AlgoBugs: Benchmarking LLMs for Corner Case Test Generation in Competitive Programming

MD. MOHIMENUL ISLAM
Dept. of Computer Science and Engineering
Daffodil International University
Dhaka, Bangladesh
islam15-5725@diu.edu.bd

***Abstract*—**

***Index Terms*—Large Language Models, Code Generation, Software Testing, Test Case Generation, Competitive Programming, Benchmark, Fault Exposure**

I. Introduction

The rapid maturation of Large Language Models (LLMs) has transformed their role in software engineering, evolving from simple code completion tools to sophisticated agents capable of complex program synthesis [1], [2]. To rigorously evaluate their advancing capabilities, the research community has increasingly turned to the domain of competitive programming (CP). Unlike simpler coding tasks, CP problems demand a deep understanding of algorithms, data structures, and performance constraints, serving as a potent crucible for testing the limits of AI reasoning [3], [4].

However, the validity of any evaluation in this domain is fundamentally limited by the quality of the test cases used to judge correctness. Early benchmarks relying on existing or simple mutation-based tests were often insufficient, as they failed to probe the subtle corner cases that distinguish a truly robust algorithm from a plausible but flawed one [3], [5]. This critical dependency has catalyzed a new research direction focused on automated, high-quality test case generation, with LLMs themselves being employed to create more challenging and comprehensive test suites [6], [7].

The frontier of this research has become increasingly adversarial. Recent benchmarks such as TestCase-Eval [8] and TCGBench [9] have operationalized the "hacking" phase of programming contests into a formal task: **targeted fault exposure**. In this task, an LLM is given a problem description and a specific buggy solution, and it must generate a single test case input that causes the code to fail. This is a significant leap forward, as it directly measures an LLM's debugging and analytical reasoning. Yet, while these benchmarks have established that this is an exceptionally difficult task for modern LLMs [8], their evaluation remains monolithic. They can determine *if* a model can find a bug, but not *what kind* of bug it is adept at finding, or where its reasoning is weakest.

This paper introduces `AlgoBugs`, a novel benchmark designed to fill this precise diagnostic gap. We argue that to truly understand and improve LLM reasoning, we must move from a single success score to a fine-grained diagnostic profile. The core contributions of our work are:

- A new, challenging benchmark, `AlgoBugs`, specifically designed for the diagnostic evaluation of LLMs on the targeted fault exposure task.
- A sophisticated taxonomy of real-world algorithmic bugs, moving beyond naive classifications to capture subtle errors in areas like dynamic programming, greedy algorithms, and resource management.
- A dataset curated using a "gold-standard" workflow, mining recent {`Wrong Answer`, `Accepted`} submission pairs from the Codeforces platform to ensure bug authenticity and mitigate data contamination [11].

By evaluating models against this structured taxonomy, `AlgoBugs` can reveal not just an overall success rate, but a detailed profile of a model's reasoning capabilities—for instance, showing that a model excels at finding boundary errors but consistently fails to generate test cases that expose flawed greedy choices. This level of diagnostic insight is essential for guiding the next wave of improvements in LLM architecture and training.

II. Related Work

Our work is situated at the intersection of three key research areas: benchmarks for evaluating LLMs on code, automated test case generation, and bug-centric evaluation for fault exposure.

A. Benchmarking LLMs on Code

Early benchmarks for code generation, such as HumanEval [1] and MBPP, focused on function-level synthesis from docstrings. While foundational, these benchmarks do not fully capture the algorithmic reasoning required for complex problem-solving. To address this, the research community shifted towards the more demanding domain of competitive programming. The APPS benchmark [4] introduced thousands of problems from platforms like Codeforces, providing a more challenging test of algorithmic competence. Similarly, the CodeContests dataset [3] became a key resource for training and evaluating systems like AlphaCode. A persistent challenge in this area is data contamination, where models may have been trained on problems present in the evaluation set. Recent work emphasizes using problems released after an LLM's

knowledge cutoff date to ensure a true test of reasoning rather than memorization [11], [21].

B. Automated Test Case Generation

The quality of evaluation is directly tied to the quality of the test cases. Recognizing that manually crafted tests are a bottleneck, researchers have explored automated methods. Traditional techniques like mutation testing were adapted for code datasets like CodeContests [3], but often struggle to generate inputs that satisfy the complex constraints of CP problems [5], [22].

More recently, LLMs themselves have been employed as test generators. The CodeContests+ paper introduced a sophisticated Generator-Validator (G-V) agent system, where one LLM agent generates a program to produce test cases (including "corner cases" and "tricky cases"), and another agent validates their correctness against the problem constraints [5]. This represents the state-of-the-art in generating high-coverage test suites. However, the primary goal of such systems is often to create better datasets for training models via reinforcement learning, rather than to benchmark the test generation capability itself [5], [22].

C. The Adversarial Turn: Fault Exposure Benchmarks

The most recent and relevant line of work has shifted from general test generation to a more adversarial task. This shift was enabled by the creation of large-scale datasets of real-world bugs, such as Codeflaws [7] and FixEval [13], which provide pairs of buggy and correct submissions from programming contests.

Building on this foundation, new benchmarks have emerged to evaluate an LLM's ability to perform targeted debugging. **TestCase-Eval** [8] introduced two pivotal tasks: "Fault Coverage" (evaluating a test suite's ability to find bugs in a large pool) and **"Fault Exposure"** (evaluating an LLM's ability to generate a single test case to break a *specific* buggy solution). Similarly, **TCGBench** [9] evaluates the ability of LLMs to generate *programs* that produce these targeted, bug-exposing test cases. These works established that targeted fault exposure is an extremely challenging task where SOTA models lag significantly behind human experts [8], [9].

D. Research Gap and Our Contribution

While TestCase-Eval and TCGBench have defined the important task of targeted fault exposure, their evaluation remains monolithic. They measure *if* a model can find a bug, but not *what kind* of bug it is adept at finding. This is the critical gap our work addresses.

`AlgoBugs` builds directly on the targeted fault exposure paradigm but introduces a crucial diagnostic layer. By curating a dataset of real-world bugs and classifying each one according to a sophisticated, multi-dimensional taxonomy (see Section III-A), we can move beyond a single success score. Our benchmark is designed to reveal the specific strengths and weaknesses of an LLM's reasoning, providing a detailed profile of its performance across different categories of algorithmic flaws. This diagnostic capability, which is currently absent in the literature, is essential for understanding the failure modes of current models and guiding future research.

III. Methodology

This section details the systematic approach designed to create and utilize the `AlgoBugs` benchmark for evaluating the corner case test generation capabilities of Large Language Models (LLMs). The methodology, depicted in Fig. 1, is divided into two primary stages: the design and curation of the benchmark dataset, and the experimental protocol for evaluating LLMs using this dataset.

A. The AlgoBugs Benchmark: Design and Curation

The core contribution of this work is the creation of a novel, diagnostic benchmark named `AlgoBugs`. The design philosophy is to move beyond monolithic performance scores and enable a fine-grained analysis of LLM reasoning capabilities against a structured classification of algorithmic errors. The curation process is inspired by the rigorous standards set by established bug-centric datasets like Defects4J [20] and Codeflaws [7].

1) A Taxonomy of Algorithmic Corner Cases: To enable diagnostic evaluation, we first established a formal taxonomy of common, yet challenging, bug types found in competitive programming. This taxonomy moves beyond naive classifications (e.g., simple complexity traps) to capture the subtle errors that differentiate expert human programmers from automated systems. The taxonomy is grounded in patterns observed in the literature [16] and practical competitive programming experience. The main categories are:

- **Algorithmic Design and Logic Errors:**
  - *Incorrect DP State/Transition:* The overall algorithmic paradigm (e.g., dynamic programming) is correct, but the state representation or the transition logic is flawed.
  - *Flawed Greedy Choice:* The solution employs a greedy heuristic that is correct for most cases but fails on specific inputs designed to trick the greedy choice.
  - *Misunderstanding of Problem Constraints:* The solution fails to handle a subtle requirement of the problem, such as a graph being disconnected or containing self-loops [16].
- **Resource and Performance Failures:**
  - *Time Limit Exceeded (TLE):* This includes not only naive complexity bugs but also high-constant-factor solutions that fail on maximum-sized inputs, or algorithms that are vulnerable to worst-case inputs (e.g., a long chain in a graph traversal). This is a known weakness for LLMs [16], [17].
  - *Memory Limit Exceeded (MLE):* Bugs that are exposed by inputs forcing the creation of unexpectedly large data structures.
- **Boundary, Edge Case, and Data Type Errors:**

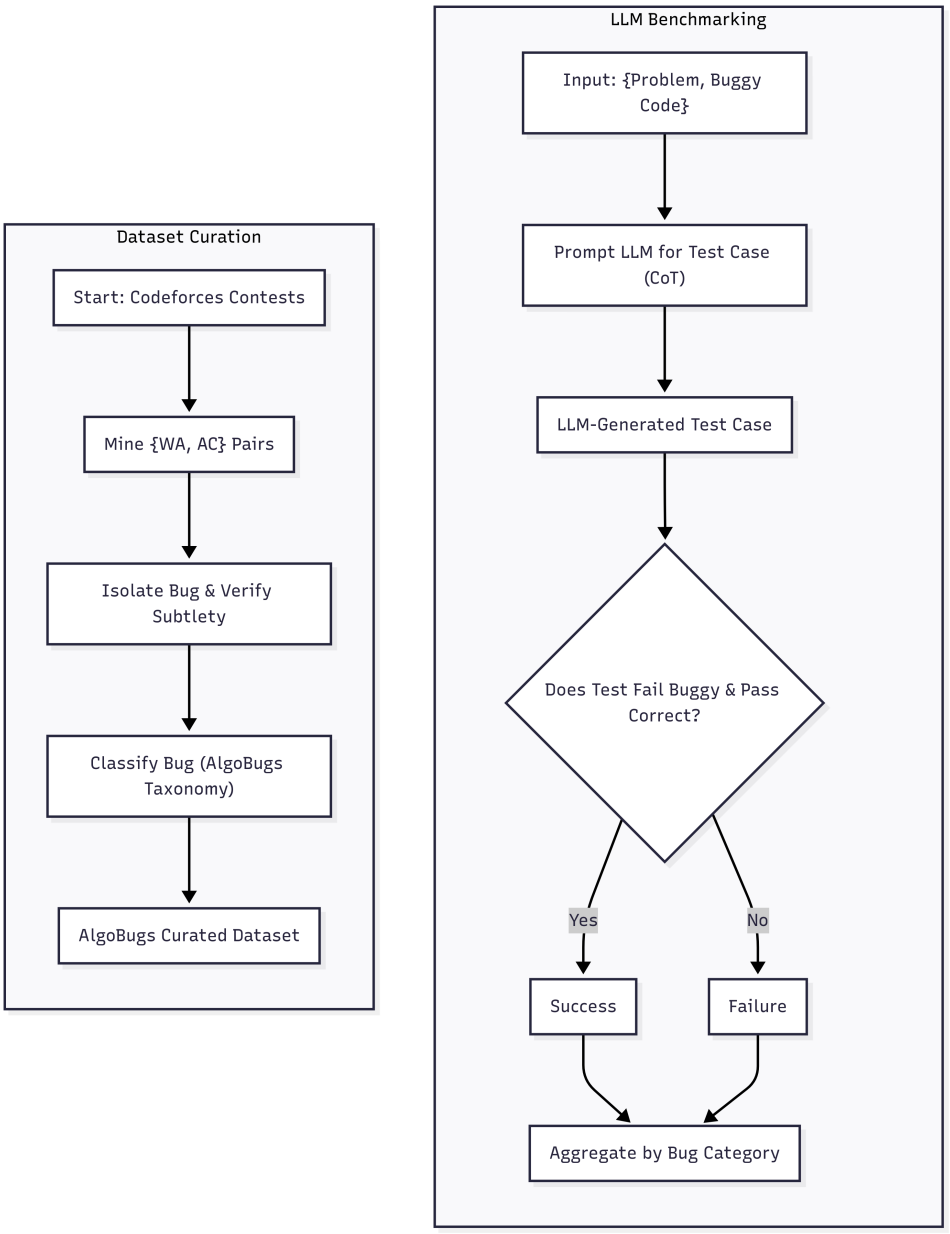

Fig. 1. The end-to-end methodology for the AlgoBugs benchmark, from data curation to diagnostic evaluation.

- *Numeric Boundary/Overflow:* Errors related to minimum/maximum constraint values, zero, or intermediate calculations exceeding the capacity of standard data types (e.g., `int` vs. `long long`).
- *Structural Edge Cases:* Errors exposed by empty inputs, single-element collections, or inputs where all elements are identical [16].

2) Data Sourcing and Curation Protocol: To ensure the highest authenticity and scientific validity, `AlgoBugs` is populated with real-world bugs mined from the Codeforces platform. This approach follows a "gold-standard" workflow, mitigating data contamination and ensuring the bugs are representative of genuine human errors [11], [13], [14].

The curation protocol for each data point is as follows:

1) **Problem Selection:** Problems are selected from recent Codeforces contests held after the knowledge cutoff dates of the LLMs being evaluated, to minimize data contamination [21].
2) **Pair Mining:** We identify submission pairs from the same user for the same problem, consisting of a "Wrong Answer" (or TLE/MLE) submission followed by an "Accepted" submission.
3) **Bug Isolation:** The two submissions are compared using a `diff` utility. Only pairs with small, localized changes are kept to ensure the bug is isolated.
4) **Subtlety Verification:** The buggy solution is compiled

and run against the public sample test cases provided in the problem description. The data point is only accepted if the buggy solution *passes all public sample tests*. This is a critical quality gate to ensure the bug is a true corner case and not a trivial error.

5) **Taxonomy Classification:** The isolated bug is manually analyzed and classified according to the taxonomy defined in Section III-A1.

3) Dataset Structure: Each curated data point is stored in a structured, machine-readable format. A master `metadata.json` file indexes each problem, its buggy variants, the bug category, a natural language description of the bug, and paths to the relevant source files (`problem_statement.md`, `solution_correct.cpp`, and `solution_buggy.cpp`). This organization ensures clarity and reproducibility.

B. Experimental Setup

The experiment is designed to evaluate an LLM's ability to perform targeted fault exposure. This task was chosen as it represents a more challenging and novel research frontier compared to general test suite generation [8], [9].

1) The Targeted Fault Exposure Task: For each data point in the `AlgoBugs` benchmark, the LLM is provided with the full problem description and the source code of a single buggy solution. The LLM's task is to analyze the code and generate a single test case input that causes the buggy solution to fail.

2) Prompt Engineering Strategies: To ensure a comprehensive evaluation, we will test multiple prompting strategies to assess their impact on the model's reasoning capabilities.

- **Zero-Shot Prompt:** A baseline prompt that directly asks the LLM to generate a "hacking" test case without providing any examples [10].
- **Few-Shot Prompt:** Provides 2-3 examples of other {`buggy_code`, `hacking_test_case`} pairs to give the model context before presenting the target task [18], [19].
- **Chain-of-Thought (CoT) Prompt:** The primary experimental prompt. It explicitly instructs the LLM to first analyze the buggy code step-by-step to identify a potential flaw, and *then* generate a test case based on that analysis. This technique is known to improve performance on complex reasoning tasks [7], [14], [15].

C. Evaluation Metrics

The evaluation is designed to be objective and execution-based, which is more reliable than match-based metrics [1], [13].

1) **Test Case Validity:** An LLM-generated test case is first validated by running it against the `solution_correct.cpp`. The test case is considered valid only if the correct solution passes.
2) **Fault Exposure Success Rate (Primary Metric):** For a valid test case, we measure success as a binary outcome. The "hack" is successful (TRUE) if the `solution_buggy.cpp` fails on the generated test case (Wrong Answer, TLE, MLE, or Runtime Error). Otherwise, it is a failure (FALSE).
3) **Success Rate per Bug Category (Diagnostic Metric):** The primary novelty of our evaluation is to aggregate the success rates based on the bug categories defined in our taxonomy. This will allow us to produce diagnostic results, such as "Model X achieves an 80% success rate on *Boundary Errors* but only 30% on *Resource and Performance Failures*."

REFERENCES

[1] M. Chen, et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.
[2] D. Guo, et al., "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," arXiv preprint arXiv:2401.14813, 2024.
[3] Y. Li, et al., "Competition-level code generation with alphacode," Science, vol. 378, no. 6624, pp. 1092-1097, 2022.
[4] D. Hendrycks, et al., "Measuring coding challenge competence with APPS," arXiv preprint arXiv:2105.09938, 2021.
[5] Z. Wang, et al., "CodeContests+: High-quality test case generation for competitive programming," arXiv preprint arXiv:2506.05817, 2025.
[6] Z. Wang, et al., "TestEval: Benchmarking large language models for test case generation," in Findings of the Association for Computational Linguistics: NAACL 2025, 2025.
[7] S. H. Tan, et al., "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, pp. 89-91.
[8] Z. Yang, et al., "Can LLMs generate high-quality test cases for algorithm problems? TestCase-Eval: A systematic evaluation of fault coverage and exposure," arXiv preprint arXiv:2506.12278, 2025.
[9] Y. Cao, et al., "Can LLMs generate reliable test case generators? a study on competition-level programming problems," arXiv preprint arXiv:2506.06821, 2025.
[10] N. Dong, "A Survey of Prompt Engineering for Large Language Models," Medium, Sep. 2024.
[11] J. Huang, et al., "Competition-level problems are effective LLM evaluators," arXiv preprint arXiv:2402.15886, 2024.
[12] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.
[13] M. M. A. Haque, et al., "FixEval: Execution-based Evaluation of Program Fixes for Competitive Programming Problems," in Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, 2023, pp. 1-14.
[14] J. Wei, et al., "Chain-of-thought prompting elicits reasoning in large language models," in Advances in Neural Information Processing Systems, 2022.
[15] F. L. et al., "Uncertainty-Guided Chain-of-Thought for Code Generation with LLMs," arXiv preprint arXiv:2503.15341, 2025.
[16] Z. Wang, et al., "An empirical study on the robustness of LLM-generated code," arXiv preprint arXiv:2503.20197, 2025.
[17] Y. Cao, et al., "Can LLMs generate reliable test case generators? a study on competition-level programming problems," arXiv preprint arXiv:2506.06821, 2025.
[18] PromptPanda, "Few Shot Prompting Explained: A Guide," promptpanda.io, 2025.
[19] Learn Prompting, "Shot-Based Prompting: Zero-Shot, One-Shot, and Few-Shot Prompting," learnprompting.org, 2025.
[20] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 437-440.
[21] J. Huang, et al., "Competition-level problems are effective LLM evaluators," arXiv preprint arXiv:2402.15886, 2024.
[22] Z. Wang, et al., "CodeContests+: High-quality test case generation for competitive programming," arXiv preprint arXiv:2506.05817, 2025.