Enhancing Genetic Improvement Mutations Using Large Language Models

Alexander E.I. Brownlee[1][0000−0003−2892−5059], James Callan[2][0000−0002−5692−6203], Karine Even-Mendoza[3][0000−0002−3099−1189], Alina Geiger[4][0009−0002−3413−283X], Carol Hanna[2][0009−0009−7386−1622], Justyna Petke[2][0000−0002−7833−6044], Federica Sarro[2][0000−0002−9146−442X], and Dominik Sobania[4][0000−0001−8873−7143]

[1] University of Stirling, UK alexander.brownlee@stir.ac.uk
[2] University College London, UK
[3] King's College London, UK
[4] Johannes Gutenberg University Mainz, Germany

Abstract. Large language models (LLMs) have been successfully applied to software engineering tasks, including program repair. However, their application in search-based techniques such as Genetic Improvement (GI) is still largely unexplored. In this paper, we evaluate the use of LLMs as mutation operators for GI to improve the search process. We expand the *Gin* Java GI toolkit to call OpenAI's API to generate edits for the JCodec tool. We randomly sample the space of edits using 5 different edit types. We find that the number of patches passing unit tests is up to 75% higher with LLM-based edits than with standard Insert edits. Further, we observe that the patches found with LLMs are generally less diverse compared to standard edits. We ran GI with local search to find runtime improvements. Although many improving patches are found by LLM-enhanced GI, the best improving patch was found by standard GI.

Keywords: Large language models · Genetic Improvement

1 Introduction

As software systems grow larger and more complex, significant manual effort is required to maintain them [2]. To reduce developer effort in software maintenance and optimization tasks, automated paradigms are essential. Genetic Improvement (GI) [15] applies search-based techniques to improve non-functional properties of existing software such as execution time as well as functional properties like repairing bugs. Although GI has had success in industry [12, 13], it remains limited by the set of mutation operators it employs in the search [14].

Large Language Models (LLMs) have a wide range of applications as they are able to process textual queries without additional training for the particular task at hand. LLMs have been pre-trained on millions of code repositories spanning many different programming languages [5]. Their use for software engineering tasks has had great success [9], showing promise also for program repair [17, 19].

Kang and Yoo [10] have suggested that there is untapped potential in using LLMs to enhance GI. GI uses the same mutation operators for different optimization tasks. These operators are hand-crafted prior to starting the search and thus result in a limited search space. We hypothesize that augmenting LLM patch suggestions as an additional mutation operator will enrich the search space and result in more successful variants.

In this paper, we conduct several experiments to explore whether using LLMs as a mutation operator in GI can improve the efficiency and efficacy of the search. Our results show that the LLM-generated patches have compilation rates of 51.32% and 53.54% for random search and local search, respectively (with the MEDIUM prompt category). Previously LLMs (using an LLM model as-is) were shown to produce code that compiled roughly 40% of the time [16, 18]. We find that randomly sampled LLM-based edits compiled and passed unit tests more often compared to standard GI edits. We observe that the number of patches passing unit tests is up to 75% higher for LLM-based edits than GI INSERT edits. However, we observe that patches found with LLMs are less diverse. For local search, the best improvement is achieved using standard GI STATEMENT edits, followed by LLM-based edits. These findings demonstrate the potential of LLMs as mutation operators and highlight the need for further research in this area. To the best of our knowledge this is the first work investigating the use of LLMs for enhancing GI mutations. We refer the reader to the survey by Fan et al. [6] for other uses of LLMs in Software Engineering .

2 Experimental Setup

To analyze the use of LLMs as a mutation operator in GI, we used the GPT 3.5 Turbo model by OpenAI and the GI toolbox Gin [3]. We experimented with two types of search implemented within Gin: random search and local search. Requests to the LLM using the OpenAI API were via the Langchain4J library, with a temperature of 0.7. The target project for improvement in our experiments was the popular JCodec [7] project which is written in Java. 'Hot' methods to be targeted by the edits were identified using Gin's profiler tool by repeating the profiling 20 times and taking the union of the resulting set.

For the random sampling experiments, we set up the runs with statement-level edits (copy/delete/replace/swap from [14] and insert break/continue/return from [4]) and LLM edits, generating 1000 of each type at random. A timeout of 10000 milliseconds was used for each unit test to catch infinite loops introduced by edits; exceeding the timeout counts as a test failure. For local search, experiments were set up similarly. There were 10 repeat runs (one for each of the top 10 hot methods) but the runs were limited to 100 evaluations resulting in 1000 evaluations in total, matching the random search. In practice this was 99 edits per run as the first was used to time the original unpatched code.

We experimented with three different prompts for sending requests to the LLM for both types of search: a SIMPLE prompt, a MEDIUM prompt, and a DETAILED prompt. With all three prompts, our implementation requests five

```
Give me 5 different Java implementations of this method body:
```
<code>
```
This code belongs to project <projectname>.
Wrap all code in curly braces, if it is not already.
Do not include any method or class declarations.
label all code as java.
```

Fig. 1. The MEDIUM prompt for LLM requests, with line breaks added for readability.

different variations of the code at hand. The SIMPLE prompt only requests the code without any additional information. The MEDIUM prompt provides more information about the code provided and the requirements, as shown in Figure 1. Specifically, we provide the LLM with the programming language used, the project that the code belongs to, as well as formatting instructions. The DETAILED prompt extends the MEDIUM prompt with an example of a useful change. This example was taken from results obtained by Brownlee et al. [4]. The patch is a successful instance of the INSERT edit applied to the jCodec project (i.e., an edit that compiled, passed the unit tests and offered a speedup over the original code). We use the same example for all the DETAILED prompt requests used in our experiments; this is because LLMs are capable of inductive reasoning where the user presents specific information, and the LLM can use that input to generate more general statements, further improved in GPT-4 [8].

LLM edits are applied by selecting a block statement at random in a target 'hot' method. This block's content is `<code>` in the prompt. The first code block in the LLM response is identified. Gin uses JavaParser (https://javaparser.org) internally to represent target source files, so we attempt to parse the LLM suggestion with JavaParser, and replace the original block with the LLM suggestion.

3 Results

The first experiment compares standard GI mutations, namely INSERT and STATEMENT edits, with LLM edits using differently detailed prompts (SIMPLE, MEDIUM, and DETAILED) using Random Sampling. Table 1 shows results for all patches as well as for unique patches only. We report how many patches were successfully parsed by JavaParser (named as Valid), how many compiled, and how many passed all unit tests (named as Passed). We excluded patches syntactically equivalent to the original software. Best results are in **bold**.

We see that although substantially more valid patches were found with the standard INSERT and STATEMENT edits, more passing patches could be found by using the LLM-generated edits. In particular, for the MEDIUM, and DETAILED prompts 292 and 230 patches passed the unit tests, respectively. For the INSERT and STATEMENT edits only 166 and 91 passed the unit tests, respectively. Anec-

Table 1. Results of our Random Sampling experiment. We exclude patches syntactically equivalent to the original software in this table. For all and unique patches we report: how many patches passed JavaParser, compiled, and passed all unit tests.

	Unique				All			
EditCategory	Patches	Valid	Compiled	Passed	Patches	Valid	Compiled	Passed
STATEMENT	896	**819**	199	80	967	**869**	227	91
INSERT	785	785	**284**	**161**	860	860	295	166
SIMPLE	193	0	0	0	1000	0	0	0
MEDIUM	324	260	183	154	645	463	**331**	**292**
DETAILED	332	268	126	110	606	456	250	230

Table 2. Local Search results. We exclude all empty patches. We report how many patches compiled, passed all unit tests, and how many led to improvements in runtime. We report best improvement found and median improvement among improving patches.

EditCategory	Patches	Compiled	Passed	ImpFound	BestImp(ms)	Median(ms)
STATEMENT	990	213	105	71	**508.0**	137.0
INSERT	948	414	264	136	313.0	81.0
SIMPLE	990	2	2	2	176.0	**137.5**
MEDIUM	990	**530**	**520**	164	395.0	75.5
DETAILED	990	379	369	**196**	316.0	95.0

dotally, the hot methods with lowest/highest patch pass rates differed for each operator: understanding this variation will be interesting for future investigation.

It is also notable that LLM patches are less diverse: over 50% more unique patches were found by standard mutation operators than the LLM using MEDIUM, and DETAILED prompts. With the SIMPLE prompt, however, not a single patch passed the unit tests, since the suggested edits often could not be parsed. Thus detailed prompts are necessary to force LLM to generate usable outputs.

We investigated further the differences between MEDIUM and DETAILED prompts to understand the reduction in performance with DETAILED (in the unique patches sets) as MEDIUM had a higher number of compiled and passed patches. In both prompt levels, the generated response was the same for 42 cases (out of the total unique valid cases). However, DETAILED tended to generate longer responses with an average of 363 characters, whereas MEDIUM had an average of 304 characters. We manually examined several DETAILED prompt responses, in which we identified some including variables from other files, potentially offering a significant expansion of the set of code variants GI can explore.

The second experiment expands our analysis, comparing the performance of the standard and LLM edits with Local Search. Table 2 shows the results of the Local Search experiment. We report the number of compiling and passing patches as well as the number of patches were runtime improvements were found. Furthermore, we report the median and best improvement in milliseconds (ms). In the table, we excluded all empty patches. As before, best results are in **bold**.

Again, we see that more patches passing the unit tests could be found with the LLM using the MEDIUM, and DETAILED prompts. In addition, more improve-

ments could be found by using the LLM with these prompts. Specifically, with MEDIUM and DETAILED, we found 164 and 196 improvements, respectively, while we only found 136 with INSERT and 71 with STATEMENT. The best improvement could be found with 508 ms with the STATEMENT edit. The best improvement found using LLMs (using the MEDIUM prompt) was only able to improve the runtime by 395 ms. We also examined a series of edits in Local Search results to gain insights into the distinctions between MEDIUM and DETAILED prompts due to the low compilation rate of DETAILED prompt's responses. In the example, a sequence of edits aimed to inline a call to function CLIP. The DETAILED prompt tried to incorporate the call almost immediately within a few edits, likely leading to invalid code. On the other hand, the MEDIUM prompt made less radical changes, gradually refining the code. It began by replacing the ternary operator expression with an if-then-else statement and system function calls before eventually attempting to inline the CLIP function call.

4 Conclusions and Future Work

Genetic improvement of software is highly dependent on the mutation operators it utilizes in the search process. To diversify the operators and enrich the search space further, we incorporated a Large Language Model (LLM) as an operator.

Limitations. To generalise, future work should consider projects besides our target, jCodec. Our experiments used an API giving us no control over the responses generated by the LLM or any way of modifying or optimizing them. Though we did not observe changes in behaviour during our experiments, OpenAI may change the model at any time, so future work should consider local models. We experimented with only three prompt types for LLM requests and within this limited number of prompts found a variation in the results. Finally, our implementation for parsing the responses from the LLMs was relatively simplistic. However, this would only mean that our reported results are pessimistic and an even larger improvement might be achieved by the LLM-based operator.

Summary. We found that, although more valid and diverse patches were found with standard edits using Random Sampling, more patches passing the unit tests were found with LLM-based edits. For example, with the LLM edit using the MEDIUM prompt, we found over 75% more patches passing the unit tests than with the classic INSERT edit. In our Local Search experiment, we found the best improvement with the STATEMENT edit (508 ms). The best LLM-based improvement was found with the MEDIUM prompt (395 ms). Thus there is potential in exploring approaches combining both LLM and 'classic' GI edits.

Our experiments revealed that the prompts used for LLM requests greatly affect the results. Thus, in future work, we hope to experiment more with prompt engineering. It might also be helpful to mix prompts: e.g., starting with MEDIUM then switching to DETAILED to make larger edits that break out of local minima. Further, the possibility of combining LLM edits with others such as standard copy/delete/replace/swap or PAR templates [11] could be interesting. Finally, we hope to conduct more extensive experimentation on additional test programs.

Data Availability. The code, LLMs prompt and experimental infrastructure, data from the evaluation, and results are available as open source at [1]. The code is also under the 'llm' branch of github.com/gintool/gin (commit 9fe9bdf; branched from master commit 2359f57 pending full integration with Gin).

Acknowledgements. This work was supported by the UKRI EPSRC grant no. EP/P023991/1 and the ERC advanced fellowship grant no. 741278.

References

1. Artifact of Enhancing Genetic Improvement Mutations Using Large Language Models. Zenodo (Sep 2023). https://doi.org/10.5281/zenodo.8304433
2. Böhme, M., Soremekun, E.O., Chattopadhyay, S., Ugherughe, E., Zeller, A.: Where is the bug and how is it fixed? An experiment with practitioners. In: Proc. ACM Symposium on the Foundations of Software Engineering. pp. 117–128 (2017)
3. Brownlee, A.E., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: genetic improvement research made easy. In: GECCO. pp. 985–993 (2019)
4. Brownlee, A.E., Petke, J., Rasburn, A.F.: Injecting shortcuts for faster running Java code. In: IEEE CEC 2020. p. 1–8
5. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
6. Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., Zhang, J.M.: Large language models for software engineering: Survey and open problems (2023)
7. Github - jcodec/jcodec: Jcodec main repo, https://github.com/jcodec/jcodec
8. Han, S.J., Ransom, K.J., Perfors, A., Kemp, C.: Inductive reasoning in humans and large language models. Cognitive Systems Research p. 101155 (2023)
9. Hou, X., Liu, Y., Yang, Z., Grundy, J., Zhao, Y., Li, L., Wang, K., Luo, X., Lo, D., Wang, H.: Large language models for software engineering: A systematic literature review. arXiv:2308.10620 (2023)
10. Kang, S., Yoo, S.: Towards objective-tailored genetic improvement through large language models. arXiv:2304.09386 (2023)
11. Kim, D., Nam, J., Song, J., Kim, S.: Automatic Patch Generation Learned from Human-Written Patches (2013), http://logging.apache.org/log4j/
12. Kirbas, S., Windels, E., Mcbello, O., Kells, K., Pagano, M., Szalanski, R., Nowack, V., Winter, E., Counsell, S., Bowes, D., Hall, T., Haraldsson, S., Woodward, J.: On the introduction of automatic program repair in bloomberg. IEEE Software **38**(4), 43–51 (2021)
13. Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., Scott, A.: Sapfix: Automated end-to-end repair at scale. In: ICSE-SEIP. pp. 269–278 (2019)
14. Petke, J., Alexander, B., Barr, E.T., Brownlee, A.E., Wagner, M., White, D.R.: Program transformation landscapes for automated program modification using Gin. Empirical Software Engineering **28**(4), 1–41 (2023)
15. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: A comprehensive survey. IEEE Transactions on Evolutionary Computation **22**, 415–432 (2018)

16. Siddiq, M.L., Santos, J., Tanvir, R.H., Ulfat, N., Rifat, F.A., Lopes, V.C.: Exploring the effectiveness of large language models in generating unit tests. arXiv preprint arXiv:2305.00418 (2023)
17. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. In: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR). pp. 23–30. IEEE Computer Society (2023)
18. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., Zhang, L.: Universal fuzzing via large language models. arXiv preprint arXiv:2308.04748 (2023)
19. Xia, C.S., Zhang, L.: Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. arXiv preprint arXiv:2304.00385 (2023)