Improving LLM-Generated Code via Genetic Improvement: A Summary of Recent Advances

Giovanni Pinna[1], Damiano Ravalico[1], Luigi Rovito[2], Luca Manzoni[1] and Andrea De Lorenzo[2]

[1]Department of Mathematics, Informatics and Geosciences, University of Trieste, Via Valerio 12/1, Trieste, 34127, Italy
[2]Department of Engineering and Architecture, University of Trieste, Via Valerio 6/1, Trieste, 34127, Italy

Abstract

Recent advancements in Natural Language Processing (NLP) have enabled Large Language Models (LLMs) to generate source code from textual problem descriptions. However, even state-of-the-art LLMs frequently produce syntactically correct which is, however, semantically incorrect. The LLM generated code might have a correct general structure, while treating incorrectly some corner cases. Furthermore, existing test cases might be difficult to use to help the LLM guide the construction of a semantically correct solution. To address this limitation, two recent studies propose the application of Genetic Improvement (GI) techniques, particularly Grammatical Evolution (GE), to automatically enhance the correctness of LLM-generated code. These approaches demonstrate that evolutionary computation can substantially improve the functional validity of code produced by both open-source and proprietary LLMs.

Keywords

Language Models, LLMs, Genetic Improvement, Grammatical Evolution, Evolutionary Computation

1. Introduction

The automatic generation of source code from natural language descriptions is a long-standing goal in software engineering. With the emergence of LLMs such as OpenAI's GPT-4, Meta's LLaMA, and Gemini, researchers have made significant strides in bridging the gap between human intentions and machine-readable code. Despite these advances, LLM-generated code often fails to satisfy functional correctness, particularly when confronted with ambiguous prompts or complex problem requirements.

Addressing this challenge, recent research explores the integration of evolutionary computation techniques with LLMs to enhance the quality of generated programs. Genetic Improvement (GI) leverages evolutionary algorithms to iteratively refine existing code, guided by fitness metrics derived from test cases. This combination offers a promising avenue for improving both the correctness and robustness of code synthesized by LLMs [1].

2. Related Works

Automatic code generation spans decades of research, from early studies on program synthesis and transformation [2, 3, 4] to more recent advances using deep learning and LLMs [5, 6]. Techniques include rule-based approaches, GANs, formal methods, and evolutionary algorithms. Evolutionary Computation, particularly Genetic Programming (GP) [7] and Grammatical Evolution (GE) [8], has been employed for tasks such as code optimization [9], hardware synthesis [10], and control system generation [11].

With the rise of LLMs like Codex and ChatGPT, neural models have become dominant in code generation. However, these models often struggle with functional correctness, leading to interest in hybrid solutions. Benchmarks like PSB2 [12] and HUMANEVAL [13] have helped compare LLM-based and evolutionary methods.

Ital-IA 2025: 5th National Conference on Artificial Intelligence, organized by CINI, June 23-24, 2025, Trieste, Italy
giovanni.pinna@phd.units.it (G. Pinna); damiano.ravalico@phd.units.it (D. Ravalico); luigi.rovito@units.it (L. Rovito); lmanzoni@units.it (L. Manzoni); andrea.delorenzo@units.it (A. D. Lorenzo)

GI has been adopted to address a variety of tasks, including network usage reduction [14], performance optimization [15], and software transplantation [16]. Recently, GI has been successfully applied to enhance LLM outputs, and new methods evolve not only code but also prompts [17, 18]. Recent research has also explored the use of LLMs to generate patches or fixes directly [19]. While effective, such methods often lack robustness. Hybrid approaches combining LLMs and evolutionary strategies still remain underexplored.

Studies in prompt engineering [20] and benchmark evaluation [13] highlight both the strengths and brittleness of LLMs in coding tasks. GI offers a principled post-processing mechanism to improve robustness and correctness. Our approach differs from prior grammar-constrained GI [21] by focusing on semantic correctness through test-driven evolution.

3. Methodology

The proposed method consists of three main phases:

Code Extraction and Preprocessing The user supplies a natural language problem description, which is passed to an LLM. The LLM produces a textual answer that should include the code needed to solve the given problem. From the LLM output, the actual code is extracted using delimiters and cleaned of comments and extraneous text. Since the extracted code may contain syntax errors, we apply a simple heuristic to recover a valid program: (i) identify the first line causing a syntax error, (ii) remove it, and (iii) repeat until the code is syntactically correct or empty (return a default syntactically-valid empty program in this latter case). In the end, we parse the cleaned code into an Abstract Syntax Tree (AST). Since function parameter names and variable names do not affect the semantic correctness of the program, they are renamed as v_i with i ranging from 0 to the number of variables minus one.

Dynamic Grammar Specialisation A base grammar in Backus-Naur Form (BNF) for the target programming language is extended dynamically. Constants, keywords, and function names are dynamically extracted from both the problem description and the initial LLM-generated code. In particular, we use KeyBERT [22] to search for relevant keywords in the problem description. Then, the extracted constants, keywords, and function names are added to the initial base grammar to create a new, extended grammar that will be used to parse the initial LLM solutions. This ensures that evolved candidates are tailored to the specific problem domain. Libraries used in the LLM output are also incorporated to promote the reuse of reliable routines.

Evolutionary Optimization To refine LLM-generated code, we adopt a GP variant tailored for code edits, that is, GE. GE is applied using the specialized grammar. The initial population consists of multiple copies of the extracted code. During the evolution, each individual in the population represents a program variant obtained via AST-level transformations. A lexicase selection mechanism is used to select candidates that perform well on diverse test cases. A fitness function computes the number of failing test cases, optionally penalizing the magnitude of output deviations. Mutation and crossover operators are applied at the AST level to preserve syntactic correctness. These edits are defined as localized sub-tree replacements or insertions. These modifications often target semantically relevant nodes, such as conditionals, function calls, or loop constructs. The process iterates over multiple generations to produce functionally correct solutions.

4. Experiments and Results

To evaluate the effectiveness of GI in correcting errors in LLM-generated code, we conducted a series of experiments on the PSB2 benchmark suite, which contains 25 diverse coding problems. Five LLMs are used: Alpaca 7B/13B, LLaMA 2 13B, ChatGPT, and GPT-4. For each problem and model, we generate 10 candidate programs using the same textual prompt composed by the following request:

```
Write a single Python function to solve the following problem and
insert the necessary modules:
```

followed by the problem description. These programs are used to initialize the GI process.

Each generated code sample was evaluated for correctness based on task-specific test sets. When a test case failed, we applied our tree-based GI approach to attempt automatic repair.

Each GI run uses a population size of 1000 individuals and evolves them over 100 generations. A subset of Python is used as the target language. The test sets for each problem consist of 1000 examples, and fitness is measured based on the number of correct outputs. Lexicase selection and AST-based genetic operators improve diversity and precision.

Our method improves code correctness for cases where the initial LLM outputs are partially or completely incorrect. On average, across all models, GI improves the number of passed test cases by 50–75%. Notably, even state-of-the-art models like GPT-4 benefit from GI when their outputs are structurally sound but logically flawed. The improvements are statistically significant, confirming the effectiveness of combining GI with LLMs. A comparison with a self-correction baseline—where the LLM is re-prompted with failed test cases—shows that GI consistently outperforms such strategies without additional API calls or fine-tuning.

While the combination of GI and LLMs shows promise, several limitations must be acknowledged:

1. **Oracle Dependency**: GI effectiveness heavily depends upon the quality and coverage of the test cases suite. Incomplete or missing test cases may lead to overfitting or insufficient improvement;
2. **Scalability**: As program size and complexity increase, the search space becomes increasingly large. Techniques like localization or grammar-based constraints help but do not fully resolve this issue;
3. **LLM Bias**: LLMs often generate plausible but incorrect code, leading to deceptive baselines that misguide the fitness function or mutations during the GI process;
4. **Lack of Guarantees**: GI may introduce semantic changes that pass tests but violate specifications or security policies.

5. Conclusions

The combination of LLMs and GI constitutes a compelling approach to automated program synthesis. By leveraging evolutionary search within a dynamically specialized grammar, the proposed method improves the semantic correctness of LLM-generated code without modifying the LLM itself. The enhancements introduced in the extended study—namely, lexicase selection and a granular fitness function—further strengthen the generalization of the framework capabilities and computational efficiency.

Future research directions include extending the approach to additional programming languages, integrating automated grammar extraction for new domains, and exploring the co-evolution of prompts and code. Moreover, incorporating developer feedback during the search process may guide mutation towards more relevant or explainable changes. Nonetheless, augmenting GI with interpretable edits that explain *why* a repair was applied could enhance trust and adoption. Beyond correctness, future GI frameworks could optimize for code readability, performance, and security, by using multi-objective optimization.

As LLMs continue to evolve, complementary methods like GI will play a crucial role in ensuring the correctness, reliability, and usability of AI-generated code.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] G. Pinna, D. Ravalico, L. Rovito, L. Manzoni, A. De Lorenzo, Enhancing large language models-based code generation by leveraging genetic improvement, in: European Conference on Genetic Programming (Part of EvoStar), Springer, 2024, pp. 108–124.

[2] S. Gulwani, O. Polozov, R. Singh, et al., Program synthesis, Foundations and Trends® in Programming Languages 4 (2017) 1–119.

[3] Z. Manna, R. J. Waldinger, Toward automatic program synthesis, Communications of the ACM 14 (1971) 151–165.

[4] M. Ward, Proving program refinements and transformations, Ph.D. thesis, University of Oxford PhD Thesis, 1989.

[5] A. Bahrini, M. Khamoshifar, H. Abbasimehr, R. J. Riggs, M. Esmaeili, R. M. Majdabadkohne, M. Pasehvar, Chatgpt: Applications, opportunities, and threats, in: 2023 Systems and Information Engineering Design Symposium (SIEDS), 2023, pp. 274–279.

[6] P. Vaithilingam, T. Zhang, E. L. Glassman, 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 '22, Association for Computing Machinery, New York, NY, USA, 2022.

[7] J. R. Koza, Genetic programming as a means for programming computers by natural selection, Statistics and computing 4 (1994) 87–112.

[8] M. O'Neill, C. Ryan, Grammatical evolution, IEEE Transactions on Evolutionary Computation 5 (2001) 349–358.

[9] Y. Zhang, Y. Li, X. Wang, An optimized hybrid evolutionary algorithm for accelerating automatic code optimization, in: Third International Seminar on Artificial Intelligence, Networking, and Information Technology (AINIT 2022), volume 12587, SPIE, 2023, pp. 488–496.

[10] U. R. Karpuzcu, Automatic verilog code generation through grammatical evolution, in: Proceedings of the 7th annual workshop on Genetic and evolutionary computation, 2005, pp. 394–397.

[11] M. Löppenberg, A. Schwung, Self optimisation and automatic code generation by evolutionary algorithms in plc based controlling processes, in: 2023 IEEE 21st International Conference on Industrial Informatics (INDIN), 2023, pp. 1–6. doi:10.1109/INDIN51400.2023.10218168.

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

[13] M. Chen, J. Tworek, et al., Evaluating large language models trained on code, 2021. URL: https://arxiv.org/abs/2107.03374. arXiv:2107.03374.

[14] J. Callan, W. B. Langdon, J. Petke, On reducing network usage with genetic improvement, in: 13th International Workshop on Genetic Improvement@ ICSE, 2024.

[15] W. B. Langdon, M. Harman, Optimizing existing software with genetic programming, IEEE Transactions on Evolutionary Computation 19 (2015) 118–135. doi:10.1109/TEVC.2013.2281544.

[16] J. Petke, M. Harman, W. B. Langdon, W. Weimer, Using genetic improvement and code transplants to specialise a c++ program to a problem class, in: Genetic Programming: 17th European Conference, EuroGP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers 17, Springer, 2014, pp. 137–149.

[17] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, Y. Yang, Connecting large language models with evolutionary algorithms yields powerful prompt optimizers, 2024. URL: https://arxiv.org/abs/2309.08532. arXiv:2309.08532.

[18] A. Chen, D. Dohan, D. So, Evoprompting: language models for code-level neural architecture search, Advances in Neural Information Processing Systems 36 (2024).
[19] V. Liventsev, A. Grishina, A. Härmä, L. Moonen, Fully autonomous programming with large language models, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 1146–1155.
[20] M. Saletta, C. Ferretti, Exploring the prompt space of large language models through evolutionary sampling, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 1345–1353. URL: https://doi.org/10.1145/3638529.3654049. doi:10.1145/3638529.3654049.
[21] N. Tao, A. Ventresque, V. Nallur, T. Saber, Grammar-obeying program synthesis: A novel approach using large language models and many-objective genetic programming, Computer Standards & Interfaces 92 (2025) 103938.
[22] M. Grootendorst, Keybert: Minimal keyword extraction with bert, Zenodo (2020).