

Quantum-Guided Test Case Minimization for LLM-Based Code Generation

Huixiang Zhang

*Department of Computer Science
Lakehead University
Thunder Bay, Canada
hzhan102@lakeheadu.ca*

Mahzabeen Emu

*Quantum Communications and Computing Research Center
Department of Electrical and Computer Engineering
Memorial University of Newfoundland
St. John's, NL, Canada
memu@mun.ca*

Abstract—Precisely controlling Large Language Models (LLMs) to generate efficient and concise code is a central challenge in software engineering. We introduce a framework based on Test-Driven Development (TDD) that transforms code specification into a combinatorial optimization task. The framework first prompts an LLM to generate a test suite, then formulates the Test Case Minimization (TCM) problem as a Quadratic Unconstrained Binary Optimization (QUBO) model. This QUBO paradigm is compatible with both classical solvers and emerging hardware such as quantum annealers. Experimentally, quantum annealing solves the core TCM task 16 times faster than simulated annealing. This performance underpins our end-to-end framework, which reduces total token consumption by 36.5% and significantly improves code quality. This work demonstrates a powerful synergy between generative AI and combinatorial optimization in software engineering, highlighting the critical importance of precise model formulation.

Index Terms—large language models, test case minimization, quadratic unconstrained binary optimization

I. INTRODUCTION

Large Language Models (LLMs) are changing software development. However, their powerful code generation capabilities are often accompanied by a core challenge: a lack of output control. Developers frequently receive code that is functionally correct but unnecessarily redundant and insufficient. The critical task has become how to precisely guide an LLM to produce concise code [1].

Test-Driven Development (TDD) offers a paradigm to address this. The test cases serve as a precise and unambiguous guide for the code. A new problem arises when applying TDD to LLMs. Feeding a comprehensive test suite directly into a model incurs prohibitive token and inference costs due to its large size, making it economically and practically infeasible. This creates a central conflict. We need the precision of test cases, but we cannot afford the cost of their completeness [2].

To resolve this conflict, we position the Test Case Minimization (TCM) problem at the heart of our workflow. We adopted a TCM Quadratic Unconstrained Binary Optimization (QUBO) model for our framework [3]. This provides a powerful mathematical framework and, more importantly, opens the door to acceleration using frontier technologies like quantum computing. In modern software development, where Continuous Integration/Continuous Deployment (CI/CD) is the standard, testing efficiency dictates delivery velocity. A slow

test selection process is a major bottleneck. Our work shows that the over 10 times speedup from quantum computing can transform TCM from an infrequent, offline task into a more easy process. This research therefore aims to answer the following central questions:

- Can an automated framework combining TDD with combinatorial optimization improve the quality and reduce the cost like token consumption of LLM-generated code?
- For the core optimization task within this framework, what is the practical performance advantage of quantum computing over classical algorithms, and what does this advantage mean for software development pipelines?

II. FRAMEWORK OVERVIEW

Our framework is designed to automatically generate and optimize test suites for code modules. It leverages LLMs for test case generation and employs combinatorial optimization to minimize the size of the test set. The workflow consists of three stages: (1) Comprehensive Test Generation, (2) Test Suite Optimization, and (3) Optional Code Refinement.

The input to this stage is an existing code module that requires testing. Our objective is to generate a comprehensive test suite for this code, where each test has an explicit functional label. We first provide the code under test to an LLM. Using a specifically designed prompt, we instruct the LLM to analyze the code and generate a highly diverse and redundant test suite, denoted as $T_{comprehensive}$. A core requirement of this prompt is that the LLM must assign a clear label to each generated test case, indicating the specific feature that it is designed to validate. This explicit mapping from tests to features is a critical prerequisite for the subsequent optimization stage.

The core task of this stage is to reduce the comprehensive test suite $T_{comprehensive}$ into an efficient, minimal subset T' , using the mapping of the test to the characteristic of the previous stage. We formulate this task as a Quadratic Unconstrained Binary Optimization (QUBO) problem. We assign a binary decision variable t_i to each test case in $T_{comprehensive}$, where $t_i = 1$ if the test is selected and 0 otherwise. Our goal is to find an optimal assignment of variables that minimizes a unified objective function.

$$\min \left(\sum_i \text{cost}(t_i) + \lambda \sum_j \text{Penalty}_j \right)$$

This objective function intuitively balances two core goals:

- $\sum_i \text{cost}(t_i)$: This is the cost term. Attempts to minimize the total cost of the selected test cases. The cost can be the number of tests where each $\text{cost}(t_i)$ is 1.
- $\lambda \cdot \sum_j \text{Penalty}_j$: This is the penalty term. For each functional requirement j that is not covered by any selected test, Penalty_j incurs a positive penalty value. The coefficient λ controls the strength of this penalty.

Solving this QUBO model yields a combination of test cases that minimizes the total cost while ensuring that all functional requirements are covered. This optimal combination forms our final minimized test suite, T' .

A. Optional Code Refinement

This optional stage leverages the minimal test suite T' to refine and improve the quality of the original code. We provide the original code and T' to an LLM, instructing it to refactor for efficiency and readability under the strict constraint that all tests must pass.

The minimal test suite T' acts as a precise and unambiguous functional specification in this process. It establishes a clear boundary for the LLM's modifications. This allows the model to safely remove redundant logic or unnecessarily complex implementations from the code without breaking core functionality. The final output of this process is a refined, higher-quality version of the code, term as C_{final} .

III. EXPERIMENTAL EVALUATION

Our experimental evaluation is designed to answer two core questions: 1) What is the performance of quantum computing versus classical methods on the core task of test suite optimization? 2) What is the impact of our TDD framework on the final output of LLM-based code generation?

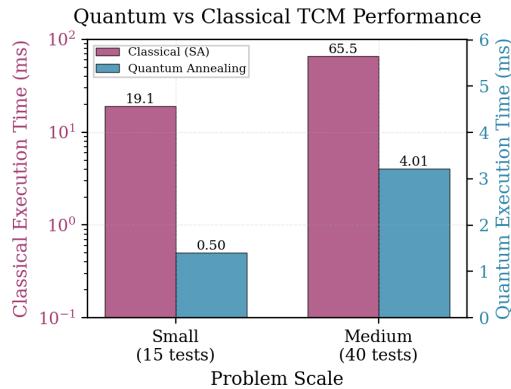


Fig. 1. Quantum Annealing are faster than Simulated Annealing for the core TCM optimization task.

We first benchmarked the core task of our framework's Test Suite Optimization stage. We compared the solving time

of a physical quantum annealer against a classical simulated annealing solver on small- and medium-scale TCM problems.

As shown in Figure 1, the quantum annealer shows a speed advantage. In the medium-scale problem, its resolution time (4.008 ms) was more than 16 times faster than that of simulated annealing (65.5 ms). This leap in performance is significant. It suggests that the test optimization process can be transformed from a slow, offline batch job into a real-time tool suitable for per-commit execution within a CI/CD pipeline.

Next, we evaluated the impact of our framework on the LLM code generation task. We defined two modes:

- **Baseline:** The original full test suite generated by the LLM is used directly as the specification to guide the generation of the code.
- **TDD-Guided:** Our framework is used to first find a minimal test suite via QUBO optimization, which then guides the code generation.

We assessed the final generated code in three dimensions: total token consumption or cost, code quality, and generation time. For code quality, we used Cyclomatic Complexity, an industry-standard metric that evaluates code complexity by counting the number of linearly independent paths. A lower score indicates simpler, more maintainable code.

TABLE I
PERFORMANCE COMPARISON: BASELINE VS TDD-GUIDED

Metric	Baseline	TDD	Improvement
Total Tokens	897.75	570.25	36.5%
Complexity Score	27.75	20.50	26.1%
Generation Time (s)	40.23	18.38	54.3%

The results, presented in Table 1, show that our TDD-Guided approach has improvements across all metrics:

- **Cost Reduction:** Total tokens were reduced by **36.5%**, which translates directly to lower API costs and reduced computational requirements.
- **Quality Improvement:** The code's cyclomatic complexity was lowered by **26.1%**, providing quantitative evidence that the code generated from a minimal test set is structurally simpler and of higher quality.
- **Efficiency Gain:** The generation time was shortened by **54.3%**. This demonstrates that despite the added optimization step, providing the LLM with a highly focused and concise input dramatically reduces its inference burden, leading to a faster overall process.

REFERENCES

- [1] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.
- [2] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen, "CodeT: Code Generation with Generated Tests," in *Proceedings of the The Eleventh International Conference on Learning Representations*. OpenReview.net, 2023.
- [3] X. Wang, A. Muqeet, T. Yue, S. Ali, and P. Arcaini, "Test Case Minimization with Quantum Annealers," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, pp. 5:1–5:24, 2024.