2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW) | 979-8-3503-1956-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSREW60843.2023.00040

Automatic Bug Fixing via Deliberate Problem Solving with Large Language Models

Guoyang Weng
Heidelberg University & SAP SE
Heidelberg, Germany
Email: guoyang.weng@sap.com

Artur Andrzejak
Heidelberg University
Heidelberg, Germany
Email: artur.andrzejak@uni-heidelberg.de

***Abstract*—Developers dedicate a significant share of their activities to finding and fixing defects in their code. Automated program repair (APR) attempts to reduce this effort by a set of techniques for automatically fixing errors or vulnerabilities in software systems. Recent Large Language Models (LLMs) such as GPT-4 offer an effective alternative to existing APR methods, featuring out-of-the-box bug fixing performance comparable to even sophisticated deep learning approaches such as CoCoNut. In this work we propose a further extension to LLM-based program repair techniques by leveraging a recently introduced interactive prompting technique called Tree of Thoughts (ToT). Specifically, we ask a LLM to propose multiple hypotheses about the location of a bug, and based on the aggregated response we prompt for bug fixing suggestions. A preliminary evaluation shows that our approach is able to fix multiple complex bugs previously unsolved by GPT-4 even with prompt engineering. This result motivates further exploration of hybrid approaches which combine LLMs with suitable meta-strategies.**

1. Introduction

Program defect detection and repair consume a considerable amount of developers' time. This predicament motivates the need for Automated Program Repair (APR) approaches that could identify and correct bugs automatically, reducing the load on developers [4], [5], [9]. With recent advancements in Large Language Models (LLMs) [1], [7], there's an opportunity to leverage these models in APR tasks. Study [10] shows that also in the case of APR prompt engineering can improve the rate of bug fixes. However, zero-shot prompting can fail to handle more intricate bug cases, as shown in this work.

To enhance the bug fixing performance of LLM models we propose an multistage bug fixing technique that we term as the "Multistage Bug Fixer (MBF)". Our approach leverages a recently introduced interactive prompting technique called Tree of Thoughts (ToT) [11] to split the bug fixing process into multiple steps, each involving generation of partial solutions and their evaluation via LLM. This strategy exploits the surprising fact that generative models like GPT-4 are still better in classification-like task than in token generation.

The MBF approach produces multiple hypotheses about the location of a bug and, based on aggregated responses, prompts for bug-fixing suggestions. Through this method, our approach is capable of correcting complex bugs that even GPT-4 failed to resolve using traditional prompt engineering.

Our findings indicate that the proposed multistage approach holds potential to enhance bug fixing capabilities of LLMs and lays the groundwork for further exploration of APR strategies combining LLMs and meta-strategies. Furthermore, we compare the bug fixing ability of ChatGPT (powered by GPT-3.5 model) and GPT-4 on the QuixBugs benchmark.

The complete source code and evaluation results are available in a GitHub repository[1]. We refer to this resource for the details of the description.

2. Approach

Multistage Bug Fixing. We describe here the proposed approach and its prototypical implementation as a tool *Multistage Bug Fixer (MBF)*. Our approach divides the program repair process into two phases, each involving multiple interactions with a LLM (in our case GPT-4).

In phase 1, we ask LLM for multiple hypotheses $E_1, \ldots, E_m$ about fault location and explanation and then let LLM vote on its own solutions. In phase 2, we ask LLM to generate multiple bug fix proposals $F_1, \ldots, F_m$ given the highest voted explanation E^* and the buggy code. Subsequently, LLM votes on the fix proposals and we output the highest rated proposal F^* as the final solution. Algorithm 1 shows a pseudocode of this approach.

The exact prompts used in each stage can be found in the repository in the directory `prompts/mbf`. Note that in the evaluation we provide as input in Steps 1a and 2a not only buggy code but also details of the expectation, for example, the unit tests, or specifically the inputs of the routine, expected output and the actual output. We used $m = 5$ and $n = 5$ as the number of explanations and the number of bug fixes, respectively.

Our algorithm can be interpreted as a ToT version using Breadth-First Search (BFS) algorithm with a depth of 2 and

1. https://github.com/guoyang-weng/apr-via-llm-metastrategy

979-8-3503-1956-9/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSREW60843.2023.00040

Algorithm 1 Program Repair Process

```
procedure MULTISTAGEBUGFIXER(buggyCode)
    Phase 1: Explanation Generation and Voting
    for i = 1 to m do                                   ▷ Step 1a
        E_i ← LLM_GENEXPLANATION(buggyCode)
    end for
    E* ← LLM_VOTEEXPLS(E_1, ..., E_m)                   ▷ Step 1b

    Phase 2: Bug Fix Generation and Voting
    for i = 1 to n do                                   ▷ Step 2a
        F_i ← LLM_GENFIX(E*, buggyCode)
    end for
    F* ← LLM_VOTEFIXES(F_1, ..., F_n)                   ▷ Step 2b
    return F*
end procedure
```

a breadth limit $b = 1$, keeping only one choice per step. A simple zero-shot vote prompt was used to sample $k = 5$ votes at both Steps 1b and 2b. This setup for the code fixer task is similar to the creative writing task mentioned in [11].

Our implementation leverages the code[2] accompanying the paper [11] which introduced the ToT concept.

Baseline Bug Fixer. We also evaluated a simple baseline approach for LLM-based APR called *Baseline Bug Fixer (BBF)*. Here, the LLM is asked for a code fix via a single prompt containing buggy code and an explanation of the expected program behavior. The prompt template can be found in our repository in the directory `prompts/bbf`.

3. Evaluation

Experimental settings. We use the QuixBugs dataset of 40 defective Python files [6], and an additional example *arithm_expr_eval.py*[3] from [3] with a manually introduced bug (see file `src/mbf/data/buggy_code/arithm_expr_eval.py` in our repository). We use GPT-4 (July/August 2023) with Azure OpenAI Services API via SAP Business Technology Platform (SAP BTP).

GPT-3.5 and Baseline Bug Fixer. We compare first the results of the baseline approach BBF against the results for QuixBugs stated in [10]. The latter study has essentially the same setup as BBF but obviously uses GPT-3.5 (results in [10] are from Jan. 2023, prior to availability of GPT-4).

According to [10], GPT-3.5 cannot solve 9 cases from QuixBugs listed in Table 1. Contrary to this, the BBF using GPT-4 can solve 7 of these 9 cases (using one trial in each case), and all of previously solved cases (we do not include unit test information for BBF). As for *arithm_expr_eval.py*, BBF could correct the manually inserted bug but introduced another bug leading to a test failure.

Overall, BBF could fix 38 out of 41 cases with a simple zero-shot prompting, and failed for 3 cases. Adding unit test information to BBF input did not help significantly for these 3 unsuccessful cases (only 2 successes in 15 trials). Directory `evaluation/bbf` shows an example interaction of BBF for the case *lcs-length*.

TABLE 1. BUG FIXING RESULTS FOR THE BASELINE APPROACH BBF VS. THE TOT-BASED APPROACH MBF. BUG CASES ARE FROM QUIXBUGS [6] EXCEPT FOR THE LAST ONE (MODIFIED) FROM [3]. COLUMN "MBF" SHOWS MBF RESULTS WITHOUT UNIT TEST INFORMATION IN THE PROMPT, AND THE LAST COLUMN WITH THIS INFORMATION.

Benchmark problem	BBF	MBF	MBF w/UT
kheapsort	✓		
lcs-length	✓		
lis	✓		
longest-common-subsequence	✓		
rpn-eval	✓		
shortest-path-lengths	✓		
to-base	✓		
wrap	×	(random)	✓
topological-ordering	×	(random)	✓
arithm_expr_eval	×	×	✓
Solved / Total	7/10	-/3	3/3

Multistage Bug Fixer. We evaluated the MBF approach on the 3 last functions of Table 1 for which BBF failed: *wrap*, *topological-ordering*, and *arithm_expr_eval*. Table 1 shows that MBF succeeded in all cases if the unit test information was added (last column). Without this information, results were less impressive and random: *arithm_expr_eval* always failed, and the other cases succeeded in about 50% of the trials. A complete log of an example MBF run can be found in the repository in `evaluation/mbf`.

Discussion of the results. The evaluation indicates that meta-strategies can improve the bug fixing performance of LLMs. However, note that the results are not deterministic or unstable [2], and it seems to be significant to provide additional information like unit test details. There are also performance and cost penalties of ToT-based approaches since multiple interactions per issue are needed.

4. Conclusions and Future Work

Despite of the limited scope, the conducted study demonstrates the potential of meta-strategies like Tree of Thoughts on top of LLMs for automated program repair. Our multistage approach could fix multiple cases for which direct prompting with GPT-4 has failed.

Future work will follow several threads. First, we will add information from validation and tracing execution. For example, for each explanation (results of Step 1a) we can supply LLM with information about the variable values in the suspicious code lines for failed vs. passed tests, similar to spectral methods in fault localization [8]. Second, results of unit tests can be exploited to optimize the approach by generating more hypotheses and trials for harder cases. Finally, inspired by the aider project[4], we will attempt using *ctags* to handle defects in large code projects.

2. github.com/princeton-nlp/tree-of-thought-llm
3. github.com/jilljenn/tryalgo/blob/master/tryalgo/arithm_expr_eval.py
4. https://aider.chat/docs/ctags.html

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. arXiv, 2005.14165.

[2] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time?, 2023. arXiv, 2307.09009.

[3] Christoph Dürr and Jill-Jênn Vie. *Competitive Programming in Python 128 Algorithms to Develop Your Coding Skills*. Cambridge University Press, 2020.

[4] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra. Automatic program repair. *IEEE Software*, 38(04):22–27, jul 2021.

[5] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[6] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *SPLASH 2017*, SPLASH Companion, page 55–56, New York, NY, USA, 2017.

[7] OpenAI. Gpt-4 technical report, 2023. arXiv, 2303.08774.

[8] Alexandre Perez, Rui Abreu, and Arie van Deursen. A theoretical and empirical analysis of program spectra diagnosability. *IEEE Trans. Software Eng.*, 47(2):412–431, 2021.

[9] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. A survey on machine learning techniques for source code analysis, 2022. arXiv, 2110.09610.

[10] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023. arXiv, 2301.08653.

[11] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate problem solving with large language models, 2023. arXiv, 2305.10601.