Bibel, W.: Syntax-directed, semantics-supported program synthesis. Artif. Intell. **14**(3), 243–261 (1980)

5. Blot, A., Petke, J.: MAGPIE: machine automated general performance improvement via evolution of software. arXiv preprint arXiv:2208.02811 (2022)
6. Brown, T.B., et al.: Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020)
7. Budinsky, F.J., Finnie, M.A., Vlissides, J.M., Yu, P.S.: Automatic code generation from design patterns. IBM Syst. J. **35**(2), 151–171 (1996)
8. Chen, M., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
9. Chen, T., et al.: {TVM}: an automated {End-to-End} optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), pp. 578–594 (2018)
10. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2019)
11. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., O'Neill, M.: PonyGE2: grammatical evolution in Python. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1194–1201 (2017)
12. Fernando, C., Banarse, D., Michalewski, H., Osindero, S., Rocktäschel, T.: Promptbreeder: self-referential self-improvement via prompt evolution. arXiv preprint arXiv:2309.16797 (2023)
13. Grootendorst, M.: KeyBERT: minimal keyword extraction with BERT. Zenodo (2020)
14. Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis. Found. Trends® Program. Lang. **4**(1–2), 1–119 (2017)
15. Guo, Q., et al.: Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. arXiv preprint arXiv:2309.08532 (2023)
16. Helmuth, T., Kelly, P.: PSB2: the second program synthesis benchmark suite. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 785–794 (2021)
17. Helmuth, T., Kelly, P.: Applying genetic programming to PSB2: the next generation program synthesis benchmark suite. Genet. Program Evolvable Mach. **23**(3), 375–404 (2022)
18. Holm, S.: A simple sequentially rejective multiple test procedure. Scand. J. Stat. 65–70 (1979)
19. Karpuzcu, U.R.: Automatic Verilog code generation through grammatical evolution. In: Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation, pp. 394–397 (2005)
20. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Stat. Comput. **4**, 87–112 (1994)
21. Kruskal, W.H., Wallis, W.A.: Use of ranks in one-criterion variance analysis. J. Am. Stat. Assoc. **47**(260), 583–621 (1952)
22. Langdon, W.B.: Genetic improvement of programs. In: 2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 14–19. IEEE (2014)
23. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. arXiv preprint arXiv:2305.01210 (2023)
24. Liu, Z., Tang, Y., Luo, X., Zhou, Y., Zhang, L.F.: No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. arXiv preprint arXiv:2308.04838 (2023)

25. Liu, Z., Dou, Y., Jiang, J., Xu, J.: Automatic code generation of convolutional neural networks in FPGA implementation. In: 2016 International Conference on Field-Programmable Technology (FPT), pp. 61–68. IEEE (2016)
26. Liventsev, V., Grishina, A., Härmä, A., Moonen, L.: Fully autonomous programming with large language models. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2023, pp. 1146–1155. Association for Computing Machinery, New York (2023)
27. Löppenberg, M., Schwung, A.: Self optimisation and automatic code generation by evolutionary algorithms in PLC based controlling processes. arXiv preprint arXiv:2304.05638 (2023)
28. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**(1), 50–60 (1947)
29. Manna, Z., Waldinger, R.: Knowledge and reasoning in program synthesis. Artif. Intell. **6**(2), 175–208 (1975)
30. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM **14**(3), 151–165 (1971)
31. Marino, F., Squillero, G., Tonda, A.: A general-purpose framework for genetic improvement. In: Handl, J., Hart, E., Lewis, P.R., López-Ibáñez, M., Ochoa, G., Paechter, B. (eds.) PPSN 2016. LNCS, vol. 9921, pp. 345–352. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45823-6_32
32. Menabrea, L.F.: Sketch of the analytical engine invented by Charles Babbage, ESQ. In: Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age (1843)
33. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Proceedings of the 2nd Symposium on Information and Communication Technology, pp. 179–188 (2011)
34. Miller, J.F., Harding, S.L.: Cartesian genetic programming. In: Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 2701–2726 (2008)
35. Moreira, T.G., Wehrmeister, M.A., Pereira, C.E., Petin, J.F., Levrat, E.: Automatic code generation for embedded systems: from UML specifications to VHDL code. In: 2010 8th IEEE International Conference on Industrial Informatics, pp. 1085–1090. IEEE (2010)
36. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Trans. Evol. Comput. **5**(4), 349–358 (2001)
37. OpenAI: GPT-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
38. Ouyang, S., Zhang, J.M., Harman, M., Wang, M.: LLM is like a box of chocolates: the non-determinism of ChatGPT in code generation. arXiv preprint arXiv:2308.02828 (2023)
39. Paolone, G., Marinelli, M., Paesani, R., Di Felice, P.: Automatic code generation of MVC web applications. Computers **9**(3), 56 (2020)
40. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., et al. (eds.) EuroGP 2014. LNCS, vol. 8599, pp. 137–149. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44303-3_12
41. Pluhacek, M., Kazikova, A., Kadavy, T., Viktorin, A., Senkerik, R.: Leveraging large language models for the generation of novel metaheuristic optimization algorithms. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, pp. 1812–1820 (2023)

42. Rugina, A.E., Thomas, D., Olive, X., Veran, G.: Gene-auto: automatic software code generation for real-time embedded systems. DASIA 2008-Data Syst. Aerosp. **665**, 28 (2008)
43. Ryan, C., Collins, J.J., Neill, M.O.: Grammatical evolution: evolving programs for an arbitrary language. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, pp. 83–96. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055930
44. Sandnes, F.E., Megson, G.M.: A hybrid genetic algorithm applied to automatic parallel controller code generation. In: Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, pp. 70–75. IEEE (1996)
45. Serruto, W.F., Casas, L.A.: Automatic code generation for microcontroller-based system using multi-objective linear genetic programming. In: 2017 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 279–285. IEEE (2017)
46. 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 Genetic and Evolutionary Computation Conference, pp. 1019–1027 (2022)
47. Sun, H., Nie, Y., Li, X., Huang, M., Tian, J., Kong, W.: An automatic code generation method based on sequence generative adversarial network. In: 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC), pp. 383–390. IEEE (2022)
48. Taori, R., et al.: Alpaca: a strong, replicable instruction-following model. Stanford Center for Research on Foundation Models **3**(6), 7 (2023). https://crfm.stanford.edu/2023/03/13/alpaca.html
49. Touvron, H., et al.: LLaMA: open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)
50. Touvron, H., et al.: LLaMA 2: open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)
51. Vaithilingam, P., Zhang, T., Glassman, E.L.: Expectation vs experience: evaluating the usability of code generation tools powered by large language models. In: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA 2022. Association for Computing Machinery, New York (2022)
52. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
53. Walker, J.A., Liu, Y., Tempesti, G., Tyrrell, A.M.: Automatic code generation on a MOVE processor using cartesian genetic programming. In: Tempesti, G., Tyrrell, A.M., Miller, J.F. (eds.) ICES 2010. LNCS, vol. 6274, pp. 238–249. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15323-5_21
54. Wang, Y., et al.: Self-instruct: aligning language models with self-generated instructions. arXiv preprint arXiv:2212.10560 (2023)
55. Ward, M.: Proving program refinements and transformations. Ph.D. thesis, University of Oxford (1989)
56. Zhang, Y., Li, Y., Wang, X.: An optimized hybrid evolutionary algorithm for accelerating automatic code optimization. In: Third International Seminar on Artificial Intelligence, Networking, and Information Technology (AINIT 2022), vol. 12587, pp. 488–496. SPIE (2023)
57. Zheng, L., et al.: Ansor: generating {High-Performance} tensor programs for deep learning. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020), pp. 863–879 (2020)