Using Large Language Models for Bug Localization and Fixing

Tung Do Viet
The University of Aizu
Fukushima, Japan
m5252105@u-aizu.ac.jp

Konstantin Markov
The University of Aizu
Fukushima, Japan
markov@u-aizu.ac.jp

***Abstract*—As part of their learning journey, students frequently encounter challenges and make errors, especially with algorithmic programming questions. Regrettably, providing tailored solutions for these mistakes can impose a significant burden on instructors in terms of time and effort. To address this, automated program repair (APR) techniques have been explored to generate such fixes automatically. Previous research has investigated the use of symbolic and neural approaches for APR in the educational domain. However, both types of approaches necessitate substantial engineering endeavors or extensive data and training. In this study, we propose the utilization of a large language model trained on code to construct an APR system specifically designed for student programs. Our system has the capability to rectify semantic errors by employing a few-shot example generation pipeline solely based on the input code. We assess the performance of our system on one dataset of algorithm implementations, namely QuixBugs. The results demonstrate that the novel example generation pipeline not only enhances the overall system's performance but also ensures its stability.**

***Index Terms*—bug localization, bug fixing, program repair, large language model. few-shot prompting, in-context learning**

I. Introduction

In recent years, programming education has seen significant growth, leading to the challenge of providing effective learning support to beginner students. Manual assistance from teaching assistants is often impractical on a large scale, driving the need for automated tools to offer tailored feedback for programming errors. Automated program repair (APR) has emerged as a field within software engineering [1], [2], introducing various methodologies [3], [4] to generate fixes for student mistakes in introductory assignments. The APR system aims to generate a patch that aligns with the given specifications, primarily defined by the instructor's test cases, while minimizing code alterations to enhance the learning experience for students. Automated systems for repairing student programming assignments have historically relied on two main approaches: symbolic techniques [3], [4] and neural methods [5]. Symbolic techniques require significant engineering effort and expertise in program analysis and repair, tailored to the specific programming language used by students. Neural methods, on the other hand, often require large amounts of data, making them more suitable for specialized applications like Massive Open Online Courses (MOOCs). Additionally, these systems may specialize in either syntax repair or semantic repair, with semantic repair assuming the code contains no syntactic errors.

We propose a prompting-based system using a large language model (LLM) to tackle bug localization and program repair challenges. Our approach uses pre-trained LLMs, avoiding the need for custom logic or retraining. In bug localization and program repair, we adopt the approach of using LLMs as proposed in [6]. To overcome stability challenges, we introduce a novel framework based on few-shot examples. The pipeline involves code summarization-generation and code modification. This enables generating question-related examples applicable to various problem classes.

We evaluate our system by applying it to each of the 2 chatbot LLM models: Galpaca [7] [8], GPT-3.5-turbo [9] on one student programming dataset: QuixBugs [10]. The programs in the dataset are filtered to contain only semantic mistakes. The accquired results has proven the effectiveness and efficiency of our approach where it dominates in most of the benchmarks. We also carried out two ablation studies to understand the impact of our design decisions. To our knowledge, the choice of building blocks for the current pipeline outperform all of the other variants.

To summarize, we make the following contributions

- We propose an approach to automatically repair mistakes in student programs using a large language model trained on code. Our approach follows a Socratic Models [11] to propagate messages through contiguous blocks to achieve the final goal. In contrast to prior work, our approach relies on no manual symbolic logic nor training data and thus can be applied to any domain.
- We implement our system based on an open source LLM and a popular commercial LLM. We then evaluate it on a dataset of real student programs drawn from an online programming challenge. We compare our few-shot prompting performance to baselines ranging from zero-shot to manual fixed few-shot (related and unrelated). The results show that our proposed method outperforms other baselines in most of the benchmarks. Thus, our novel approach becomes the default choice to design few-shot examples, especially in the case of bug localization/fixing.

The remainder of this paper is structured as follows, Section II introduces background and related works on the bug localization/fixing and large language models. After that, Section

III describes our proposed approach in details. Next, Section IV presents our experimental setup followed by the results and discussions from Section V. Finally, Section VI concludes our findings.

II. Background and Related Works

A. Bug Localization

Automated bug localization aims to identify bugs in programs accurately. It can be classified into different levels of complexity: bug detection, line-level bug localization, and token-level bug localization. Bug detection lacks real-world applications and is often unnecessary when test suites are available. Token-level localization can lead to many false positives, while line-level localization strikes a balance, receiving significant attention in program analysis [12].

Spectrum-based fault localization (SBFL) techniques [13] use execution traces of passing and failing tests to identify potentially faulty code elements. They calculate the suspiciousness of code elements using statistical analysis methods like Ochiai [12] and Tarantula [14]. However, SBFL may not directly associate a code element with the test failure. Mutation-based fault localization (MBFL) [15] addresses this limitation by mutating code elements and evaluating their impact on test outcomes. Researchers have explored other fault localization techniques including learning-based, data mining-based approaches [16], [17]. In the end, traditional bug localization methods rely on code-specific features like control flow, data flow, and ASTs, demanding significant engineering for representation design. This restricts their utility across domains and languages. To address this, we present a new bug localization approach that incorporates embedded programming knowledge from extensive repositories. This design avoids manual efforts and caters to diverse domains.

B. Program Repair/ Bug Fixing

Automated program repair (APR) is an emerging field focused on automatically rectifying programming errors [18]. These repair techniques take a faulty program and a correctness specification (typically a test suite) as inputs and generate a corrected program by making slight modifications to satisfy the given specification. Conventional repair tools employ different approaches for generating patches based on program semantics and the provided specification. Semantic-based repair tools, such as SemFix [19] and Angelix [20], utilize symbolic execution to produce patches. On the other hand, search-based repair tools like GenProg [21] and TBar [22] explore a predefined search space and leverage dynamic execution results to find correct patches. Thus, search-based conventional methods are limited in their search spaces which can not match the demand of a general program repair solution.

In addition to conventional methods, the utilization of neural machine translation (NMT) models has emerged as an alternative approach for program repair. Specifically, employing a robust language model, NMT-based APR techniques have exhibited superior performance compared to most rule-based approaches [23], [24]. NMT models use deep learning to encode flawed source code into a latent representation, which is then decoded into the correct target code. Through iterative loss minimization and weight updates, NMT models learn to grasp the connection between flawed and correct code, avoiding manual design of fixed patterns or features. However, source code's unique traits challenge NMT models in bug fixing [24]. Current methods lack repository-wide knowledge, relying on limited snippets. NMT-based APR lacks language syntax awareness, resulting in incomplete code comprehension. This can be solved by integrating deep models with ample knowledge, making large language models ideal for program repair [24].

C. Automated Program Repair Using Large Language Model Prompting

Expanding transformer-based language models, with factors like model size, data, and compute, boosts performance in downstream NLP tasks [25]. Large language models (LLMs) exhibit emergent capabilities, including few-shot learning, zero-shot problem solving, chain-of-thought reasoning, and instruction following [26]–[28]. LLMs with 100B+ parameters support few-shot learning via in-context prompts [25], ushering the ”pre-train and prompt” era. Prompting gains wide adoption for various NLP tasks [29].

Various endeavors have been made to perform bug localization and program repair using large language prompting techniques. In a recent study [6], Codex is employed as an assistant to process diverse forms of input, including code, code with hints, code with docstrings, docstrings alone, and input-output examples. This approach involves inserting the input between two comments: ”fix the bug in the following function” and ”fixed function” while requesting the desired output in subsequent lines. Although achieving satisfactory performance on the QuixBugs dataset [10], this prompting mechanism is restricted to a zero-shot prompting template. Additionally, the fact that Codex is not a chatbot leads to a clear distinction from few-shot prompting, thus overlooking a crucial aspect of prompting. Another simultaneous effort, known as REPEATNPR [30], leverages program dependence analysis to extract slicing-based contextual information, enhancing the APR task. Furthermore, this work introduces an ensemble framework that integrates multiple APR models through a filtering mechanism. Despite the sophisticated framework, REPEATNPR still relies on a dataset split for training. Given our objective of developing a domain-agnostic method without the need for training, we pursue an alternative path distinct from REPEATNPR.

III. Approach

Figure 1 describes the overview of our framework. To construct a set of in context examples for few-shot prompting, we first ask the LLM to summarize the buggy code. After that, we generate a presumably correct implementation of the summary. Subsequently, a chosen line is modified to result in a buggy version of the code. Finally, we construct the an example from the generated correct-buggy pairs of programs.

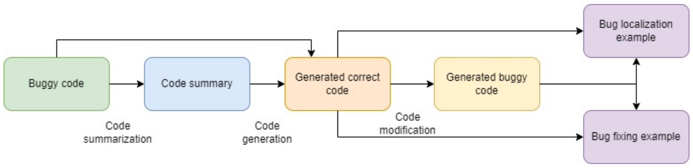


Fig. 1. Overview of our method

A. General Prompt Template

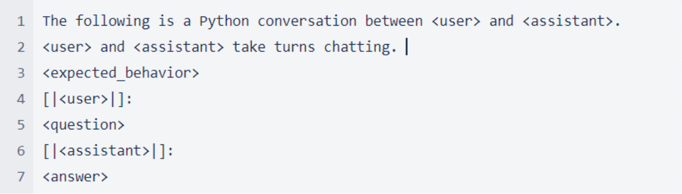
In this section, we describe the general prompt template for subsequent sections. Using chatbot models, prompts follow a consistent approach, including the example generation pipeline. Each prompt begins with a system message defining the LLM's role and behavior in a conversation with the user for a specific task (lines 1-3, Fig. 2). Depending on the task, we fill in the corresponding `<user>` and `<assistant>` roles. LLMs trained on different tasks receive distinct instructions, including these roles. We condition the LLM using the `<expected_behavior>` tag to specify the desired output type. The user's question, with a concise text description and code block, follows, separated using ```` ```python ```` tags for code parsing. For zero-shot prompting (default), the prompt fills slots from lines 1 to 6. For few-shot prompting, additional examples adhere to a similar format (lines 4 to 7, including the answer) and are inserted before line 4.

B. Bug Localization Prompting

In this section, we describe the process of input processing, prompt creation, and outputs parsing for the bug localization task. This process is applicable regardless of the presence of few-shot examples. For bug localization, we set `<expected_behavior>` as `Analyst always considers the previous query carefully. Analyst always extracts one buggy line in the provided Python code.` to ask the LLM for exactly one answer on the buggy line location, acting as a zero-shot chain-of-thought [28]. The question `Please analyze the following code and locate the buggy line.` aligns with common scenarios encountered in practice. After inputting the buggy program within backticks, we define the output format, including line index and content. The inferred content of the line is expected, not directly copied from the input buggy code (see Section IV-B). By explicitly specifying the desired format, we enable parsing all outputs by extracting text within backticks, even in cases of zero-shot prompting.

C. Program Repair Prompting

Similar to bug localization, we begin by replacing the extracting command in the `<expected_behavior>` by `Analyst always provides the corrected code after fixing the bug`. In addition to the buggy program, the LLM is given the ground-truth buggy line location (and its content). To complete the process, the output format entails another block of Python code.



```
1  The following is a Python conversation between <user> and <assistant>.
2  <user> and <assistant> take turns chatting.
3  <expected_behavior>
4  [|<user>|]:
5  <question>
6  [|<assistant>|]:
7  <answer>
```

Fig. 2. General prompt template (line 1-6) with the expected answer (line 7). Few-shot examples are inserted between line 3 and line 4.

D. Generate few-shot demonstrations

Here, we introduce our main contribution: creating few-shot examples solely from input codes, eliminating the need for domain expertise. Each example includes correct and buggy programs labeled with the buggy line's location. Our method combines code summarization and code generation to generate partially correct code, and code mutation to create the corresponding buggy version. This approach labels bug localization and repair tasks, serving as integrated few-shot demos in the prompt. In the following sections, we describe each component in details.

1) Code Summarization and Generation: First, we ask the LLM to summarize our input buggy code with a simple `<expected_behavior>` of `Analyst always provides a summary of the code inside the backticks.` and a simple `<question>` of `Please tell me what does the code do.`. Subsequently, we prompt the LLM to produce a code snippet, typically a function, that embodies the summarized information. In order to enhance the LLM's conditioning for the desired function, we extract the function prototype, typically found in the first line of the code, and incorporate it into the code generation prompt. Furthermore, we modify the role of the assistant to `Coder` as it is commonly associated with code generation tasks. In practice, given a code generation prompt, we will generate several candidates and evaluate them on a small set of public test cases provided with the questions to pick the most suitable generated code which is preferably bug-free. If none of the generated codes passed any test cases, we go back and re-generate them.

2) Code Modification: In this stage, our objective is to generate a modified version of the synthesized correct code with one controlled buggy line. Using the LLM, we provide appropriate instructions for natural modifications applicable across domains. Initially, we emphasize the need for a new code with specific modifications. To identify a line eligible for modification, we scan the entire program for variables. We examine each line involving any of the variables, excluding assignments of variable-primitive values (e.g., empty list, constants) and lines with print, assert, and raise keywords. From the identified candidates, we randomly select one line for modification and integrate it into the prompt template. Similar to the code generation pipeline, we go back and generate new modifications if none of them change our selected lines.

IV. Experiment

A. Datasets

QuixBugs is a benchmark for program repair [10], consisting of buggy implementations of 40 classical computer science algorithms in Python and Java. Each buggy program has a corresponding correct program with a single edit. Test suites with multiple input-output examples are provided for each program. In this work, we focus on the Python version of the dataset.

B. Data Processing

To prepare bug localization formats, we use the difflib Python package to identify discrepancies between the faulty program and its correct version. The output contains multiple changes, and we determine the number of bugs by counting non-equal changes. We convert batch changes into single-line changes for line-based bug localization. The input includes the faulty code, and the output is a list of buggy lines represented by their indices. Practically, we take only the first predicted line as our answer because QuixBugs has only one bug per program. To address the program repair task, the input includes the faulty program and its corresponding ground-truth buggy line indices. The desired output is simply the model-generated corrected program.

C. Evaluation Metrics

1) Bug Localization Metrics: To evaluate the bug localization results, we implement several standard metrics:

- **Top-1 accuracy** [31] $A^{top-1} \in [0.0, 1.0]$. Given a localization prompt, we ask our LLMs to output only one answer and compare it with the ground truth.
- **Top-3 accuracy** [31] $A^{top-3} \in [0.0, 1.0]$. In addition to A^{top-1}, we output 1 line before and 1 line after the model prediction as buggy lines. We argue that LLM sometimes can be confuse between indices starting from 0 and indices starting from 1.

2) Bug Fixing Metrics:

- **Exact Match Accuracy** [6] $A^{exact} \in [0.0, 1.0]$. Given a repair prompt, we ask our LLMs to output the program after fixing the bug. If a program is exactly equal to its corresponding correct version, we count the result as correct.
- **Execution Accuracy** [6] $A^{exe} \in [0.0, 1.0]$. Given a fixed program, we run it with the hidden test cases of the problem. If the program passed all the tests, we count the result as correct.

D. Large Language Model Backbones

In this work, we use 2 different LLM backbones: Galpaca-30b, and GPT-3.5-turbo. While the former is a variant of the well-known open source LLM called LLaMA [32] from MetaAI, the latter is a proprietary software from OpenAI [9] that has already proven its peak performance on multiple tasks.

E. Baseline Methods

To create a fair comparison between baseline methods and ours, we only consider different LLM prompting approaches ranging from few-shot to zero-shot. All of the prompts are then fed to the same set of LLMs to answer the same questions without any exceptions.

- **Related fixed few-shot prompting**. In this baseline, we provide the prime number checking problem as a fixed few-shot example for all of the input buggy codes. We alter one line in a program meant for prime checking, treating the modified version as a bug-containing program. We follow the same data processing pipeline (refer to IV-B) for this example. The example choice is open, as long as it has no dataset link and consists of over three lines to avoid triviality
- **Unrelated fixed few-shot prompting**. A related example improves the prompt's performance, while an unrelated one could yield the opposite. To assess this, we manually design a function checking if a string starts with a number which should be unrelated to the algorithmic questions in the dataset.
- **Zero-shot prompting**. In this baseline, we simply ask the LLMs to answer the bug localization (program repair) question without any conditioning. This baseline serves as a lower bound to test the embedded knowledge in each LLM. Incorrect formats from the outputs are expected (see III-B, III-C).

F. Settings

In order to serve Galpaca-30b on our system consisting of 2 NVDIA RTX 3090Ti-24GB GPUs, we initialize the checkpoints with the 8-bit integer format (int8) parameters. With that setting, it takes 2 x 18 GB of GPU RAM. We implement our system with the open-source framework PyTorch. For the implementation of the selected backbone models, we use the publicly available source codes provided by the authors or the publicly released checkpoints. For GPT-3.5-turbo, we simply use the OpenAI API.

V. Result and Discussion

A. RQ1: How does our proposed method perform against other baselines

1) Bug Localization: In all prompts, we use only 1 set of few-shot example to fit the maximum tokens of 1024. According to Table I, our proposed method outperforms all other baseline methods with GPT-3.5-turbo backbone. However, we only rank the second place with Galpaca-30b backbone. This suggests that handcrafted examples created by experts still works as an upper bound, especially in the case of the model with less computing capacity.

2) Bug Fixing: To create a comprehensive comparison between our proposed method and the baseline methods, we set up 3 scenarios for bug fixing: fixing without bug location (1) [6], fixing with provided bug location (2) [6], and joint bug localize-fixing (3) where we use the predicted

TABLE I
BUG LOCALIZATION PERFORMANCE

Backbone	Zero-shot		Related fixed few-shot		Unrelated fixed few-shot		Generated few-shot (ours)	
	A^{top-1}	A^{top-3}	A^{top-1}	A^{top-3}	A^{top-1}	A^{top-3}	A^{top-1}	A^{top-3}
Galpaca-30b	0.0	0.05	**0.15**	**0.4**	0.075	0.2	0.125	0.25
GPT-3.5-turbo	0.15	0.35	0.125	0.375	0.075	0.275	**0.175**	**0.4**

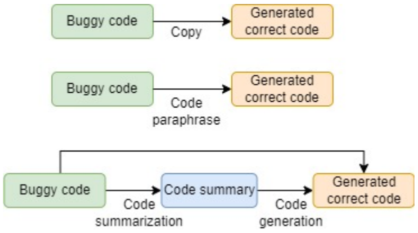


Fig. 3. 3 different approaches to generate (partially) correct codes from buggy codes. Top: Use directly buggy code. Middle: Paraphrase buggy code. Bottom: Summarize buggy code and generate new code (ours)

bug location as an input of the bug fixing. According to Table II, our proposed outperforms other baseline methods in 4 out of 6 scenarios. Similar to the previous comparison, expert-crafting examples still can outperform our method in some scenarios. Besides, the poorly unrelated fixed examples always has suboptimal performance. Thus, making effective in context examples requires in-depth knowledge of both algorithm and programming. Therefore, our proposed solution is a perfect candidate to eliminate human effort yet perform well in most scenarios.

B. RQ2: What is the contribution of the code summarize-generation block in our system?

To validate our chosen building blocks' efficacy, we compare our method, ”summarize buggy code and generate,” with two alternatives: ”use buggy code only” and ”paraphrase buggy code”. In the first baseline, we directly copy buggy code, falsely treating it as new and correct. This leads to misleading bug localization/fixing examples, as the modified code could accidentally improve. In the second baseline, we ask the LLM to generate code with the same meaning as the buggy version, often resulting in another buggy code or occasionally a correct one. This approach lies between our proposed method and naive copying in terms of code quality. Fig. 3 gives a closer look at the difference between the code generation methods. According to Table III, our proposed method dominates in 3 out of 4 scenarios. This clearly proves that the summarize buggy code and generate approach can generate examples with higher quality and thus can lead to better prompts (and results).

C. RQ3: Can we replace the LLM prompting with a heuristic approach for the code modification task?

To demonstrate the effectiveness of our chosen code modification component, we compare it against a heuristic approach [33]. In the heuristic approach, given the same input as the generated code and the chosen line to modify, we use a set of heuristic rules to randomly alternate between changing an operand or changing an operator. For example, if our chosen line contains a statement $a+b$, we can choose to either change the $+$ sign into another in the set $[-, *, /]$ or change a to c if c appears before this statement. We argue that our proposed LLM prompting for code modification give a more diverse set of modifications by not defining any fixed set beforehand. According to Table IV, our LLM prompting outperforms the heuristic approach in 3 out of 4 scenarios. Consequently, the freedom of choices from our proposed method leads to the best performance for the code modification task and thus the whole pipeline.

VI. CONCLUSION

In this work, we introduce a novel approach to automatically generate few-shot examples for both bug localization and bug fixing toward semantic mistakes in student programs. At the core of our approach is a large language model trained on code and textual instructions. We leverage code summarize-generation and code modification using LLMs to create zero-cost few-shot examples to boost the performance and stability of our main tasks. We evaluate our approach with 2 backbones: Galpaca, and GPT-3.5-turbo on QuixBugs dataset. With the obtained results, we demonstrate that the novel approach with its components creates an optimal choice to design few-shot examples for bug localization/fixing task without any human effort.

REFERENCES

[1] F. Long, P. Amidon, and M. C. Rinard, “Automatic inference of code transforms for patch generation,” *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[2] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, “Verifix: Verified repair of programming assignments,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, pp. 1 – 31, 2021.

[3] S. Gulwani, I. Radiček, and F. Zuleger, “Automated clustering and program repair for introductory programming assignments,” *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 465–480, 2018.

[4] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, “Refactoring based program repair applied to programming assignments,” *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 388–398, 2019.

[5] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, “Compilation error repair: For the student programs, from the student programs,” *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 78–87, 2018.

[6] J. A. Prenner and R. Robbes, “Automatic program repair with openai's codex: Evaluating quixbugs,” *ArXiv*, vol. abs/2111.03922, 2021.

[7] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, A. S. Hartshorn, E. Saravia, A. Poulton, V. Kerkez, and R. Stojnic, “Galactica: A large language model for science,” *ArXiv*, vol. abs/2211.09085, 2022.

TABLE II
BUG FIXING (PROGRAM REPAIR) PERFORMANCE OF 3 SCENARIOS: 1. BUG FIXING WITHOUT BUG LOCATION 2. BUG FIXING WITH BUG LOCATION 3. JOINT BUG LOCALIZATION-FIXING

Backbone	Scenario	Zero-shot		Related fixed few-shot		Unrelated fixed few-shot		Generated few-shot (ours)	
		A^{exact}	A^{exe}	A^{exact}	A^{exe}	A^{exact}	A^{exe}	A^{exact}	A^{exe}
Galpaca-30b	1	0.125	0.125	0.075	0.1	0.05	0.125	**0.15**	**0.15**
	2	0.15	0.2	0.175	0.225	0.15	0.2	**0.25**	**0.275**
	3	**0.1**	0.1	0.05	**0.125**	0.025	0.025	**0.1**	0.1
GPT-3.5-turbo	1	0.475	0.775	**0.525**	**0.875**	0.45	0.775	0.45	0.775
	2	**0.625**	0.825	0.5	0.7	0.575	0.8	**0.625**	**0.875**
	3	0.475	0.6	0.275	0.45	0.325	0.525	**0.525**	**0.625**

TABLE III
ABLATION STUDY 1: CODE GENERATION APPROACHES

	Use buggy code only		Paraphrase buggy code		Summarize buggy code and Generate	
Bug localization	A^{top-1}	A^{top-3}	A^{top-1}	A^{top-3}	A^{top-1}	A^{top-3}
Galpaca-30b	**0.15**	**0.45**	0.05	0.25	0.125	0.25
GPT-3.5-turbo	0.075	0.25	0.05	0.2	**0.175**	**0.4**
Bug fixing	A^{exact}	A^{exe}	A^{exact}	A^{exe}	A^{exact}	A^{exe}
Galpaca-30b	0.15	0.175	0.1	0.15	**0.25**	**0.275**
GPT-3.5-turbo	0.5	0.7	0.475	0.7	**0.625**	**0.875**

TABLE IV
ABLATION STUDY 2: CODE MODIFICATION APPROACHES

	Heuristic Modification		LLM Modification	
Bug localization	A^{top-1}	A^{top-3}	A^{top-1}	A^{top-3}
Galpaca-30b	0.0	**0.25**	**0.125**	**0.25**
GPT-3.5-turbo	0.075	0.175	**0.175**	**0.4**
Bug fixing	A^{exact}	A^{exe}	A^{exact}	A^{exe}
Galpaca-30b	0.075	0.15	**0.25**	**0.275**
GPT-3.5-turbo	**0.625**	**0.875**	**0.625**	**0.875**

[8] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford alpaca: An instruction-following llama model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[9] OpenAI, "Gpt-3.5: Language model," OpenAI, 2021. [Online]. Available: https://chat.openai.com

[10] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.

[11] A. Zeng, A. S. Wong, S. Welker, K. Choromanski, F. Tombari, A. Purohit, M. S. Ryoo, V. Sindhwani, J. Lee, V. Vanhoucke, and P. R. Florence, "Socratic models: Composing zero-shot multimodal reasoning with language," *ArXiv*, vol. abs/2204.00598, 2022.

[12] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[13] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

[14] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 467–477.

[15] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[16] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an rbf neural network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149–169, 2011.

[17] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon, "D&c: A divide-and-conquer approach to ir-based bug localization," *arXiv preprint arXiv:1902.02703*, 2019.

[18] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[20] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[21] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[22] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[23] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[24] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[26] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[28] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[29] V. Sanh, A. Webson, C. Raffel, S. H. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, T. L. Scao, A. Raja *et al.*, "Multitask prompted training enables zero-shot task generalization," *arXiv preprint arXiv:2110.08207*, 2021.

[30] Y. Zhang, G. Li, Z. Jin, and Y. Xing, "Neural program repair with program dependence analysis and effective filter mechanism," *ArXiv*, vol. abs/2305.09315, 2023.

[31] R. Gupta, A. Kanade, and S. Shevade, "Deep learning for bug-localization in student programs," 2019.

[32] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[33] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Improving automatically generated code from codex via automated program repair," *ArXiv*, vol. abs/2205.10583, 2022.