Test Driven Generation:
Optimizing Multi-Agent Code Generation By Conditioning On Unit Tests
(Experimental Protocol)

Matthew Billman
XCS224U / 70 Remsen St #7F, Brooklyn NY 11201
mgbvox@gmail.com
https://github.com/mgbvox/tdg

1 Supplementary Background

Three new papers came to my attention recently that have required me to update my hypothesis. These are briefly summarized below.

Self-Edit (Zhang et al., 2023) included human-defined test cases in the code generation prompt and found this approach to improve generated output over prompt-only inputs. This was the topic of my original hypothesis, and validates it.

Code Chain of Thought (Code-CoT) generation (Huang et al., 2024a) used an iterative generate - evaluate - analyze flow for code and test generation, establishing a pipeline for the incremental improvement of generated code, albeit within a single context window (a single chat-style LLM session).

AgentCoder (Huang et al., 2024b) adapted Self-Edit and Code-CoT, introducing a multi-agent framework wherein one or more LLMs undertake different "roles" in a collaborative, iterative Chain of Thought generation process. The roles used were *Programmer*, *Designer*, and *Executor*. They achieved state-of-the-art results on HumanEval$^+$, outperforming vanilla GPT-4.

2 Hypotheses

Prior work has demonstrated the value of providing code-level guidance (in the form of tests and other example code snippets) alongside natural language prompts for code generation.

I hypothesize that integrating human-generated tests into to the multi-agent Chain of Thought (CoT) paradigm espoused by AgentCoder and Code-CoT, as well as adding a *Navigator* role to reason on inputs and outputs at a more abstract level in several steps of the generation process, should further improve performance on code generation benchmarks.

Specifically, I will introduce and explore the feasibility of a framework I call Test-Driven Generation (TDG), leveraging principles from Test-Driven Development (TDD) to formalize Pair Programming Navigator guidance by requiring LLM Driver output to pass tests as a prerequisite for acceptance.

I will further explore recursive code generation in the case of Driver generation failure, feeding test error logs back into the Driver LLM in a post-mortem pass over several hops (as needed) to proactively fix bugged Driver output. This will be attended to by multiple LLMs collaborating together in the style of AgentCoder.

3 TDG Algorithm

TDG will be implemented as a pytest plugin, running at test time either within a terminal or interactive development environment debugger (PyCharm was used for development).

For a given problem description prompt p, test set $t \in T$, an optional user-defined metric set $m \in M$, role set $r \in R$, and sample count k:

Define our generative agents as G_{role}, where valid roles are Navigator (nav), Programmer (dev), and Test Designer (test).

Define $\xi_{test}(c, t) \in 0, 1$ to be the test executor module operating on candidate c for test t, where 0 indicates test failure, and 1 indicates test passage. The ξ_{test} module will be an inner pytest invocation executed on dynamically generated python modules derived from c and t.

Define $\xi_{met}(c, m) \in \mathbb{R}$ to be the metric executor module operating on candidate c for metric m. The ξ_{met} module will be a python function operating on the generated code string as plain text and user-defined metric callback functions.

Generate a candidate solution set $c \in C$ where $len(C) = k$ via the following pipeline:

1. Generate a navigator analysis prompt $N_{pre} = G_{nav}(p, T)$ from the prompt and human derived tests.

2. Augment T with additional tests $T_{gen} = G_{test}(N_{pre}, p, T)$ from G_{nav} and the aforementioned prompt and tests, giving $T_{aug} = T + T_{gen}$.

3. Generate our candidate solution set $C = G_{dev}(N_{pre}, p, T_{aug})$.

4. Select optimal solution $\hat{c} \in C$:

   - Let a given candidate's score be defined as:

     $$Score(c) = \prod_{T} \xi_{test}(c, t) * \sum_{M} \xi_{met}(c, m) \tag{1}$$

     Note that we must take the product over $\xi_{test}(c, t)$ since all tests must pass for a candidate to be valid in the first place.
   - In the case where no user-defined metrics exist (which will be the case for evaluation against our benchmark datasets):

     $$\sum_{M} \xi_{met}(c, m) = 1 \iff len(M) = 0 \tag{2}$$

   - Select $\hat{c}$ from C by

     $$\hat{c} = \max_{C}(Score(c) | Score(c) > 0) \tag{3}$$

5. If $\hat{c}$ exists, return $\hat{c}$.

6. If no $\hat{c}$ exists:

   (a) Collect tracebacks from the top $f | f <= k$ test runs to obtain failure set F.

   (b) Analyze tracebacks to obtain $N_{fail} = G_{nav}(F, C, N_{pre}, p, T_{aug})$

   (c) Pass this back into TDG as additional Navigator context and iterate until $\hat{c}$ is found.

4 Data

For model evaluation, I will use the HumanEval^{+} dataset (Liu et al.), a successor to OpenAI's HumanEval (Chen et al., 2021). HumanEval^{+} is not properly formatted for pytest, so test examples will be modified to conform to pytest's specific formatting requirements. All input/output mappings in HumanEval^{+} test cases will be preserved.

Given that TDG requires test cases to be included in the initial model prompt, I will include a random sub-sample of test cases from the training set.

Time permitting, I will also evaluate on the MBPP dataset (Austin et al., 2021), adapting test cases similarly.

5 Metrics

See [3] for additional metrics.

In addition to the metrics defined in [3], I will evaluate the performance of set C over T by calculating $pass@k$.

Further, the optimal $\hat{c}$ will be compared with the canonical solution code sample c^{*} by removing all docstrings and comments, then calculating:

- Exact String Match
- Levenstein String Distance
- GPT4 Embedding Cosine Distance (Time and Cost Permitting)

I do not expect $\hat{c}$ to closely resemble c^{*} frequently, as there are often many optimal ways to solve a given problem. It will be nevertheless interesting to see what patterns of similarity (if any) emerge.

6 Models

I will build a model-agnostic interface class allowing the use of arbitrary generative LLMs. Development will be done with OpenAI's GPT-3.5-Turbo, with evaluation performed on GPT4-Turbo, Google's Gemini, and Anthropic's Claude-3-Opus.

As TDG inference is performed entirely in context, no models need to be fine-tuned for the initial portion of the development. Time permitting, I may investigate fine-tuning GPT4 on the codeparrot/github-code-clean dataset via the GPT Fine Tune API.

7 General Reasoning

Given that tests typically encode substantial information about the ultimate structure of the system under test, including tests in the initial prompt should result in generated code that is more aligned with the intent of the developer.

Multi-agent generation via AgentCode has already demonstrated remarkable improvement over existing state-of-the-art code generation models. If multiple task-specific agents (Programmer, Designer, Executor) are better than a task-agnostic

LLM agent, I suspect including roles from other proven collaborative programming paradigms (Pair Programming, specifically the Navigator role) might also augment performance.

This begs the question - is the performance increase seen with the AgentCoder roles due to their specific intent or simply to the fact that there are more LLMs thinking about the same problem? If due to role intent, what other specific roles (Navigator, Project Manager, Market Analyst...) might demonstrate improved performance? If due to increased LLM count, can we demonstrate an increase in performance by removing roles from the prompt entirely, simply asking multiple LLMs the same question and aggregating the results? Can we infer - or can yet another LLM infer - an optimal number or set of roles for a code generation process, given the task at hand?

This research will operate under the assumption that role intent is key, and that adding roles from other programming disciplines should beget a performance increase. Further research could and should absolutely be done to test these other hypotheses.

8 Summary of Progress

I have already implemented a single-agent generation plugin at `https://github.com/mgbvox/tdg`. The next steps will be to add the multi-agent pipeline and implement failure recursion.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. ArXiv:2108.07732 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. ArXiv:2107.03374 [cs].

Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024a. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation. ArXiv:2308.08784 [cs] version: 2.

Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2024b. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. ArXiv:2312.13010 [cs] version: 2.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct?

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. ArXiv:2305.04087 [cs].